THESIS FOR THE DEGREE OF DOCTOR OF PHILOSOPHY

On Impact and Tolerance of Data Errors with Varied Duration in Microprocessors

Örjan Askerdal

Department of Computer Engineering
School of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, SWEDEN, 2003

On Impact and Tolerance of Data Errors with Varied Duration in Microprocessors

Örjan Askerdal

ISBN 91-7291-285-5

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie 1967
ISSN 0346-718X

School of Computer Science and Engineering
Chalmers University of Technology
Technical Report 12D
ISSN 1651-4971

Department of Computer Engineering
School of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg
Tel. +46 (0)31-772 10 00
www.ce.chalmers se

Author email: askerdal@ce.chalmers.se

Chalmers Reproservice
Göteborg, Sweden, 2003

On Impact and Tolerance of Data Errors with Varied Duration in Microprocessors

Örjan Askerdal
Department of Computer Engineering, Chalmers University of Technology

Abstract

The evolution of high-performance and low-cost microprocessors has led to their almost pervasive usage in embedded systems such as automotive electronics, smart gadgets, communication devices, etc. These mass-market products, when deployed in safety-critical systems, require safe services albeit at low recurring costs. Moreover, as these systems often operate in harsh environments, faults will occur during system operation, and thus, must be handled safely, i.e., tolerated.

This thesis investigates the efficiency of adding software-implemented fault tolerance techniques to commercial off-the-shelf (COTS) microprocessors. Specifically, the following problems are addressed:

- Which faults need to be tolerated considering the architecture, implementation and operational environments for COTS processors?
- Which software-implemented fault-tolerance techniques are effective and efficient to use?
- How can the efficiencies of such designs be evaluated?

The main contribution of this thesis is the development of novel approaches for estimating the effects of data errors with varied duration, and for ascertaining the efficiency of applied fault-tolerance techniques. These approaches are based on identifying the characteristics that determine which effects data errors will have on the system. Then these characteristics can be varied at a high abstraction level and the effects observed.

The first approach is based on response analysis methods for understanding the effects of data errors on control systems. The second is a *VHDL* simulation-based fault injection method, based on insertion of specific components (so-called saboteurs) for varying the characteristics. As most system development processes start at a high abstraction level, we expect our approaches to be applied early in the process, and be a useful complement to traditional post-design assessment approaches such as fault-injection.

Keywords: error effect analysis, error propagation analysis, fault injection, fault tolerance, dependability

List of Papers

This thesis is based on, and extends the following work:

- Askerdal, Ö., Gäfvert, M., Hiller, M. and Suri, N., "A Control Theory Approach for Analyzing the Effects of Data Errors in Safety-Critical Control Systems", Pacific Rim International Symposium on Dependable Computing, pp.105-114, 2002.
- Gäfvert, M., Wittenmark B. and Askerdal, Ö., "On the Effect of Transient Data Errors in Controller Implementations", to appear in American Control Conference, 2003.
- Askerdal, Ö., Suri, N. and Torin J., "Use of Complementary Techniques for Detection of Low-Level Errors Caused by both Transient and Persistent Faults, Based on Analysis of Double Execution", Technical Report No. 00-24, Department of Computer Engineering, Chalmers University of Technology, Sweden, 2000.
- Askerdal, Ö. and Suri, N., "On-Line Error Detection in Control Systems", Technical Report No. 01-17, Department of Computer Engineering, Chalmers University of Technology, Sweden, 2001.
- Askerdal, Ö., Wiklund, K., Mendelson, A. and Suri, N., "Accurate Impact Evaluation of Hardware Faults with Varied Time Duration", In review, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, 2003.

Acknowledgments

It is now about six years since I joined the Department of Computer Engineering, Chalmers. So much has happened during this time that I can barely remember who the boy starting was. I have of course improved my research skills during these years, but there is so much more I learned about people, life, love...

However, I would like to take this opportunity to thank you all for supporting me during these years. Thank You!

There are though of course a few people I would like to thank specifically...

First, of course, I need to thank my family (My parents Bo and Monica, and my brother Mikael) who always supported me, especially when I didn't understood people, life, love...

Coming to the department, it was Lars-Åke Johansson that employed me as an "exjobbare" together with Jonas Wåhlström (at that time Olsson). Thank you both for getting me interested in distributed computer systems!

Through that employment, I started to know the other people in the "Bil-gruppen". Professor Jan Torin, Arne Dahlberg, Rolf Snedsböl, Henrik Lönn, Vilgot Claesson (at that time Klasson), and Kristina Forsberg (at that time Ahlström). After finishing my "civilingenjörs-examen", I was employed as a Ph.D. student by Professor Jan Torin who supported me for three years (Thank you, Jan!), and you all become my friends. Especially, I would like to thank Henrik for introducing me to how to do research, Rolf for introducing me to how to teach, and Kristina for all nice, and important, chats.

After three years Professor Neeraj Suri came around, this guy who I could not figure out (and still cannot), provided a lot of Yada, Yada, and who really CONFUSED me. But, he did not just confuse me, he also taught me a lot about research (how to write papers, how to make presentations, how to review work, etc.), introduced me to people, organized nice dinners, and supported me, for at least two and a half year. Thank You Neeraj!

Along with Neeraj, a group was formatted (an accidental typo, should be formed!), DEEDS, constituting of Vilgot Claesson (I cannot remember if it still was Klasson at this time or not), Martin Hiller, Arshad Jhumka, and later on also Robert Lindström, and Andréas Johansson. We have been a hard-working group, but there has always

been time for jokes, cakes, and everything else that makes life worth to live. You are all close friends of mine (girlfriends and wives included) and I owe you a lot!

During these years, I have also got to work with other people. Already from the start, Marcus Rimén (presently at Semcon) has with odd and even intervals discussed, criticized, and assisted my research. Thank you, Marcus! Stefan Asserhäll at Saab Ericsson Space has provided me with simulation models and often helped me out when I have had questions on the models. I would also like to thank the funders, managers, members of the projects I have been working in during my time as a Ph.D. student (X-By-Wire, DICOSMOS, PÅLBUS, SAAB). The work within these projects has always been interesting.

I would also particularly like to thank my other co-authors during these years: Kristian Wiklund (Ericsson), Avi Mendelson (Intel), Magnus Gäfvert (Lund Institute of Technology), Björn Wittenmark (Lund Institute of Technology), and Joakim Aidemark (Chalmers). It has been great to work with you all and I have really learned a lot.

I would also like to thank all people at this department for providing me with all the help and things which I have been taken for granted. Thank you so much!

Especially, I would like to direct a special thank you to all my fellow Ph.D. students which I have met during these years, at the department, at different councils, at floor-ball, etc. You are just too many to name, but you will not be forgotten!

I would also like to give a special thank you to all people that has given me comments on this thesis Per Johannessen, Håkan Forsberg, Daniel Eckerbert, Joakim Aidemark, Jonny Vinter, Vilgot Claesson, Martin Hiller, Arshad Jhumka, Robert Lindström, Andréas Johansson, Neeraj Suri, and Cristian Constantinescu. Your comments have been invaluable!

Finally, but absolutely not least, I would like to thank life and Klara for making me so fortunate to have someone to love as much as I love you! THANK YOU!

Now, a new chapter of my life is starting (which you cannot find in this thesis), hope to see you there!

Contents

CHAPTER 1

Introduction

Dependability has always been of prominent concern to mankind. The early humans fought to find and construct dependable settlements, shelters, clothes and tools to adapt to nature and survive. Looking back, one can say that this battle has been very successful, even if the resources are of uneven quality and erratically distributed and nature occasionally subverts the most careful of plans. In the modern world, based on our advances in technology, most people today associate dependability with the functionality and punctuality of different objects, for example: public transportation, banking, telecommunication etc.

Considering computers, the concept of dependability has also changed over time. The earliest computers were constructed of components such as relays and vacuum tubes that would fail as often as once every hundred thousand or million cycles, which is far too often to ensure correct completion of even modest calculations [Sieworek and Swarz, 1992]. This changed as more durable components, such as the transistor, were introduced which made computers become one of the most important and dependable tools for the human to make new discoveries. As an example, in the first space flight to the moon, computers were used to facilitate navigation. Today, the refined hardware development processes result in small size, high performance and low-cost devices. Consequently, computers are being pervasively used and also embedded in diverse products such as vehicles, home gadgets, communication devices etc.

The pervasiveness of our use of computers also results in our increasing dependence on their correct operation. In most cases a computer failure would only mean that the program running on the personal computer crashes or that the microwave

oven stops working. However, there are systems where a computer fault, if not handled correctly, will cause much greater damage and even impact human safety. Examples of such systems are brake-by wire systems in vehicles and navigation systems in aircraft among others. Additionally, the fact that people do not associate such systems with computers have the psychological effect that for these systems to be publicly accepted, the computers must be extremely dependable.

Another complicating aspect is that many systems are real-time systems, meaning that their operational safety is not only dependent on the system delivering the correct results but also at the correct time. Moreover, as many of these systems are complex and operate in harsh environments, it is hard to prevent faults from occurring. Thus such systems must inherently be designed to tolerate faults. Furthermore, the severe cost-sensitivity of the consumers and manufacturers requires cost-efficient solutions for provision of dependable devices and systems. Therefore, the development of highly dependable and cost-efficient computer systems is an extremely challenging task. *This thesis contributes to this task by exploring how microprocessors in such systems can be developed to be fault tolerant.*

The intent of this introduction is to describe how safety-critical systems are developed, and also to develop the context of the thesis research. First the concepts and terminology of safety, which is a specific attribute of dependability, used in this thesis are defined. Second, the development process of safety-critical systems and more specifically fault tolerant systems (systems that continue to operate correctly even in the presents of faults) is described. After that, the main functionalities and aspects of control systems are discussed, as these systems are one common type of safety-critical systems, and used in this thesis to exemplify different methods and techniques. Then, the specific research questions targeted and the contributions of this thesis are presented. Finally, the organization of the thesis is detailed.

1.1 Safety Definitions

Firstly, we outline our definitions of the different concepts used in the thesis. With a *component*, we mean a building block in the form of hardware, software or both. Specifically, we use the term *task* for software building blocks and *unit* for hardware building blocks. A *system* consists of several components of which at least one is a microprocessor and should provide a desired, specified service. Sometimes the term *computer system* is used to stress this fact. With a *system developer*, we mean a man or woman that develops system, i.e., intelligently conjoins components such that the specified service can be delivered.

When developing a system, it is always necessary to specify requirements on the

design so that it is possible to determine when a proper design has been reached. It is important that the specified requirements are measurable in some way so that it can be verified that the requirements are fulfilled. Therefore, it is necessary to exactly define what is meant with different concepts and properties. Throughout this thesis, the definitions originally developed in [Laprie, 1992], modified and expanded in [Avižienis *et al.*, 2001], are used.

Safety is classified as an attribute of dependability, which is defined as "the ability to deliver service that can justifiably be trusted". Depending on the application of the system, dependability can have different meanings, i.e., different attributes could be of interest:

- Availability: Readiness for correct service.
- Reliability: Continuity of correct service.
- Safety: Absence of catastrophic consequences on the user(s) and the environment.
- Confidentiality: Absence of unauthorized disclosure of information.
- Integrity: Absence of improper system state alterations.
- Maintainability: Ability to undergo repairs and modifications.

The main attribute considered in this thesis is safety (availability, reliability and maintainability is also considered to some extent). Sometimes, the term *safety-critical systems* is used in this thesis, meaning systems where a computer fault may impact human safety.

A system failure is said to have occurred when the delivered service deviates from fulfilling the specified system requirements, i.e., incorrect service is delivered. It is important to note that all system failures do not result in catastrophic consequences. Failures occur as a result of some incorrect internal state, i.e., there exists an error. The adjudged or hypothesized cause of an error is denoted as a fault.

The terminology of faults-errors-failures is bound to the abstraction level of observation. At that abstraction level, the fault is said to be active when it causes the component/system to assume an incorrect state (a fault that is not active is called dormant), i.e., the fault generates an error, see Figure 1.1. In turn, if the error propagates to the border of that abstraction level, a failure is said to have occurred. This failure will be considered a fault at the next higher abstraction level. This is described in more detail in Chapter 3. *Therefore, to have a safe system, either faults*

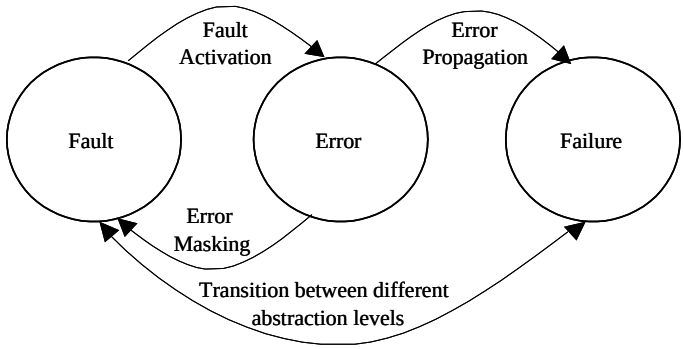


Figure 1.1: The fault-error-failure cycle.

should be prevented from occurring and/or the fault-error-failure cycle should be interrupted before a failure occurs at the system level.

The sources of faults can be classified into six major criteria:

- Phase of creation or occurrence: Does the fault occur during operation or was it introduced already during development?
- System boundaries: Is the fault source internal or external of the system?
- Domain: Is it a hardware or software fault that has occurred?
- Phenomenological cause: Is the fault caused by human or the environment?
- Intent: Was the fault introduced deliberately or not?
- Persistence: Was the fault of transient or permanent nature?

The faults themselves are classified in detail in [Avižienis *et al.*, 2001]. However, as this classification is more complex than needed for the context of this thesis, we use a simpler fault classification, namely:

- Developmental faults: Faults introduced during system development (specification, design or implementation, including manufacturing).
- Physical faults: Faults caused by the environment occurring during operation.
- Interaction faults: Faults caused by humans occurring during operation.

Faults are sometimes also grouped based on whether they are reproducible (termed as solid or hard) or not (termed as elusive or soft). Furthermore, the term intermittent faults is often used. In [Avižienis *et al.*, 2001] this definition is used to include all elusive developmental faults and transient physical faults as they manifest similarly. Errors produced by intermittent faults are usually termed soft errors. However, intermittent faults sometimes have different meanings (e.g., something in between transient and permanent faults) in literature.

In the context of the thesis work, we utilize an abbreviated classification with emphasis on the following fault types, namely:

- Transient faults: Faults generating errors only at one single point in time.
- Persistent faults: Faults generating errors at more than one point in time.

The advantages with these two definitions are that: all faults could be classified as either transient or persistent, transient faults will be closely connected to the bit-flip fault model, and the persistent fault class will include, for instance, permanent faults and elusive developmental faults. Sometimes, also the term permanent fault is used to stress that the fault can be removed only by removing the faulty component or repairing it.

Four techniques can be used for development of dependable (including safety) computing systems:

- Fault forecasting: How to estimate the present number, the future incidence, and the likely consequences of faults.
- Fault prevention: How to prevent the occurrence or introduction of faults.
- Fault removal: How to reduce the number or severity of faults.
- Fault tolerance: How to deliver correct service in the presence of faults.

In this section, safety and how safety can be enforced has been defined. Next, we will describe how the defined concepts in this section can be practically implemented in the system development process.

1.2 Development of Dependable Computer Systems

Many different system development processes have been proposed all with the aim of producing systems of consistent quality, reliably produce systems with complex behavioral requirements, predict when systems will be complete, predict how much

systems will cost to develop, identify milestones during development, enabling mid-course corrections when necessary, and enable efficient team collaboration for moderate and large-scale systems. However, this section does not aim to survey all different processes, but describe the general stages of development processes.

All development processes include the stages: analysis, design, implementation, and testing or use similar terms. Examples of such processes are the waterfall process, the v-process, the spiral model, and extreme programming [Pradhan, 1996], [Douglass, 2001], [Peters and Pedrycz, 1999], [Wells, 2003]. The tasks of the different steps are:

- Analysis: The functionality and requirements are specified from the problem to solve.
- Design: Solutions for solving the problem and meeting the requirements are proposed.
- Implementation: The designs are implemented.
- Testing: The functionality and requirements are validated (are we developing the right system?) and verified (are we developing the system correctly?).

As modern computer systems generally are very complex, these steps are generally iterated starting at high abstraction level with few details and then integrating more and more details until a final complete design is reached, i.e., a top-down approach. Thus, the first analysis step concerns which service the system should deliver and the first design step which basic components that are needed to deliver this service. The first implementation is thus generally a model of these components that are simulated in the testing step to verify that the functionality and requirements are likely to be met. The testing will then provide information for the analysis step for determining the functionality and requirements of each basic component.

Subsequently, the development cycle continues with more detailed models leading to prototypes and till a final complete design. However, low-level testing may sometimes discover faults in the high-level designs implying that the process needs to backtrack and repeat the design iterations. *The longer the process needs to be backtrack the more costly it becomes. Thus, one very important task for development processes is to avoid such re-iterations.*

A recent iteration and cost saving trend is to use standard components (both hardware and software components), so-called Commercial Off The Shelf (COTS) components. However, the drawbacks are that these components are generally not optimized for specific systems and the detailed design of these components are not

accessible for system developers. We will now briefly go through what these development steps involve considering the dependability aspects.

As previously described, a development cycle needs to start by analyzing the problem to solve and defining the requirements on the services, performance, dependability, etc for the design (solution). For dependability analysis this specifically implies progressing through the different attributes of Chapter 1.1 and determining the requirements. For safety, which is the attribute this thesis mainly focuses on, the first step could typically mean to determine the maximum accepted system failure rate at different working conditions, whether single point of failures should be allowed and which values specific quantities are allowed to obtain.

When the first requirements for a sufficient solution for the problem have been specified at the analysis step, different strategies (designs) for solving the problem can be proposed. It is also necessary to determine how the requirements can be tested for the different designs. For development of dependable systems this specifically means determining the techniques (listed in Chapter 1.1) to use to reach a sufficient design and to be able to test that it is sufficient.

The first design (solution) to start with is often a system that is believed to fulfill the service requirements, but for which it is not known if the dependability requirements are met. In order to determine if and how the dependability requirements can be met, dependability analysis methods are used in the testing step. Examples of such analysis methods are Failure Mode and Effect Analysis (FMEA), Functional Failure Analysis (FFA), Fault Tree Analysis (FTA), Markov Modeling, Petri Nets, Formal Methods (FM), etc (the interested reader is referred to [Johannessen, 2001], [Clarke and Wing, 1996], [Betous-Almedia and Kanoun, 2002]). Different methods test different properties and require different implementations.

The target of the first analysis is often to identify the components that are critical for the safety of the system. In later development steps, more quantitative analysis can be applied for computing the Mean Time To Failures (MTTF) that are achieved using different fault tolerance strategies, as based on component failure rates for instance collected from [US Department of Defense, 1991]. A comparison of different reliability prediction models is presented in [Jones and Hayes, 1999].

If the proposed design meets the safety requirements, the development process can be taken to a lower abstraction level. However, if the requirements are not met, the design must be modified such that faults are prevented from occurring or are tolerated (i.e., the fault-error-failure cycle interrupted before a system failure occurs) to such level that the requirements are met.

Fault prevention may for instance imply changing to more robust components, redesigning the components/system, and/or shielding. This is most efficient to apply

to improve the safety against faults that else would be very common and/or affect several separate parts of the system.

Fault tolerance generally implies error detection (identification of incorrect states) and recovery (transforming the system into an acceptable state where faults are not activated again).

It is generally not possible to efficiently (considering dependability, functionality, and cost) to prevent all faults from occurring, as the environment for safety-critical systems can be very harsh, all components are exposed to wear, and the complexity of components is increased. Thus, to meet the safety requirements fault-tolerance techniques must be used. As mentioned previously, the focus of this thesis is on exploring how efficient fault tolerant systems can be developed. Therefore, in the next section, development of fault tolerant systems will be described in more detail.

1.2.1 Development of Fault Tolerant Systems

After it has been determined that fault tolerance is necessary to meet the safety requirements on the system, the fault tolerance development process can be started. First, in order to develop fault tolerant components/systems, it is necessary to know which types of faults need to be tolerated, i.e., the analysis step.

Looking at the four means to develop dependable services (see, Chapter1.1), fault tolerance distinguishes itself by being the only "active" method, i.e., the only method that can be used to avoid failures during system operation. To paraphrase Murphy's law: "If something bad can happen it will, and it will happen during that time when it is most damaging", fault tolerance techniques should ultimately be applied to handle all unexpected faults. However, as fault-tolerance is expensive to implement, it is an effective strategy to methodically constrain fault occurrences using the processes of fault forecasting, fault prevention and fault removal. Thus, the refined list of faults that are necessary to apply fault tolerance techniques for are now discussed.

Developmental faults are introduced when the system is developed, i.e., before it is operating. Many people consider developmental faults, specifically software faults, to be the most important challenge for the dependability community. Therefore, some tolerance techniques for detecting such faults have been proposed, in general based on diversity, [Avižienis, 1985], [Ammann and Knight, 1988]. Regardless, their efficiency is debatable as the technique increases the complexity, and thus, potentially the number of faults. However, one cannot ignore that even if fault removal techniques are improved, they will probably never be perfect, especially consider-

ing the complexity growth of computer systems. Therefore, this thesis assumes that proper testing has been applied, meaning that developmental faults left after testing have a low activation frequency, as faults with high activation frequency should already have been detected. Thus, developmental faults following this assumption, i.e., with low activation frequency, are considered in this thesis.

Physical faults occur when the system is operating due to wear-out and/or environmental disturbances. Even if new materials with better characteristics will be introduced, and manufacturing, design, testing and shielding techniques improve, the physical fault occurrence rate is expected to increase [Constantinescu, 2002], [Avižienis, 2000], [Shivakumar *et al.*, 2002], [Hazucha, 2000]. Therefore, it will be necessary to apply tolerance techniques for such faults in order to get cost-efficient solutions. Consequently, physical faults are the major type of faults considered in this thesis.

Interaction faults are caused by that the user misunderstands how the system should be used, or deliberately attacks the system, i.e., faults occurring due to misuse of the system. These faults differ greatly from the other two classes as they are introduced by humans during system operation. Thus, in many cases, a different fault tolerance strategy (many faults are probably more cost-efficiently to avoid through fault prevention) is required. Therefore, to restrict the content, such faults are not treated in this thesis.

To summarize, *this thesis focuses on tolerating physical faults and developmental faults with low activation frequency*.

After specifying the desired class of faults to tolerate, the next step is to determine how these faults should be handled, i.e., the design step. As mentioned previously, fault tolerance generally implies error detection and recovery. To establish this, redundancy is required. There are three classical forms of redundancy in practice: spatial-, temporal-, and information- redundancy. Generally, spatial redundancy means physical replication of units; temporal redundancy implies re-execution of tasks, and information redundancy often entails coding type techniques to provide supplemental information onto the basic data content.

The choice of strategy to use is determined by the effect/impact[1] each fault has on the system, and the cost. First, in order for fault tolerance to work, it is required that not all replicas (redundant tasks) fail in the same time. Second, for systems which are manufactured in relatively small numbers as in the aerospace industry, the total cost is primarily determined by the fixed cost for developing the system. Thus, techniques based on error masking through spatial redundancy may be efficient as they often are relatively simple to accomplish. However, for systems that

[1]These two terms will be used interchangeable in the continuation of this thesis.

are manufactured in large numbers, e.g., the automotive industry, the total cost will mainly depend on the recurring cost (the manufacturing costs including the material cost etc), strategies with less spatial redundancy are preferred, even if such solutions may become more complex.

As safety-critical systems lacking a fail-safe state (a fail-safe state is a state where no catastrophes can occur) require that units are at least duplicated to be able to recover, it seems natural to provide fault tolerance by executing the tasks on both units and comparing the results, i.e., a duplex systems. However, duplex systems have the following problems:

- How should it be determined which unit that is erroneous?
- How should the comparison be performed? Should it be on one of the units (in that case, what happens if this is the erroneous unit) or by an external comparer (which must be assumed to be reliable)?
- The units must be synchronized, which reduces the performance.

Therefore, fault masking via voting across a number of replicas, so called N-Modular Redundancy (NMR), can be used. To tolerate m faults, $2*m+1$ replicas are required, [Pradhan, 1996], [Yeh, 1996]. In this way the system can recover without specific error detection. However as this approach conceals fault occurrences, the number of faults may eventually exceed m. Therefore, practical implementations often log the errors (i.e., error detection). The main advantage with fault masking is that it is simple to implement. However, it requires a large number of replicas which makes it costly and/or time consuming.

Error detection and recovery techniques are potentially less costly and time consuming but can be complex to implement. The development of systems based on error detection and recovery is now described in more detail.

Error Detection and Recovery

As it is not practical to recover a component/system before an error has been detected, error detection often precludes recovery[2]. However, when developing the system, it is more practical to design the fault recovery policy first as this sets the requirements for the error detection, i.e., determines which errors that need to be detected and separated. Therefore, the description of the different steps starts with the recovery.

[2]It should be noted that some techniques perform immediate recovery, as for instance Error Correcting Coding (ECC) or N-Modular Redundancy.

Recovery is aimed at transforming the system into a correct state where faults are not activated again. This is achieved through error handling (eliminating errors from the system state) and fault handling (locating faults and preventing them from being activated again). Starting backwards, fault handling involves four steps:

- Fault diagnosis: Identification and recording the cause(s) of error(s), in terms of both location and type.
- Fault isolation: Performing physical or logical exclusion of the faulty component(s) from further participation in service delivery.
- System reconfiguration: Switching in spare components or reassigning tasks among non-failed components.
- System reinitialization: Checking, updating and recording the new configuration and updating system tables and records.

It should be noted that sometimes some of these steps are left out. For instance, if the fault is transient, there are components performing the same tasks (so called active replicas), or the system is allowed to degrade, i.e., reduce service in a controlled manner (e.g., set the system in a fail-safe state). A classical example is the Anti-Lock Braking System (ABS) for automobiles, which when functioning correctly, reduces the braking distance for vehicles, but when turned off the normal braking performance is obtained. Thus, the off position is a fail-safe state for the ABS.

Fault diagnosis can be performed based on information from how the error was detected, a-priori information on how faults are activated and resulting errors propagated, and from specific fault diagnosis tests. It is of greatest importance that the fault diagnosis does not incorrectly point out a fault that should be handled in a different way than the fault that really occurred, since this not only leaves the fault untreated, but also may reduce the resources of the system.

Error handling can be achieved through:

- Rollback, where the state transformation consists of returning the system back to a saved state that existed prior to error detection; that saved state is called a checkpoint.
- Compensation, where the erroneous state contains enough redundancy to enable error elimination.
- Rollforward, where the state without detected error is a new state.

The decision on how to handle errors is determined by the cost for the specific system, information of the specific system and the chosen fault treatment strategy.

When recovery strategies for all fault types are selected, the requirements on the error detection is known. Errors are generally detected by checking some property of the system. This can be achieved for instance through determining the property twice, either from redundant components or from the same component, but separated in time, or comparing it to in advance information about the property.

Generally, a number of quantities are used to determine the detection efficiency of the error detection, namely:

- Error Detection Coverage: The percentage of the errors occurring that are detected by the technique.
- Error Detection Latency: The time from that the error occur until it is detected by the technique.
- False Alarms: The number of times the technique is triggered without an error has occurred.

In many cases, the most efficient (detection efficiency, diagnosis, cost and overhead) solution for error detection in a system is based on several complementing error detection techniques, rather than one, [Steininger and Scherrer, 2002]. The choice of which error detection technique to use is not only dependent on the cost for using them and which error detection efficiency that is required, but also on what support they give for fault and error diagnosis.

In this subsection we have described the necessary steps to be able to tolerate faults. Thus, the fault tolerance coverage, i.e., the probability that a fault is tolerated is the product of the probabilities that the resulting errors from the fault are detected and handled correctly, the fault diagnosed and isolated correctly, and the system reconfigurated and reinitialized correctly.

In order to determine the fault tolerance requirements, the analysis methods mentioned in Chapter 1.2 can be used. However, for testing of fault tolerance designs, fault injection is the most common used technique.

Fault Injection

Fault injection means evoking faults or errors in a system in order to observe their effects. This can be used for understanding which effects faults have on the system, but also to test and evaluate the efficiencies of fault tolerance techniques. Thus, fault injection can be used for fault forecasting, fault prevention and fault removal.

If the injected fault is activated (i.e., generates an error) during the observation time, the fault is said to be effective, else it is determined to be latent (i.e., it may have been activated later if the observation time had been extended), or ineffective (i.e., if it can be established that errors never will be generated). Faults can be injected in many different ways and at many different abstraction levels and locations. Some of these are, [Iyer and Tang, 1996], [Folkesson, 1999], [Karlsson *et al.*, 1995]:

- Simulation-based fault injection implies that faults are injected into a model of the system that is simulated.
- Software-Implemented Fault Injection (SWIFI) injects faults through software into registers and the program memory of the system.
- Scan-based fault injection (SCIFI) means that faults are injected through use of the scan chains of the device.
- Physical fault injection implies that faults are physically injected into hardware, i.e., at least a prototype must have been developed.

Simulation-based fault injection has the advantage that it can be applied early in the development process, i.e., as soon as a model of the system exists. The controllability and observability of the experiments are also usually very high. Furthermore, since simulations are used, there is no risk of physical damage due to the fault injection. The drawbacks are that as for all simulations, the experiments can be very time consuming for detailed models. Therefore, the models need to be abstracted, i.e., less detailed models are used, which on the other hand implies a risk of giving less accurate results. Thus, only a subset of the possible faults can be evaluated and only on models, not on physical prototypes or real systems.

Software-implemented fault injection has the advantage that faults can be injected into physical prototypes and systems with only a small risk of destroying them. It has the potential of modeling all types of software faults and many hardware faults as well, [Christmansson *et al.*, 1998]. However, it is not clear how faults should be injected as it is hard to determine which software faults that are not detected during testing (which are the faults that are of main interest to investigate), and how hardware faults should be modeled.

Scan-based fault injection has the advantage that faults can be injected into physical prototypes and systems with only a small risk of destroying them. Moreover, it can use the traditional hardware fault models developed for simulation-based fault injection. The main drawback is that the scan-chains generally only are accessible for the device developers, i.e., not for system developers. Another potential

problem is that scan-chains often only access limited parts of the device, i.e., only a subset of the possible faults can be investigated.

Physical fault injection has the major advantage that the real physical prototypes or system is evaluated with real faults. The drawbacks are that the observability and controllability often are low which can make it hard to explain the causes of the measured results. Furthermore, there is a risk of destroying the devices, the experiments can be time-consuming since it can be hard to evoke faults, and even if real faults are injected, generally not all of types of faults can be injected, i.e., the results may be biased towards a certain fault type. Examples of physical fault injection techniques are:

- Pin-level fault injection: the pins of an integrated circuit are forced to specific levels.
- Radiation induced fault injection: the system is exposed of radiation.
- Power supply disturbance fault injection: the power supply to the system is disturbed.
- Electro-magnetic interference fault injection: the system is exposed to electromagnetic waves.
- Laser fault injection: a laser beam is pointed to parts of the system that are sensitive to Single Event Upsets (SEU).

In this section we have described the process of developing safe, specifically fault-tolerant systems. In the next section we will describe control systems as such systems are one of the most common safety-critical systems and used in this thesis when discussing and exemplifying different methods and techniques.

1.3 Control Systems

Conceptually, a control system is set to control a certain physical process (see Figure 1.2). Examples of physical processes that generally are controlled are the temperature in the compartment of a car, the braking of a truck, the travel direction of a space shuttle etc. The control is performed by first determine the current state of the process by measuring certain quantities, $y(k)$, directly using sensors (**S** in Figure 1.2), or by measuring related properties and estimating the quantities from these. Then, the controller compares the current state with the desired state, i.e., the reference signal, $u_c(k)$, that is provided by the user (which may be a human user or

an external computer system) and that may be fixed or variable, via some interface (**I** in Figure 1.2). The difference between the current and desired state is used by the controller to compute how the process should be controlled, i.e., the control signal, $u(k)$, is computed, such that the process state is as close to the desired state as possible. The actual physical control of the process is carried out by a set of actuators (**A** in Figure 1.2).

In this thesis we assume that controllers are implemented on microprocessors, and that they execute in discrete steps, i.e., a controller reads reference values and sensor values and calculates control signals periodically. This is most, at least for safety-critical systems, but there also exist systems where the controller computation is event-triggered, [Årzén *et al.*, 1990]. In Figure 1.2, and subsequently, k indicates the k^{th} step in time.

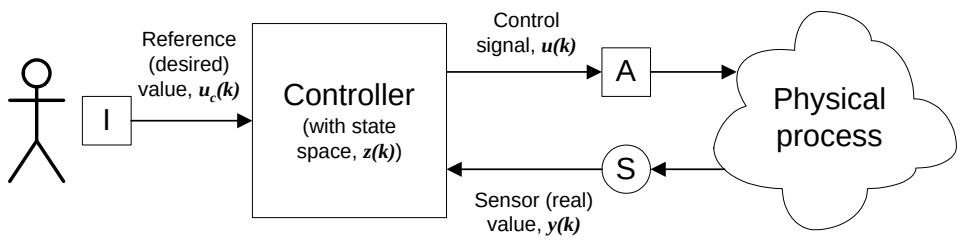


Figure 1.2: A general figure of a controller and the corresponding physical process.

Using pseudo-code, the controller could be implemented as:

```
repeat:
   uc := read_from_interface();
   y := read_from_sensors();
   u := compute_control_signal(z,uc,y);
   write_to_actuator(u);
   z := update_controller_state(z,uc,y);
   wait_for_next_sample;
```

It is easy to realize that control systems generally are real-time systems as most physical processes have time-requirements (consider for instance a brake-by wire system in an automobile).

1.3.1 Distributed Dependable Control Systems

During the previous decade, the performance of microprocessors increased drastically, whereas the size and cost decreased significantly. At the same time, the

requirements of low recurring costs, low weight, high dependability, high functionality and high controllability have increased. These aspects brought the trend to distribute the functionality to several computers instead of having one central computer. The advantages are less cabling (less weight), fewer single points of failure (higher dependability), and improved functionality and controllability. However, distributing systems means that the different parts need to communicate and often also to be synchronized, which increases the complexity. Details on communication and synchronization solutions can be found in [Lönn, 1999], [Claesson, 2002].

The distribution of control systems is often determined of where the physical properties can be measured and controlled, i.e., the sensors and actuators are located dependent on the architecture of the physical system. In many cases are simple computers integrated with the sensors/actuators and additional computers placed according to which computation efforts that are required at different part of the system. For the future, it can be expected that the systems will become even more distributed, such that sensors and actuators are directly connected to the communication network, rather than to computer nodes. This implies that it will not matter which computer connected to the network that computes which control algorithm, and thus, the computers can be seen as a multiprocessor system. However, through out this thesis we assume that sensors and actuators are mainly connected to specific computer nodes and that these nodes compute the data from and to these components. Later, in the future work section in Chapter 6, the implications of this possible development will be discussed.

The Computer Node

A generic computer node is shown in Figure 1.3. Such a node conceptually contains two interfaces, a communication interface and an application interface. The communication interface handles the information exchange with other nodes (e.g., sensor and/or control signal values). The communication is generally performed according to a protocol, stating the packaging and encoding of data. Received data is transferred through the memory to the application interface, where it is utilized together with data received from internal data sources (e.g., local sensors) in the control algorithms. The control signals are then transferred to their corresponding actuators.

The actual implementation differs among computer nodes. In some nodes, communication and calculation of control signals may be managed by the same main processor, whereas in others, the main processor only handles calculations and a specific controller manages network communication. For such nodes, data can be transferred between the controller and main processor through an internal bus or through a dual-port memory. Other variations are the types of memory used (EEP-

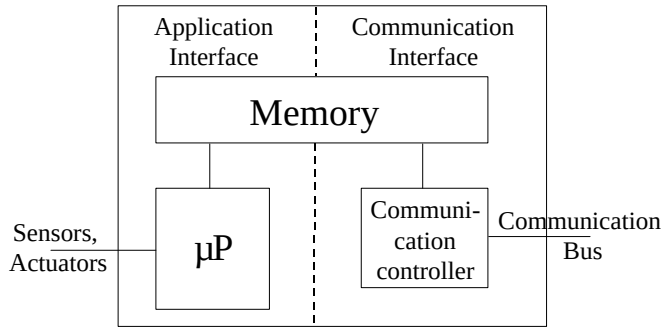


Figure 1.3: A general architecture of a computer node.

ROM, FLASH, RAM, etc.).

Another issue is whether to use an operating system or not. As more and more software is used and which have been developed by different developers, operating systems will become necessary. In this thesis, operating systems are not specifically discussed, but are seen as a software component, and thus, faults in the operating system are indirectly included when developmental faults are discussed.

The Application Microprocessor of a Computer Node This thesis focuses on how to develop fault tolerant microprocessors for use in safety-critical systems. Therefore, the requirements on the functionality and dependability of the microprocessor in the application interface are now discussed in more detail. As described in the previous section, microprocessors are generally used in the application interface even if they in some cases also handle the communication. In this thesis we focus on the role of the microprocessor in the application interface. The requirements of the microprocessor for such use can be divided into delivered service and dependability:

Service

As we are dealing with control systems, the most important services (tasks to execute) for the microprocessor at the application interface are:

- Reading input values from directly connected sensors and other computer nodes (through the memory on the computer node).
- Computing the control signals and updating the state of the controller based on read input values.
- Distributing the control signals to the actuators, either directly or through other computer nodes.

It is important that these functions are performed at specific time points in time as the control algorithm is designed in most cases are designed for periodic sampling and any difference in sampling time, i.e., jitter (see for instance [Lönn, 1999]), will reduce the control performance.

However, the microprocessor may also perform other tasks. For instance, handling interfaces between the system and the user that are not directly related to the control system, and storing data on how the system is used in order to be able to adapt it for the specific user and how to improve future developed systems. As the microprocessors constitute the major intelligence of the system, it is also often responsible for diagnosing the sensors and actuators of the system, and store information that can be used for repairs. Determining whether a sensor or actuator is working properly is generally achieved through functional redundancy (an error detection technique based on information redundancy) and/or test sequences, see Appendix A for more details.

Considering these functions. it is easy to understand that a microprocessor fault that is not treated correctly will in many case cause severe damage[3].

Dependability

As the focus of this thesis is on safety-critical systems, the requirements on dependability are high. For many such distributed systems, the computer nodes are required to be *fail-silent*, i.e., that the nodes fail "cleanly" by just stopping to send messages, [Powell *et al.*, 1988]. The main advantages with this requirement are that:

- It is comparably expensive to recover from errors that have spread to several nodes.
- Communication between nodes is expensive which makes error detection between nodes unattractive.
- Error propagation to actuators can be limited, as today, actuators often are connected to computer nodes rather than to the communication network.

Thus, today (this may change in the future) fail-silent computer nodes simplifies the design and reduces the cost.

As the microprocessor generally is the most complex component of the computer node, and thus, the most failure prone, it is of utmost importance that faults in this

[3]In this thesis we consider diagnosis of the network, the status of other computer nodes and system reconfiguration to be handled by the communication controller. Examples on references on how to protect the communication controller are [Temple, 1998], [Steininger and Temple, 1999], [Steininger and Vilanek, 2002].

component can be detected and handled to meet the requirements on fail-silence. How to achieve this is the main focus for this thesis.

1.4 Restrictions and Targeted Problems

As was discussed in the previous section, microprocessors have a key role in safety-critical control systems. Thus, it is very important that the microprocessors are working correctly. Therefore, the main research question for this thesis is:

Q1: How should a system developer in the future design a computer node with a microprocessor so that safety, cost and functionality requirements are met?

It is assumed in the thesis that fault tolerance techniques will be necessary, i.e., that fault prevention in the future will not be enough to reach safety requirements. Furthermore, the discussed fault tolerance techniques are mainly focused on physical faults (even developmental faults with low activation probabilities are targeted). In this thesis we have focused on systems that are produced in large numbers (e.g, systems in automobiles) which implies that the cost requirements mainly consider the recurring costs rather then the fixed costs.

For the future, there seem to be mainly three different directions for a system developer to design a computer node with microprocessors. First, use specific designed fault tolerant on-chip processors, second, use commercial processors, and third, use commercial processors but adding software and/or hardware to improve the fault tolerance[4].

The number of microprocessors that have been designed for tolerating a wide range of faults is limited. The main reason for this is that the market for fault tolerant microprocessors has been considered too small (it is not only the microprocessor itself that need to be developed, but also compilers and other development tools). In spite of this, there exist some microprocessors that are designed to be fault tolerant. Examples of such processors are: *G5* [Check and Slegel, 1999], *Thor* [Saab Ericsson Space, 1999], and *LEON* [Leon, 2003], [Gaisler, 2002]. The *G5* processor is based on massive hardware redundancy on-chip and has been designed mainly for banking systems, whereas *Thor* and *LEON* have been designed for space-flight applications. *Thor* and *LEON* are designed to tolerate faults occurring in harsh environments, and offers good observability and controllability for system developers. However, their performance is not comparable with modern commercial processors.

[4]It is generally not possible for system developers to redesign commercial processors to improve the on-chip fault-tolerance as low-level details are concealed by the microprocessor developers.

Commercial processors include fault tolerant techniques. However, it is important to note that these techniques are not applied for the sole purpose of increasing the dependability, but as the fault rate is so high that performance would be reduced if the techniques were not applied. Today, most internal buses, registers and memories are protected with Error Detection and Correction Codes (EDCC). For the future, as the scaling of the device geometries and the increased complexity are likely to increase the fault occurrence rate, [Hazucha, 2000], [Shivakumar *et al.*, 2002], fault tolerance techniques can be expected to occur more and more frequently on-chip of commercial processors. Many of the new proposed error detection techniques are based on implementing time redundancy on chip, e.g., executing instructions several times and compare the results (examples of such are [Rotenberg, 1999], [Rashid *et al.*, 2000], [Mendelson and Suri, 2000], and [Lai *et al.*, 2002]). Such techniques can detect almost all types of errors caused by transient faults, and some even errors caused by persistent faults. However, how to differentiate between faults of different duration and how to recover from persistent faults are sparsely discussed. The cost for these techniques is the additional execution time, which cannot totally be avoided even if these techniques try to make use of resources that are idling. Thus, there is a performance trade-off of adding error detection techniques. On one hand, faults that are not handled will reduce performance, but on the other hand, error detection techniques may constantly decrease performance.

If the yield (error detection and correcting codes are often used to compensate for memory elements with developmental faults) or performance of commercial processors is not increased by tolerating a certain fault, such techniques will generally not be implemented on chip. However, for safety-critical systems, most faults need to be tolerated. Therefore, specific techniques for tolerating faults can be added, implemented in software or as off-chip hardware. Other advantages with such techniques are that they are controllable for system developers and support fault diagnosis, which may not be the case for on-chip techniques. Even if such solutions generally will not be as efficient as designed on-chip, they can still be competitive as they be executed on commercial processors optimized for performance.

This thesis is focused on investigating the third alternative, designing fault tolerant microprocessor based on adding software-implemented techniques to commercial processors. The motivation for this choice is that commercial processors can today not be used without modifications and fault-tolerant microprocessors are expensive and/or have less performance. By adding fault tolerance techniques to commercial processors, the hope is that it should be possible to balance the fault tolerance coverage, cost and performance better. Furthermore, such solutions should be portable as it is based on software, and thus, should be possible to add to any

microprocessor.

However, even if this decision was taken, it is important to emphasize that *the aim of the thesis is NOT to say that systems in the future will be built of commercial microprocessors and the safety requirements reached through software-implemented software techniques.* It may very well be the case that such systems will be based on specific designed microprocessors (due to the market for safety has increased) or simple commercial microprocessors (due to that extensive fault tolerance techniques become standard) or adding hardware-implemented fault tolerance techniques (due to that hardware becomes cost-efficient). However, even if any of these scenarios will occur, we still believe that this thesis will provide important insights, which at least can be used for comparison.

1.4.1 Adding Software-Implemented Fault-Tolerance Techniques

By focusing the work to investigate whether commercial microprocessors can be used in safety-critical applications by adding software-implemented techniques, the research question *Q1* transformed into the sub question:

Q1.1: How efficient (safety, cost, functionality) can a fault tolerant processor based on commercial processors with added software-implemented fault tolerance techniques be?

As we focus on systems that are manufactured in large numbers, it is important to restrict the recurring costs, and thus, fault masking should not be used at the node-level as it requires triplication of the microprocessors. This implies that efficient error detection with recovery should be applied. With efficient error detection, we mean:

- Maximum detection coverage for errors that result in failures, i.e., detect as many errors as possible that will result in catastrophic consequences if not handled.
- Minimum of false alarms, i.e., should not be triggered if no error exist.
- Minimum of unnecessary alarms, i.e., should not be triggered if the error has insignificant impact on the system.
- Minimum detection latency, i.e., should be triggered as fast as possible after an error has occurred.
- Maximum fault diagnosis, i.e., should provide as much information as possible on which error that triggered the detection.

- Maximum portability, i.e., should be portable to any system.
- Maximum scalability, i.e., should scale if the system is modified or redesigned.
- Minimum overhead, i.e., should have as low overhead (cost (mainly recurring costs), performance, memory, power) as possible.

Efficient recovery means:

- All detected errors should be eliminated.
- All faults should be correctly diagnosed.
- All faults requiring to be actively handled (in contrast to transient faults), should be correctly isolated, reconfigured or degraded, and reinitialized.
- Should have as low overhead (cost (mainly recurring costs), performance, memory, power) as possible.

As was described previously in Chapter1.2, a development process starts with the analysis step. Therefore, in order to identify the major problems of developing fault tolerant processors based on applying software-implemented fault tolerance techniques to commercial microprocessors, a number of studies were performed:

- Investigation of how the occurrence rate of different fault mechanisms are affected by the device scaling, Chapter 2.
- Identification of how errors manifest them selves in a microprocessor at different abstraction levels, Chapter 3.
- Survey of different types of error detection techniques, Appendix A.

The major conclusions from these studies are that storing and transferring parts are likely to be well protected in commercial processors whereas the transformation of data is sensitive. Moreover, in the future, the duration of faults can be expected to be more varied (today, most research is focused on transient faults).

Based on these results, a design was proposed in Chapter 3, based on the use of complementary techniques for detection of different types of errors. In order to determine the efficiency of the proposed design, evaluations were necessary. As low-level details seldom are accessible for system developers, the next research question became:

Q1.2: How can commercial processors with added fault-tolerant techniques be designed and evaluated with only high-level information accessible?

This are the focus of Chapter4-5 where analysis (Chapter4) and fault injection (Chapter5) methods for evaluating the effect of data errors caused by faults with varied duration, and the efficiency of applied error detection techniques are presented.

1.5 Major Contributions

One contribution of this thesis is the identification of the problems with developing fault tolerant microprocessors, based on the studies of modern and future data architectures, occurrence rates of future architectures, and existing error detection techniques, Chapter2-3 and Appendix A. From these studies, a design based on complementary techniques for detection of different errors caused by faults with varied duration were proposed. Even if the efficiency of this design has not been evaluated, no major hindrances for building fault tolerant processors by adding software-implemented techniques to commercial processors were detected.

The main contribution of this thesis is the development of a novel approach for evaluating the effects of data errors caused by faults of varied duration and the efficiency of applied error detection techniques. Most previous work has tried to model low-level faults and then observed which errors and effects they result in, i.e., bottom-up approaches. However, as low-level details of commercial microprocessors seldom are accessible for system developers or researchers in academia, a different, top-down strategy, has been progressed.

First, the characteristics of data errors (property, number, magnitude, state and repetition frequency) that determines their effects and the possibility to detect them, are identified, Chapter 3. Then, in Chapter 4, errors with different characteristics are divided into different sets. Each set is modeled as a specific disturbance. Then, analysis methods from control theory are applied to determine the impact of the different errors.

The analysis can also be used for estimating the efficiency of executable assertions. The main advantage with the analysis is that it provides information early in the design process. However, with the proposed analysis methods, not all combinations of the error characteristics can be analyzed.

The enabling of using analysis for estimating the effect of data errors is a major contribution as most previous work has been based on fault injection experiments which are time consuming, and as it requires more detailed implementations, must be applied later on in the design process.

To investigate those errors that are hard to analyze, simulation-based fault injection, Chapter 5, is proposed. One problem with simulation-based fault injection is that the simulations must be performed at high abstraction level (as low level

models seldom are accessible), which means that there is a risk that the accuracy of such simulations can be low. Therefore, a method based on VHDL-simulations at Register Transfer Level (RTL) is developed. Here the errors are injected through saboteurs, a specific VHDL-component added only for the purpose of stressing the system. The saboteurs are developed so that the error characteristics can be varied.

The proposed high-level simulation-based fault injection methods are major contributions as they enable accurate estimations of the effects of faults with varied durations. Most previous high-level approaches have only considered transient faults and those that have considered faults with longer durations, have not been able to model there effects as accurately. Another advantage with this approach is that it also can be used for evaluating the efficiency of applied error detection techniques.

One problem with simulations is that it is generally not practically possible to investigate all combinations of the error characteristics. This problem is suggested to be solved in this thesis by applying iterative fault collapsing. First errors with certain characteristics are simulated. If the effect of these errors are negligible, errors with characteristics that generally implies less effects (e.g., lower magnitudes and frequencies) do not need to be simulated. Next the error characteristics are tuned so that the errors will have greater impact (e.g., the magnitude and frequency are increased etc.). This continues until all errors can be classified into desired classes. In this way the number of simulations is reduced. Similarly, for investigating which errors that are detected by a certain technique, the characteristics are varied depending on whether the errors are detected or not.

1.6 Thesis Organization

The organization of this thesis follows the steps of system development processes. In Chapter 2, the problem is analyzed and in Chapter 3 a design is suggested. Then in Chapter 4 and Chapter 5 testing at different abstraction levels (system level and RTL) is discussed.

More precisely, this thesis is organized as follows: A short overview of the architecture of modern and future microprocessors are given in Chapter 2. Also, in this chapter, the effect of device scaling on the occurrence rate of different fault mechanisms is briefly discussed. The effect of faults and how the generated errors propagate in microprocessors is discussed in Chapter 3. The characteristics which determines the effect of data errors are identified. Furthermore, a design of complementary software-implemented error detection techniques is proposed and the problems of evaluating it discussed. Subsequently, in Chapter 4, methods for analyzing the impact of data errors on control systems are presented. As a complement for er-

rors/systems that are hard to analyze, two simulation-based fault injection methods are presented in Chapter 5. In Chapter 6, the main conclusions are summarized and how the results of this thesis may be affected by future development is discussed.

The appendices of this thesis have the following content:

- Appendix A, a survey of error detection techniques.
- Appendix B, analysis on the probability of detecting errors caused by persistent faults with double execution.
- Appendix C, ideas on design of software-implemented self-tests for detection of persistent faults.
- Appendix D, details on the control theory used in Chapter 4.
- Appendix E, the VHDL-code for a saboteur.

CHAPTER 2

Modern Microprocessors and Fault Mechanisms

As stated in the introduction, this thesis focuses on the design of fault tolerant microprocessors using commercial processors with software-implemented fault tolerance techniques. Fault tolerance can be defined as: "How to deliver correct service in the presence of faults", [Avižienis *et al.*, 2001]. Thus, in order to design a fault tolerant component, the correct service of that component must be known. In this chapter, the function of a microprocessor and its basic functionality are described. After that, architectures of modern processors are presented.

One of the reasons for the fast development of processors with increasing performance is the successful scaling of transistors, i.e. reducing their size. However, the scaling may also make the transistors more sensitive to discrepancies and disturbances. Therefore, also the microprocessor implementation and scaling trends, and their implications on the occurrence rates of different faults, are briefly described and discussed.

A related problem is that as the number of transistors increases on chip, more complex microprocessors are developed which increases the fault probability. Furthermore, the time required for testing the circuits increases exponentially with the complexity, which makes exhaustive testing practically impossible. Thus, there is a risk that the number of developmental faults, not detected during testing, will increase. This is discussed briefly in this chapter.

The main result from these surveys is that we expect the fault occurrence rate of commercial processors to increase in the future, not only for transient, but also

persistent faults, due to scaling and increased complexity (however, few field data exists to support this claim). Thus, for commercial microprocessors used in safety-critical systems, techniques should be applied in order to tolerate transient faults as well as persistent faults.

2.1 The Functionality of a Microprocessor

In Chapter 1.3.1, the service requirements of microprocessors used in the application interface of a computer node, were described. However, the functionality of a microprocessor itself was not discussed, but will be discussed here.

In Cambridge dictionaries online, [Cambridge, 2003], a microprocessor is defined as "a part of a computer that controls its main operations" and where computer is defined as "an electronic machine which is used for storing, organizing and finding words, numbers and pictures, for doing calculations and for controlling other machines." Looking in the Computer User magazine online dictionary [Computer User, 2003], a microprocessor is defined as: "A computer with its entire CPU on one integrated circuit," where CPU is defined as "Central Processing Unit. The CPU controls the operation of a computer. Units within the CPU perform arithmetic and logical operations and decode and execute instructions. In microcomputers, the entire CPU is on a single chip." Similar definitions are given in other dictionaries.

Thus, microprocessors can organize data and perform simple calculations (logic operations, addition, subtraction, multiplication and division). Thus, comparing this to the wide range of actions a human can perform, microprocessors would be inferior. However, when putting the simple functions (instructions) of a microprocessor together in sequences (programs), very advanced computations can be performed, and since these sequences can be run at a very high speed, the microprocessor can, for some tasks, be a very efficient tool. Therefore, historically, performance has also always been the main driver of the microprocessor development.

In the next section, the architectures of modern microprocessors for increasing performance, is described. Following this, different performance optimizations used in modern processors and future trends, are discussed.

2.2 Architecture of Modern Processors

As mentioned previously, the main functionality of a microprocessor is to organize data and compute simple arithmetic and logic operations. The first microprocessors

generally executed one instruction at a time, had a large number of different, rather advanced, instructions and only a few registers, i.e. so called Complex Instruction Set Computers (CISC). This made assembler programs short and easy to design.

However, in the early 1980s, [Hennessy and Patterson, 1996], it was realized that the most commonly used instructions were the simple load, store and branch instructions. Thus, microprocessors that would be optimized for simple instructions would show better total performance compared to processors with a lot of different complex instructions. Therefore, the trend changed towards microprocessors that only could perform the simplest operations, but on the other hand, could perform these very fast, i.e., so called Reduced Instruction Set Computers (RISC). This made assembler programming more complex, but high-level languages and compilers were improved to compensate for this.

The main idea for increasing the execution performance of simple instructions, was to introduce pipelining. Pipelining exploits that the execution of different instructions still have many similarities. Thus, the execution of the instructions can be divided into stages which are common to all or several of the instructions. The advantage with pipelining is that several instructions can be executed simultaneously in the same way as the production line in the automotive industry, i.e., some parts are assembled at one station and then forwarded to the next station where new parts are added and so on until the end of the line where the automobile is complete. For a microprocessor this means that when a program is started, the first instruction is fetched to the first pipeline stage. When this stage is finished, it continues to the second stage and the second instruction is fetched into the first pipeline stage. In this way, ideally, all resources of the microprocessor can be utilized simultaneously, and an instruction will be completed at every pipeline stage shift. Thus, this a very efficient way of executing instructions.

A common way to divide the functionality of a microprocessor is to use the following stages, [Hennessy and Patterson, 1996]:

1. **Instruction Fetch**
2. **Instruction Decode/Register Fetch**
3. **Execute/Address Calculation**
4. **Memory Access**
5. **Write Back**

In the **instruction fetch** stage, the next instruction of the program to execute is read (fetched) from a memory (storage). The microprocessor keeps track of the next

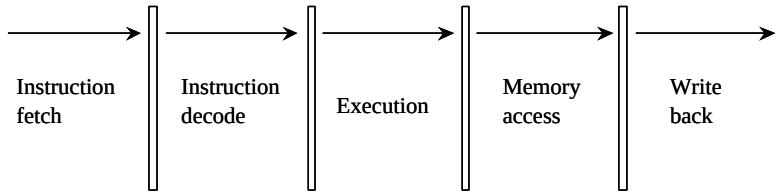


Figure 2.1: The stages of a pipelined processor.

instruction to execute by storing its memory address in an internal register, the program counter register. The instruction consists of the operation code (a recognizing pattern) together with required data or the necessary information (e.g., the memory address) to obtain it.

At the **instruction decode/register fetch** stage, the operation code is decoded in order to determine and generate the required control signals for the following pipeline stages so that the desired operation is performed.

The **execute/address calculation** is the stage where the actual computation takes place. This means that in this stage, different operations are performed for different functions. In most processors addition, subtraction, logic functions, multiplication, division and shifting can be performed. For instructions that read (load) data from, or write (store) data to the memory, the memory address where the data should be loaded from or stored in, is computed.

The **memory access** stage is only used for the load and store instructions. At this stage, data is loaded from (stored at) the computed memory location.

At the **write back** stage, the result from the operation is stored into the desired register. Store instructions do generally not use this stage.

It should be noted that modern processors often divides the functionality in much finer stages (some state of the art processors use more than 20 pipeline stages, [Intel, 2003(B)]).

In order to synchronize the instruction shifting between the stages, a clock signal is used, i.e., the execution at each stage is only allowed for a maximum time (the inverse of the clock frequency). After that amount of time, the data is transfered to the next execution stage. Therefore, the clock cycle time must be set according to the maximum time for the most time-consuming stage, and thus, it is important to balance the execution time so that it is similar for all stages.

As was previously described, pipelining is very efficient as, ideally, all resources of the microprocessor can be utilized simultaneously, and at least one instruction will be completed at every pipeline stage shift. However, the pipeline execution can be

slowed down due to dependencies between the instructions. There are three major dependencies: **structural**, **control**, and **data hazards**.

Structural hazards can for instance occur in the execution stage where some operations can take several clock cycles to perform, e.g., multiplication. This means that the execution of following instructions must be halted which decreases performance. Therefore, microprocessors generally have parallelized execution stages, which makes it possible for several instructions to execute simultaneously as long as they do not use the same functional unit (e.g., adder, multiplier, etc.). Through this design, and by allowing instructions to be executed in another order than at which they were programmed (so called out-of-order execution with in-order-commit, i.e., the instructions can be executed out of order, but the results from the executions are written to the memory in order), the pipeline flow is not restricted by time-consuming instructions. Many microprocessors also have replicated functional units so that instructions do not need to be halted even when the same type of resource is needed for two consecutive instructions.

Control hazards occur at conditional branches in the program. As it is not known which branch to follow until after the condition has been evaluated, the pipeline need to be halted until it is determined which instruction to fetch next. However, modern processors make advanced predictions on which branch will be taken based on the specific type of branch, history, etc. If it turns out that it was the correct branch was taken (predicted), the execution just continues. Otherwise the instructions that have started to execute from the incorrect branch must be thrown away and not considered, i.e., the pipeline is flushed.

Another way to avoid the problems with control hazards is to fetch several instructions instead of just one, and use out-of-order execution. In this way, some control branches can be avoided by executing only instructions that should be executed for all branches while the condition has not been evaluated. Similarly, compilers can analyze programs and change the order of instructions so that control hazards are avoided without changing the functionality of the program.

Data hazards occur when an instruction requires data that is computed by the previous instructions. As this data is not yet computed and stored when the instruction is fetched, the pipeline may need to be halted. In order to reduce the number of cases where the processor needs to be halted, data is often fed back to previous pipeline stages (i.e., so-called data forwarding) so that the closest following instructions do not need to wait for it to be written into the register bank. There exist some techniques for predicting the data that is required, but this a much harder problem than to predict the control flow, hence, the performance gain is generally less. Out-of-order execution can resolve some of the hazards by simply executing independent

instructions in between.

Another major bottleneck for the performance of modern microprocessors is memory accesses, i.e., loading and storing data from/to memory. In order to enhance the performance, several memory layers are generally used. At the lowest layer, the main memory is located[1]. This memory can hold a large amount of data, but it can be very time consuming to access. On top of the hierarchy, is generally one or several cache-memory layers with shorter access times, i.e., they are faster, but have limited storing capacity. The idea is to predict which data set the program will require in the nearest future and transport that from the main memory through the cache-layers to the highest layer (the cache with the fastest access-time) before it is required. If this is successful, the access time seen by the microprocessor (which is the same as the performance penalty) can be reduced significantly. However, the problem is to correctly predict which data will be required, and how to guarantee that all the memory layers will have the same data content after writing, without slowing down the instruction (program) execution.

Looking at modern architectures of today, they are characterized by the rapid device scaling (e.g., the reduction of the transistor size) of previous years (which will be discussed in more detail later in Chapter 2.4). This has resulted in chips containing large numbers of transistors. Therefore, it is now space available to replicate units on chip. There are some different concepts for how to most efficiently utilize replicate units. In **Chip Multi Processors (CMP)**, the whole microprocessor is replicated on chip, i.e., there are several microprocessors on the same chip which the workload can be distributed between, [Hammond *et al.*, 1997], [Diefendorff 1999], [Belcastro, 1998], [Bossen *et al.*, 2002], [Barroso *et al.*, 2000].

Another approach is **Simultaneous MultiThreading (SMT)** architectures, which have several pipelines on chip, but only one fetch and decode unit, [Lo *et al.*, 1997], [Theobald *et al.*, 1999], [Chao, 2002]. The advantages with this structure are that it is still possible to run a number of tasks (threads) simultaneously, without replicating all units, and the scheduling of at which pipeline each instruction should be executed is simplified as it can be controlled with one unit. If a task is stalled due to a hazard, the microprocessor can still continue to run the other threads, which gives higher performance. Another possibility is to run different parts of a task speculatively. For instance if the task has a conditional branch, then all branches can be followed simultaneously by running them as speculative threads. When the condition is evaluated, all pipelines that ran incorrect branches are flushed, whereas the pipeline with the correct branch is allowed to commit, i.e., the results of its execution can be stored in the register bank and the memory.

[1]Many systems use external memories (e.g., hard discs), which can be seen as an even lower layer.

The difference between simultaneously multithreading and chip multi processors is that SMT can reuse some functionality between the pipelines which save chip area, power and potentially can increase performance. On the other hand, the complexity of SMT is much higher than CMP where the microprocessors are identical. Furthermore, CMP may be more reliable than SMT since fewer parts are common between the different processors, i.e., CMP may have fewer single-points of failure.

There also exist other approaches for increasing the performance by utilizing the large number of transistors that the chips can hold. For instance, larger memories can be placed on-chip and/or specific processors can be added to the memory to improve the prediction accuracies, and thus, reduce the memory access times.

In this section, the architecture of modern microprocessors have been described. In the next section, implementation of microprocessors will be discussed. As almost all microprocessors today are implemented in silicon in form of integrated circuits, we will focus the discussion to these. However, alternative platforms such as optical, biological, quantum, etc processors have been proposed, [Mule *et al.*, 2002], [Theis, 2003], [Lewin 2002], [Oskin *et al.*, 2002]. In the following sections, the device scaling trend and how different fault mechanisms are affected by the scaling will be discussed. This will provide information on which faults are likely to occur in the future.

2.3 The Integrated Circuit

The main building block of integrated circuits is the transistor. The most common transistor type is Metal Oxide Semiconductor (MOS) transistors, see Figure 2.2. These transistors consist of four connections: the source, the drain, the gate, and the

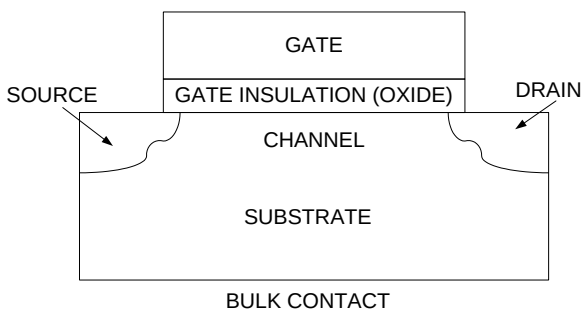


Figure 2.2: The structure of a general MOS transistor.

bulk. The gate is separated by an oxide layer. The source and drain are connected

to doped silicon that is surrounded by oppositely doped silicon, the substrate, like two islands. By applying a voltage to the gate, a channel of free charge carriers can be formed between the source and drain. If the applied voltage exceeds a certain threshold voltage, a current will flow between the source and drain. This means that *whether a current will flow between the source and drain can be controlled by the voltage applied at the gate*.

Thus, logical expressions can be implemented by putting several transistors together, but also memory elements, etc. Often, one N-channel (Negative doped) and one P-channel (Positive doped) transistor are connected to form an inverter, as seen in Figure 2.3, i.e., so called Complementary MOS logic (CMOS-logic). Using these as the basic building block, arbitrarily complex gates/circuits can be built.

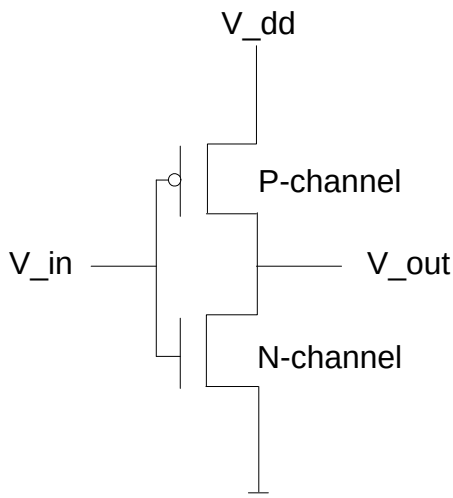


Figure 2.3: The structure of a general CMOS inverter.

As it is desired to hold as many transistors as possible on chip, transistors are often connected through layers of interconnects, metal planes and vias (holes which sides are filled with metal so that different metal planes can be connected). To connect the integrated circuit to other external components, the internal points of the circuit that should be connected using pads connected to bond-wires that in turn are connected to the pins of the package.

Having described the components of the integrated circuit, the continuing implementation optimization (e.g., reduction of the transistor size, supply voltage, increasing the clock frequency, etc.), i.e., device scaling is now discussed.

2.4 Device Scaling

It is desirable to scale transistors (i.e., reduce their size) as the resistance and parasitic capacitance decrease which make it possible to clock the circuit at a higher rate (increase performance), the power consumption per transistor will decrease, and the number of transistors that can fit on a single chip is increased. Therefore, developers have since the beginning of the integrated circuit area, tried to come up with methods to scale the size of the transistors. The problem with scaling can be described as follows: Decreasing the size of the transistor implies thinner oxide layers, i.e., reduced oxide resistance. Thus, to maintain the electrical characteristics (avoid that the electric field increases which could result in an oxide breakdown), the gate voltage must also be decreased. This is not directly a problem as it also reduces the power consumption which is desirable. However, reducing the oxide thickness also means that the probability for charge carriers tunneling through the oxide is increased. Moreover, also the threshold voltage is decreased which increases the influence of leakage currents. This implies that the power consumption do not scale at the same rate as the transistor size, [Borkar, 1999], [Eckerbert, 2002]. This is a major problem. It also results in a reduced signal-noise ratio, i.e., increases the sensitivity of the signal levels.

Another scaling problem is that when the clocking frequency is increased, the wire capacitance is increased which delays the signal distribution. Thus, the synchronization accuracy between different parts of the chip is reduced as differences in wire length gets significant. Another issue is the scaling of the interconnects as the resistance increases when the interconnect cross section area decreases.

A major problem with scaling is also how to test the integrated circuits when their complexity increases. Testing is also complicated due to that physical size of the devices is reduced and their power sensitivity (overheating) and clocking frequencies are increased. Furthermore, the manufacturing processes need to be more accurate when the device size decreases.

During the life-time of the integrated circuit, there have always been threats to the performance increase predicted according Moore's law (the number of transistors on chip will double each year), [Moore, 1965]. However, researchers have so far been able to come up with new solutions, keeping the performance increase up. Today, a large number of research projects addresses the issues mentioned above. Some suggested solutions are:

- New materials, e.g., the use of low resistivity metals, development of insulators with low dielectric constants.

- Advanced circuit design, e.g., using asynchronous building blocks.
- New transistor implementations, e.g., double and triple gate, strained silicon, etc.
- Improved lithography techniques.
- New testing techniques, e.g., more advanced scan chains (components added in chains to devices in order to be able to observe internal states) and Built-In Self-Tests (BIST).

Some examples of references of the large number of papers targeting problems and predictions of future design of integrated circuits are: [Hawkins *et al.*, 1999], [Torin, 1999], [Kam *et al.*, 2000], [Bryant *et al.*, 2001], [Kundu *et al.*, 2001], [Meindl, 2001], [ITRS, 2002], [Antoniadis, 2002], [Constantinescu, 2002], [Geppert, 2002], [Otten *et al.*, 2002], [Grundman *et al.*, 2003], [Hamilton, 2003], [Intel, 2003(A)], [Meindl, 2003].

Now, after describing the implementation of microprocessors and the device scaling trend, different fault mechanisms and how they are affected by the scaling trend will be described.

2.5 Fault Mechanisms

As described in the introduction, most faults can be divided into the classes: developmental, physical, and interaction faults. It was also motivated that fault tolerance is best suited for handling physical faults. However, as it seems like developmental faults cannot completely be removed through testing, these are also discussed in this section. Furthermore, as there are many different types of manufacturing faults, these are treated in a specific subsection. The listing of possible fault mechanisms are not in any way a complete survey but aimed at showing the recent trends. Furthermore, there exist a lot of dependencies between the different fault mechanisms (for instance, wear-out effects will generally have a faster lapse if spot defects exist on chip) which are not evident as the mechanisms are explained separately. The content of this section is to a high extent based on [Gil *et al.*, 2002] and it should be noted that *that the discussion of this section is supported by very few field data.*

Developmental faults:

As the complexity of components and the number of subcomponents of systems increases, the number of developmental faults can be expected to increase, simply due to the fact that there are more things to keep in mind, and there are more components that can break (a chain is not stronger than its weakest link), [Avižienis, 2001], [Avižienis, 2000]. Another problem is that the number of test cases increases when the number of transistors on-chip increases.

All software faults are defined as developmental faults (the software code of a system can change during operation, but this must be preceded by a hardware fault). Many faults are of course detected during compilation, code inspection, testing etc., but it is considered as an utopia to be able detect all software faults before operation. The set of faults left undetected after testing, are generally hard to characterize[2] (else, testing techniques for detecting them would have been developed). This is also the case for hardware developmental faults (hardware manufacturing faults may be an exception).

Furthermore, the set of faults left undetected after testing is probably also dependent on the program languages, development processes, and test techniques, being used. Thus, this set will probably change over time. These facts complicate development of techniques for tolerating development faults.

Examples of famous developmental faults are [Lions, 1996], the Ariane 5 rocket failure, and [Blum and Wasserman, 1996], the Intel Pentium floating-point division unit bug.

Manufacturing Defects:

The manufacturing process of integrated circuits (e.g., microprocessors) involve many steps such as applying different types of masks, etching, splitting (sawing), assembling, solding, bonding, etc. It also involves a large number of different metals and other substances. Thus, manufacturing of integrated circuits are itself a very complex process, and the increased number and the decreased size of transistors make the manufacturing process even harder.

In the rest of this section, different manufacturing defects will be discussed.

[2] However, an attempt to classify software faults occurring during operation was made in [Chillarege *et al.*, 1992].

2.5.1 Spot Defects

A spot defect is a permanent deformation located to a specific point of the die structure. These faults can occur due to dust particles, discrepancies in the silicon or in the metal of the interconnects, or due to incomplete manufacturing, as incorrect etching, plasma or mask processing, etc. The effects of these faults are that the impedance of some subcomponent (transistor etc.) is changed. Spot defects are often modeled as opens (unwanted impedances, mainly resistive, on a signal line that is supposed to conduct perfectly), shorts (unwanted connections between a signal line and the supply voltage or ground), and bridges (unwanted connections between two signal lines). As the line width decreases, the signal lines will get closer which increases the probability for discrepancies to result in faults.

Spot defects may not immediately generate faults, but can also speed-up wear-out and start to generate faults after a certain operation time period. Thus, the spot defects may not be visible directly after manufacturing, but may start to impact systems which have operated for a while.

Due to the reduction of transistor sizes, the critical size at which spot defects will impact the device will also decrease. Thus, the cleanliness and accuracy of the manufacturing processes most be improved when scaling, else their is a risk that the average operating time before spot defects will impact systems is reduced, i.e., the life-time of the device can be reduced.

For more details, see: [Hawkins *et al.*, 1994], [Segura *et al.*, 1995], [Soden and Hawkins, 1995], [McVittie, 1996], [Riordan *et al.*, 1999], [Lisenker and Mitnick, 2000].

2.5.2 Stress Voiding

During the wafer manufacturing process, the application of different masks may introduce mechanical stress. These may result in increased resistance, which can result in for instance, undesired delays.

2.5.3 Package and Assembly

Several faults can be introduced during packaging and assembling, e.g.,

- **Die Attach Faults**: If the die is not correctly mounted to the lead frame, the mechanical and/or thermal stress can be increased. This can with time lead to cracks in the die.

- **Bonding Faults**: If the bonding is not correct, the bonds may lift, resulting in open circuits. There is also the possibility that adjacent bond wires are shorted.

- **Delamination Faults**: If moisture is introduced during packaging, this may expand during operation and cause for instance delamination between the die and the lead frame.

As the circuits get smaller, the requirements on packaging increases.

Environmental Disturbances:

Many embedded systems are working in harsh environments. Here a number of different disturbances, possibly inducing faults, that can occur in such environments will be discussed.

2.5.4 Vibrations

Vibration is a phenomenon, which can occur when a component is exposed to mechanical forces. When an electric circuit is exposed to these forces, faults may occur due to imperfect manufacturing. For instance loose bondings, mechanical stress in the material, narrow interconnects can wear out faster due to vibrations. For more details, see: [Yu *et al.*, 2002]

2.5.5 Temperature Variations

In many embedded systems the temperature can vary quite a lot (for instance, in the engine compartment of an automobile the temperature can reach $125°$C). Varying temperatures affect electric circuits in a number of ways. First, as metal expands more than other parts of the circuit when the temperature increases, the mechanical stress is increased. Second, a temperature increase will reduce the mobility of the charge carriers, i.e., the internal resistances of the transistor will be higher, which results in longer delays.

Another very important effect is that the leakage currents increase when the temperature increases. This increases the power consumption.

For more details, see [Kundu and Galivanche, 2001], [Makabe *et al.*, 2000].

2.5.6 Radiation

Radiation occurs when an element decomposes. The radiation consists of different types of particles with different weight. Two different causes of radiation is discussed.

- **Radiation of Cosmic Rays**: In space, elements decomposes all the time causing different types of radiation called cosmic rays. When such cosmic rays enter the atmosphere and collide with the atoms there, particles are produced. These particles can in turn hit integrated circuits and if the particles movement energy is high enough, significant charges can be induced.

  For more details, see [Henson *et al.*, 1999], [Hazucha, 2000], [Massengill *et al.*, 2000], [Constantinescu, 2002], [Karnik *et al.*, 2002], [Shivakumar *et al.*, 2002].

- **Radiation of α-Particles**: Radioactive impurities, i.e., elements that spontaneously decomposes, exist in the packaging of integrated circuits. The radiation generated by the impurities may induce significant charges in the circuit.

  For more details, see [Constantinescu, 2002], [Karnik *et al.*, 2002], [Oldiges *et al.*, 2002].

Charges that are induced into the integrated circuit can make a RAM-cell, flip-flop or even combinational logic to change value. When transistors are scaled, less charges have to be induced to change the logical value, but the probability that the transistor is hit is reduced even more. However, as the size of the integrated circuit is not decreased (only more transistors are contained), the probability that a fault could occur in the circuit is increased.

In some cases, induced charges can activate so-called parasitic transistors, i.e., undesired device paths which cannot be activated during normal conditions. This can result in a short-circuit across the device until it burns up or the power to it is cycled. Such faults are called latch-ups and it is important to design the circuit to avoid such faults.

2.5.7 Electromagnetic Interference

As soon as an electric potential difference and/or electric current exist, an electromagnetic field is generated which will affect adjacent conductors. This can in some cases give raise to undesired disturbances that could result in errors in the system.

The interference is dependent on how far away the source (the generator of the electromagnetic field) is, its strength (charge), the transfer path, and the physical shapes of the source and receptor (the affected conductor).

Two different effects must be considered which impacts are determined by the distance between the source and receptor. These are:

- **Antenna-Induced Charges**: This effect occurs when charges are introduced due to sources at long distance. This is generally not a major problem for printed circuit boards as they are enclosed and the length of the conductors (possible antennas) are relatively short[3]. However, [White and Kim, 1996], [Kim *et al.*, 2000], [Eure *et al.*, 2001] have investigated the effects of strong EMI sources. Such sources can introduce bursts of faults at which several components/subsystems may be affected. As the consequences can be so severe, fault prevention techniques, e.g., shielding, are generally more efficient than attempting to tolerate such faults.

- **Cross Talk**: Cross talk means that a conductor induces charges (capacitive and inductive) on one or several other conductors on-chip. As the source in this case is much closer, the effects may be severe. Furthermore, as the circuits are scaled, the conductors (interconnects) are routed closer and closer which increase the effect, [Clayton, 1992].

2.5.8 Electro Static Discharge (ESD)

Charges can be introduced into integrated circuits in other ways than through radiation. One such example is through contact with electro static charged components/objects. Even though the discharging generally has a very fast lapse the impacts can be severe as the voltages can be very high. Therefore, integrated circuits should be protected from electro static discharges, for instance through employing specific diodes. However, such protection is getting more expensive as the transistors are scaled, e.g., making it more difficult to lead away the charges and at the same time stop charges on the circuit from taking the same way, i.e., avoid additional leakage currents. Moreover, the oxide thickness reduction makes the transistors more sensitive to ESD.

For more details, see: [Ker *et al.*, 2002], [Clayton, 1992], [Huh *et al.*, 1998], [Vinson and Liou, 2000].

[3]For transmission lines, as for instance communication buses, the effect can be severe, [Clayton, 1992].

2.5.9 Supply Voltage Disturbances

As integrated circuits need to be supplied with voltage, they are sensitive to disturbances affecting the supply voltage source or the supply voltage wiring. Such disturbance could for instance occur due to overloads (the supply source is to heavily utilized) or electro magnetic interference. As the supply voltage is scaled, disturbances will have greater influence, [Kundu and Galivanche, 2001], and there is a risk of temporarily changing the behavior of transistors as well as permanently damaging components.

Wear-Out (Aging)

Wear-ot faults are caused by the stress that is always present for operating components/systems. Wear-out is a slow process that does not immediately results in faults (therefore, aging is sometimes used to denote this mechanism). If manufacturing discrepancies exist, these are often the first places where the wear-out effects become visible. In most cases, wear-out faults first manifest as errors only at specific operating points, i.e., system states. Therefore, it is hard to distinguish these errors from those caused by transient faults. However, the difference is that if they are not taken care of, they will with increasingly be activated more often, resulting in more severe effects.

We now discuss different wear-out effects in more detail:

2.5.10 Electromigration

As interconnects are scaled they get more sensitive to the currents flowing through them. If they are properly designed, there is generally no danger that they will break, but if the current is flowing only in one direction of the conductor, there is a danger that it will actually move metal atoms. This phenomenon is called electromigration and can result in resistance changes and even open circuits.

For more details, see: [Besser *et al.*, 2000], [Fischer *et al.*, 2000], [Fischer *et al.*, 2002], [Lienig *et al.*, 2002].

2.5.11 Corrosion

If the integrated circuit is exposed to moisture, the conductors can corrode which will change their resistance and eventually, lead to open circuits.

2.5.12 Gate Oxide Faults

The gate oxide is the part of the transistor that should isolate the gate from the silicon. When the transistor is scaled, the thickness of the oxide is reduced. This requires the gate voltage to be reduced in order to avoid failures. If defects exist in the oxide or the oxide is thinner at some places due to process variations, the electric field may be stronger at these spots. If the strength of the field gets too high, the oxide will break down, resulting in short-circuits, but even weaker fields can cause damage as the wear-out lapse may be speeded up. Some common gate oxide wear-out mechanisms are:

- **Hot Electron Injection**: When the transistor is conducting, a current flows in the silicon channel created by the electric field between the gate and the substrate. Some of the electrons in the current may have extremely high energy. Such electrons are called hot. If these electrons gains a direction towards the oxide, they may not just "bounce" but actually penetrate the oxide, due to their high energy. Once inside the oxide, their movement energy reduces fast and they get trapped in the oxide. This phenomena is called hot electron injection and the trapped electrons reduces the isolation capacity of the oxide, which may eventually cause a break-down.

  Hot electrons may also be trapped into the substrate of the transistor, reducing the quality of the transistor. Other mobile particles, as for instance ions, may also cause similar effects.

  For more details, see: [Lee *et al.*, 2000], [Zhang *et al.*, 2001], [Kaczer *et al.*, 2002], [Zhang *et al.*, 2002], [Crupi *et al.*, 2003].

- **Tunneling**: According to quantum mechanics, there is always a certain probability that a particle can move through an object. This effect is called tunneling. The probability for tunneling is of course dependent on the thickness of the object, and thus, when scaling devices and the oxide thickness is reduced, the probability for tunneling increases, resulting in increased leakage currents. For more details, see: [Nicollian *et al.*, 1999].

2.6 Fault Occurrence Rates for Integrated Circuits

Based on the discussion of the previous section, we believe that a large number of fault mechanisms will have higher impact when the size of the transistors are scaled (However, it should be emphasized that very little field data exists to support this

claim). Furthermore, the increased system (hardware and software) complexity is likely to increase the number of undetected developmental faults. There is though a large number of engineers and researchers addressing this issue by inventing and developing new and more: transistor implementations, materials, manufacturing processes, design tools, testing methods, shielding techniques, etc. However, the main goal of most of this research is to make it possible to develop microprocessors with higher performance, working in stable environments, and where occasional faults are acceptable and have low impact. This is generally not the case for embedded control systems where the operational environments often are harsh with extreme levels of particle radiation, electromagnetic interference, vibrations, temperature changes etc., and where the consequences of faults can be catastrophic. Therefore, we believe that the occurrence rate of transient faults, but also of persistent faults (important as the impact of these faults generally are more severe), will increase. Some significant reasons for this are:

- The wear-out phenomena, discussed in the previous section, are expected to have faster lapses as the sizes of transistors decrease, implying that the critical dimension for stress and channel region defects introduced during manufacturing (too small to be directly detectable through post-silicon testing) also will be decreased. Also, the increased number of transient faults may increase the stress and accelerate the aging process.

- Some modern commodity components integrate parts with different clock domains. Thus, faults may propagate from one clock domain to another. Therefore, for fault tolerance techniques applied at the system level, transient faults on a "slow clock" subsystem may appear as persistent faults on a different, faster part of the same die. Furthermore, the constantly increasing clocking frequency of circuits has the side effect that disturbances that previously only generated errors at one single point in time now can generate errors at several points in time, i.e. resulting in persistent effects.

- Research on high performance and low power processors may require on-chip systems to be designed using GALS (Globally Asynchronous Locally Synchronous) design, [Beerel, 2002]. This new design technology involves building the system of "voltage" or "frequency" islands, which are connected with an asynchronous event based network. In such a design, it is hard to bound the fault activation and error propagation times. So faults in such systems would be classified as persistent.

Another problem, [Driscoll *et al.*, 2003], is the increased observations of Byzantine faults. The generated errors by such faults can be interpreted differently for different connected components, implying that different parts of the system may estimate data and/or states differently.

We believe that these findings require novel strategies when designing fault tolerant systems in the future. First, techniques need to be added for tolerating not only transient faults but also persistent faults, specifically considering their severe impact potential. Second, tolerance against developmental faults with low activation frequency needs to be provided. Third, Byzantine faults need to be considered during design.

The work in this thesis aims at aiding system developers and researchers in academia to develop systems based on commercial microprocessors that are tolerating faults with varied duration.

CHAPTER 3

Design of Efficient Fault Tolerant Microprocessors

As was discussed in the previous chapters, we believe that fault tolerance will be necessary to meet the requirements for safety-critical systems. In this section, as error detection is one of the major concepts to achieve fault tolerance (see the introduction for details), the effect of the fault mechanisms (i.e., the generated errors) described in Chapter 2, on different abstraction levels is discussed. Looking at the microprocessor level, the effects (errors) can be divided into three main classes:

- **Crash Failures**: The microprocessor has incorrectly halted.
- **Control-Flow Errors**: The instructions have been executed in an incorrect order.
- **Data Errors**: Incorrect data has been produced.

After that, a design for efficiently tolerating faults based on complementary error detection techniques (a survey of different error detection techniques is found in Appendix A) is suggested. This solution has not been evaluated, but we believe that it describes the problems and feasibility of applying software-implemented fault tolerance techniques to commercial processors. The choice of the error detection techniques is motivated by detection efficiency, cost and overhead considerations.

Finally, evaluation of the design is discussed. Previous evaluations have mainly considered injection of transient faults. However, as was discussed in Chapter 2, we believe that in the future, persistent faults must also be considered. As there is a

high risk of damaging the circuits when injecting such faults physically, generally simulation and analysis methods must be used. There are mainly two problems related to such evaluations of persistent faults:

- It is important to use a fault model that covers the different types of real faults.
- Detailed low-level circuit descriptions are generally not accessible for system developers. Thus, high-level models must be used, which makes it hard to reach accurate evaluation results.

We claim that these problems are mainly related to data errors as the impact of crash failures and control-flow errors are rather independent on the fault duration. Therefore, we propose a new approach for evaluating the effect of data errors as well as the coverage of applied errors detection techniques. The approach is based on identification of the data error characteristics that impacts the system and varying these characteristics rather than injecting faults according to a model.

We believe that this is approach is a useful complement to traditionally techniques. In following chapters, we develop analysis methods (Chapter 4) and a simulation-based fault injection technique (Chapter 5) for such evaluations.

3.1 Error Propagation Between Abstraction Levels

As was described in Chapter 1.1, the terminology of faults-errors-failures is bound to the abstraction level of observation. At that abstraction level, the fault produces an error (incorrect state) when it is activated. In turn, if the error propagates to the border of that abstraction level, a failure is said to have occurred, which is considered as a fault at the next higher abstraction level. Thus, in order to determine which impacts the physical faults described in Chapter 2 will have on the system, the propagation of errors need to be mapped.

Figure 3.1 describes the error propagation between abstraction levels. Starting from the physical faults described in Chapter 2 (a discussion on the effect of developmental faults is held last in this section), it was found that a hardware fault implies a change in voltage, current, and/or resistance, i.e., the electric characteristics are changed[1]. This change could be temporarily or persistent dependent on the nature of the fault. When this change occurs, an error is said to have been generated on the *electrical level*.

[1]The fault-error cycle could be described at even lower levels as particle diffusion laws etc, but this is out of the scope of this thesis.

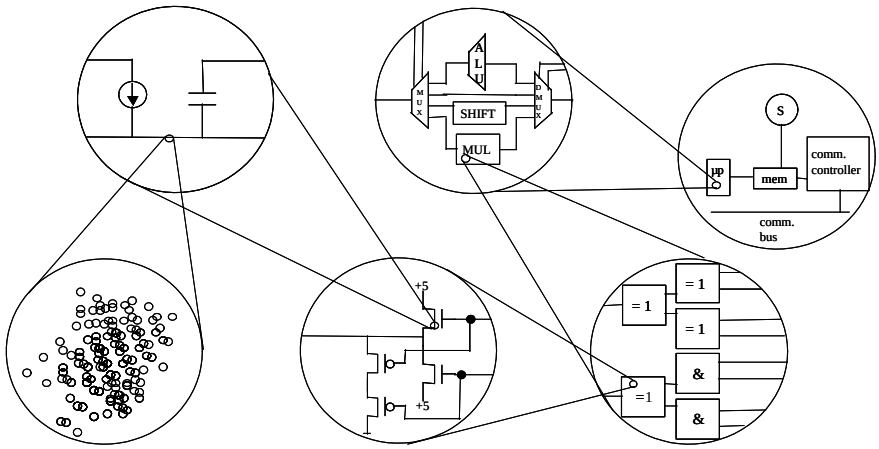


Figure 3.1: Error Propagation between different abstraction levels.

As microprocessors mainly are built of transistors, changes in the electric characteristics may result in that a transistor is switched on or off at incorrect times. If this happens, this mean that the error at the electrical level has propagated to the *switch level*. Whether this happens or not depends on how much the electrical characteristics are changed (**Influence**), at what time (**Occurrence Time**), for how long time (**Duration**), and the function of the specific effected transistor (**Location**). Small changes with short durations do generally not have any major impact on the function of the transistor. However, it is important to note that if the transistor function is degraded and the device is clocked at discrete times, even small changes with short durations may have non-negligible effects. If the error at the electrical level does not affect the function of the transistor, it is said to be masked.

Looking at the error at the transistor level, the effect of the transistor switching on or off at incorrect time may be that the output of a gate or memory cell assumes an incorrect value. In such case, the error has propagated to the *gate-level*.

Gate-level errors may correspondingly result in combinational logic, sequential logic or memory errors, dependent on the location of the fault. If so, the error has propagated to the *Register Transfer Level (RTL)*.

As the components are rather complex and different from each other at this level, the errors they will generate will also be different. The effect of faults on some of the components and functions at RTL will now be discussed in more detail.

Instruction Fetch Unit: This stage can be influenced not only by faults in the components of the stage itself, but also by memory faults. Faults can for instance

result in that an illegal or incorrect instruction is fetched or that the instructions are fetched in an incorrect order.

Instruction Decoder: At this stage the control signals for the specific operation are generated. If a fault occurs in this stage it can mean that an incorrect instruction is executed.

Functional units: In the execution stage the output or effective address, is computed. For branch instructions, this will mainly result in that the instructions are executed in incorrect order, whereas for other instructions it will mainly result in incorrect results.

Memory Access: At this stage, the memory is accessed. For branch instructions, this will mainly result in that the instructions are executed in an incorrect order, whereas load and store instruction will mainly result in incorrect results.

Register access: At this stage, the results are written to the register bank. This will mainly result in that incorrect data is stored in the registers, but also control-flow errors are possible if the register content can be used to determine jump addresses, i.e., so-called indexed addressing.

Thread, Prediction, Instruction Reorder Logic: Faults in these logics can result in that instructions are executed in an incorrect order.

Registers and Data Buses: Faults in these units should mainly result in that incorrect data are delivered.

Oscillator and Clock Signals: A fault in these units can result in that parts of or the complete processor stops operating.

However, independently of the specific fault, we have divided the errors into three main classes:

- **Crash Failures**[2]: The microprocessor has incorrectly halted.
- **Control-Flow Errors**: The instructions have been executed in an incorrect order[3].
- **Data Errors**: Incorrect data has been produced.

A microprocessor level error will in many cases affect the system[4]. If the de-

[2] As a microprocessor crash is an external incorrect state, we use the term failure rather then error.

[3] With correct order, we mean that the instructions are run in such order that they will give the same result and not less performance than the order that was designed by the programmer. Therefore there exists often several correct orders, which is utilized by out-of-order multiprocessors, see Chapter 2.

[4] The error propagation can be divided into more abstraction levels in between the microprocessor level and system level (e.g., a computer node level), but as we in this stage do not want to make any assumptions about the system architecture, and as we do not add any hardware-implemented fault tolerance techniques, we assume that all microprocessor faults will propagate to the system level.

livered service by the system deviates from the fulfilling the system specification, a system failure is said to have occurred. For real-time systems, requirements generally are to deliver correct data at the correct time, and thus, the notions data errors and timing errors are generally used for such systems.

So far, only physical faults have been discussed. However, hardware developmental faults will be activated in the same way as the physical faults. Errors caused by software faults will though in the beginning propagate a bit differently.

Assuming that the software has been properly compiled, a software fault implies that some incorrect information (generally an instruction or data) is stored in the memory. Such faults will be activated when this part of the memory is used, i.e., when the instruction or data including the fault is fetched. Therefore, one can say that software faults that are activated generates errors at the RTL. After that, the errors propagate in the same way as for physical faults. However, how often errors will be generated is of course not dependent on the use of any hardware unit, but on how often the incorrect software is accessed.

Summary

As was seen in the previous section, faults in different components will result in different errors. It was seen that whether a fault is activated or not at a certain abstraction level depend on:

- **Influence:** How much the characteristics are changed.
- **Occurrence Time:** At what time did the fault occur (in which state the component/system is and the value of the input data).
- **Duration:** How long time the fault persist.
- **Location:** The function of the faulty component.

3.1.1 Implications of Fault Tolerance

Today, fail silent computer nodes (i.e., the nodes should fail “cleanly” by just stopping to send messages, [Powell *et al.*, 1988]) are desirable for distributed safety-critical systems, which implies that faults should be tolerated on node-level. The reasons for this are:

- It is comparably expensive to recover from errors that have spread to several nodes.

- Communication between nodes is expensive which makes error detection between nodes unattractive.

- Error propagation to actuators can be limited, as today, actuators often are connected to computer nodes rather than to the communication network.

Thus, fail-silent computer nodes simplifies design and reduces cost.

As the microprocessor generally is the most complex component of the computer node, and thus, generally the most failure prone, it is of utmost importance that faults in this component can be tolerated to meet the requirements on fail-silence.

As fault masking generally implies high recurring costs, fault tolerance in systems produced in large number is often implemented by error detection and recovery. On which abstraction level fault-tolerance should be applied is is a trade-off. If detection is applied on a high level, detection is generally less costly, but the errors may have propagated to a large portion of the system which make recovery more costly. On the other hand, the total cost for error detection at low abstraction levels are comparatively high, but as errors are detected early (low latency) recovery is often less costly.

In the next section, how to design a cost-efficient fault tolerant microprocessor based on adding software-implemented techniques to commercial microprocessors is discussed.

3.2 Design through Software-Implemented Techniques

In the introduction (Chapter 1), different strategies for designing fault tolerant systems were discussed. An efficient fault tolerant design means, first and foremost, that faults are tolerated with a very high probability. However, to also be practical implementable, the design must also be cost and resource-efficient (e.g., performance, memory, device area, power). It is also an advantage if the design is portable to different hardware platforms (different brands of microprocessors etc.) and scalable (efficient fault tolerant design was defined in detail in Chapter 1.4). Therefore, it is these criteria we target when we now discuss different fault tolerance solutions based on error detection and recovery.

It is sometimes more practically to design the recovery policy first, as it sets the requirements on the error detection. Thus, first, recovery is discussed and after that, efficient techniques for detection of the errors identified in the previous section, will be suggested.

3.3 Recovery

As described in the introduction, recovery is aimed at transforming the system into a correct state where faults are not activated again. This is achieved through error handling (eliminating error from the system state) and fault handling (locating faults and preventing them from being activated again). Starting backwards, fault handling involves four steps:

- Fault diagnosis: Identification and recording the cause(s) of error(s), in terms of both location and type.
- Fault isolation: Performing physical or logical exclusion of the faulty component(s) for further participation in service delivery, i.e., it makes the fault dormant.
- System reconfiguration: Switching in spare components or reassigning tasks among non-failed components.
- System reinitialization: Checking, updating and recording the new configuration and updates system tables and records.

It is hard to give any general solutions for the implementation of fault isolation, system reconfiguration and system reinitialization steps as some systems should be set in a fail-safe state when a unrecoverable[5] error has been detected, whereas other systems require the function to be transferred to a spare unit. However, generally, such measures are costly as they reduce the resources of the system. Therefore, it is very important that these steps are performed correctly and only performed when absolutely necessary in order to avoid failures (e.g., for minor faults, i.e., faults with negligible effects as transient faults, these measures should be avoided). To achieve this, these measures must be robust, meaning that it can with high probability be guaranteed that the measures were not incorrectly, triggered or executed, due to a fault. Robust design will be discussed in Chapter 3.4.3.

It should be noted that sometimes, some of these measures are left out. For example, when it is not required due to the nature of the fault (e.g., some transient faults), there already exists components performing the same tasks (so called active replicas), or the system can be allowed to degrade, i.e., loose functionality in a controlled manner.

[5]With an unrecoverable error/fault we mean an error/fault that the microprocessor cannot recover completely from itself.

It is also important to note that the use of replicated components are not efficient for handling of developmental faults (except for some manufacturing faults) unless diverse replicas are used. However, design diversity is costly, so for systems where the recurring costs are critical (which we consider in this thesis), we assume that the system has been accurately tested so that developmental faults are activated with low frequency. This means, that it should be possible to recover from such faults just by changing the state of the system slightly (e.g., re-executed the task with new sensor values), i.e., without any isolation, reconfiguration or reinitialization.

The goal with fault diagnosis is to determine which fault that has occurred, and which measures should be taken. Thus, it is very important that the fault diagnosis correctly points out the fault that really occurred, since otherwise, the fault is left untreated, and correct parts of the system may be shut-down. Thus, the fault diagnosis must also be robust designed.

Fault diagnosis can be performed based on a-priori information on how faults are activated, information from how the error was detected, and from specific fault diagnosis tests. As diagnosis can be a complex task, and the time requirements can be strict, the diagnosis is often precluded by error handling, i.e., eliminating errors from the system state, and storing status and events. Then, at specified times or at times when the processor is less loaded, the stored information can be analyzed and detailed tests run.

To give an example, consider a situation where a certain executable assertion interrupt the execution of a specific task. The result from that task is then flagged erroneous, possibly recovered, and the event stored. Then, the next task is executed (unless the result of the erroneous task is very critical. In that case it can be re-executed). If no further interruptions occur within a specified time, the fault is diagnosed as a transient fault, and no further actions are necessary. However, if the same assertion is triggered at the next execution of the same task also, it might be necessary to disconnect that application (fault isolation). Then, accurate diagnosis test can be run to determine the fault during the normal execution time of that application. If the function has no fail-safe state, the task must be executed on a spare unit.

Another distinct case is if several different error detection techniques are triggered several times. Then it can be assumed that a unrecoverable fault has occurred and the node has to disconnect completely from the network. In order for this node to be reconnected to the network, extensive tests must be run. It should be noted that what is determined as a minor or a unrecoverable fault, is in many cases dependent on how often the fault is activated, and not, whether the fault is a transient fault or a persistent fault.

It is important to note that it generally is impossible to recover from developmental faults (except for random manufacturing faults) through spare components, unless these components are diverse designs. Furthermore, it is generally also impossible to diagnose such faults, as if the diagnosis would have been run after manufacturing (which should be the case if proper testing has been performed), it would have been detected at that time. Thus, it is very important to guarantee that developmental faults only are activated very occasionally, so that they can be treated as transient faults.

3.4 Error Detection

As identified in Chapter 3.1, errors at the microprocessor level can be divided into crash, control-flow, and data errors. In this section, in order to efficiently cover these errors, a number of detection techniques are suggested (more details about different error detection techniques can be found in Appendix A). The major targets for error detection is to detect, in time, those errors that else could result in catastrophic consequences and to provide information for fault diagnosis. It should also be efficient in terms of cost, performance, memory, device area, power, and portability (see Chapter 1). Techniques based on hardware replication are avoided, as the focus is on systems with strict requirements on recurring costs.

Error detection techniques can be divided into system-dependent and systematic techniques. System-dependent techniques utilize information about the system to determine whether an error has occurred or not, whereas systematic techniques detects errors through general properties as coding or executing tasks several times on the same or different processors. System-dependent approaches often have low recurring costs as they are based on information redundancy, but the fixed costs may be higher than for systematic techniques, as they need to be designed, or at least tuned, for the specific system of interest. However, this also implies that it is possible to design them to only detect severe[6] errors (an example of this is given in Chapter4.5), which often is hard to do with systematic techniques, reducing the risk of unnecessary recovery.

We will now present suitable, software-implemented, techniques to run on a microprocessor in order to detect crash, control-flow, and data errors.

[6]With a severe error/fault we mean an error/fault that, if not handled correctly, may result in catastrophic consequences.

3.4.1 Crash Failures

When a crash failure has occurred, the microprocessor has halted, and thus, is not executing any instruction. This does not mean that the processor is fail-silent as its output signals may be locked to undesired values, and thus, it is important to detect these errors. Furthermore, if such error is detected, it may be possible to restart the processor.

As the microprocessor cannot execute any instructions, these errors cannot be detected on the erroneous chip unless it contains two separate oscillators, where one oscillator can be used to drive a watchdog timer for monitoring that the processor (driven by the other oscillator) has not crashed. Such chips exist [Temple, 1998], but is not standard in commercial processors. Therefore, to detect crash failures, on single oscillator circuits, an external component driven by another oscillator must be used. An example of such detection technique is the so-called "I'm alive" message protocol, see [Pradhan, 1996], [Dimmer, 1985], where messages are transmitted between different computer nodes with even and predetermined time intervals. The use of these messages can be extended to show not only that the processor has not crashed, but also that it delivers correct results. The more information that is exchanged and verified between the different computer nodes, the more reliance can be put on that the microprocessor is working correctly, but the overhead will also be greater. Other issues that must be considered are that the messages can be effected by other faults faults and how to determine whether it is the receiver or sender that is erroneous when an error is detected.

In distributed safety-critical systems membership agreement protocols (see for instance [Claesson, 2002]) are generally used to detect crashed computer nodes. However, as the communication generally is handled by a separate communication controller, messages may be transmitted correctly even if the microprocessor has crashed, but such agreement should at least guarantee that the oscillator of the communication controller is working properly (independent on whether the communication controller and the microprocessor are driven by the same oscillator or not). Therefore, by having the communication controller checking that the information it receives from the microprocessor (generally through the memory) is regularly updated, also crash failures of the microprocessor can be detected. A simple example of such check is to add a task to the microprocessor that increases a variable stored in the memory, within even intervals. This variable should then regularly be checked by the controller, Thus, in these systems, crash failures are detected without adding any specific techniques.

For other type of systems, we believe that crash failures are best detected through

use of simple "I'm alive" messages on the communication network.

3.4.2 Control-Flow Errors

A control-flow error implies that the task has not been executed according to the intended control-flow, i.e., it has been executed in an incorrect order. For detection of control-flow errors, software-implemented signature checking as for example [Alkhalifa *et al.*, 1999], [Oh *et al.*, 2002(A)] are efficient to use. However, watchdog timers should be added to verify that the signature is checked repeatedly[7]. One approach is to reset and start one timer whenever a new task is started. Then a second timer can be reset and started as soon as the task has finished executing. In this way at least one timer will always be running to detect control-flow errors.

3.4.3 Data Errors

A data error means that any stored, transmitted or transformed data assumes an incorrect value. Data that is stored or transmitted is generally easy to protect with error detecting and some times even correcting codes. Such techniques are applied in most modern microprocessors, and thus, we believe that it is not necessary to add any software to detect such errors.

However, the transformation of data is not as efficient to protect with coding, as more complex codes are required, [Pradhan, 1996]. Looking at the function of the microprocessor, the main data transformation performed are instruction decoding, address computation, and data computation. For instruction decoding and address computation faults, generating data errors, they are also likely to generate control-flow errors, which can be detected with the applied control-flow error detection techniques. However, for those faults that only result in data errors, e.g., faults in the functional units of the execution pipeline state, techniques specifically for detection of data errors, need to be added. We will first discuss this with specific focus on control systems, and then briefly discuss other types of systems.

Control Systems

Contradictory to crash and control flow errors, the impacts of data errors are in many cases negligible, [Vinter *et al.*, 2001], [Cunha *et al.*, 2001]. Therefore, it is important to avoid spending resources on errors that would have no significant impact

[7] Most such errors would be detected with the detection techniques applied for detection of crash failures, but the latency for such detection may be high. Therefore, in order to limit the error effects as much as possible, watchdog timers should be used to decrease the detection latency.

on the system. In this sense, system-dependent error techniques are preferable as they are specifically designed for the specific system, i.e., can be tuned to detect only the severe errors. One such technique, which has shown to be able to detect many errors, is executable assertions, i.e., software code that checks the validity of a property (generally a value of a signal/variable) or a set of properties in order to find faults either during system development or operation. We will now discuss how executable assertions can efficiently be applied in order to reduce the impact of data errors without using up resources. Also discussed is how the system can be put in a fail-safe state or the control handed over to a spare unit in time to avoid system failures when a unrecoverable error has been detected. As the problem consist of two parts, we now treat them one by one.

Automatic Reduction of Data Error Impacts

The impact of data errors can be reduced without using up resources by detection and automatic recovery. Automatic recovery can be performed by replacing the erroneous signal value with the closest acceptable value, [Hiller, 2000], the previous value, [Vinter *et al.*, 2001], or by correction through a modified anti-windup function, [Gäfvert *et al.*, 2003]. All these are general techniques with low overhead, but it is out of the scope of this thesis to investigate which of them that offer the most efficient recovery (there may very well exist other solutions that are even more efficient).

The most natural way to set the thresholds[8] for assertions are to use the physical signal bounds. Such magnitude and rate bounds were computed and applied in [Stroph and Clarke, 1999] for the open-loop system, and in [Gäfvert *et al.*, 2003], magnitude bounds for the closed-loop system (which gives tighter bounds than for the open-loop) were used . However, for signals that have a large acceptable value space, these assertions may not be efficient enough. In this case, dynamic bounds, [Stroph and Clarke, 1999], may be used, though this increases the complexity. It should be noted that it is always possible that the signal is disturbed after the check. Still the assertions are not useless as the error may be detected at the next check, and many other errors will be detected directly.

Another alternative is simply to reduce the acceptable value space (tighten the thresholds) for the assertions, but if the specifications of the systems are not simultaneously adjusted, false alarms can occur, demanding to be handled when the system is stressed. As systems often are very resource demanding when they are stressed, we believe that false alarms should be avoided as far as possible.

Still, if the assertions are considered to be insufficient, the computation of the

[8] We use the term threshold to denote the borders for the acceptable value space of executable assertions.

control algorithm, or part of it as suggested in [Vinter *et al.*, 2002], can be replicated (systematic error detection) either on another computer (spatial redundancy) or on the same computer (time redundancy), see Appendix A. The overhead for these solutions are higher, but not extreme.

Detection of Unrecoverable Errors

Determining when an error is unrecoverable and the system should be set in a fail-safe state or the control handed over to a spare unit, is a multi-faceted problem. First, how can an assertion be designed so that it will work even in the presence of an unrecoverable, internal error? Second, which signals should the assertions check? Third, how should the threshold for the acceptable value space be chosen?

These problems are rarely treated in literature. In [Cunha *et al.*, 2001], and [Cunha *et al.*, 2002] solutions targeting the process industry are proposed. These are not directly applicable to safety-critical systems as they are either ad-hoc or very time-consuming. However, these papers give valuable insights which we will make use of when we try to solve the problem for safety-critical systems.

Robust assertions, [Silva *et al.*, 1998], is a technique for protecting assertions from errors, based on double execution and a simple code (magic numbers). The code is used to check that both executions of the assertions and a comparison of them had been correctly performed. These solutions provide protection against transient faults occurring during the execution of the assertion (else the system could be requested to recover unnecessarily). Is a mayor advantage if the used assertions are checks, not so much because the overhead is low, but since this increases the chance that the assertion is executed correctly despite of the unrecoverable error[9]. For more complex versions, the risk is higher that the assertion also is affected by the fault.

However, robust assertions seem to have two indistinctions. First, how should errors in the the magic number (the code) be handled? To set the system in a fail-safe state or hand over the control may not be the most efficient solution (which is the suggested measure in [Silva *et al.*, 1998], [Cunha *et al.*, 2001]), as such errors can be caused by control-flow errors. An alternative is to delay this measure until several similar errors have been detected in adjacent updates of the signal, when it can be established that a severe fault has occurred. However, this increases the complexity of the check, as some sort of buffer is necessary, which makes the robust assertions more vulnerable.

Second, these assertions seem to be vulnerable to faults in the compare oper-

[9]To as far as possible avoid the assertions to be affected by the same errors as the computation of the control algorithm, it would be desirable to execute the assertions on external hardware/computer. However, this would require synchronization which reduces the performance, and it leaves the problem of determining which unit that is erroneous, the controller or the checker?

ation, e.g., “stuck-at equal” comparison errors (all comparisons are claimed to be equal). In order to avoid this we suggest to add a few additional comparisons which are known to provide not equal or equal in advance so that “stuck-at equal” and “stuck-at not equal” errors can be detected, i.e., a simple test of the compare function (this test is similar to the self-tests proposed in Appendix C). If this test fails it should be re-executed in order to remove the possibility of transient faults. This test should also be assigned a magic number (a code) so that it can be guaranteed that the test has been executed.

In order to determine which signals the assertions should check, a signal monitoring the health of the system must be determined. Then, the thresholds for the assertions on this signal should be set so that the recovery can be performed in time to avoid system failures. This is a delicate balance as tight thresholds may unnecessary set the system to a fail-safe state or hand over the control, which are safety-critical measures them-selves. On the other hand, setting wide thresholds may imply that the probability for system failure is increased.

In Chapter 4.5, an example of how to design executable assertions for control systems is presented.

General Systems

Control systems can be considered to be rather deterministic as the state of the system generally cannot change dramatically between samples due to the inertia of physical processes (The sample rate is often designed so that the system should not be able to change state arbitrarily between samples). This makes system-dependable error detection techniques efficient as they are based on estimating the state of the system. However, for systems that are not equally predictable, systematic techniques must be applied (as discussed previously, this may also be necessary for control systems). Such techniques are generally based on space or temporal redundancy. As discussed previously, space redundancy requires additional hardware which increases the fixed cost, and thus, temporal redundancy is preferred.

Double execution, i.e., executing tasks twice and comparing the results is one of the most commonly used temporal redundancy technique, see Appendices A and B. In some implementations, diverse tasks are executed, [Johnson, 1989], [Lovric 1996] in order to increase the coverage for errors caused by persistent faults. However, there are several disadvantages with diverse tasks. The tools for developing diverse programs generally targets only one type of microprocessors, and is thus not portable. The development process may also be very time-consuming and which detection coverage that can be reached varies between applications and micropro-

cessors. Even if some of these problems are targeted in [Jochim, 2002], we suggest that the same task should be executed twice.

Double execution can detect some errors caused by persistent faults, see [Aidemark *et al.*, 2003] and Appendix B. The advantage with this is that such errors need to be detected as they generally have more severe impacts, but the drawback is that it complicates the error handling and fault diagnosis. Even if some errors caused by persistent faults are detected by double execution, the coverage is generally not sufficient. Therefore, we propose to add self-test tasks (see Appendix C). Previous online test approaches has mainly been based on on-line Built-In Self-Tests (BISTs), since software self-tests has been considered too slow. However, we challenge this viewpoint because of the following reasons:

- The tests do not need to detect persistent faults in the entire microprocessor, but can be focused to detect data errors, i.e., to the functional units. This reduces the required test length drastically.
- In contrast to manufacture testing, on-line testing is not aimed to detect all existing faults, but to avoid data errors to be activated in such rate that a system failure could occur. For instance, it was discussed and shown in [Vinter *et al.*, 2001] and [Cunha *et al.*, 2001] that many data errors caused by transient faults are tolerated by control systems.
- BIST solutions require support from the manufacturing company and implies a switch between normal execution and executing the BIST which is a potential safety risk. This is not the case with software-implemented tests as these are executed as normal tasks.

The drawback with systematic error detection techniques are not only that the overhead generally is high, but also that it can be hard to differentiate between minor and severe errors. One approach for filtering errors with low impacts, is to accept small differences at the comparisons between the two executions and for the self-test. It is also possible to add executable assertions to facilitate the differentiation.

As most errors detected by double execution will be caused by transient faults (assuming that the occurrence rate of transient faults is higher), the task triggering this technique should be re-executed, if necessary, and the event stored for further diagnosis. If a fault is triggered by the self-test, the self-test should be re-executed to exclude transient faults. If the fault persist, the microprocessor need to be disconnected, and necessary functions transferred to spare units, so that detailed diagnosis can be performed. The executable assertions should be applied to facilitate differentiation between minor and severe errors.

3.5 Coverage and Overhead

To summarize, we propose to use "I'm alive" messages for detection of crash failures (on other computer nodes or by the communication controller). As it may be difficult and time-consuming to restart the microprocessor after a crash failure, important services must be transferred to spare units.

For detection of control-flow errors we suggest to use control-flow checking and watchdog timers. When a control-flow error is detected, if necessary, the task should be re-executed. Else, the next task should be executed. If several errors are detected close in time, it can be necessary to disconnect the node for more detailed diagnosis at which important services must be transferred to spare units.

For detection of data errors we suggest to use executable assertions to monitor deterministic properties. If a value that cannot occur during normal operation is detected, automatic recovery can be applied (for instance using the method suggested in [Gäfvert *et al.*, 2003]).

Also, we suggest to use assertions for monitoring the health of the system. If these assertions are triggered, the system should be set in a fail-safe state or the control handed over to a spare unit. These assertions should be robust designed,as reconfiguration due to false alarms are expensive and may be safety-critical.

For properties which are not deterministic enough for assertions to be efficient, systematic approaches must be applied. We suggest to use double execution complemented with self-test tasks and executable assertions. As most errors detected by double execution will be caused by transient faults (assuming that the occurrence rate of transient faults is higher), the tasks triggering this technique should be re-executed, if necessary, and the event stored for further diagnosis. If a fault is triggered by the self-test, the self-test should be re-executed to exclude transient faults. If the fault persist, the microprocessor need to be disconnected, and necessary functions transferred to spare units, so that detailed diagnosis can be performed. The executable assertions should be applied to facilitate differentiation of minor and severe errors.

As the approach has not been evaluated, it is hard to estimate the total detection coverage and the overhead related to the solution. Thus, below, *only very preliminary estimates are presented, which are nor directly based on any implementation.*

We believe that the total detection coverage should be well above 90%, as individual techniques have been shown to have this efficiency, at least for certain applications, [Wilken and Shen, 1990], [Rimén, 1995], [Hiller, 2000].

With the suggested approach, the recurring cost overhead should be low as no additional hardware is used. Moreover, as no additional hardware is required and no

diversity is used, the solutions should be rather independent of the chosen hardware platform (i.e., portable to any microprocessor). Furthermore, the power consumption should be about the same as for the application tasks, without any applied fault tolerance techniques, but the energy consumption will increase as the workload is increased.

The memory overhead is expected to be around $60 - 100\%$ as the "I'm alive" messages overhead should be negligible, signature checking around $10 - 20\%$, watchdog timers around $5 - 10\%$, executable assertions around $20 - 30\%$, self-tests around $20 - 30\%$, and double execution around $5 - 10\%$ (assuming that the task only is stored in one place of the memory).

The performance overhead is predicted to $150 - 210\%$, as the "I'm alive" messages overhead should be around $5 - 10\%$, signature checking around $10 - 20\%$, watchdog timers around $5 - 10\%$, executable assertions around $10 - 20\%$, self-tests around $20 - 30\%$, and double execution around $100 - 120\%$.

It should be noted that the accuracy of these predictions may be low. Therefore, especially for estimating the detection coverage for the different types of errors, the solution need to be evaluated. How to perform such evaluation will be discussed in the next section.

3.6 Evaluation of Detection Coverage

Traditionally, investigations of the dependability of systems has been based on listing a number of faults and estimating their effect through analysis, see Chapter 1.2, or fault injection experiments, see [Cunha *et al.*, 2001], [Vinter *et al.*, 2001], Chapter 1.2.1. These methods require that the injected faults capture the behavior of all real faults that can occur (i.e., that a correct fault model is used).

For faults generating crash failures and control-flow errors, the effects are rather similar and it is rather easy to accurately model faults. However, as the impact of different data errors can have large variations, it is much harder to find a fault model that captures the behavior of real faults. For commercial microprocessors which are very complex it is very difficult to identify all possible fault cases, and thus, suitable fault models. Furthermore, the low-level details of such processors are seldom available for developers of safety-critical systems. Therefore, these bottom-up methods are not flawless. The implications of modeling faults at different abstraction level is shown in Figure 3.2.

In the continuation of this thesis we present a new methodology that could be used as a complement to the traditional evaluation methods. Instead of looking at which effects faults have, i.e., which errors they generate, the characteristics of data

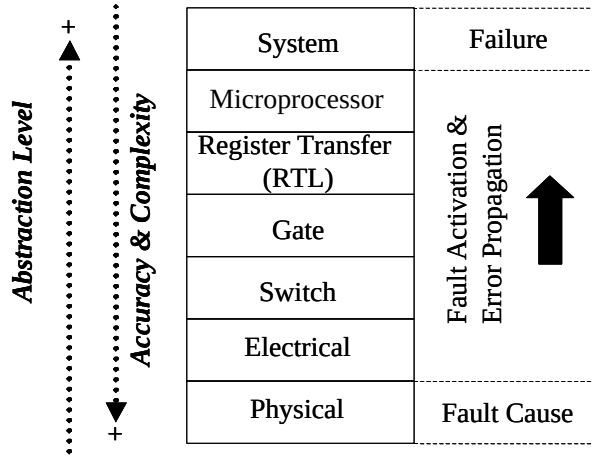


Figure 3.2: Traditional fault modeling dependency on abstraction level.

errors that determines their impacts are identified. Then these characteristics are varied and the effects on the system are observed. This makes it possible to use top-down approaches for determining, at high abstraction levels, the effect of any data error on the system, e.g., determine which errors the system is most sensitive to. After that, it can be determined which faults (if any) that result in severe errors and fault tolerance techniques can applied to handle these. We believe that the methodology has the following advantages:

- The risk of missing certain fault cases is decreased.
- The first analysis can be performed at high level.
- Fault tolerance techniques can be focused to the most damaging faults.
- The same approach can be used for estimating the efficiency of applied fault tolerance techniques.

A similar methodology aimed at profiling the sensitivity of different software modules, was proposed in [Hiller, 2000], [Hiller *et al.*, 2001], [Hiller *et al.*, 2002]. However, as it was mainly focused on the profiling methodology, it did not in detail investigate which characteristics that influence the effects of data errors on the system (bit-flips was injected with PROPANE, a software-implemented fault injection tool).

Also [Kim and Shin, 1994] and [Kim *et al.*, 2000] uses a similar approach for investigating the effect of data errors on control systems. However, here the characteristics are randomized chosen in a limited interval rather than varied (which we propose) and only the effect on the system stability is evaluated.

In the continuation of this chapter, the characteristics of data errors are identified. Then, in Chapter 4 and 5, different approaches using this methodology is presented.

3.7 Identification of the Characteristics of Data Errors

As previously defined, a data error has occurred when a property (e.g., a signal or variable) has assumed an incorrect value, i.e., incorrect data has been produced. As seen from the discussion in this chapter, whether a fault is activated (i.e., an error is generated) or not at a certain abstraction level depend on:

- **Influence:** How much the properties are changed.
- **Occurrence Time:** At what time did the fault occur (in which state the component/system is and the value of the input data).
- **Duration:** How long time the fault persist.
- **Location:** The function of the faulty component.

These quantities also determines the characteristics of the data error. The fault influence and the fault duration determine the probability of an error being generated, which determines how often errors are generated. The occurrence time determines system state when the error occurs, which also influences its effect.

Continuing, the fault duration determines how often errors are generated. For instance, if the fault is transient, only one error may be generated at one point in time. However, if the fault has longer duration, several errors may be generated with or without even intervals.

The location of the fault determines which part of the system that is affected, but also how the error propagates, divides (several errors can be generated) and manifests.

Therefore, the effects of a data errors depend on a limited number of characteristics, namely:

- **Property:** Which property that is erroneous.
- **Number:** The number of properties that are erroneous.

- **Magnitude:** The difference between the correct value and the erroneous value of a certain property.
- **State:** The occurrence time of the error.
- **Repetition:** How often the error is repeated.

To summarize, by varying the above defined error characteristics it should be possible to mimic the effect of all data errors, independently on which fault that generated them. The problem with this methodology is that it is not practical possible to explore the effect of all possible combinations of the characteristics. Thus, the characteristic combinations to explore must be intelligently chosen so that the evaluations will be accurate. Different approaches for accomplishing this will be discussed in the continuation of the thesis.

In Chapter 4 analysis methods for estimating the the effect of data errors on control systems are presented. Here data errors with different characteristics are classified as different types of disturbances, which makes it possible to apply traditional control theory methods for estimating there effects.

As the analysis can not determine the effect of all types of errors, a simulation-based fault injection technique is developed in Chapter 5 to estimate the effect of these errors. As low-level details of microprocessors are seldom accessible for system developers, the simulations must be performed at a high-level. Usually, high-level simulations imply less accurate results, but by modeling the error characteristics derived above, the most important information can be gained.

CHAPTER 4

Analyzing the Effect of Data Errors in Control Systems

As was discussed in the previous chapter, evaluations of fault tolerance techniques are a major problem for system developers and researchers as few low-level component (e.g., microprocessor) details are accessible. In this chapter analysis methods for understanding the effects of data errors on linear control systems are developed. These analysis methods can be used to determine which fault tolerance techniques that need to be added, but also to estimate the efficiency of system dependent error detection techniques.

First, in this chapter, suitable failure criteria for control systems is discussed based on related work. It was decided to use failure criteria based on the control error, that is the difference between the desired and actual value of a physical property. Then, the linear control system model is defined. After that, a top-down modeling approach and suitable models for data errors are presented. The approach is based on modeling data errors with varied characteristics of the signals (the control and state signals) affecting the physical process. This differentiates from most previous approaches as they generally have been based on injecting faults at low levels, i.e., bottom-up approaches.

The analysis methods consist of:

- **Sensitivity Analysis:** Determines which error repetition frequencies the system is most sensitive to.
- **Impulse Response Analysis:** Describes the effect of occasional errors for

instance caused by transient faults.

- **Norm Analysis:** Determines how much the physical property is affected by a certain error magnitude.
- **Step Response Analysis:** Describes the effect of errors that are repeated several samples in a row, caused by persistent faults.
- **White Noise Analysis:** Describes the effect of errors that occur randomly, for instance caused by persistent faults that are activated seldomly.

These analysis methods are exemplified through a simple model of a brake-slip controller. It is also shown how the analysis can be used for designing executable assertions and to determine their efficiencies.

To be able to use the analysis, models of the controller and the physical process must exist. Such are often developed for designing the controller.

The major advantages with the new approach are that it can be applied early in top-down system development processes and that, since the models are linear, the effects of errors with different characteristics can be found from simple computations. The problems with the approach is that it is hard to model some errors caused by persistent faults. Another problem is to determine which characteristics errors caused by real faults will have, i.e., which error characteristics that are of interest to analyze. For the errors that are hard to model, simulation is suggested, either in the same signals (Chapter 4.4.5) for instance with *MATLAB* simulations or through VHDL simulations (Chapter 5). The problem of analyzing realistic error characteristics is discussed in Chapter 5.

These analysis methods do not cover complete evaluations of the effect of data errors in control systems, but we believe that it can be a useful complementing tool to traditional fault injection.

Readers that would like more details about the analysis methods and how they are used at control design are referred to [Åström and Wittenmark, 1997].

4.1 Related Work and Definition of Failure Criteria

Understanding the effect of data errors on computer functionality is an intensively researched area, e.g., [Pradhan, 1996]. As defined in the introduction of this thesis, efficient fault tolerance implies that resources should be focused on those faults that will have severe impacts on the system. Therefore, it is desirable to be able to determine the impacts of different faults when designing safety-critical systems.

For control systems, recent results [Cunha *et al.*, 2001], [Vinter *et al.*, 2001] show that many data errors will have a limited effect on control performance, i.e., control systems often have an inherent resilience or inertia to data errors. The results in [Cunha *et al.*, 2001], [Vinter *et al.*, 2001] were obtained experimentally using fault-injection (fault-injection is described in greater detail in Chapter 1.2.1) a technique suitable for validation purposes. However, this technique requires a prototype of the system (or at least a detailed model), which generally is not available in the early design phases. Thus, methods for estimating the effects of data errors earlier in the design process are preferred.

Methods for analyzing the effects of errors have previously been developed. For instance, [Kim and Shin, 1994], [Wittenmark *et al.*, 1995] have investigated the effect of timing errors on control systems. Analysis of the effects of data errors caused by EMI bursts on system stability was investigated in [Kim *et al.*, 2000]. These data errors were modeled as a time-invariant Poisson process with random magnitudes bounded within a certain interval. However, data errors may occur of other reasons than EMI bursts, and then, have other characteristics (distributions). Moreover catastrophic failures in certain safety-critical systems may occur before the system reaches instability. For some industry processes, nuclear power plants, and space ship stabilization, etc., stability may be a valid failure criteria. However, for tracking systems, i.e., systems where the reference signal is changing frequently, failure may occur even if the stability is not lost.

As an example, take the steering of an automobile. If the driver turns the steering wheel 15 degrees to the right, but it is interpreted as a 15 degree turn to the left, for instance due to a bit-flip in the sign bit, a catastrophic situation may occur, even though the erroneous value is in the valid domain of the angle sensor. Thus, the severity of errors is for some systems more related to how much the reference (desired) value of a controlled physical process property differ from the actual value of this property, i.e., the control error of the system (e.g., in a system controlling the steering of a car, the control error would be the difference between the steering angle set by the driver through the steering wheel and the actual steering angle of the front wheels of the car). Therefore, for such systems, we instead suggest to use the following (application dependent) requirements on the control error:

- **C1:** The control error must not exceed a certain specified limit.
- **C2:** A control error above a certain level is only acceptable (from a control perspective) for a limited duration.

If any of these two requirements is violated, we say that the system has failed. Similar criterion as **C1** was used in [Cunha *et al.*, 2001], [Vinter *et al.*, 2001].

Note that a control error may be non-zero even if the control system is fault free. A change in reference (desired) value may generate a control error, as well as external disturbances (e.g., for the steering system, deep tracks in the road). Thus, the impact of a fault on the system is not only dependent on the specific fault, but also the current operational state (the current operational state is called the operating point) of the system, e.g., for the steering example, whether the driver drives straight ahead or turns sharply.

As most systems have a certain inherent inertia, the control error is dependent on i) the dynamics of the system, ii) how the reference value changes, and iii) external disturbances. Thus, the effect of a fault on the system is not only dependent on the specific fault, but also the current operating point of the system. We assume that the developer has identified the most sensitive operating point and set the requirements on the control error for that point.

4.2 The Controller

In this section, the models for a generic controller structure implementing a control function at a level detailed enough to communicate the ideas presented in this chapter, are described.

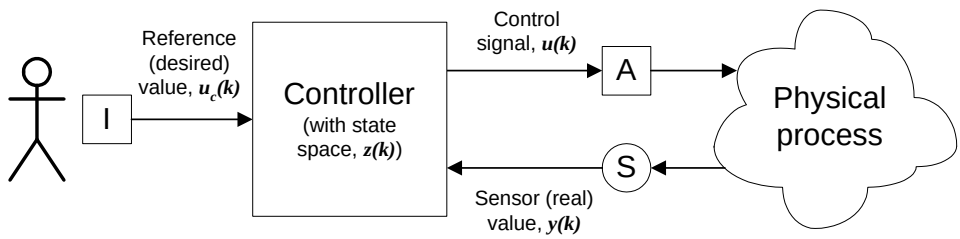


Figure 4.1: General figure of a controller and the corresponding physical process.

As was described in the introduction (Chapter 1), conceptually, a control system is set to control a certain physical process (see Figure 4.1). This is achieved using a set of actuators (**A** in Figure 4.1) for affecting the physical process and sensors (**S** in Figure 4.1) for monitoring the effects of the actuators. The user of the controller (which may be a human user or an external computer system) provides the controller with a reference signal $u_c(k)$ via some interface (**I** in Figure 4.1). The controller will then attempt to change the physical process, with control signal $u(k)$ to the actuators, such that the sensor value $y(k)$ is as close to the reference value as possible. As

controllers implemented on microprocessors are considered, we assume that it executes in discrete steps, i.e., the controller reads reference values and sensor values and calculates control signals periodically. In Figure 4.1, and subsequently, k will indicate the k^{th} step in time.

Typically, a control system has non-linear effects (a non-proportional relationship between the value of the control signal and the resulting physical value read by the sensors). However, linear models are in many cases sufficient approximations at the normal system operating point and reduce the complexity. Therefore, in this chapter, the error effect analysis are focused on linear system models. The generalized feedback controller in Figure 4.1 is defined as:

$$\begin{aligned} u(k) &= Cz(k) + D^{u_c}u_c(k) + D^{y}y(k) \\ z(k+1) &= \Phi z(k) + \Gamma^{u_c}u_c(k) + \Gamma^{y}y(k) \end{aligned} \tag{4.1}$$

The equations in Eq. (4.1) provide a generic linear controller with internal states, e.g., a PI-controller (we have a controller with a proportional feedback (P) and an integrated feedback (I)) or a state-feedback controller with observer states and integral action. The control signal $u(k)$ is calculated using the reference value $u_c(k)$, the sensor value $y(k)$, and an intermediate value $z(k)$. This intermediate value is actually a set of various values constituting the state space (history) of the controller. The state space can for instance contain integrator states, which are used to provide a certain level of history that guarantee that the correct output is reached despite constant disturbances, and observer states, which are used to estimate signals which influence the control algorithm, but may be impossible to measure. In the example in Chapter 4.4 one integrator state and one observer state is used.

Φ, Γ^{u_c}, Γ^{y}, C, D^{u_c} and D^{y} are matrices containing control constants, used as weights in the equations. The control constants are amplifying factors that are set by the designer to give the desired control performance, e.g., making the physical value assume the reference value as fast as possible (after a change of the reference value) without exceeding the reference value (i.e., overshooting).

Now the general structure of the controller has been introduced. In the next section, how data errors can be described with disturbance models used in control theory is discussed. The motivation for this is that if data errors can be described as disturbances, disturbance analysis can be used to estimate the effect of errors.

4.3 Modeling of Data Errors

In Chapter 3.7 the characteristics that determine the effect of an data error were identified to be:

- Which property that is erroneous (*Property*).
- The number of properties that are erroneous (*Number*).
- The difference between the correct value and the erroneous value of a certain property (*Magnitude*).
- The occurrence time (*State*).
- How often the error is repeated (*Repetition*).

Thus, error models used for estimating the impact of errors must accurately consider these characteristics. Based on the assumptions and restrictions made in Chapter 4.2, error models, using the disturbance mathematical framework from control theory, considering these characteristics will now be defined.

As discussed in the previous section, the controller can only affect the physical process through the computed control signal (see Eq. (4.1)). Therefore, for a computer node fault to have an impact on the system, they must affect the calculation of this signal (directly or through propagation). Thus, the data errors (disturbances) disrupting the calculated control signal (u) and/or the state space of the controller (z) can be modeled as additive terms[1] as follows:

$$\begin{aligned} u(k) &= Cz(k) + D^{u_c}u_c(k) + D^{y}y(k) + \boldsymbol{\eta_u(k)} \\ z(k+1) &= \Phi z(k) + \Gamma^{u_c}u_c(k) + \Gamma^{y}y(k) + \boldsymbol{\eta_z(k)} \end{aligned} \tag{4.2}$$

where $\boldsymbol{\eta_u(k)}$ and $\boldsymbol{\eta_z(k)}$ are the functions describing disturbances due to faults in the computer (i.e., errors) executing the control equations. Thus, the *properties* that are considered are the control and the state signals. As these are the signals of the control algorithm, this can be seen as a top-down approach compared to traditional fault injection where faults generally are injected at low levels.

As linear systems are considered, the impact of an error is not dependent directly on the *state* of the system. This means that the impact can be computed for any *state*. However, in order to determine the worst case, the most sensitive operating point must have been identified. The total impact of several errors (i.e., the *number* characteristic) can, due to the linearity, be found by superimposing the impacts of each individual error. Thus, the analysis can be limited to study the individual errors (these will be discussed more in Chapter 4.4.2).

[1]The reason to why the errors are modeled with additive terms is that the magnitude of the errors are bounded to the word-length for representing the control signal. This implies that, which will be seen in Chapter4.3.2, any error magnitude can be modeled with additive terms.

Now, the expressions for $\eta_u(k)$ and $\eta_z(k)$ to fit errors with different *repetition* frequencies are defined. Then, it will be described how to model the *magnitude* of the errors.

4.3.1 Repetition Frequency Classes for Data Errors

In this subsection, the data errors are divided into classes based on the rate of occurrence, i.e., the *repetition* frequency.

Class A: Persistent Errors This class consists of errors which will affect almost all calculated samples (from when the error first occurs and then subsequent samples). If the errors are caused by faults in the memory storing the control constants, they can substantially change the behavior/structure of the system (Disturbances with this effect are denoted structural in control theory. Tools for more detailed analysis of structural disturbances are provided by robust control theory, see e.g., [Zhou and Doyle, 1998]). If the errors have similar magnitude at longer time intervals, for instance caused by faults in memory cells storing the most significant bits of variables (which change their value infrequently), they can be modeled by the step-function[2], $\theta(k-s)$. In this chapter this type of persistent errors is mainly handled. However, Chapter 4.4.5 briefly treats other types of persistent errors.

Class B: Sporadic Errors This class includes errors that occur so infrequently that the system has time to return to a correct state before the next error occurs, i.e. the effects of errors are not superimposed. Thus, these errors can be modeled as temporary impulse disturbances, described with the pulse-function[3], $\delta(k-s)$. Such errors are for instance caused by transient faults, occurring with low intensity, and persistent faults in the functional units (adder, multiplier, etc.) of the microprocessor that are activated occasionally and propagated with low probabilities.

Class C: Frequent Errors This class consists of errors not covered by the two previous classes, i.e., errors that do not affect every sample, but yet many enough for their effects on the system to be superimposed. The effects of these errors can be modeled as stochastic processes, i.e., a function that assumes random values and whose properties are described by its mean and variance, which either will assume 0 (when no error affects the system) or the magnitude, m, of the error (when an error does affect the system), see Chapter 4.3.2. Such errors could be caused by transient faults occurring with medium intensity, memory faults in the least significant bits of variables (which change value relatively often) and persistent faults in the functional

[2] $\theta(k-s) = 0$ for $k < s$ and 1 for $k \geq s$, where s denotes the point in discrete time where the error occurs.

[3] $\delta(k-s) = 1$ for $k = s$, else 0, where s denotes the point in discrete time where the error occurs.

units, that have medium high propagation probabilities.

4.3.2 Data Formats and Error Magnitudes

The error magnitude will be dependent on the formats used for representation of data in the calculations performed by controller. Two commonly used formats to represent numerical data are floating-point and fixed-point values. Floating-point values give better accuracy than fixed-point values for a given number of bits, but require either floating-point units (i.e., more expensive microprocessors) or additional software routines (adding to the total execution time of the calculations). We will now define the error magnitudes that can occur for the different formats.

In the IEEE floating-point standard [IEEE, 1985], numbers are represented with a sign bit, s, a fraction part with value f, (23 bits for the single precision format) and an exponent part with value e, (8 bits for the single precision format). A decimal number a is represented in the single precision format as:

$$a = (-1)^s(1.f)2^{(e-127)} \tag{4.3}$$

The range of representable values[4] is then for the single precision format $R = [-(2-2^{-23})\cdot 2^{128}, (2-2^{-23})\cdot 2^{128}]$, with a resolution (minimum difference between two non-identical numerical values) of $Q = 2^{-23}$. With this format, the magnitude of a bit error is dependent on the values of the other bits, but if it occurs in the most significant bits of the exponent, it will result in very high error magnitudes, see Table 4.1. Even if the physical limitations of actuators, sensors, etc., limit the immediate effects of such errors, the state update (see Eq. (4.1)) can be seriously perturbed. This problem is addressed in [Vinter *et al.*, 2001] by adding executable assertions that check that data do not exceed their specified limits.

Table 4.1: Example of error magnitudes for single-bit errors in different data formats.

Erroneous bit	*Error magnitude, floating-point, single format (e_* is the value of the remaining exponent bits)*	*Error magnitude, fixed-point integer, N=32, M=23*
0	$2^{-23} \cdot 2^{(e-127)}$	2^{-23}
16	$2^{-7} \cdot 2^{(e-127)}$	2^{-7}
31	$(1.f) \cdot (2^{(e_*+1)} - 2^{(e_*-127)})$	2^{8}

[4]Often some values are used to represent special entities, e.g., ∞, reducing the usable range.

In the fixed-point format [Hanselmann, 1987], numbers are represented as two-complement integers, with a total of N bits, of which $M < N$ bits are used as fractional bits. Hence, a decimal number a is represented by the bit sequence $a_{N-1} \ldots a_0$, such that

$$a = 2^{-M} \left(-a_{N-1} 2^{N-1} + \sum_{i=0}^{N-2} a_i 2^i \right) \tag{4.4}$$

The $(N-1)th$ bit carries the sign information. The range of representable values is then $R = [-2^{N-M-1}, 2^{N-M-1} - 2^{-M}]$, with a resolution of $Q = 2^{-M}$.

With this representation, the magnitudes of the errors will be in the same range as the control signals (assuming that the data has been properly scaled). Thus, a single bit error in bit $n \in [0, N-2]$ will have the magnitude:

$$m = (-1)^{a_n} 2^{n-M} \tag{4.5}$$

where a_n is the correct bit value. A bit error in the sign bit $n = N-1$ will have the magnitude:

$$m = -(-1)^{a_{N-1}} 2^{N-M-1} \tag{4.6}$$

Some examples of magnitudes for single-bit errors are given in Table 4.1. Note that burst errors, affecting multiple bits, will correspondingly have magnitudes being the sum of the individual bit error magnitudes.

Fixed-point values will be used in the continuation of this chapter, but as only the range of magnitudes differ between the two formats, the proposed analysis methods are valid also for floating-point values.

4.3.3 Error Effects on the Generalized Controller

Looking at the errors described in previous subsections, expressions for modeling the errors effects on the generalized controller of Eq. (4.1) are now defined. The expressions are summarized in Table 4.2, based on the discussions in Chapter 4.3.1 and Chapter 4.3.2 The first column defines the error class, the second column is the type of the disturbance that the error will cause and the third column shows the mathematical expressions describing the disturbances, η, for modeling the error in Eq. (4.2). Here, $m_A(k)$ is a piecewise constant error magnitude, $m_B(s)$ the error

magnitude at time s, $m_C(k)$ a stochastic process describing the error magnitude when a frequent error occurs, θ the step-function, δ the impulse function and s the point in discrete time when the the first error occurs (i.e., the error occurs in a specific sample).

Table 4.2: Effects of errors occurring at time s.

Error class	*Nature of resulting disturbance*	*η: Mathematical expression for disturbance*
A:Persistent	Step	$m_A(k)\theta(k-s)$
B:Sporadic	Impulse	$m_B(s)\delta(k-s)$
C:Frequent	Stochastic Process	$m_C(k)\theta(k-s)$

4.4 Analysis Methods

In the previous section the disturbance-models for the effects different errors have on the system were defined. In this section. first, an example of a control system is given. Then it is shown how different analysis methods can be used to understand the effect of errors on control systems, using the disturbance-models (Chapter 4.3.3) on the example.

Example — Brake-Slip Controller

A brake-slip controller is used to control the wheel-slip λ on a car during braking. This is used to avoid situations where the car looses its grip of the road and starts skidding. The sampling time is set to $t_h = 0.01$ s. The brake-slip controller normally operates in the region $0 < \lambda < 0.1$, where 0 corresponds to no slip, i.e., no braking, and 0.1 very hard braking where the wheels are almost locked. It is essential that the brake force on the left and right side of the car is balanced, otherwise there will be a resulting torque on the vehicle, and the car will turn during braking, which may lead to a hazardous situation.

The process can be described as a linear discrete-time system:

$$\lambda(k) = \frac{B(q)}{A(q)}u(k) \tag{4.7}$$

where the polynomials B and A are of degree 1 and 2, respectively. The generalized controller in Eq. (4.1) were used and instantiated according to traditional

control design for achieving desired control performance (for design details, see Appendix D.2). This resulted in the following values of the control constants:

$$\begin{aligned} \Phi &= \begin{pmatrix} 1 & 0 \\ 0 & 0.2534 \end{pmatrix}, \qquad \Gamma^{u_c} = \begin{pmatrix} 0.1237 \\ 0.01378 \end{pmatrix}, \\ \Gamma^{y} &= \begin{pmatrix} -0.1237 \\ 0.7328 \end{pmatrix}, \qquad C = \begin{pmatrix} 63.55 & 94.64 \end{pmatrix}, \\ D^{u_c} &= 13.01, \qquad D^{y} = -161.4 \end{aligned} \tag{4.8}$$

The first value in the state space of the controller (i.e., the top values of Φ, Γ^{u_c} and Γ^{y}) is an integrator state and the second value (the bottom values) is an observer state.

To validate that the desired control performance was reached with this instantiation, a case where the driver suddenly applies full brakes, was investigated. At normal driving conditions (dry asphalt road), this manoeuvre corresponds to a slip change from $\lambda = 0$ (no braking) to $\lambda = 0.1$ (full braking). Figure 4.2 shows how the output signal (Figure 4.2(a)) and the control signal (Figure 4.2(b)), are changed during the manoeuvre. Please note that different scales are used between different plots in this and subsequent figures.

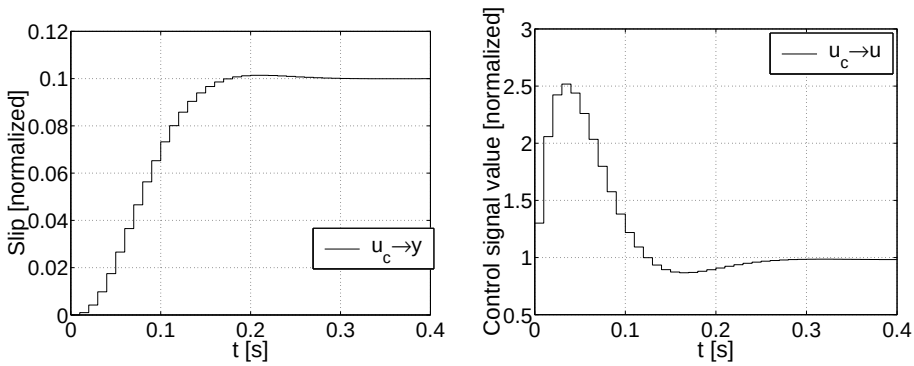
As can be seen in Figure 4.2(a), the output value assumes the reference value within 0.4 s and without any major overshooting (i.e., exceeding the reference value). Looking at the control signal Figure 4.2(b), it first increases quickly to make the output signal assume the desired value (the reference value) as fast as possible. Then it decreases to avoid overshooting and stays constant on the level required to give the desired slip of $\lambda = 0.1$. The maximum required braking was considered to be possible to deliver with the actuators. Therefore, as this control instantiation gives acceptable performance it was used.

To avoid hazardous situations, the system developer must specify the failure criteria **C1** and **C2**, see Chapter 4.1. The requirements should be set from experience of the system and detailed studies. We lack this essential information and as a consequence cannot specify accurate criteria. However, as we would like to illustrate different analysis methods, we simply set an ad-hoc requirement:

- **C1**: The system has failed if the magnitude of the control error exceeds 0.01 when no reference signal changes has taken place for $0.4s$.

The last condition is necessary as else normal reference changes would be defined as failures. For instance, in Figure 4.2(a), the maximum control error is 0.1 (i.e., ten times higher than **C1**) as the desired value is 0.1 at time 0, but the actual value still is 0 at this time. We do not set any requirement on **C2** for this example.

In the continuation of this chapter, Figure 4.2 can be used for comparison of how the system signals are changed due to normal changes of the reference values (braking) and how they are changed due to different types of errors caused by computer node faults (the effects of errors on control system are detailed in the following subsections).



(a) (b)

Figure 4.2: Step-input command-signal response of closed-loop system. The plots have different scales.

To study the effect of bit errors, the data format needs to be set. Assume that the available word-length is $N = 32$ bits. Based on the numerical values in Eq. (4.8) and the signal magnitudes of Figure 4.2, the fixed-point numerical implementation is chosen as $M = 23$, which gives the numerical range $R = [-256.0, 256.0 - Q]$ with a resolution of $Q = 2^{-23} \approx 1.19 \cdot 10^{-7}$.

In the subsequent subsections, it is described how the effects of data errors on control systems can be investigated using different analysis methods, adopted from control theory, and exemplify these using the control system example (i.e., the brake-slip controller) given in this subsection. It should be noted that the emphasis is on describing the analysis methods, not on the results of this particular example.

4.4.1 Sensitivity Analysis

In control theory, the sensitivity function is used to determine how the system is affected by disturbances occurring in a particular signal. It is calculated by determining the impulse response function, $h(k)$, for how changes of that specific signal

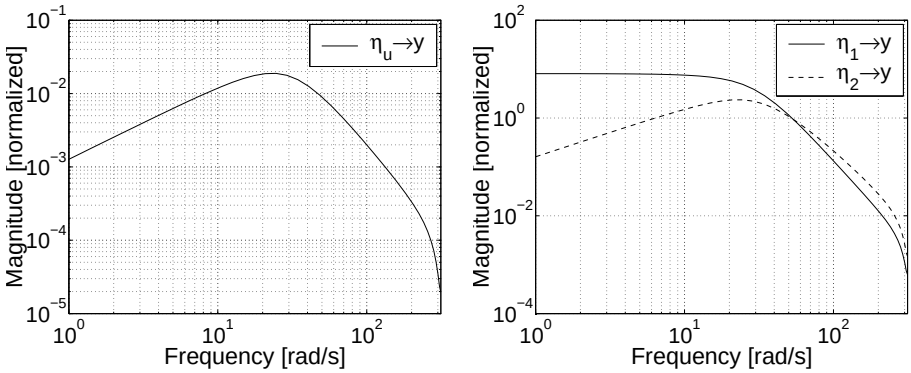
affect the output value. This function is then transformed from the time domain to the frequency domain (to the closed-loop transfer-function, $H(z)$) to determine how much disturbances with different frequencies are amplified (or attenuated). This makes it possible to determine which occurrence rates of disturbances the system is most sensitive to. Thus, for data errors this corresponds to investigating which *repetition* frequencies of errors the system is most sensitive to. For more details about the sensitivity functions, see Appendix D.1.

As shown in Chapter 4.3.3, errors can be described as disturbances in the calculation of the control signal, η_u, and state space, η_z, (Henceforth, disturbances affecting the integrator state are denoted as η_1 and disturbances affecting the observer state as η_2.) see Eq. (4.2). Thus, the effects of different repetition frequencies of errors on the system, can be found by calculating the sensitivity functions for each disturbance.

To exemplify this approach, the sensitivity functions for errors affecting the brake-slip controller example, were calculated. The results are shown in Figure 4.3, where the x-axis shows angular frequency and the y-axis how much harmonic (sine shaped) disturbances (errors) of each angular frequency are amplified. As control signals are sampled, errors will be rectangular shaped, consisting of various frequencies. However, the repetition frequency of the error will be the main frequency component, i.e., cause the largest effects, and thus, analyzing the effect of this frequency will in most cases give a good assumption.

As can be seen, the curve for errors affecting the integrator state (η_1 in Figure 4.3(b)) assumes the highest value, i.e., the system is most sensitive for these errors. The highest value for the sensitivity function of η_1, about 8, is assumed at 0 Hz, which means that the magnitudes of errors affecting multiple consecutive samples, i.e., persistent errors, of the calculation of the integrator state, will be amplified about 8 times and have the largest effect on the system output. As the sensitivity is greater than 0, the errors will result in a steady-state influence on the process output (i.e., the system will never return to the reference value).

For the control signal (η_u in Figure 4.3(a)) and the observer state (η_2 in Figure 4.3(b)) persistent errors (frequency = 0 Hz) are rejected (asymptotically assumes 0, meaning that the effect of the error eventually will decrease to 0). Instead the maximum value for their sensitivity functions are reached at 23.2 rad/s = 3.7 Hz. Thus, for errors affecting the control signal and observer state, the largest slip effect (the error that is amplified most) will be for errors occurring with this frequency.



(a) (b)

Figure 4.3: Sensitivity functions from η_u, η_1 and η_2 to output, y. The plots have different scales.

4.4.2 Impulse Response Analysis

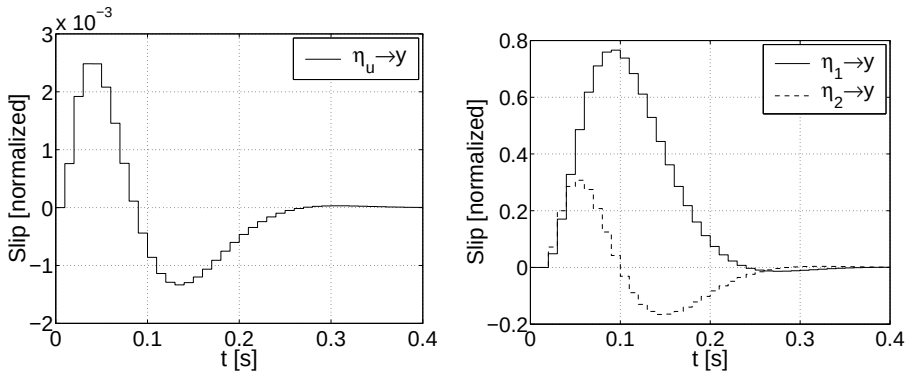
To more accurately estimate the effects of sporadic errors with different magnitudes, one can study how the output value is affected by impulse disturbances occurring in the control algorithm (η_u and η_z in Eq. (4.2)), by calculating the impulse response functions, $h(k)$ (described in Chapter 4.4.1 and in Appendix D.1). These functions show how the output value changes due to impulses (sporadic errors) with magnitude 1, affecting the calculation of the different signals of Eq. (4.2). As linear systems are considered, *the effect on the system will be proportional to the magnitude of the error.* Thus, the effect of an error with magnitude m will be[5] $mh(k)$. Thus, generally errors with large magnitude will affect systems more than errors with low magnitude. However, errors with small magnitudes are generally harder to detect with system-dependent error detection techniques as they differentiates less from correct values.

The impulse response analysis also reveals how fast the output returns to the correct value, due to the inherent robustness of the system (i.e., without adding any error recovery), after an error has occurred. This time can be used to determine how often errors can occur without their effects being superimposed, i.e., it can be used for classifying errors into sporadic errors or frequent errors. It should be noted that

[5]For an error that manifests as an impulse with magnitude m at time $s = 0$, i.e., $\eta = m\delta(k)$, the effect on the process output will be described by $y(k) = \sum_{l=0}^{\infty} h(l)m\delta(k-l) = mh(k)$.

due to the linearity, the time constant is independent of the magnitude of the error. This means that if the response to an error with a certain magnitude has decreased to $x\%$ of its maximum value after a certain time, errors with other magnitudes will also have decreased to $x\%$ of their maximum value, after the same time.

To exemplify this analysis method, the output and control signal responses to sporadic data errors for the example described in Chapter 4.4 were calculated. Figure 4.4 shows how the slip is affected (the nominal value is set to 0) by an impulse (sporadic error) affecting the calculation of the control signal (Figure 4.4(a)) and state space (Figure 4.4(b)). Figure 4.5 shows how the control signal is affected for the same impulse affecting the control signal (Figure 4.5(a)) and state space (Figure 4.5(b)).



(a) (b)

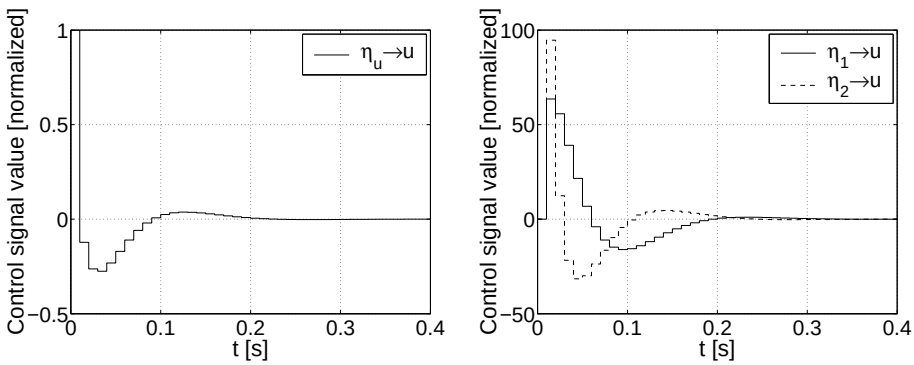
Figure 4.4: Output signal responses to sporadic data errors. The plots have different scales.

It can be seen that sporadic errors have larger effect when affecting the states compared to the control signal (i.e., the slip diverges more from its nominal value of 0 in Figure 4.4(b) compared to Figure 4.4(a). This is due to the large scaling of the state variables when calculating the control signal (matrix C in Eq. (4.8)) and that the errors will be stored in the states of the controller (the old values of the state z in Eq. (4.2) are used for calculating the new state and the control signal), and thus, affect several samples, whereas control signal errors will only directly affect one sample.

Furthermore, it can be seen that the integrator state is more sensitive to sporadic errors than the observer state (the magnitude is higher for the solid line $\eta_1(k)$in

Figure 4.4(b) than for the dashed line $\eta_2(k)$). However, looking at Figure 4.5(b) it can be seen that errors in the observer state generate control signals with higher magnitudes (the dashed line has a higher maximum magnitude than the solid line). This implies that if an assertion would be placed on the control signal (which seems natural and has been suggested for instance in [Vinter *et al.*, 2001]), it would either (dependent on how the threshold is set) let unacceptable integrator errors through (that would violate the failure criteria) or unnecessarily and even incorrectly (i.e., cause false alarms) detect observer errors.

It can also be seen from Figure 4.4 that the system returns to the nominal slip, 0, about 0.4 s (in this case 40 samples, $t_h = 0.01$ s) after the error occurred. This means that if errors do not occur more frequently than this, the effects of them will not be superimposed.



(a) (b)

Figure 4.5: Control signal responses to sporadic data errors. The plots have different scales.

4.4.3 Norm Analysis

As discussed in Chapter 4.1 we base our failure criteria on magnitude and duration of the control error. Thus, to get numerically comparable values, the following norms are useful:

$$\|h\|_1 \triangleq \sum_k |h(k)| \qquad \text{and} \qquad \|h\|_\infty \triangleq \sup_k |h(k)| \tag{4.9}$$

where sup denotes the supremum function. The norm $\|h\|_\infty$ is the maximum absolute value of the error resulting from the disturbance. The norm $\|h\|_1$ is the sum of the absolute control errors over time. The essential difference is that $\|h\|_\infty$ only gives information on the largest size of the error. When combined with $\|h\|_1$, information on the duration of the error is also assumed. The choice of controller state-realization (i.e., which values for the control constants of Eq. (4.1) that are chosen) will affect the size of $\|h\|$. In the example described in Chapter 4.4, all state realizations are scaled such that a step-input command-signal results in state-variables with unit stationary value.

The maximum single bit error, n, which do not violate the failure criterion **C1** can thus be found through:

$$n < n_{lim\infty} := M + \log_2(\mathbf{C1}/\|h\|_\infty) \tag{4.10}$$

where M is the number of fractional bits used in the fixed-point representation. It follows that multiple bit errors satisfying $n < n_{lim} - 1$ do not lead to failure. Similarly, $\|h\|_1$ can be used to determine when **C2** is violated. However, to avoid that small control errors, having no real effect on the system, eventually sum up and violate **C2**, control errors below a certain level should be neglected, and the norm reset.

The impulse-response norms for the example described in Chapter 4.4 are shown in Table 4.3. The largest bit for which a single-bit or multiple-bit error does not result in failure (i.e., the control error exceeds 0.01, see Chapter 4.4), according to Eq. (4.10), is shown in Table 4.4.

Table 4.3: Impulse-response norms.

Norms	$h_u(k)$	$h_1(k)$	$h_2(k)$
$\|h\|_1$	2.45e-02	8.28e+00	3.08e+00
$\|h\|_\infty$	2.48e-03	7.66e-01	3.07e-01

4.4.4 Step Response Analysis

Persistent errors change the dynamics of the closed-loop system, and thus, control performance is generally substantially affected. Therefore, most of these errors need to be handled, and thus, the analysis method in this section is focused on how fast errors need to be detected in order for the system to recover before it fails.

Table 4.4: Largest bit-number for which a single or multiple bit sporadic error is tolerated.

	η_u	η_1	η_2
Single bit error	25	16	18
Multiple bit errors	24	15	17

In many cases, errors occurring several samples in a row with high constant magnitude (i.e., $m_A(k) = m_A$ in Table 4.2), are some of the most harmful errors, i.e., the errors that in shortest time will result in failures (which error repetition frequencies that result in the largest impact on the system can be identified with sensitivity analysis, see Chapter 4.4.1). Such errors will, at least for the first samples, have similar effects as step disturbances (the step function, θ, is described in Chapter 4.3.1). Thus, the system response to step disturbances can be studied to find when (after how many samples) the failure limits are reached for different error magnitudes[6].

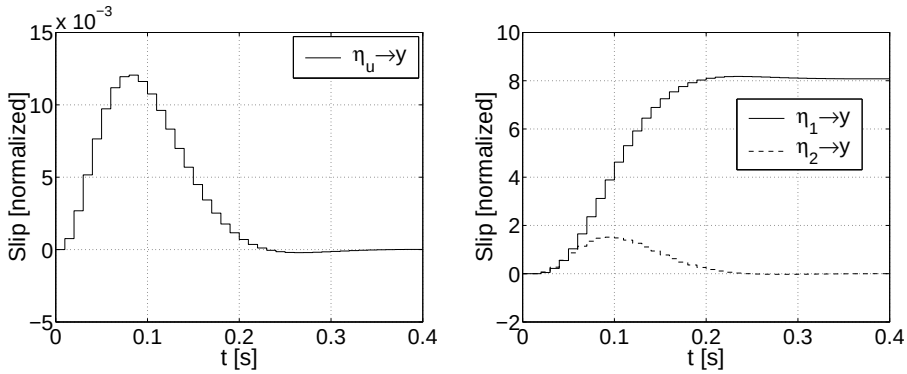
To exemplify this approach the step response functions for the brake-slip controller described in Chapter 4.4 were calculated. The result is shown in Figure 4.6. For persistent errors with constant magnitude affecting the calculation of the control signal (Figure 4.6(a)) or the observer state (the dashed line, $\eta_2(k)$, in Figure 4.6(b)) of Eq. (4.2), the slip returns to the nominal value 0 within 0.4 s. This is due to the fact that the controller uses an integrator state designed to mask the effect of constant disturbances. However, when such errors affect the integrator state directly (the solid line in Figure 4.6(b)), the slip will never return to its nominal value.

As previously discussed in Chapter 4.4.2, the time constant is independent of the error magnitude. Using this, the number of samples between the fault occurrence and until the **C1** or **C2** criteria are violated, can be used as a measure of the required system recovery time. As an example, consider an error occurring in the integrator state (the solid line in Figure 4.6(b)). Closer inspection of the plot shows that after 0.02 s (two samples) the control error reaches from 0 to 0.0483 for an error with magnitude[7] 1, which is well above **C1**$= 0.01$ (**C1** was specified in Chapter 4.4). This means that recovery must have been initiated before 0.02 s after occurrence of

[6]For a fault that manifests as a step with magnitude m on state z occurring at time $t = 0$, i.e., $\eta_z = m\theta(k)$, the effect on the process output will be described by $y(k) = \sum_{l=0}^{\infty} h_z(l) m\theta(k-l) = m\sum_{l=0}^{k} h_z(l)$. Using this equation the first sample (if any), k, at which the requirements are no longer fulfilled can be found. That is, the time within which the system needs to be recovered to avoid system failures.

[7]For an error occurring in the most significant bit (i.e., with magnitude $2^{31} \cdot 2^{-23} = 2^8$), the control error will be (due to the linearity): $0.0483 \cdot 2^8 \approx 12.4$, i.e., much higher.

the error.



(a) (b)

Figure 4.6: Output responses to step fault disturbances. The plots have different scales.

4.4.5 White Noise Response Analysis

For determining the effects of frequent errors we propose two different approaches, which are not exemplified. For errors which can be described as uncorrelated stochastic processes (also termed white noise), with variance σ^2 (for instance errors caused by faults in memory cells storing low significant bits of variables), a requirement on the maximum tolerated variance on the slip, can be specified. To determine which errors violate this requirement, the resulting variance[8] of errors with different occurrence frequencies can be calculated.

The second approach to estimate the effects of frequent errors is to run simulations. As we are looking at linear systems, the effects of several errors occurring closely in time (i.e., another error occurs before the system has returned to the correct state after the previous error) can be simulated by superimposing the effect of each error. Thus, by adding the error as an additional input and defining which samples it is active and the magnitude at each specific active sample, the effect should

[8]The variance can be calculated through: $E[y^2] = E[\left(\sum_l h(l)\eta(k-l)\right)\left(\sum_j h(j)\eta(k-j)\right)] = \sum_l h(l)^2\sigma^2 = \|h\|_2^2\sigma^2$, where η determines which calculation of the control algorithm Eq. (4.2) that is affected and $\|h\|_2^2 \triangleq \sum_k h(k)^2$.

be possible to find. In order to reduce the number of error cases to simulate, the linearity can be utilized as if a specific magnitude exceeds the failure criteria, all higher magnitudes will also violate the criteria. Furthermore, as the system in this case was most sensitive to persistent errors in the integrator state, it can be expected that if a certain error repetition frequency exceeds the failure criteria, all higher repetition frequencies will also violate the criteria.

4.5 Design of Executable Assertions

In the previous part of this chapter, it has been shown how different analysis methods can be used to estimate the effect of data errors on control systems. Now, in this section, it is shown how these methods can be used for designing and estimating the efficiency of executable assertions, a systematic error detection technique (see, Appendix A).

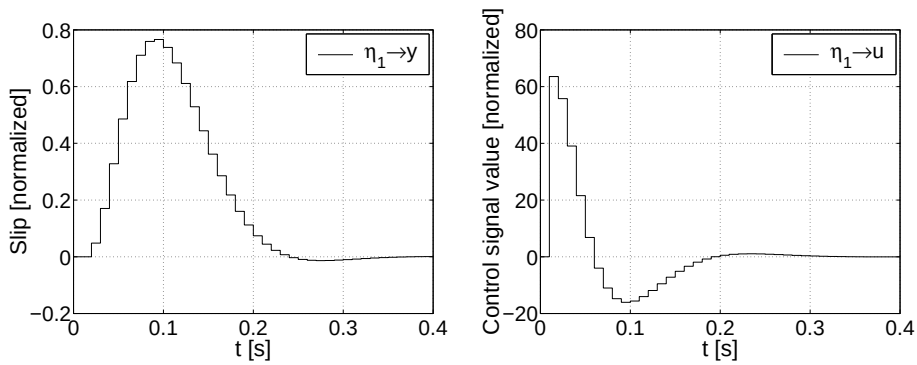
As was defined in Chapter 3.4.3, it is desirable to design assertions that 1) will reduce the impact of data errors, and 2) determine when the controller cannot handle an error, i.e., an unrecoverable error has occurred and the system must be set in a fail-safe state or the control handed over to a spare computer. It is now exemplified how such design can be performed for the brake-slip controller example, using the previous described analysis methods.

First, in order to reduce the impacts of data errors, we apply the anti-windup technique suggested in [Gäfvert *et al.*, 2003] (this technique can be seen as an executable assertion with automatic recovery), which sets the bounds for the control signal as: $|u(k)| \leq M \sum_{j=0}^{k} |h_c(j)|$, where M is the maximal absolute value of the reference signal, $u_c(k)$, and h_c is the impulse response function sequence of the closed-loop system from the reference signal to the control signal. According to the specification for the brake-slip controller, the reference signal (the slip), is bounded as $0 \leq \lambda \leq 0.1 = M$, and computation gave $\sum_{j=0}^{k} |h_c(j)| \approx 42.91$. Thus, the anti-windup observer is designed to be activated for $|u(k)| > 4.3$, i.e., it will reduce the impact of all errors resulting in a control signal higher than 4.3. As linear systems are used, the maximum control error due to a sporadic data error will be an error resulting in a control signal value change of 4.3.

To determine the maximum control error due to a sporadic error, the impulse response function from the integrator state signal (integrator state errors were identified to be the worst errors in Chapter 4.4) to the process output, $h_u(k)$, is computed, Figure 4.7(a), and the corresponding control signal, Figure 4.7(b). From closer inspection, the maximum control error is about 0.77 and the maximum generated control signal was 63.55. As the maximum accepted control signal was 4.3, the maxi-

mum control error due to a sporadic data error will occur when an integrator error results in a control signal just below 4.3. This will happen for data errors with magnitudes $4.3/63.55 \approx 0.07$. The resulting control error will be $0.07 * 0.77 \approx 0.05$. This can be compared with the maximum control error if no restrictions on the signals are used which would be $256 * 0.77 = 197.12$.

It can be seen that the maximum control error exceeds the failure criterion, **C1**, which we set in Chapter 4.4. However this criterion was set ad-hoc, so whether the control error can be considered acceptable or not should be judged from tests and experience, as in one hand the magnitude is high, but on the other, it returns back to normal quite fast and the error itself will occur seldom. If it is considered acceptable by the system developer, the process is finished, else some of the proposed measures in Chapter 3.4.3 can be used to design an assertion, which then can be evaluated using the analysis described above. This iteration should be continued until an acceptable design is found.



(a) (b)

Figure 4.7: Output response and corresponding control signal to sporadic data errors in the integrator state.

Second, as discussed in Chapter 3.4.3, to determine when the controller cannot handle a specific error, a signal monitoring the health of the system should be defined. As we determined to use the control error as failure criteria, this signal is suitable to use. Thus, an robust assertion, according to Chapter 3.4.3, should be applied to this signal.

In order to set the threshold for the assertion, the most severe errors should be identified. For the brake-slip controller, persistent errors with constant high magni-

tude in the integrator step were identified to be one of the most severe errors through sensitivity analysis, Chapter 4.4.1. Therefore, to exemplify the analysis methods, the step response function is now computed and shown in Figure 4.8(a) together with the resulting control signal, Figure 4.8(b), for such errors.

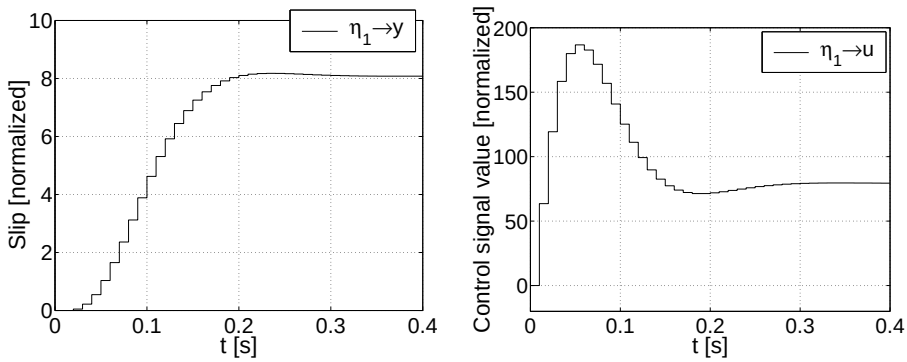
From closer inspection of Figure 4.8(b), the maximum generated control signal was 186.72. As the maximum accepted control signal was 4.3, due to the modified anti-windup described above, the magnitude of the most severe persistent error with constant magnitude, can be found as $m = 4.3/186.72$. Thus the response function needs to be scaled (multiplied) with this factor. Then from the scaled figure the times when the failure criteria is violated can be found (i.e., when do the response exceed the maximum allowed magnitude and duration).

Figure 4.9 shows the scaled response to a persistent error with constant magnitude $m = 4.3/186.72$ in the integrator state. The specified failure criteria is also included in the figure. From closer inspection of Figure 4.9 it was found that the failure criteria is exceeded at 0.04 s. Now assume that it takes two samples for the controller to set the system in a fail-safe state (or for a spare computer to recover the system). As the assertions are working on the control error that already exist in the system, this means that the detection must take place at least two samples (0.02 s) before the failure criteria is exceeded, i.e., at time $0.04 - 0.02 = 0.02$ s. Through the inspections, it was found that the control error is 0.0011 at this time. Thus, the threshold for the assertion checking the magnitude of control error should be set to 0.0011. However, this is a very low level, even lower than the assertions designed for automatically recovery. This implies, first, that the automatic recovery would be useless, and second, that the assertion would probably have very high false and unnecessary alarm rates. Therefore, either the failure criterion **C1** should be relaxed (remember that it was set ad-hoc for this example), or another type of error detection technique must be applied for detection of these errors.

It should be noted that when the failure criterion **C2** is used, similar assertions can be applied to monitor this property.

4.6 Summary

In this chapter, analysis models and methods were adopted from control theory for understanding the effect of data errors on control systems. The results obtained from the analysis of an automotive brake-slip controller showed that many transient faults are tolerated by the control system and that the part most sensitive to data errors is the integrator state. This indicates that many results found from fault injection experiments [Cunha *et al.*, 2001], [Vinter *et al.*, 2001] can be estimated from anal-



(a) (b)

Figure 4.8: Output response and corresponding control signal to persistent errors with constant magnitude in the integrator step.

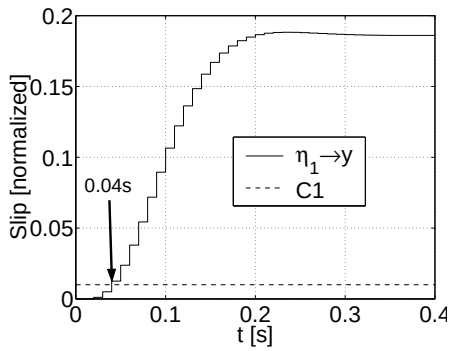


Figure 4.9: The scaled output response to a persistent error with constant magnitude in the integrator step, together with the specified failure level **C1**.

ysis already in early design phases. Such information is valuable for developers of safety-critical systems when deciding on which fault tolerance techniques are efficient to use.

More specifically, using the described analysis it is possible to estimate: i) the control performance degradation (i.e., the control error) different errors cause, ii) the maximum allowed recovery time before a system failure occurs, iii) how often errors can occur without their effects being superimposed, iv) the error magnitude dependency on the data format used.

Furthermore, it was also shown how executable assertions can be designed and evaluated with similar analysis. These analysis methods provide, among other things, information on how much the impact of different data errors are reduced due to applied assertions and recovery.

The analysis methods are best suited for sporadic errors and persistent errors with constant magnitude. As these often are considered to be the most frequent and severe errors, we believe that the analysis can provide important results during development of safety-critical control systems. However, for other error classes, and for verification purposes, simulations may be necessary if not complementary analysis methods can be developed.

CHAPTER 5

Experimental Evaluations of Data Errors

In the previous chapter, analysis methods for estimating the effect/impact of data errors were discussed and developed. The main advantage with the proposed analysis is that it can be applied early in the design process. However, for certain errors, and for validation and verification purposes, the analysis cannot be used. For such detailed evaluations, fault injection (see Chapter 1.2.1) is often suitable.

As the first implementations generally are models of the system, simulation-based fault injection is preferable. The main drawback with simulations is that models are used, and thus, the accuracy of the results is dependent on the accuracy of the models. This is a major problem for the use of commercial processors as low-level (detailed) descriptions of such seldom are accessible for system developers. Thus, high-level simulations must be used which generally are less accurate compared to low-level simulations as less locations are controllable and observable for fault injections. In addition, most existing evaluation techniques focus on a specific class of faults such as transient or permanent faults and/or require detailed (low level) descriptions of the processors, which seldom are accessible for developers of embedded systems. Therefore, first in this chapter, how the error generation is dependent on the fault injection abstraction level, is investigated. The investigation is performed by comparing the characteristics of errors (important error characteristics were identified in Chapter 3.7) generated from faults injected at gate-level with errors generated from faults injected at RTL. The results showed that fault activation (error generation) probability is much more dispersed for errors generated from

gate-level fault injections and the average activation probability is lower. Thus, the impact of faults injected at RTL may significantly differ from the impacts of faults injected at gate-level.

Then, related research on modeling of persistent fault for those cases when the characteristics differs, is presented. Most of this research targets accurate modeling of specific fault types and requires some sort of detailed information, e.g., fault libraries, or do not show how the simulated errors correspond to errors caused by real faults.

Therefore, in order to be able to model the effects of real faults without detailed low-level information, a high-level simulation-based fault injection approach based on insertion of specific components, so-called saboteurs [Jenn *et al.*, 1994], for fault injection into VHDL-models is developed. The novelty of this approach is that the characteristics of errors that determines the impact on the system can be varied for the signals that are reachable at the chosen abstraction level. Furthermore, not only the impacts of data errors can be evaluated, but also the efficiency of applied error detection techniques.

If the developer has little knowledge about the characteristics of the faults that can occur and/or which impacts they will have, faults with great varying characteristics should be injected, since the approach itself has no inherent correspondence to the characteristics of real faults. However, the impact of a fault will in many cases be the same if it occurs within a certain time interval or is located within a certain area. Also, generally the longer the duration of the fault is, the more severe is the resultant impact. Thus, using this information for intelligently choosing injection times, and locations and durations for the faults, the required number of simulations can be heavily reduced.

If the developer has some knowledge about the faults that can occur or the robustness of the system, this can be utilized to focus the injections to faults with specific characteristics. As an example, if it is known that the system tolerates most transient faults, the injections can be focused mainly to persistent faults. When the system developer has determined the characteristics of the faults to inject, the rest of the evaluation process is fixed, and thus, can be automated.

The saboteur approach is implemented and evaluated in two different examples. In the first example, it is shown that the error characteristics can be varied as desired. The second example describes evaluation of the detection coverage of double execution for data errors caused by persistent faults. From the latter example it can be seen that as it is possible to vary the characteristics of errors with the saboteur approach, the influence of these characteristics on error impact and detection coverage, can be evaluated. Furthermore, for this specific implementation, it was not

possible to deactivate the overflow exception which then affected the estimation of the coverage for double execution. However, with the saboteurs, it was possible to generate errors that did not result in overflows, and thus, most of the influence of the overflow exception on the estimated coverage of the double execution technique could be removed.

In the end of this chapter, it is described how the approach can be modified for use at other abstraction levels (the approach is also portable to any simulation tool) and for other fault injection methods (e.g., scan-chain and software-implemented fault injection).

5.1 Traditional Simulation-Based Fault Injection

Traditionally, the impacts of faults have been determined through fault injection Chapter 1.2.1, i.e., injecting faults into the system and observing their impacts on the delivered service and whether resulting errors are detected. Faults can be injected either physically, through scan-chains, simulations or in software. Most of this previous work considers injection of transient faults. However, as was seen in Chapter 2, the occurrence rate of persistent faults can be expected to increase in commercial microprocessors. Therefore, it is important to be able to estimate the effect of such faults as well.

The problem of evaluating the impact of faults with long duration using physical fault injection is that there is a risk of permanently destroying the circuit. Furthermore, internal scan-chains are seldom accessible for system developers and it is difficult to mimic all types of hardware faults with software-implemented fault injection. Simulation-based fault injection does not suffer these problems, but the accuracy of the result is, as for all simulations, highly dependent on the level of details that are provided by the model. As microprocessor manufacturers seldom make low level (detailed) models of the microprocessors available, the simulation is invariably based on high-level models, and thus, there is a risk that the evaluation gives inaccurate results.

Accurate modeling of transient hardware faults at Register Transfer Level (RTL) has been discussed for instance in [Yount and Siewiorek, 1996]. The modeling was intended to resemble the effects of real faults, but to avoid combinatorial explosions, only a subset of the faults was chosen. Since modeling of persistent faults also needs to consider the time aspect, i.e., such faults can be activated several times, the combinatorial growth explosion will be even more critical for such modeling. Furthermore, fault activation depends not only on the application for which the system is used, but also on how the faulty component is implemented. Since detailed infor-

mation of the implementation may not be available for most system developers, a high-level approach that can model faults with varied time duration is necessary.

The simplest and the most commonly used fault model is to manipulate the value of some signal in the structural model of the component/system for the desired fault duration. For transient faults this implies halting the simulations, changing the value of the desired signal(s) and then continuing the simulation, i.e., *bit-flipping*. For permanent faults, this means forcing the desired signal to hold a certain value during the continuation of the simulation, i.e. the so-called *stuck-at fault* model. The accuracy of using these models at high level, considering the identified characteristics in Chapter 3.7, is now briefly discussed.

As it is possible to halt the simulations at any time to manipulate signals, it is simple to model the occurrence-time (*State*) by injecting the fault (manipulating the value of the signal(s)) at this time. Moreover, as the tasks and hardware is simulated, and assuming that relevant input data is used (i.e., that the system responses to typical input data are known), the error generation is included inherently (*Repetition* frequency). Furthermore, the *Magnitude* and *Number* can be controlled by which and how many bits that are flipped or set stuck.

However, if only high-level component/system models are accessible, which generally is he case for system developers and researchers within the academia, the number of locations (*Property*) where faults can be injected is limited. As the location of the fault affects how, when and to which output(s) resulting errors propagate to, the *Magnitude*, *Number*, and *Repetition* frequency, are indirectly restricted. This means that the number of errors that propagate becomes higher than in reality, as no errors are masked at lower abstraction levels than the faults are injected at. Also, the number of multiple errors can be expected to be lower when faults are injected at high levels as the propagation paths are fewer.

For transient faults, it is generally possible to compensate for both these inconsistencies by biasing the number of effective errors and by also injecting multiple faults. However, for persistent faults, the *Repetition* frequency for stuck-at faults injected at high-level can become higher as high-level signals generally has a larger influence on the functionality of the component. Thus, faults injected at high-level simulations can be expected to be activated more often. This inconsistency can generally not be compensated for by biasing.

The *Repetition* frequency of an error is one of the characteristics determining its impact on the system, but also how easy it is to detect. Generally, the higher the *Repetition* frequency is of an error, the greater will the impact be, but at the same time, the resulting errors will be easier to detect. Therefore, it is important that the set of faults that can be injected have the same error characteristic distributions as

the total set of faults.

It should be noted that in Chapter 4, there were also limitations on which signals (properties) that could be analyzed as erroneous. However, it was identified that in order for any microprocessor fault to have any impact on the controlled physical process, the resulting errors must propagate to any of the signals that were accessible. This implies that as it is possible to vary the characteristics of these errors freely, the impact of all possible errors can be modeled. Also, for high-level simulations, for any fault to have any effect on the system, the resulting errors must eventually propagate through a reachable signal. However, the difference between the analysis and high-level simulations, is that it is not possible to vary the characteristics of errors injected in high-level signals freely using the bit-flip or stuck-at fault model. *Therefore, it is possible that these fault models discriminate errors with certain characteristics, when used at high-level simulations.* If so, it is possible that results from such evaluations is not valid in reality.

To investigate in which cases the error characteristics differ between faults injected at low and high-level, a comparison between fault-injection at RTL and gate-level was performed. The results are presented in the next section.

5.2 Investigation of Abstraction Level Dependency

Prior to developing a new evaluation approach, first, in this section, how the accuracy of evaluations using traditional fault models is affected by the fact that fewer signals (locations) are accessible at higher abstraction levels, is investigated. This investigation is performed by comparing the characteristics of faults injected at gate-level and RTL. The estimation was performed on Functional Units (FU), i.e., the units of the execution stage of the processor pipeline, e.g., ALU, multipliers, shifters, etc. These components are of specific interest to investigate as they are some of the least protected components in microprocessors (e.g., they are hard to efficiently protect with codes since they transform data), see [Mendelson and Suri, 2000].

However, even if the simulation results in this section are valid mainly for functional units (which are combinational), the fault injection approach presented in this chapter should be possible to utilize also for other types of combinational components and sequential components, see Chapter 5.4.2.

For injection of faults, the stuck-at model was used, based on the assumption that *the impact of a single-bit fault that has shorter duration than the time required to perform one computation of the functional unit, will either not effect the computation or have the same impact as a fault with duration longer than the required computation time*. We motivate this assumption by the fact that the signals in a func-

tional unit are generally only run through once during a computation, and thus, are then either correct or incorrect. Moreover, the computation (execution) time of a functional unit is in the order of nanoseconds (and decreasing).

To estimate the impacts of faults, four quantities were measured, namely:

- (**I**): The probability that an injected fault generates an error that propagates to any of the outputs of the component. This measure can be used to estimate the *repetition* frequency.

- (**II**): The number of times each input value propagated an error to any output. This measure can be used to determine whether certain operating *states* are more error prone than others.

- (**III**): The number of output signals that each error propagated to, i.e., the number of single- and multiple-bit errors. This measure is connected to both the *number* and *magnitude* characteristics.

- (**IV**): The number of times an error manifested in each specific output bit. This measure can be used as an estimate of the *magnitude* of the errors.

As faults were injected in different signals, the *property* characteristics was indirectly investigated.

All these quantities were measured by first simulating the components with the chosen input values, without injecting any stuck-at faults (the golden run). Then the components were simulated with the same input values for each stuck-at fault (i.e., with a signal held at 0 or 1).

More specifically, the estimations were performed through simulations of two different gate-level adders and multipliers, and one barrel-shifter. The obtained results were compared with simulations of the simple behavioral descriptions of the components at RTL, see Table 5.1.

The reason to simulate different gate-level implementations of the same component was that components that use a different trade-off among performance, power-consumption and circuit area, have different gate-architectures. Of these components, the *74238* and *c6288* are simpler, and thus, easier to analyze, whereas the *Thor*[1]-components are larger, and thus, more realistic. However, at the RTL, all behavioral descriptions of a certain component will be very similar and thus, only one representative case, were implemented.

[1] *Thor* is a microprocessor developed by Saab-Ericsson Space designed for space applications, [Saab Ericsson Space, 1999].

Table 5.1: The investigated components.

Component	**Description**	**Total Number of Bit-Signals**	**Number of Injected Faults**^{VI}	**Number of Applied Input Values**
74238 Adder	Gate-Level Carry Look-Ahead Adder, 9 inputs, 5 outputs	45	$36x2 = 72^{I}$	512^{IV}
Thor Adder	Gate-Level Carry Look-Ahead Adder (Synopsis standard component), 56 inputs, 28 outputs	~ 1916	$1832x2 = 3664^{II}$	1024^{III}
Thor Barrel-Shifter	Gate-Level Barrel-Shifter, $\sim$ 72 inputs, 32 outputs	~ 4175	$4078x2 = 8156^{II,III}$	$1024^{III,V}$
c6288 Multiplier	Gate-Level Radix-2 Multiplier, (ISCAS-85), 32 inputs, 33 outputs	~ 2448	$2416x2 = 4832^{I}$	1024^{III}
Thor Multiplier	Gate-Level Multiplier using Booth recoding and Wallace tree structure, 67 inputs, 67 outputs	~ 58774	$2416x2 = 4832^{II,III}$	1024^{III}
RTL Adder	Unsigned Behavior Adder, 32 inputs, 17 outputs	49	$49x2 = 98$	1024^{III}
RTL Barrel-Shifter	Behavior Barrel-Shifter, 65 inputs, 32 outputs	97	$97x2 = 194$	1024^{III}
RTL Multiplier	Unsigned Behavior Multiplier, 32 inputs, 32 outputs	64	$64x2 = 128$	1024^{III}

^{I} Faults were not injected in the input signals.
^{II} Faults were not injected in the input and output signals.
^{III} The input values were randomly selected.
^{IV} All possible input values were tested.
^{V} Only left and right shift of zeros were tested.
^{VI} Both stuck-at-0 and stuck-at-1faults were injected.

As the gate-level components were described with different formats, the simulations were run on two different simulators. The *74238* and *c6288* components were described with the original *ISCAS-85* format [Brglez and Fujiwara, 1985], for which the gate-level fault simulator *KSIM* [Wiklund, 2000] was used, which was validated with a switch-level simulator [Dahlgren and Lidén, 1991] that, in turn, was validated using *SPICE3* [Johnson *et al.*, 1991]. The other components (the components from *Thor* and the behavioral RTL components) were described with *VHDL*-code and were simulated with *ModelSim* [Mentor Graphics 1998], a commercial VHDL-simulator. The number of signals each component consists of, the number of injected faults and the number of applied input values (i.e., the number of performed computations) can be found in Table 5.1.

5.2.1 Results of the Quantity Measurements

In this subsection, the results from the four fault quantities measurements are presented and discussed as four separate points (**I-IV**).

I) The first quantity that was measured was the probability for each injected fault to generate an error that propagated to at least one of the outputs. That is, the result from the simulation, with the fault injected, differed from the golden run in at least one bit. In Figures 5.1-5.3, the probabilities different faults have for generating an error that propagates, can be seen for the specific components of Table 5.1. The minimum, maximum and average propagation probabilities for each component are summarized in the left part of Table 5.2, which also distinguish amongst injected stuck-at-0 and stuck-at-1 faults.

The first apparent observation when comparing the figures of the gate-level simulations with the RTL simulations is that the propagation probability generally is much more dispersed between different faults at the gate-level (this is most evident for the barrel-shifter, Figure 5.2, and the multiplier, Figure 5.3). This is natural since fewer signals are controllable at higher abstraction levels (RTL), i.e., there are fewer locations for which it is possible to simulate faults (the exception is the *74238* adder which is a very small component, and therefore, shows similarities with the RTL-components). Furthermore, the locations at which it is possible to simulate faults are often symmetric, resulting in faults that will have similar propagation probabilities.

It can also be seen in Table 5.2 that the average propagation probability generally is higher at RTL than at gate-level. This is due to the fact that faults are more likely to be logically masked when simulated in the components (gate-level) than when simulated at the inputs or outputs of the components (RTL). Furthermore, modeling persistent faults with the stuck-at model means assuming that a signal is stuck at a

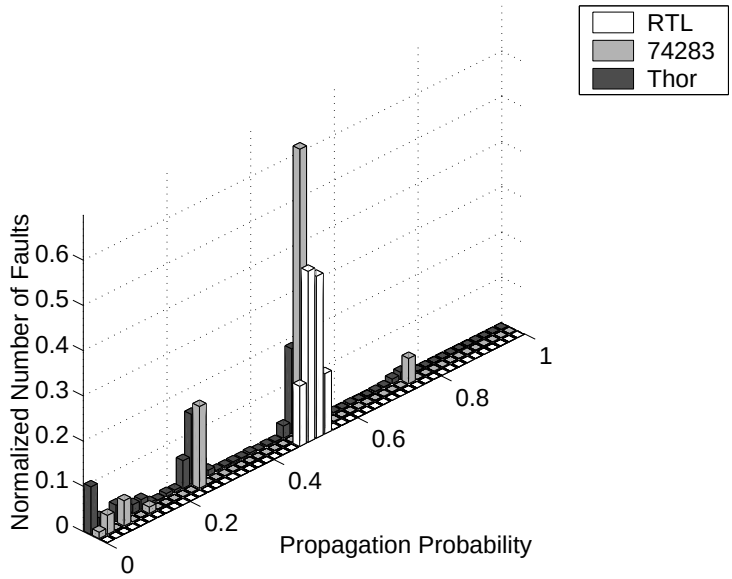


Figure 5.1: The propagation probability distributions for the different adders.

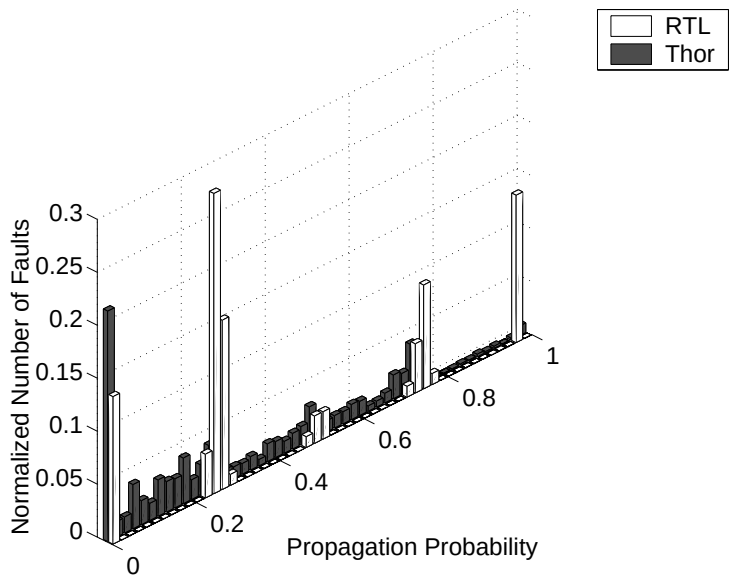


Figure 5.2: The propagation probability distributions for the different barrel-shifters.

Table 5.2: The error propagation probabilities.

Comp-onent	Propagation Probability							
	Stuck-at-0 Faults			Stuck-at-1 Faults			Input Values	
	Min	Max	Average	Min	Max	Average	Min	Max
74238 Adder	0.0156	0.500	0.3537	0.2500	0.7500	0.4722	0.3056	0.5000
Thor Adder	0	0.7686	0.3432	0	0.7637	0.3121	0.2716	0.3758
Thor Barrel	0	1	0.3379	0	0.7637	0.3121	0.2716	0.3758
c6288 Multi-plier	0	0.8213	0.3236	0.1787	0.9531	0.5705	0.4385	0.4538
Thor Multi-plier	0	1	0.3591	0	0.9766	0.3429	0.3363	0.3640
RTL Adder In	0.4648	0.5303	0.4977	0.4697	0.5352	0.5023	0.5000	0.5000
RTL Adder In	0.4717	0.5244	0.5005	0.4756	0.5283	0.4995	0.5000	0.5000
RTL Barrel In	0	0.5107	0.1712	0.2373	0.9785	0.5780	0.2010	0.4948
RTL Barrel Out	0.2285	0.2852	0.2557	0.7148	0.7715	0.7443	0.3402	0.4948
RTL Mul-tiplier In	0.4648	0.5303	0.4977	0.4697	0.5352	0.5023	0.5000	0.5000
RTL Mul-tiplier Out	0.1670	0.5254	0.4568	0.4746	0.8330	0.5432	0.5000	0.5000

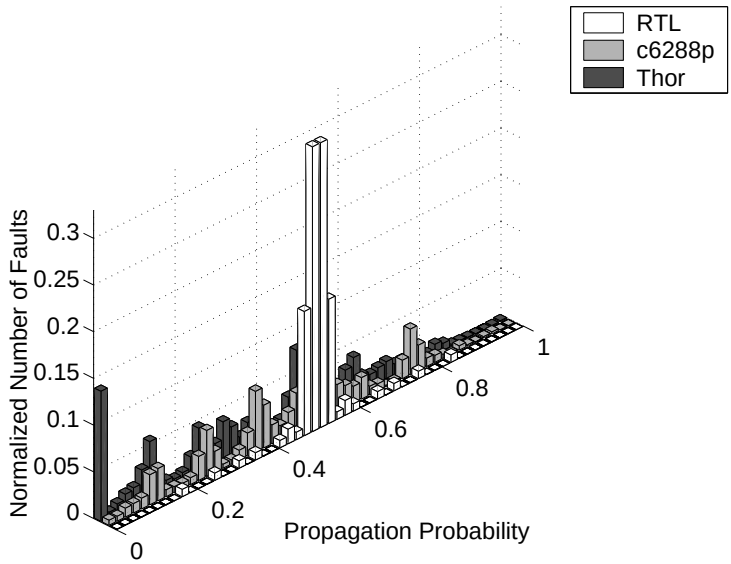


Figure 5.3: The propagation probability distributions for the different multipliers.

certain value during a certain time. This is accurate for short-circuits to ground or supply voltage in accessible signals. However, some persistent faults imply only a slightly changed resistance, and/or short-circuits to other signals (bridging faults), which means that the signal is erroneous only at certain operation points, i.e., for certain input values. *For these reasons we expect the propagation probability for real faults to be even more dispersed and to have even lower average propagation probability than the results from the gate-level simulations*. Thus, considering the propagation probabilities, only a small subset of the possible errors, will be evaluated using the stuck-at fault model at RTL. Since the propagation probability to a high extent affects the impact of faults, it is important to be able to evaluate errors with different propagation probabilities, and thus, another approach is required.

In Table 5.2, it can be seen that the average propagation probabilities for the RTL adder and multiplier are roughly 0.5. This is because that if the correct value of a signal is 0, then the injected stuck-at-0 fault will not be propagated whereas the stuck-at-1 fault will be, and the other way around. However, for the barrel-shifter, the average probability is lower. This is because injected faults in the input signals may be shifted out, and in that case, not cause any errors to propagate to the output signals. It can also be observed that for the RTL multiplier and barrel-shifter, the propagation probabilities are different between stuck-at-0 and stuck-at-1 faults in-

jected in the same signal. For the multiplier, this depends on that the least and most significant output bits with higher probability will be 0 (the result from a multiplication is with higher probability even than odd and for many multiplications the most significant output bits are not required to form the result). Thus, injected stuck-at-1 faults are activated with higher probability. For the barrel-shifter the difference comes from the fact that some bits in the register controlling the number of steps to shift, are not utilized and set to 0, and that only zeros are shifted in, i.e., the probability for a signal to be 0 is higher. Therefore, injected stuck-at-1 faults are activated with higher probability.

Also, for the gate-level simulations of *74238*, *c6288* and *Thor* barrel-shifter a difference between injected stuck-at-0 and stuck-at-1 faults can be seen in Table 5.2. This is explained for *74283* and *c6288* by the fact that these components mainly consist of NOR-gates and AND-gates, whose output signals with probability $(2^n - 1)/2^n$ attain 0 for n-input gates, resulting in stuck-at-0 faults being more likely to be masked than stuck-at-1 faults. For the *Thor* barrel-shifter, the error propagation probability is slightly higher for the stuck-at-0 faults than the stuck-at-1 faults. This component is built from different types of gates, so it is not as easy as compared to the *74238* and *c6288* to analyze the reason for the difference, even if it still may be the choice of gates.

Another observation regarding the error propagation is that generally, the more signals the component description consist of, the lower the measured average propagation probability will be simply because the number of possible data paths is higher. However, the probability for a fault to occur (the fault intensity) will be higher for components consisting of many signals, i.e., there are more possible points of failures.

Thus, the results show that the propagation probability distribution depends on both the function and the implementation of the component. However, the exact relation between these two factors has not been investigated.

II) The second quantity that was measured was the number of times each input value propagated an error to the output signals. This measure provides information whether some input values are more prone to propagating errors. This information is useful for generation of efficient test data sets, but also at system level to determine whether there exist certain operating points (system states) at which errors are more likely to propagate. The two right-most columns of Table 5.2 show the minimum and maximum normalized number of propagated errors for the input values of each component. It can be seen that for the multipliers, the variance is very low, whereas for the adders and the barrel-shifter, the variance is higher, but still low. Thus, we conclude that for these functional units the error propagation probability is at both

abstraction levels rather independent of the input value being applied (i.e., the stuck-at fault model at RTL models this quantity satisfactory). Thus, the *state* has only minor influence on which impact errors have on the system.

III) The third quantity was the distribution of single- and multiple-bit errors. This measure provides information about how the errors manifest themselves. This is an important factor for determining which impacts faults have on the system. Figures 5.4-5.6 show the distributions for the different components, where also the probabilities for an error not to propagate are added as references (the 0-values).

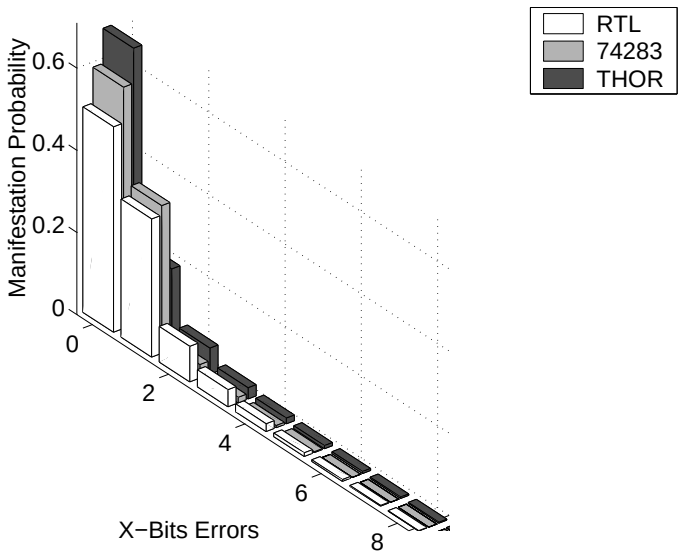


Figure 5.4: The single-, multiple- bit manifestation probability distributions for the different adders.

As can be seen, single-bit errors have the highest occurrence rate. Furthermore, the probability for multiple errors decreases exponentially with the number of erroneous bits for all components, except for the RTL multiplier. Through closer inspection of this component, it was found that the higher rate (still low compared to the rate of single-bit errors) of multiple errors (centered around 9-bit errors as seen in Figure 5.6) was caused by faults injected in the input signals of the component. This is explained by the fact that injecting a fault in one bit of the operands will, due to the nature of multiplication, most likely change several of the output bits. Thus, for some components (functions) the stuck-at fault model at RTL does not correspond to the distribution at gate-level. This also shows that even if single-bit errors are

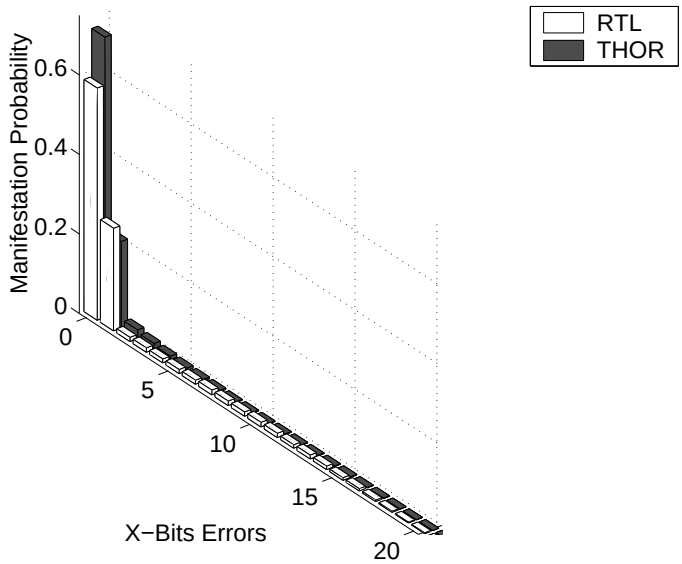


Figure 5.5: The single-, multiple- bit manifestation probability distributions for the different barrel-shifters.

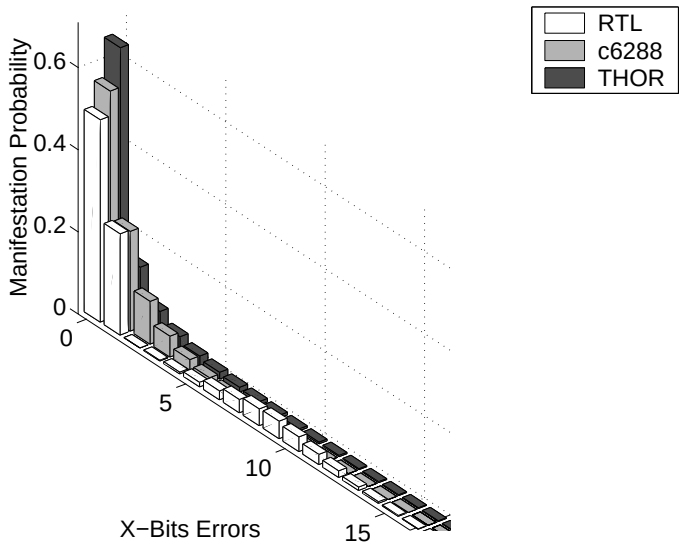


Figure 5.6: The single-, multiple- bit manifestation probability distributions for the different multipliers.

the most common type of errors at the outputs of a failed component, this may be changed if the error is propagated through an additional component.

IV) The fourth measured quantity was the distribution of manifested errors among the output signals. This quantity also provides information on how errors manifest themselves, and thus, can be used to determine how faults will affect the system. Figures 5.7-5.9 show the distributions for the adders, the barrel-shifters, and the multipliers. Please note that since the components have different number of outputs (see Table 5.1), the components have different number of data points in the figures. Therefore, to be able to compare components with different number of outputs, the manifestation probabilities have been weighted with their total number of output bits. This implies that if the manifestation is evenly distributed among all outputs of two components with different number of output bits, the data points of these two components will also get the same value in the figure. For the *74283* adder it can be seen that the manifestation probability is higher for more significant bits, with exception for the most significant bit, which has the lowest probability. This is due to the fact that errors are more likely to propagate to sum signals than carry-out signals, and the most significant bit is just the carry-out signal from the computation of the second most significant bit. The same pattern can be seen also for the RTL-adder, but is not evident for the *Thor*-adder. For the barrel-shifters the distributions are rather evenly distributed.

For the *c6288* multiplier the error manifestation probability is highest for the bits in the middle. This is because more hardware (signals) is required to form these bits than the other bits. The other multipliers show the same characteristics, but the distribution is smoother for the *Thor*-multiplier. To visualize this, the outputs of the multipliers in Figure 5.9 have been centered.

To summarize, the stuck-at fault model at RTL models this quantity satisfactory for these components.

5.2.2 Summary of the Results

The following summarizes the important results from the stuck-at fault modeling at gate-level and RTL:

- There are similarities between the two abstraction levels in how errors manifest themselves (in which bit locations and the number of locations).

- There are differences between the two levels in the distribution and average of the error propagation probability (the average is higher and the distribution is less dispersed for faults injected at high level).

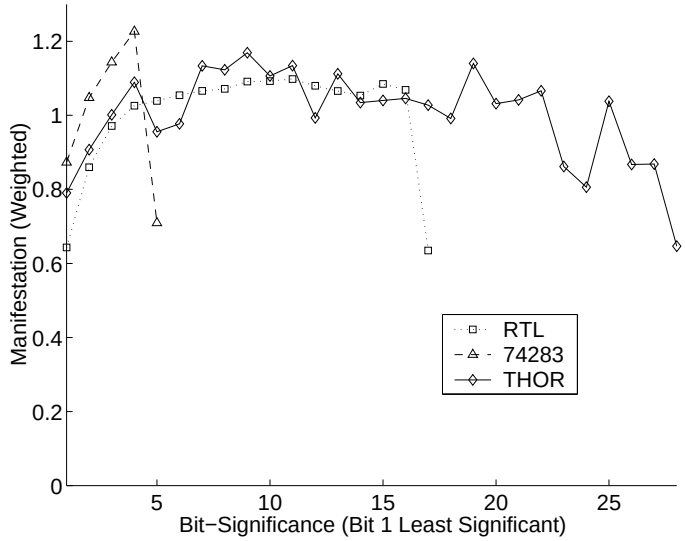


Figure 5.7: The output-bit manifestation probability distributions for the different adders.

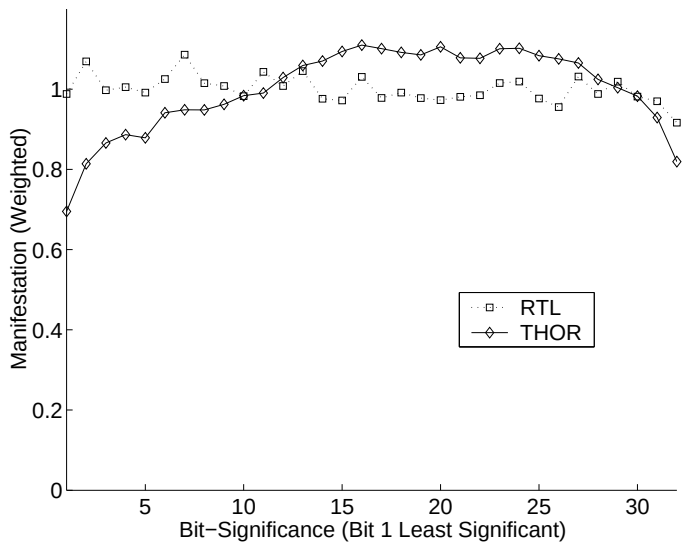


Figure 5.8: The output-bit manifestation probability distributions for the different barrel-shifters.

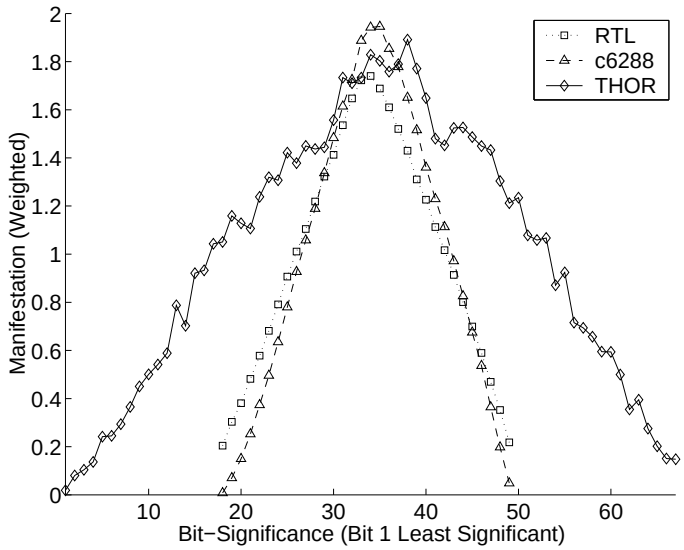


Figure 5.9: The output-bit manifestation probability distributions for the different multipliers (to visualize patterns between the multipliers, they have been centered).

From these results we conclude that the error propagation probability is greatly affected by the fact that only a limited number of signals (locations) are accessible for fault injection at high abstraction levels. This implies that the *repetition* frequency of the resulting errors will be different. This influences the effects/impacts persistent faults have on the system services as well as the detection efficiencies of different techniques Chapter 3.7. *Thus, the stuck-at model at RTL is not sufficient for evaluating the impacts of persistent faults and for ascertaining the efficiencies of online error detection techniques*. Furthermore, gate-level descriptions of microprocessors are seldom accessible for system developers. Therefore, a high-level evaluation approach that can model the characteristics of real low-level faults is required. In the next section previously proposed alternative approaches are discussed.

5.3 Related Work on High-Level Modeling of Persistent Faults

In the previous section, it was found that the stuck-at fault model at RTL cannot capture all characteristics of real low-level faults. Therefore, the stuck-at fault approach is not appropriate and a more complex high-level evaluation approach that

can model the characteristics of real low-level faults is required. In this section such previously proposed approaches are surveyed.

To be able to simulate more complex effects of persistent faults using high-level simulations, two concepts were introduced in [Jenn *et al.*, 1994]: *mutants* and *saboteurs*. A *mutant* is a model of a component with a specific fault inside. The mutant replaces the original component in the simulation when one would like to evaluate the impact of the fault. Examples of how mutants can be implemented and used are found in [Gracia *et al.*, 2001], [Leveugle and Hadjiat, 2000]. The drawbacks with these solutions are that in order to simulate all types of faults, detailed low-level information must be available. A *saboteur* is a special type of component that is inserted into the system model for enabling manipulation of signals to resemble complex faults. The saboteur will manipulate the signals when specific conditions defined in the simulator are satisfied. Some examples of implementations are [Gracia *et al.*, 2001], [Boué *et al.*, 1998]. However, these implementations of saboteurs cannot vary all characteristics of the simulated faults, and thus, can only simulate subsets of all possible persistent faults.

Approaches for injecting persistent faults similar to saboteurs and mutants, are provided in [Delong *et al.*, 1996], [Sieh et al., 1997], but they do not define how to resemble low-level faults at high-level. In [Kalbarczyk *et al.*, 1999] a method where faults are injected using fault libraries, i.e. requiring low-level details, is developed.

Within the test community, research on high-level modeling of persistent faults has been performed in order to find efficient test data sets for detecting manufacturing faults, e.g., [Ferrandi *et al.*, 2002]. However, there are differences between developing manufacturing tests and ascertaining the efficiencies of online error detection techniques: *An efficient manufacturing test set should detect as many different faults as possible where the length of the test is determined as a trade-off between the cost of the time delay introduced due to the testing and the cost for shipping faulty components. Thus, the detection efficiency of the test can be estimated directly from the hardware architecture of the component. For online detection techniques, the important thing is to detect those errors that if not handled in time will violate the dependability requirements, while ensuring that the real-time constraints of the system is not violated. Thus, the efficiency of online error detection techniques is dependent on how the specific system activates the fault (Chapter 3.7), i.e., utilizes the faulty hardware. Therefore, preferably, the application software should also be simulated to get accurate evaluations.*

There are some test approaches based on saboteurs and/or mutants that may be used: [Celeiro *et al.*, 1996], [Riesgo and Uceda, 1996],
[Aftabjahani and Navabi, 1997], [Vargas *et al.*, 1998], [Vado *et al.*, 2000],

[Shaw *et al.*, 2001]. However, these are based on detailed low-level information or can only simulate the effects of a limited number of faults.

To summarize, the limitation of existing approaches is that low-level knowledge is required or that they do not show how to resemble all types of real faults at high abstraction levels. Therefore, in the next section, a new high-level approach that can inject faults with any characteristics is developed. The problem of how to set the characteristics to match real low-level faults is also discussed.

5.4 A Novel High-Level Evaluation Approach

As shown in Chapter 5.2.1, the error propagation probabilities differ between stuck-at faults injected at gate-level and RTL. Furthermore, low-level details of components are seldom accessible for system developers. Thus, for system developers to evaluate the impacts of faults and to ascertain efficiencies of error detection techniques, more complex high-level modeling is required. As seen in the previous section, a number of approaches have previously been proposed, but either they are based on low-level details, or they do not describe how to model real faults. Therefore, in this section, a novel high-level VHDL-simulation approach for accurate modeling of fault characteristics, is developed. Here, the approach is implemented at RTL, but it should be possible to implement it to any abstraction level, see Chapter 5.9. To be able to model the characteristics of real faults, the approach utilizes the fact that for a fault to have any impact, the resulting errors must propagate to one or more outputs of the component, eventually propagating through at least one accessible signal (this was also discussed in Chapter 4.3. Therefore, by manipulating the accessible signals according to the propagation of the errors resulting from the fault, it should be possible to model any faults, also at those locations which are not directly accessible. The problem with this approach is how to manipulate the accessible signals so that they correspond to real faults.

In this section this problem is handled by first defining the desired properties for an evaluation approach. Then a fault injection approach based on saboteurs, i.e., insertion of specific components for enabling injection of faults into the system model, is developed. The novelty of the approach is that the saboteurs are designed so that the characteristics determining the impacts of faults (these were discussed in Chapter 3.7) and the possibility to detect resulting errors can be set arbitrarily. Thus, any value combinations of these characteristics can be investigated. After that, how the characteristics should be tuned if the developer has no or little knowledge about the characteristics of the faults that can occur, is briefly discussed. It is also shown how knowledge of faults can be obtained and utilized to limit the number

of injected faults. Then, it is demonstrated how a fault injection campaign is set up. In Chapter 5.6, some simple simulations are performed to verify that the fault characteristics can be set as desired and to investigate the associated complexity.

5.4.1 Desired Properties for Evaluation

In order for system developers to be able to evaluate the impacts faults have on system services and to ascertain the efficiencies of online techniques for detecting resulting errors, some basic properties are desired, namely:

- A high-level approach, as low-level descriptions of the components generally are not accessible.
- The simulation time (complexity) must be reasonably low.
- The impacts of the simulated faults/errors must correspond to the impacts of real faults.
- The modeling should not require any manual changes of the system model, i.e., it should be automated.

RTL is often the lowest level at which component descriptions are accessible for system developers. We also believe that the required simulation time will be low enough for most applications at this level. Therefore, the approach is focused to this level even if it is transferable also to other levels, see Chapter 5.9.

In order to simulate faults with characteristics corresponding to real faults, according to the results from the comparison in Chapter 5.2.1, we would like to:

- Simulate faults with any duration.
- Simulate errors with any propagation probability.
- Simulate errors that manifest themselves in correspondence to how errors caused by real faults manifest themselves.

In the next subsection, it is described how these desired properties can be implemented and automated in VHDL-simulations.

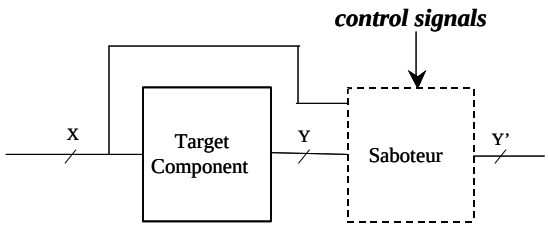
5.4.2 The Saboteur-Based Approach

VHDL-models at RTL consist of data storage and transformation units where inputs and outputs are interconnected to form more complex components/systems. The behaviors of the components are defined by program code. To be able to simulate errors in the simple components according to the previously defined properties, saboteurs are inserted at the outputs of the components one would like to simulate faults inside, see Figure 5.10(a). The saboteur is inserted in between the target-component for the fault injection and the components the outputs of the target-component are connected to. This means that all outputs (Y) are fed through the saboteur. Thus, the output signals from the saboteur (Y') will have the same format as the output signals of the target and will be connected to the same components as the target originally was.

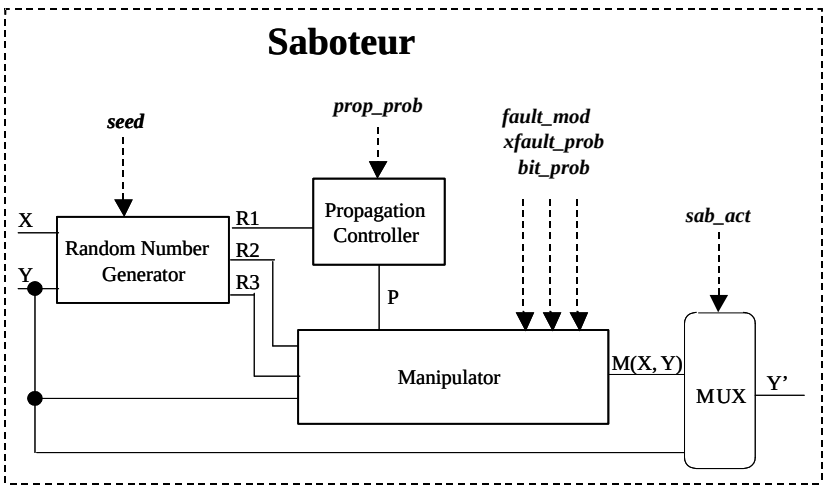
Generally, a fault will be activated and the resulting errors propagated in the same way each time the input signals assume the same values. To accomplish this, the input signals (X) are also connected to the saboteur. *It should be noted that the insertion of saboteurs into the VHDL-model of the microprocessor can be automated as it is just a question of inserting a component between existing components whose connections are well defined in the* VHDL-*code*. This is discussed in more detail in Chapter 5.4.3.

As described in Chapter 5.3, previous proposed approaches do not enable, or describe how to the characteristics of the injected faults can be varied to resemble real faults. To enable this, which is the novelty of our approach, the saboteur (the complete VHDL-code for a saboteur can be found in Appendix E) has a number of control signals which are set by the system developer to match the characteristics of faults that are believed to occur. Table 5.3 shows the formats of these signals and Figure5.10(b) shows the internal structure of the saboteur where the names of control signals are emphasized.

To be able to simulate faults with any duration (as discussed in the beginning of Chapter 5.4.1), an activation signal, *sab_act*, is added to the saboteur. When the saboteur is not activated, it just feeds through the output signals of the target component without manipulation (i.e., the simulations will be the same as for the original *VHDL*-model), but when the saboteur is activated. Then the output value will be the possibly manipulated value $(M(X,Y))$, i.e., an error may have been injected. Through the activation signal, the saboteur can be activated and deactivated at any time. Thus, transient faults can be modeled by activating a fault only once (generating one error), and persistent faults by having the saboteur activated for a longer period of time.



(a)



(b)

Figure 5.10: The structure and architecture of the implemented saboteurs.

Table 5.3: The saboteur control signals.

Signal	**Description**	**Type**	**Range**
sab_act	Signal for activating and deactivating the saboteur.	Boolean	False-True
prop_prob	Signal determining the error propagation probability for the injected fault.	Real	0.0-1.0
fault_mod	Signal determining which manipulation to use (set-0, set-1, or bit-flip).	Integer	0-2
seed	Signal that is used to produce the random generator seed so that the input values that propagate an error will differ between each injected fault.	Array(1 to 2) of real	[0.0, 0.0]-[1.0, 1.0]
xfault_prob	Signal determining the probabilities for single- and multiple- bit faults.	Array(1 to # of output signals) of real	[0.0, 0.0, ...]-[1.0, 1.0, ...]
bit_prob	Signal determining the probabilities for errors to propagate to a certain output bit.	Array(1 to # of output signals) of real	[0.0, 0.0, ...]-[1.0, 1.0, ...]

As seen in Chapter 5.2.1, it is mainly the architecture of the component that determines which input values that activate a specific fault and propagate the resulting error. This characteristic cannot be modeled at RTL without low-level information. Therefore, a random generator is used to determine which input values that should activate a specific fault and propagate the resulting errors. This means that even if the input values which activate faults will be different from real faults, the propagation probability and error manifestation, which often are the characteristics determining whether the error is detected or not (see Chapter 3.7) can be set to match errors caused by real faults.

If the target of the fault injection is a sequential component and if it is known which input signals that change the internal state of the component, then one can let the saboteur change states correspondingly. This means that a fault can be activated only for certain inputs, but also only in certain state(s). As an example, take a component with a simple state machine of three different states for which it is known when the states change (for example, the state can change for every positive clock pulse). If a saboteur should be used to model a fault that only affects one of the states, this saboteur also needs to include a state machine with three states. In this way the saboteur can change state correspondingly to the state machine and be activated only at the specific state (for the previously given example this would mean that the saboteur only is active after every third positive clock flank). However, if nothing is

known about how many states the component have or how it changes states, it will have to be treated as a combinational component.

To simulate faults accurately, we would like for each specific fault that each specific input value (when applied in the same state) always propagate errors in the same way. However, if another fault is injected, the propagation events for each input value should be changed. Therefore, the input signals of the target component are used as the seed to the random generator. This implies that a new number (between 0.0 and 1.0) will be generated for every new combination of the input signals, but each time the same combination occurs, the same number will be regenerated. The generated random number (R1) is compared with the propagation probability set for the injected specific fault (*prop_prob*) to determine whether an error should be propagated or not. This decision is communicated to the manipulator through the signal P (see Figure 5.10(b)). To enable the input values that propagate an error to be changed between each specific injected fault (independently if they have the same error propagation probability or not), the control signal, *seed* (specific for each fault), was added. This signal together with the input values of the component are used as the seed to the random number generator.

In order to make propagating errors manifest them-selves in the output signals according to the desired, set, characteristics (to determine which of the output signals of the target component that should be manipulated by the saboteur), two new random numbers (R2 and R3) were generated. The first one is used for deciding how many of the output signals that an error should propagate to (i.e., single- or multiple- bit errors). The second one is used to determine which bit(s) the error should manifest itself in. The actual decisions are made by comparing the generated numbers with the values of signals determining the desired characteristics for the specific fault (*xfault_prob* and *bit_prob*) set by the developer.

Another issue is how to manipulate the output bits. This was not investigated in the performed experiments of Chapter 5.2.1. Therefore, three different standard ways were implemented: bit-flipping, setting the bits to 0 and setting the bits to 1. Which method that is used for a specific fault is set through the control signal *fault_mod*. It should be noted that, if manipulation by setting bits to 0 or 1 is used, the error propagation probability will be lower than the set value (*prop_prob*), since if the bit has the same value as the injected value, the error is masked.

5.4.3 A Saboteur Fault Injection Campaign

In the previous section, the architecture of a saboteur for injecting faults whose characteristics can be controlled through specific control signals, was described. In

this section how to set up a fault injection campaign is demonstrated. First, in order to run a fault injection campaign, the developer has to choose the components that should be the targets for the fault injection, i.e., the components that are of interest to model faults inside. Then the developer has to determine the characteristics of the faults he/she would like to inject, i.e., make a file containing the values of the saboteur control signals (the signals of Table 5.3) for each specific fault to inject (to help the developer to get the right format for the file, we developed some simple *MATLAB* scripts).

After that, the developer needs to select which data to store from the fault injection experiments. This can be performed using the simulator tool in the same way as for standard simulations.

After the target-components have been selected, the characteristics of the faults for injection have been set and the data to store has been selected, the injection process itself can be started. The first step in this process is to insert the saboteurs according to Figure 5.10(a), by breaking up the connections between the output signals of the target components and the connected components. As the number of input and output signals are known, this step can be automated by mapping the saboteurs from a generic architecture with support from a *VHDL*-simulator. It should be noted that by inserting all saboteurs at the same time, the modified *VHDL*-model only needs to be compiled once. Single faults can still be injected since it is possible to activate and deactivate the saboteurs separately, and for event-triggered simulators the simulations are not slowed down, see Chapter 5.6.

Then, to start the campaign, a script (written in *Tcl*)controlling the simulator is run. The script first performs a golden run experiment, where all saboteurs are deactivated. After that, the script restarts the simulation, collects the values of the saboteur control signals for the first fault to be injected from the file created by the developer and performs the simulation. When finished, it restarts, collects the signal values for the next fault to inject, performs the simulation, and so on until the end of the file is reached, i.e., until the last fault has been simulated.

5.5 Reducing the Number of Error Cases

A major problem with the proposed approach is how to chose which characteristics of errors that should be investigated as it is not practical possible to investigate all combinations. This section will give suggestions on how to accomplish this.

Looking at the error *magnitude*, it is generally reasonable to assume that if it is known that a certain magnitude will generate a failure, then all errors with higher magnitudes will also result in failures if other characteristics remain unchanged. It

is generally also reasonable to assume that in most cases, errors that are repeated frequently, will affect the system more than errors that are repeated less frequently (under the assumption that the other characteristics remain unchanged). Moreover, it is also the case that the system generally is more sensitive to errors occurring at operating points that are very dynamic (meaning that state of the system is changing), compared to operating points that are static (meaning that the state of the system is unchanged). Also, it is seldom the case that two or several errors compensate each other, which means that multiple errors generally are more severe than single. Furthermore, certain properties often have a greater influence on the systems behavior than others.

Using this information, the number of cases to investigate the impact of errors can be reduced. However, it is important to note that these only are rules of thumb and may not be valid for every case. Therefore, these rules need to be used with care and random generation of error cases is a good complement to confirm the validity for the specific system.

It is also sometimes possible to though analysis determine that certain errors will have the same or similar effect. If this is possible, errors can be divided into classes in which all errors will have the same or similar effect of which it then is enough to investigate one error from each class. This is termed fault collapsing and has for instance been proposed in [Parotta *et al.*, 2000].

If the developer has some in advance knowledge about the faults that can occur or the robustness of the system, this can of course be utilized to focus the injections to faults with specific characteristics. As an example, if it is known that the system is robust to transient faults, the injections can be focused mainly on persistent faults. However, if the developer has little knowledge about the characteristics of the faults that can occur and/or what impact they will have, it can be wise to first run a small fault injection campaign with evenly distributed fault characteristics in order to identify the faults which need to be investigated in more detail. Then further campaigns can be focused to these faults.

5.6 Accuracy and Complexity Evaluation

In this section the results from fault injection campaigns, of a RTL adder, shifter and multiplier with saboteurs are presented. In each campaign 210 faults were injected, and for each fault, 1024 computations were performed, i.e., 1024 different input vectors investigated. For a real campaign, the system developer would simulate the whole microprocessor at RTL, with the saboteurs inserted at desired locations, running the system application software. *However, the intention of these campaigns*

is not to investigate the impact of different data errors or to evaluate any detection techniques, but to verify that the implementation of the saboteur meets the desired properties defined in Chapter 5.4.1.

The control signals of the saboteurs for this campaign were set (see Table 5.4) to generate faults with varying characteristics, resembling real faults in such components. *However, the characteristics were not directly set according to any of the*

Table 5.4: The properties of the injected faults.

Signal	**Values for the Adder Fault Campaign**	**Values for the Shifter Fault Campaign**	**Values for the Multiplier Fault Campaign**
sab_act	True (Permanent faults were simulated.)	True (Permanent faults were simulated.)	True (Permanent faults were simulated.)
prop_prob	Starting with 1.0 decreasing with 0.05 for every tenth fault down to 0.0 (totally 210 faults)	Starting with 1.0 decreasing with 0.05 for every tenth fault down to 0.0 (totally 210 faults)	Starting with 1.0 decreasing with 0.05 for every tenth fault down to 0.0 (totally 210 faults)
fault_mod	2 (bit-flipping)	2 (bit-flipping)	2 (bit-flipping)
seed	Selected randomly for each fault.	Selected randomly for each fault.	Selected randomly for each fault.
xfault_prob	[0.6, 0.2, 0.1, 0.05, 0.025, 0.015, 0.01, 0.0, ..., 0.0]	[0.9, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.01, 0.0, ..., 0.0]	[0.5, 0.2, 0.15, 0.1, 0.025, 0.015, 0.01, 0.0, ..., 0.0]
bit_prob	[0.0375, 0.0525, 0.060, 0.0625, 0.0625, 0.0625, 0.0625, 0.0625, 0.0625, 0.0625, 0.0625, 0.0625, 0.0625, 0.0625, 0.0625, 0.0375]	[0.03125, 0.03125, ..., 0.03125]	[0.001, 0.002, 0.003, 0.005, 0.009, 0.015, 0.02, 0.025, 0.035, 0.04, 0.045, 0.05, 0.055, 0.06, 0.065, 0.070, 0.070, 0.065, 0.06, 0.055, 0.05, 0.045, 0.04, 0.035, 0.025, 0.02, 0.015, 0.009, 0.005, 0.003, 0.002, 0.001]

gate-level simulations in Chapter 5.2.1, since, for real campaigns, this information: (a) is generally not available, (b) may differ between different component implementations (compare the results from the Thor*-components with the ones from the* c6288 *and the 74238) and (c) may differ from real faults.*

An example of information that we believe is generic, and thus, used for the simulations, is that the manifestation probability is highest for the middle bits of

multiplier. However, for specific systems, the developer will use his/her knowledge of which faults are likely to occur and focus the campaigns to those faults, see Chapter 5.4.3.

5.6.1 Simulation Accuracy

The results of the campaigns can be seen in Figures5.11-5.18. Figure 5.11 shows the number of output errors each injected fault caused for the campaign. From these results, the error propagation probabilities can be computed (as was done for the RTL and gate-level experiments and illustrated in Figures 5.1-5.3), but we chose not to do so as it would have made it more complicated to verify that the results correspond to the desired set values in Table 5.4. As can be seen the number of errors ramps down according to the desired set signal, *prop_prob*, value with a small variance due to that not all possible input values were computed. *Therefore, we conclude that the approach makes it possible to inject errors with any error propagation probability, which is necessary to be able to evaluate the impact of all types of real faults. This is not supported by other approaches*.

Figure 5.12 confirms that the number of propagated faults is similar for all input values (the number varies between 83 and 126), which was an assumption made based on the results in Chapter 5.2.1.

Figures 5.13-5.17 show the number of single-bit errors, double-bit errors etc. and to which output bits an error propagated for the saboteur campaigns. It can be seen that the results of the saboteur campaigns correspond to the values set through the control signals, *xfault_prob* and *bit_prob* in Table 5.4. It is important to note that Figures 5.11-5.17 should not directly be compared with the results of the previous simulations (illustrated in Figures 5.1-5.9) as we set the saboteur error characteristics, and thus, these characteristics only imitates real physical values. However, we believe that it is possible to, through the saboteurs, closely resemble the characteristics of real errors. *Thus, the real merit with this approach is the possibility for system developers to tune the fault characteristics arbitrarily, so that the faults that they believe are of highest importance to investigate can be injected, based on general information, and knowledge of the specific system, without requiring low-level descriptions*.

5.6.2 Simulation Complexity

In this section the complexity of the different approaches are investigated and compared through simple analysis and measurement of their simulation times. Most *VHDL*-simulators are event-triggered, i.e., the value of a signal is re-evaluated only

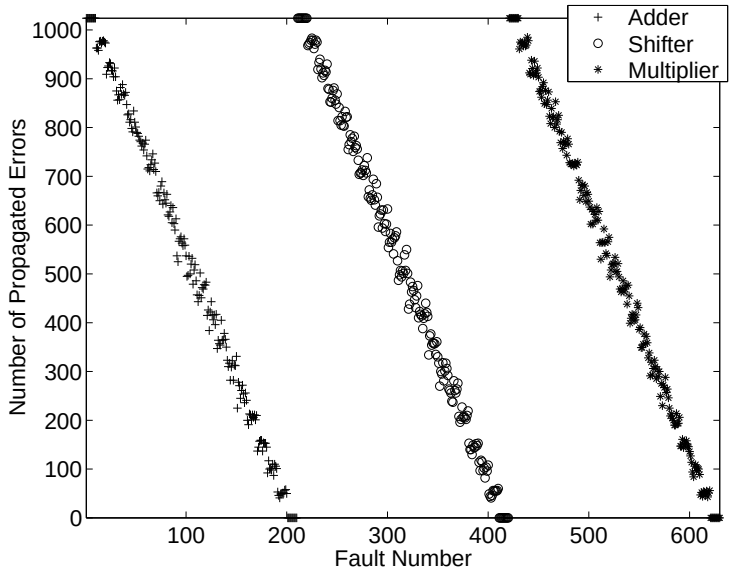


Figure 5.11: The number of times each fault was activated for the different campaigns.

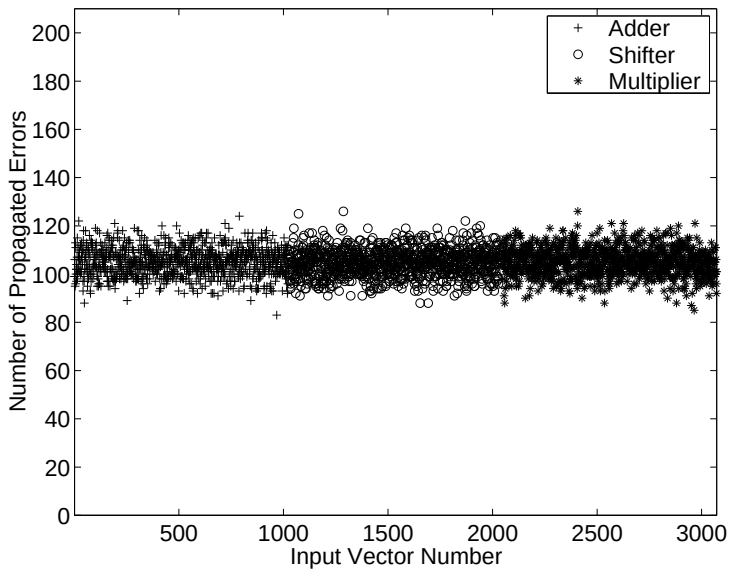


Figure 5.12: The number of times a fault was activated for each applied input vector.

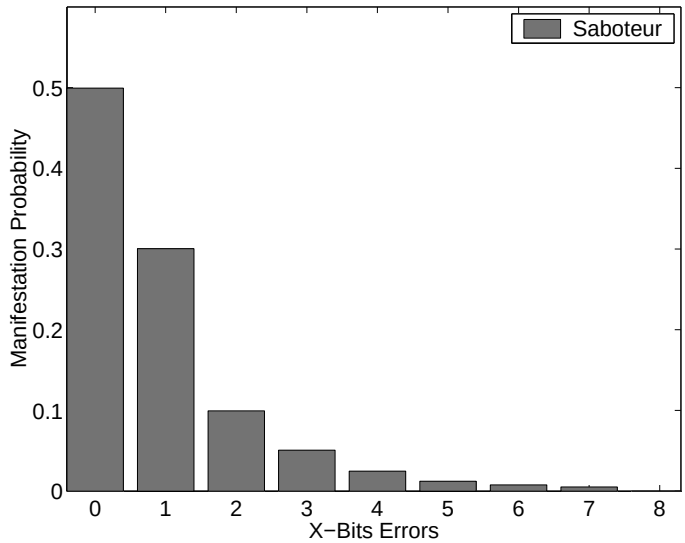


Figure 5.13: The single-, multiple- bit manifestation probability distributions for the adder saboteur.

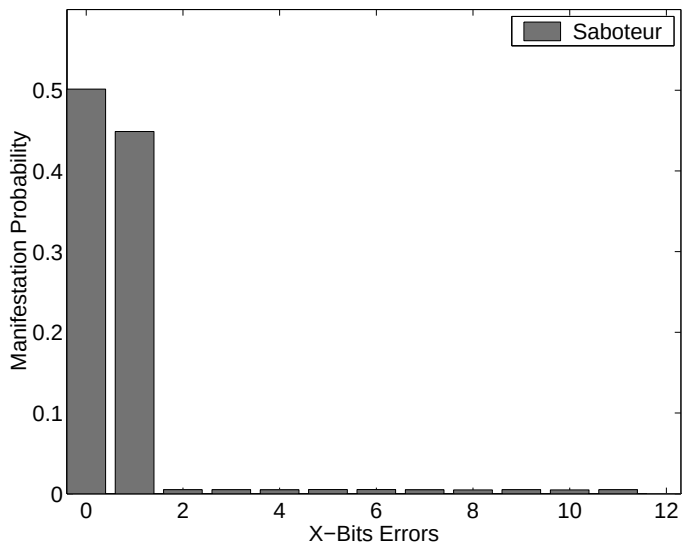


Figure 5.14: The single-, multiple- bit manifestation probability distributions for the shifter saboteur.

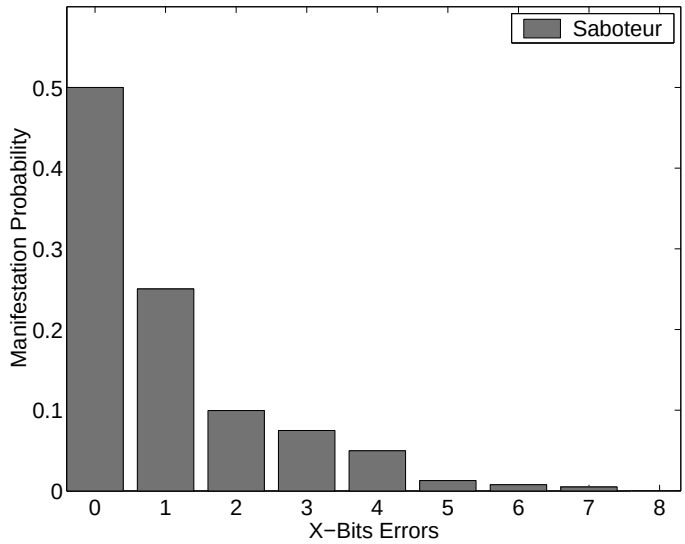


Figure 5.15: The single-, multiple- bit manifestation probability distributions for the multiplier saboteur.

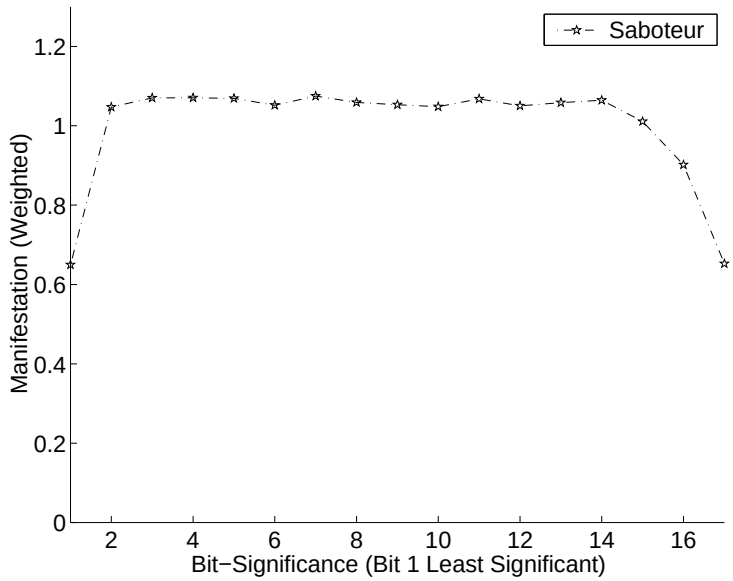


Figure 5.16: The output-bit manifestation probability distributions for the adder saboteur.

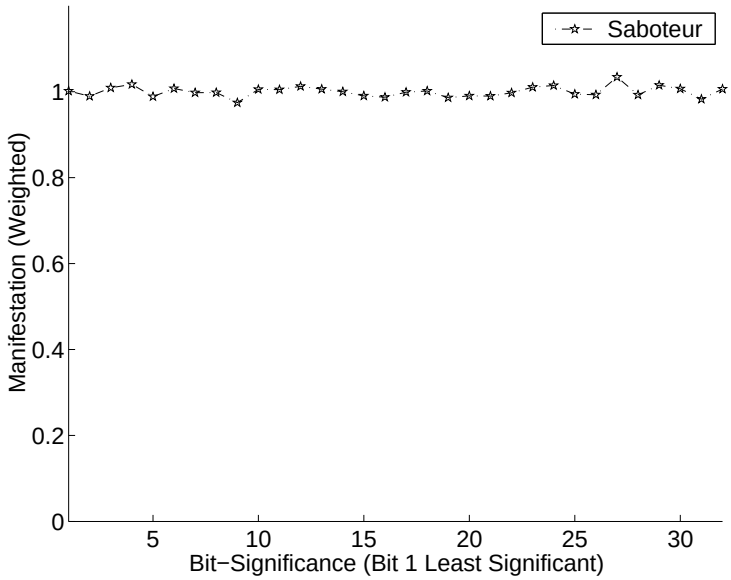


Figure 5.17: The output-bit manifestation probability distributions for the shifter saboteur.

when any of the signals its value depends on is changed (compared to time-triggered, when the values of all signals are re-evaluated with even intervals). This means that the simulation complexity is dependent on how many signals that are simulated and how they depend on each other. To understand this, compare the case when all signals depend on each other, with the case when there are no dependencies. In the first case all signals have to be re-evaluated as soon as one signal is changed, whereas for the second case, only the signal itself is changed and no other signals have to be re-evaluated. Therefore, if an event-triggered *VHDL*-simulator is used, all saboteurs can be inserted at the same time without slowing down the simulations because when a saboteur is deactivated, none of its signals need to be re-evaluated. Thus, the *VHDL*-model only needs to be recompiled once.

For the multipliers, a lot of dependencies exist (which was indicated by the fault injection experiments in Chapter 5.2.1, showing that changing one input bit often led to a change of several output bits) and thus, many signals need to be re-evaluated when the inputs are changed. For the adders and barrel-shifters, the dependencies are weaker. As it is hard to measure the exact dependencies, the number of signals that the components consist of is used as a simple measurement of the complexity,

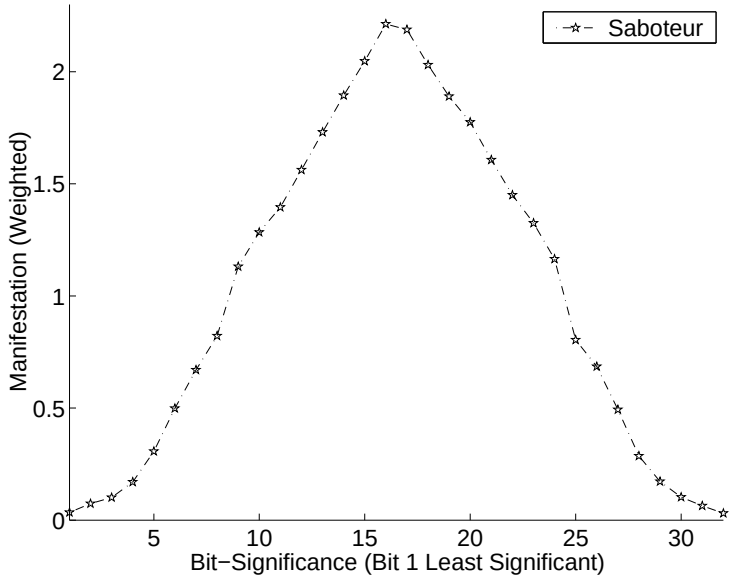


Figure 5.18: The output-bit manifestation probability distributions for the multiplier saboteur (to visualize patterns between the multipliers, they have been centered).

see Table 5.5. As another estimation of the complexity difference between gate-level and RTL simulations the approximate simulation times for the components simulated were measured (we used the the Unix command time). The approximate average times (for simulation of one input value) can be seen in Table 5.5. Since the simulations of the *74283* and *c6288* were run on a different simulator, no simulation times for these components were put in the table. The approximate increase in complexity due to the use of saboteurs (compared to the clean RTL components) has also be listed in the lower right corner of the table.

It can be seen that the relative increase in number of signals is much higher than the relative increase in simulation times. This can be explained by the fact that only a few of the added signals depend on other signals, and thus, few signals need to be re-evaluated. However, even if the complexity is increased, we believe that this will be affordable for most applications, especially when comparing it to the cost for gate-level simulations. Thus, our approach meets the desired properties defined in Chapter 5.4.1.

Table 5.5: Complexity order and simulation time for the different simulations.

Component	**Number of Bit-Signals**	**Average Simulation Time (ms)**		
74238 Adder	45	-		
Thor Adder	1916	2.50		
Thor Barrel-Shifter	4175	3.71		
c6288 Multiplier	2448	-		
Thor Multiplier	58774	141.10		
RTL Adder	49	0.51		
RTL Barrel-Shifter	97	0.41	**Increased Complexity %**	
RTL Multiplier	64	0.54	**Bit-Signals**	**Average Simulation Time**
RTL Saboteur & Adder	105	0.58	115	14
RTL Saboteur & Shifter	198	0.68	105	66
RTL Saboteur & Multiplier	165	0.69	158	28

5.7 Estimating the Error Detection Coverage of Double Execution

In order to evaluate and illustrate the capacity of the previously presented saboteur approach, the detection coverage of double execution is estimated, in this section, through gate-level, RTL, and saboteur simulations. The results from the different simulations are discussed and compared.

Double execution, i.e., executing a task twice and comparing the results of the two executions, is a common proposed error detection technique based on time redundancy. It has mostly been used for detection of data errors caused by transient faults, but as seen in Appendix B, data errors caused by persistent faults may also be detected. In [Aidemark *et al.*, 2003], the coverage of double execution for detection of errors caused by permanent faults where investigated with simulation-based fault injection of stuck-at faults at gate-level. In this section, these results will be compared with the results from stuck-at faults injected at RTL and faults injected through saboteurs. First, the three different simulations are described in detail. Then, the results are presented and discussed.

5.7.1 Simulation Details

As was discussed in Chapter 2 and 4, the functional units are some of the most vulnerable parts of commercial microprocessors. Double execution is generally applied to detect such faults. Therefore, it is desirable to use a workload that exercise such units. Moreover, double execution is a systematic error detection technique as it compares the two results of a task that is executed twice, i.e., it is independent of the actual application of the task. Furthermore, the interest of this study was primarily to compare different coverage estimation approaches. Therefore, we chose to use a simple calculation as workload, in this case a simple integer matrix multiplication: $[a_1\ a_2] \cdot \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} = a_1 \cdot b_1 + a_2 \cdot b_2$. Even if this is a very simple workload, many control algorithms consist of similar computations.

In order to avoid over-flow, the input data (a_1, a_2, b_1, and b_2) were randomly chosen in the interval $\left[0 \ldots \sqrt{\frac{max(Integer)}{2}}\right]$. For all simulations, 50 ($50 \cdot 4 = 200$ different input data) different matrix multiplications were calculated.

This task (workload) was simulated in *VHDL* (the ModelSim simulator was used, [Mentor Graphics 1998]) on a RTL model of the *Thor*-processor, [Saab Ericsson Space, 1999]. Even if *Thor* is not a commercial processor, the functional units are of similar architecture as can be seen in Table 5.1.

As seen from the workload, the adder and multiplier are the functional units that are exercised by the workload[2]. However, to simplify the fault injections, faults were only injected in the multiplier.

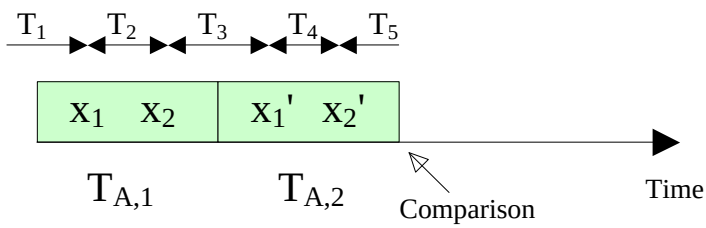


Figure 5.19: Time redundant execution of the matrix computation.

Figure 5.19 illustrates the matrix computation (the execution of task T_A in the figure). The first execution is denoted: $T_{A,1}$, and the second: $T_{A,2}$. The first and

[2]The assembler code resulting from the compilation was checked to verify that add and multiply instructions were used.

second multiplication of the task is marked as x_1 and x_2 and the intervals T_1-T_5 the times before, between, and after the multiplications. As described in Appendix B, faults occurring before the first multiplication of the first execution or after the second multiplication of the second execution cannot be detected by double execution as they will affect the two executions identically. Therefore, as the double executed workload only consists of totally 4 integer multiplications (two multiplications in each of the two executions), faults were injected at only three different times: between the first and second multiplication of the fist execution, T_2, between the second multiplication of the first execution and the first multiplication of the second execution, T_3, and between the first and second multiplication of the second execution, T_4.

As discussed previously, three different simulations were performed, namely:

- Gate-Level Simulations

  In these simulations, a structural gate-level description of a multiplier with Booth recoding and Wallace tree, was used. Stuck-at-faults were injected in 246 of totally 24635 signals (both as stuck-at-0 and stuck-at-1).

- RTL Simulations

  In these simulations, only the input and output signals of the multiplier were reachable for injection of faults. Stuck-at faults (both as stuck-at-0 and stuck-at-1) were injected in all input and output signals, totally 96 signals.

- Saboteur-Based Simulations

  Here, saboteurs were inserted after the RTL-multiplier as described earlier in this chapter (see Figure 5.10). Due to that errors generally only are detected as a difference in the results between the two executions, the detection coverage of the double execution technique is not dependent on the magnitude of the error. Therefore, in order to simplify the estimations, errors were only injected as single-bit errors in the least significant bit of the results. Thus, the only error characteristics that were varied were the property, the *state* (as described earlier in this chapter) and the *repetition* frequency (which is dependent on the activation probability of the fault) which was varied between 0 and 1 in 0.1 steps with 10 faults injected at each level, i.e., the total number of different faults injected were $11 \cdot 10 = 110$. The reason for this choice of characteristics were that we wanted a first view on the detection efficiency of double execution for faults generating errors with any *repetition* frequency.

5.7.2 Results

The results from the three different experiments can be found in the Tables 5.6, 5.7, and 5.8.

Table 5.6: Results from the gate-level simulations.

Injection Time Interval	T_2	T_3	T_4
Total Number of Injections	24600	24600	24600
Correct Result	40.7%	40.8%	49.3%
Overfbw	45.7%	45.6%	39.8%
Detected	9.2%	13.6%	10.9%
Incorrect Result	4.4%	0%	0%

Table 5.7: Results from the RTL simulations.

Injection Time Interval	T_2	T_3	T_4
Total Number of Injections	9600	9600	9600
Correct Result	17.25%	17.25%	33.46%
Overfbw	56.82%	56.82%	41.46%
Detected	18.96%	25.93%	25.08%
Incorrect Result	6.97%	0%	0%

Table 5.8: Results from the saboteur simulations.

Injection Time Interval	T_2	T_3	T_4
Total Number of Injections	5500	5500	5500
Correct Result	18.64%	21.56%	28.20%
Overfbw	0%	0%	0%
Detected	71.95%	78.44%	71.80%
Incorrect Result	9.42%	0%	0%

The Correct Result-rows contain the percentage of the injected faults that were ineffective. The Overflow-rows correspond to the number of the injected faults that raised an overflow exception. The Detected-rows contain the percentage of errors that did not raise an overflow-exception, but were detected by the double execution technique. Finally, the Incorrect Result-rows contain the percentage of errors that did not raise an overflow exception and were not either detected by the double execution technique. The reason why the overflow row is included is that it was not

possible to deactivate this exception, and that it when triggered sometimes generated an exception before the task could be executed a second time.

As can be seen when comparing the numbers in the Tables 5.6, 5.7, and 5.8, the results differ very much among the used fault injection techniques. Comparing the gate-level and RTL simulations, based on the previous discussions in this chapter, the activation frequency of the injected faults are likely to be the cause of the difference. At gate-level, a high number of locations are reachable for fault injection. However, at RTL, only the input and output signals are reachable for injection, and faults at these locations generally are more likely to be activated and propagated further than internal faults. Therefore, it would be expected that the RTL simulations have a lower number of correct results, which also is the case when comparing the Tables 5.6 and 5.7.

In order to verify this, the fault activation probabilities for the injected faults were estimated. The distributions can be seen in Figure 5.20 and the fault activation averages were: gate-level:0.51 and RTL:0.67[3]. As can be seen, the activation probabilities are higher for the RTL injections than for the gate-level. It can be seen that the RTL simulations do not have any activations between 0.02 and 0.3. Thus, the efficiency of detecting errors generated by such faults cannot be evaluated from the RTL simulations.

It can also be seen that the activation probabilities for the faults are higher than what was seen in Chapter 5.2.1. This depends on that the multiplier in the *Thor*-processor is based on sequential execution, i.e., the multiplication is divided into three sub-multiplications, which results are shifted and added to build the final result. Therefore, multiplier faults can be activated three times for each performed multiplication instruction, and thus, the activation probability will be higher.

To summarize, there is a big difference between the results of the gate-level and RTL simulations.

Now, looking at the results of the fault injections using the saboteur, there is little use of comparing the average numbers in Table 5.8 to the other injections, as one through the saboteurs determined the error characteristics. However, one major difference can be seen, the saboteur do not generate any overflow exceptions. The reason for this is that we chose to inject the errors in the least significant bit, and thus, overflow exceptions are not triggered. This shows that the saboteur fault injections have a major advantage as the error characteristics of the injected faults can be tuned to trigger specific error detection techniques. However, it should also be noted that the number of errors that are masked due to repeated multiplications is

[3]the average for the saboteur was 0.72, but as the fault activation probability was controlled for this campaign, this value cannot be compared with the averages of the other campaigns.

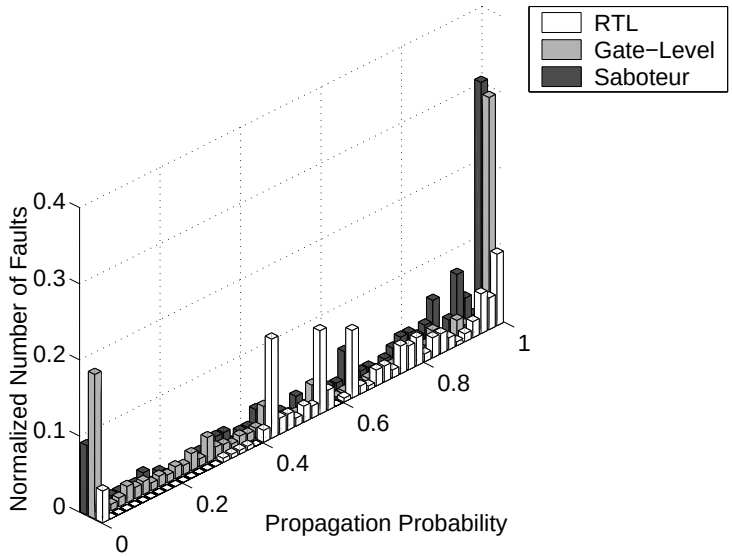


Figure 5.20: The activation probabilities for the injected faults.

increased (i.e., at the first multiplication, the fault is activated and generates an error, but this error is masked after the second multiplication). This can be seen by that number of correct results is higher when the faults were injected at time T_2 than for the ones injected at time T_3. This depends on that the injected fault always flipped the least significant bit, and through two multiplications this could mean that for the first multiplication, the least significant bit of the result is flipped from 0 to 1 and for the second multiplication, the least significant bit of the result is flipped from 1 to 0. Thus, when adding the two results, the final result will still be correct.

However, the major advantage with the saboteur approach is that the error characteristics arbitrarily can be varied to determine at which points (for which characteristics) the system is least/most sensitive, and/or the error detection technique is least/most efficient. To illustrate this, Figures 5.21-5.23 shows the error detection probability of double execution for faults with different activation frequencies (error *repetition* frequencies).

As can be seen, the detection coverage increases with the propagation probability of the fault, except for the faults with high activation probability, injected in time-interval T_3. This depends on the fact that some of the errors are masked (as discussed previously in this section).

Another difference is that the average of the time-interval T_3 increases steeper,

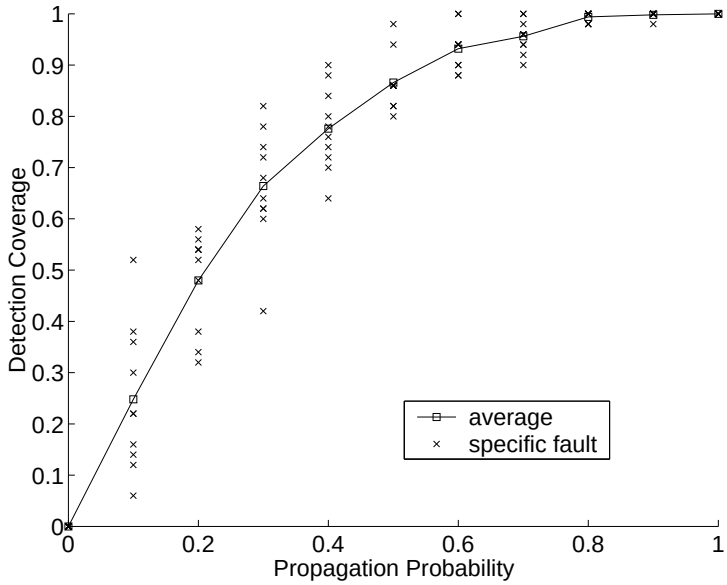


Figure 5.21: The detection coverage for the faults injected with saboteurs at time T_2.

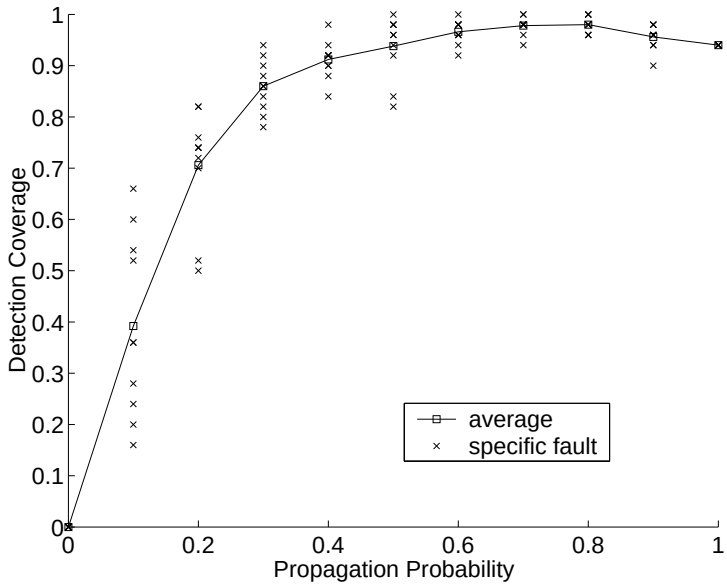


Figure 5.22: The detection coverage for the faults injected with saboteurs at time T_3.

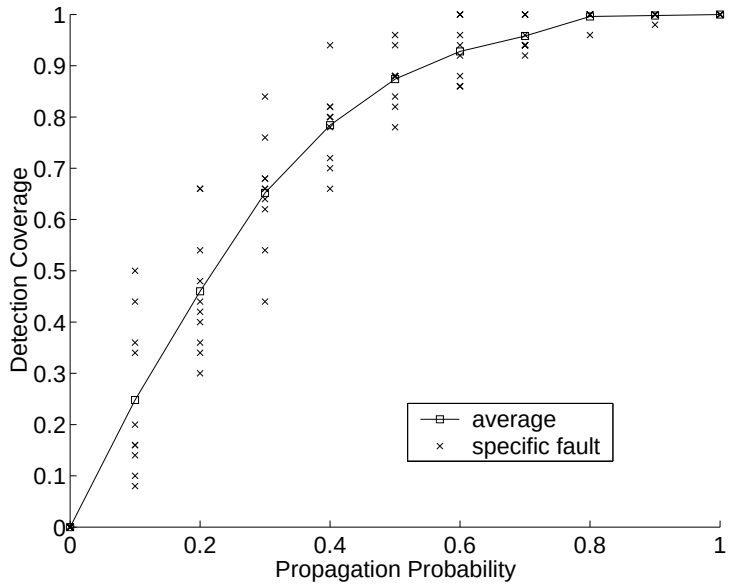


Figure 5.23: The detection coverage for the faults injected with saboteurs at time T_4.

which depends on the fact that faults injected into this interval can be activated at two times (multiplications), and thus, be detected, whereas for the other two intervals (T_2 and T_4), there is only one time (multiplication) that the faults can be activated. If the multiplier had not been sequential, one would expect the T_2 and T_4 average curves to be linear and the T_3 average to be a second degree curve. However, now, the multiplications are performed in three stages, which implies that each fault can be activated at three different times for each multiplication. Then, the average will be third degree curves (T_2 and T_4) and sixth degree curve (T_3) respectively.

5.7.3 Complexity

In order to determine the complexity of the different simulations, the number of signals in the different models and the simulation times, is listed in Table 5.9. The simulation times is an average of 5 simulations where 2 different calculations were computed and 11 faults were injected each time, that is, totally 2 golden runs were executed, and 22 faults were injected for each simulation. The number of signals of the RTL-model of the *Thor*-processor without the multiplier was 9374, i.e., the total number of signals for each simulation was 9374 + the number of signals found in the number of bit-signals column of Table 5.9.

Table 5.9: Number of simulated multiplier signals and simulation time for the three fault injection approaches.

FI Approach	Number of Bit-Signals	Average Simulation Times (s)
Gate-Level	24635	229
RTL	96	116
Saboteur	148	116

As can be seen, the complexity is not significantly increased for the saboteur simulations compared to the RTL. However, in this case only one saboteur was inserted, i.e., faults were only injected in one component, the multiplier. Furthermore, as faults were only injected as single bit-flips in the least significant bit, no distributions over the number of errors that should be activated and in which bits were needed, and thus, the saboteur was simplified compared to the one presented in Appendix E (mainly, the signals *x_faultprob* and *bit_prob* signals and the two while-loops were removed). Still, we believe that it would be possible to insert several saboteurs without increasing the simulation times significantly, specifically considering that in most cases only one saboteur is active at each time (only if faults in multiple components are simulated, more than one saboteur will be activated). As can be seen from the saboteur code in Appendix E, the VHDL-code can be optimized.

5.8 Summary

This chapter consists of mainly two parts: (1) measurements and comparisons of the characteristics of stuck-at faults modeled at gate-level and RTL, and (2) development and implementation of a saboteur-based *VHDL*-architecture for evaluating the impacts of faults with various durations and ascertaining the efficiencies of online error detection techniques at high-level simulations. The most important conclusion from the stuck-at fault injections is that the error propagation probability varies considerably across different faults, and also among abstraction levels. The reason for this is that fewer signals (locations) are accessible at high levels, which implies that fewer faults can be injected. As error propagation greatly influences the impact of faults, the stuck-at fault model is not accurate to use for effect/impact evaluations.

To facilitate more accurate evaluations, a *VHDL*-simulation approach was presented. The approach is based on insertion of specific components for injection of faults into *VHDL*-simulations, so called saboteurs. The novelty of this approach is

that it enables and describes how to compensate for the limited number of locations that are accessible for fault injection at high-level simulations so that the evaluation results will be accurate. Furthermore, the fault injection process can be automated.

The results from simulations using this new architecture showed that faults with varied time durations can be simulated, errors with any propagation probability can be modeled and the distribution of single- and multiple- bit errors can be tuned to fit real values, all within reasonable simulation time.

It was also shown that as it is possible to vary the characteristics of the injected faults through the saboteurs, the evaluations can be focused to trigger only certain error detection techniques. Another major advantage is that by using the proposed approach it is possible to determine the error characteristics for which the system is vulnerable/robust and errors detected/not detected.

Therefore, in our opinion, this approach is preferable for fault impact evaluations and evaluating the efficiencies of error detection techniques, compared to fault injections using the stuck-at fault model performed at RTL. Furthermore, we believe that this approach in many cases can give complementing results to gate-level simulations, since even if the stuck-at fault model much closer resembles real faults at this level, it may still miss some fault types, e.g., bridging faults, which can be investigated with the proposed saboteur approach.

5.9 Generalizations

The basic problem handled in this chapter is that it is hard to accurately model all types of faults at a high level where few signals are accessible. We tackled this problem by utilizing the property that for a fault to have any impact, the resulting errors must propagate to the output(s) of the component, and thus, eventually propagate through at least one accessible signal. Therefore, by manipulating the accessible signals according to the propagation of the errors resulting from the fault, it should be possible to model any faults, also those at locations which are not directly accessible (Chapter 5.4). As it is generally not practical to simulate all value combinations of the different fault characteristics, the number of simulated faults must be limited. This can be accomplished by (see Chapter 5.4.3):

- Dividing errors causing the same effects into sets in which only one error has to be simulated.

- Using knowledge on the fault effects for low-level components.

- System specific knowledge on which faults that can occur, and their impacts on the system.

Therefore, the basic idea with the proposed approach, i.e., top-down simulation of errors, can be applied at any abstraction level, for any type of simulations, for any type of components/systems, and also to scan-chain or software fault injection techniques, as long as the number of errors that are required to be simulated can be limited.

One direction for continued research is profiling of components, in order to reduce the number of faults that needs to be injected. This can be achieved through further simulation of components and in more detail investigating how much detail that can be obtained of the error propagation and manifestation only from information about the function of the component (without knowing how it is implemented), i.e., which characteristics are generic for components having the same function.

CHAPTER 6

Conclusions and Speculations About the Future

6.1 Conclusions

The target of our research has been on how to develop fault-tolerant microprocessors for use in mass market safety-critical systems, e.g., x-by-wire systems in automobiles. Thus, the research has been performed from the perspectives of developers of such systems. This means that not only the fault tolerance efficiency has to be considered, but also the costs (mainly recurring costs), portability, scalability and performance, power, memory overheads.

This thesis has focused on investigating development of fault tolerant microprocessor by adding software-implemented tolerance techniques to commercial processors. Such solutions have the following advantages:

- The only necessary hardware is the commercial processors, which implies low recurring costs.
- Commercial processors have higher performance than custom designed fault-tolerant processors.
- Commercial processors are generally employed in many different type of systems, decreasing the probability of running into previous undetected development faults.

- As the fault tolerance is based on software, the solutions should be portable and scalable.

One major problem with such development is that the low-level details of commercial processors are generally concealed by the manufactures which makes it hard to utilize on-chip tolerance techniques, and to evaluate the tolerance efficiency of these solutions.

Another problem is that the additional software introduces overhead resulting in reduced performance, increased memory usage, increased energy consumption, etc. These problems were discussed in this thesis.

First, the architecture and implementation trends of commercial processors were analyzed to determine which faults such processors are likely to be exposed to, in the future (Chapter 2). We believe that the complexity increase of such processors will increase the number of developmental faults undetected by testing. Furthermore, the transistor size reduction and the increase in number of transistors on chip, are likely to increase the vulnerability of the processors against disturbances and result in reduced life-times. This implies that the fault intensity of commercial processors will increase, not only for transient faults, but also for persistent faults (including for instance elusive developmental and permanent faults). Thus, the applied software-implemented techniques must provide tolerance for both transient and persistent faults.

Based on the results of the analysis, suitable error detection and recovery techniques were discussed in Chapter 3 (based on the survey in Appendix A). It was suggested that simple “I am alive” messages would be suitable for detection of processor crashes. For monitoring that the correct instructions are executed, use of signature checking complemented with watch-dog timers were proposed, and for detection of data errors, executable assertions, double execution, and self-tests were suggested.

For recovery, when a crash failure has been detected, the tasks handled by the processor should be set in fail-safe states or be handed over to spare computers. Then attempts to restart the processor can be made. Detected control-flow errors should be recovered from by recomputing the task or start with the next task. However, if several control-flow errors are detected in short time (the detections need to be logged), it may be necessary to set the tasks fail-safe states or hand over them to spare computers so that more accurate fault diagnosis can be performed.

Which recovery of data errors that is necessary is very dependent on the specific system. For instance, many sporadic data errors have limited effect on control systems. Therefore, it is desirable to only detect errors that requires recovery. For data errors that require recovery, the results can initially be discarded and the task,

if necessary, recomputed. However, if several data errors are detected in short time, it may be necessary to also set the tasks in fail-safe states or hand over them to spare computers, and perform more accurate fault diagnosis.

When a suitable design has been proposed, evaluations are necessary to determine if it is efficient enough, i.e., meets the specified requirements. Such evaluations of the solutions for detecting and recovery from crash failures and control-flow errors should be possible to perform, even when only high-level models are accessible, as the effects and characteristics of these errors are rather similar (a crash failure simply means that the processor is not executing and a control-flow error that incorrect instruction is executed and both these errors generally have sever impact on the system). However, evaluations of data errors are much harder as their effects and characteristics varies a lot.

In order to find a suitable method for evaluating the effects of data errors, we defined their effects on systems to depend on the following characteristics:

- Which property that is erroneous (**Property**).
- The number of properties that are erroneous (**Number**).
- The difference between the correct value and the erroneous value (**Magnitude**).
- The occurrence time (**State**).
- How often the error repeats itself (**Repetition**)

As most previous evaluation methods either are qualitative (e.g., fault tree analysis), focused on failure analysis (e.g., Markov modeling), or requires low-level details (fault injection), analysis methods for determining the effect of data errors on linear control systems were developed in Chapter 4. Previous proposed such methods have focused on identifying which errors that result in lost stability. However, as some failures may result in catastrophic consequences even if the stability is not loss we instead focused the analysis to determine how much and for how long time the controlled physical property differentiates from the desired value, due to different data errors.

As it was necessary to restrict the analysis, only the effect of data errors in the control signal and the controller states were studied. However, in order for a data error to have any effect on the controlled physical property, it must eventually propagate to any of these signals.

It was also shown that the analysis methods can be used to design and determine the efficiency of applied executable assertions.

The main advantage with the analysis methods are that they can be applied early in the development process when only models of the control system exist. We believe that such early information will be valuable for developers of safety-critical systems.

As the analysis had limitations (it only targeted control systems, had limitations in which characteristics of data errors that could be analyzed, and cannot be used for detailed verifications) simulation-based fault injection approaches were investigated in Chapter 5. First it was shown that the generated error characteristics of injected stuck-at faults depends on at which abstraction level they are injected at. This implies that there is a potential risk that the results from evaluations become inaccurate, as detailed models of commercial processors seldom are accessible.

Therefore, a new top-down fault injection approach was developed. Instead of trying to imitate as many low-level faults as possible, which is the goal of most traditional approaches, the method instead focuses on enabling variation of the characteristics of data errors (defined above) in the reachable signals. This will cover the important faults, as for any data error to have any effect, it must eventually propagate to any of the reachable high-level signals.

However, the problem is to select which characteristics of errors to investigate, as it is not possible to test all combinations and since the exact characteristics of real errors generally are not known. A suggestion to handle this problem is to first run a small campaign where all characteristics have sparse, but large, variations. From these initial results, which characteristics to investigate in greater detail may be identified.

We believe that this approach will be useful, at least as a complement to traditional bottom-up approaches, as it also can be applied early in the system development process, reduces the risk of missing relevant fault cases, and can be used for both estimating the effects of different data errors, as well as for ascertaining the efficiency of applied fault tolerance techniques.

We have in this thesis identified problems with developing fault tolerant microprocessors, by adding software-implemented techniques to commercial microprocessors. Moreover, we have identified suitable software-implemented tolerance techniques and developed methods for evaluating the efficiency of such solutions. Thus, to conclude, we believe that this thesis provides an essential step towards a framework for developing fault-tolerant processors based on commercial microprocessors with software-implemented tolerance techniques.

6.2 Speculating About the Future...

So far, we have mainly discussed today's requirements on fault tolerant microprocessors. In this section, we speculate of how these requirements may change in the future, and how this affects the results of this thesis.

As the cost/performance ratio continues to decrease for microprocessors, distributed computer control systems become more prevalent. Today, the communication network bandwidth is the major bottleneck for this development, forcing most computations of data to be performed in the nodes connected to the data source, i.e., at local nodes. Therefore, today, fault tolerance at computer node level is very important.

However, the communication bandwidth may not be a problem in the future. For instance, optical networks may be developed further so that they can be used in this type of systems. This would enable the systems to be distributed further as any data could be computed at any computer node without any major penalties. Thus, the microprocessors in such distributed systems could be seen as a multiprocessor system. This would clearly simplify recovery as tasks easily could be rescheduled to run on another microprocessor when a certain processor has failed.

This would also make it possible to re-execute tasks at different nodes to detect errors. However, as discussed previously in this thesis, duplex systems convey the problems of determining which unit that is the erroneous, synchronizing the units, and how to perform the comparison (the two last problems are also common for N-Modular Systems (NMR)). Therefore, even for such systems, fault tolerance on the individual nodes can be advantageous, at least fault diagnosis.

Another question for the future is efficient design for fault tolerance. Today, the largest market for microprocessors is for high-performance processors even though embedded system specific processors are emerging. However, as the market for embedded and fault-tolerant processors are likely to increase as the performance/cost ratio increases, the total market will be more evenly distributed, resulting in that more processors focused on these area are being developed.

One complicating issue for the development of fault-tolerant commercial processors is that the developers of safety-critical systems must be able to trust that the processors are dependable enough, i.e., must be able to evaluate them. As microprocessor manufacturers generally conceal low-level details to avoid their solutions to be copied, it is not evident how such evaluation should be performed so that both parts are satisfied.

A similar problem for microprocessors with extensive fault-tolerance provided on-chip, is which information that should be communicated to the system level. In

many cases, it is possible to perform some recovery on-chip which means that the error occurrence may not be visible at higher abstraction levels. If such recovery is not developed with care, this may complicate system level recovery. Moreover, it is not clear how developmental faults can be tolerated on-chip.

Whether COTS processors with added fault tolerance techniques will be used in the future is of course not only dependent on how efficient other solutions will be, but also on which efficiency that can be reached with such solutions. This first need to be evaluated. Furthermore, it is not enough to be able to show that faults are tolerated to a high degree, but it must also be shown that the fault intensity of such processors, is not unacceptable high[1]. This is not clear because of their extreme complexity and scaling.

However, independent on how fault tolerant microprocessors will be developed in the future, we believe that the error effect evaluation techniques developed in this thesis still will be useful for system developers and researchers in the academia, mainly because that even if detailed information should be available, low-level evaluations are very time-consuming. Thus the proposed top-down approaches should provide valuable guidance at the early design stages.

[1]Even if a system would tolerate all faults, it would be of no use if the fault intensity is that high that all its resources always are allocated by the fault tolerance techniques.

Bibliography

[Abdelhay and Simeu, 2000] Abdelhay, A. and Simeu, E., "Analytical redundancy based approach for concurrent fault detection in linear digital systems", On-Line Testing Workshop, pp. 112 -117, 2000.

[Aftabjahani and Navabi, 1997] S. Aftabjahani and Z. Navabi, "Functional Fault Simulation of VHDL Gate Level Models", VHDL International Users' Forum, pp. 18 -23, 1997.

[Ahlström and Torin 2001] Ahlström K., and Torin J., "Future architecture of flight control systems", IEEE Aerospace and Electronics Systems Magazine, Vol. 17, Issue 12, pp. 21-27, 2002.

[Aidemark *et al.*, 2002] Aidemark, J., Vinter, J., Folkesson, P. and Karlsson, J., "Experimental Evaluation of Time-redundant Execution for a Brake-by-wire Application", International Conference on Dependable Systems and Networks, pp. 210-215, 2002.

[Aidemark *et al.*, 2003] Aidemark, J., Folkesson, P. and Karlsson, J., "On the Probability of Detecting Errors Generated by Permanent Faults Using Time Redundancy", to appear at International On-Line Testing Symposium, 2003.

[Aidemark and Askerdal, 2003] Aidemark, J. and Askerdal, Ö., "Use of Time Redundancy for Detection of Data Errors Caused by Non-Transient Faults", Technical Report No. 03-09, Department of Computer Engineering, Chalmers University of Technology, Sweden 2003.

[Alkhalifa *et al.*, 1999] Alkhalifa, Z., Nair, V., Krishnamurthy, N., and Abraham, J., "Design and Evaluation of System-Level Checks for On-Line Control Flow Error Detection", IEEE Transactions on Parallel and Distributed Systems, Vol. 10, No. 6, pp. 627-641, 1999.

[Ammann and Knight, 1988] Ammann, P. and Knight, J., "Data Diversity: An Approach to Software Fault Tolerance", IEEE Transactions on Computers, Vol. 37, No. 4, pp. 418-425, 1988.

[Antoniadis, 2002] Antoniadis, D., “MOSFET Scalability Limits and “New Frontier” Devices”, Symposium on VLSI Technology, pp. 2-5, 2000.

[Askerdal *et al.*, 2000] Askerdal, Ö., Suri, N. and Torin, J., "Use of Complementary Techniques for Detection of Low-Level Errors Caused by both Transient and Persistent Faults, Based on Analysis of Double Execution", Technical Report No. 00-24, Department of Computer Engineering, Chalmers University of Technology, Sweden, 2000.

[Askerdal and Suri, 2001] Askerdal, Ö. and Suri, N., "On-Line Error Detection in Control Systems", Technical Report No. 01-17, Department of Computer Engineering, Chalmers University of Technology, Sweden, 2001.

[Askerdal *et al.*, 2001] Askerdal, Ö., Wiklund, K., Mendelson, A. and Suri, N., "A Novel Simulation Approach for Accurate Modeling of Persistent Faults", Technical Report No. 01-18, Department of Computer Engineering, Chalmers University of Technology, Sweden 2001.

[Askerdal *et al.*, 2002] Askerdal, Ö., Gäfvert, M. and Suri, N., "A Control Theory Approach for Analyzing the Effects of Data Errors in Safety-Critical Control Systems", Pacific Rim International Symposium on Dependable Computing, pp.105-114, 2002.

[Avižienis and Chen, 1977] Avižienis, A. and Chen, L., ”On The Implementation Of N-Version Programming for Software Fault-Tolerance During Program Execution, International Conference on Computer Software and Applications, pp. 149-155, 1977.

[Avižienis, 2001] Avižienis, A. and Yutao, H., “Microprocessor entomology: a taxonomy of design faults in COTS microprocessors”, Dependable Computing for Critical Applications, pp. 3-23, 1999.

[Avižienis, 2000] A. Avižienis, "A Fault Tolerance Infrastructure for Dependable Computing with High-Performance COTS Components", International Conference on Dependable Systems and Networks, pp. 492-500, 2000.

[Avižienis *et al.*, 2001] Avižienis, A., Laprie, J.-C. and Randell, B., “Fundamental Concepts of Dependability”, Technical Report No. CS-TR-739, Newcastle University, 2001.

[Avižienis, 1985] Avižienis, A."The N-version approach to fault-tolerant software", IEEE Transactions on Software Engineering, Vol. 11, No. 12, pp. 1491-1501, 1985.

[Bagchi *et al.*, 2001] Bagchi, S., Liu, Y., Whisnant, K., Kalbarczyk, Z. and Iyer, R., "A Framework for Database Audit and Control Flow Checking for a Wireless Telephone Network Controller", International Conference on Dependable Systems and Networks, pp. 225-234, 2001.

[Barroso *et al.*, 2000] Barroso, L., Gharachorloo, K., McNamara, R., Nowatzyk, A., Qadeer, S., Sano, B., Smith, S., Stets, R. and Verghese, B., "Piranha: A Scalable Architecture Based on Single-Chip Multiprocessing", International Symposium on Computer Architecture, pp. 282-293, 2000.

[Batcher and Papachristou, 1999] Batcher, K. and Papachristou, C., "Instruction Randomization Self Test For Processor Cores" VLSI Test Symposium, pp. 34-40, 1999.

[Beerel, 2002] Beerel, P., "Asynchronous Circuits: An Increasingly Practical Design Solution", International Symposium on Quality Electronic Design, pp. 367-372, 2002.

[Belcastro, 1998] Belcastro, C., "Monitoring functional integrity in critical control computers subjected to electromagnetic disturbances", American Control Conference, pp. 374 -378, 1998.

[Besser *et al.*, 2000] Besser, P., Marathe, A., Zhao, L., Herrick, M., Capasso, C. and Kawasaki, H., "Optimizing the electromigration performance of copper interconnects", International Electron Devices Meeting, pp. 119-122, 2000.

[Betous-Almedia and Kanoun, 2002] Betous-Almedia, C. and Kanoun K., "Stepwise Construction and Refinement of Dependability Models", International Conference on Dependable Systems and Networks, pp. 515-524, 2002.

[Betta and Pietrosanto, 2000] Betta, G. and Pietrosanto, A., "Instrument Fault Detection and Isolation: State of the Art and New Research Trends", IEEE Transactions on Instruments and Measurements, Vol. 49, No. 1, pp. 100-107, 2000.

[Bishop, 1993] Bishop, P., "The Variation of Software Survival Time for Different Operational Input Profiles", International Symposium on Fault Tolerant Computing, pp. 98-107, 1993.

[Blum and Wasserman, 1996] Blum, M. and Wasserman, H., “Reflections on the Pentium Division Bug”, IEEE Transactions on Computers, Vol. 45, No. 4, pp. 385-393, 1996.

[Borkar, 1999] Borkar, S., “Design Challenges of Technology Scaling”, IEEE Micro, pp. 23-29, 1999.

[Bossen *et al.*, 2002] Bossen, D., Kitamorn, A., Reick, K. and Floyd, M., “Fault-tolerant design of the IBM pSeries 690 system using POWER4 processor technology”, IBM Journal of Research and Development, Vol. 46, No. 1, pp. 77-86, 2002.

[Boué *et al.*, 1998] J. Boué, P. Pétillon and Y. Crouzet, "MEFISTO-L: A VHDL-Based Fault Injection Tool for the Expt. Assessment of Fault Tolerance", International Symposium on Fault-Tolerant Computing, pp. 168-173, 1998.

[Brglez and Fujiwara, 1985] F. Brglez and H. Fujiwara, "A Neutral Netlist of 10 Combinational Benchmark Circuits", International Symposium on Circuits and Systems, pp. 695-698, 1985.

[Bright and Sullivan, 1995] Bright, J. and Sullivan, G., "On-line Error Monitoring for Several Data Structures", International Symposium on Fault-Tolerant Computing, pp. 392-401, 1995.

[Bryant *et al.*, 2001] Bryant, R., Cheng, K.-T., Kahng, A., Keutzer, K., Maly, W., Newton, R., Pileggi, L., Rabaey, J. and Sangiovanni-Vincentelli, A., “Limitations and Challenges of Computer-Aided Design Technology for CMOS VLSI”, Proceedings of the IEEE, Vol. 89, No. 3, pp. 341-365, 2001.

[Cambridge, 2003] Cambridge dictionaries online, URL: http//www.dictionary.cambridge.org, 2003.

[Celeiro *et al.*, 1996] F. Celeiro, L. Dias, M. Santos and J. Teixera, "Defect-Oriented IC Test and Diagnosis Using VHDL Fault Simulation", International Test Conference, pp. 620 -628, 1996.

[Chao, 2002] Chao, L., Publisher, “Hyper-Threading Technology”, Intel Technology Journal, Vol. 6, Issue 1, pp. 1-66, 2002.

[Check and Slegel, 1999] Check, M. and Slegel, T., “Custom S/390 G5 and G6 microprocessors”, IBM Journal of Research and Development, Vol. 43, No. 5/6, pp. 671-680, 1999.

[Chen and Dey, 2001] Chen, L. and Dey, S., “Software-Based Self-Testing Methodology for Processor Cores”, IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 20, No. 3, pp. 369-380, 2001.

[Chillarege *et al.*, 1992] Chillarege, R., Bhandari, I., Chaar, J., Halliday, M., Moebus, D., Ray, B. and Wong, M.-Y., “Orthogonal Defect Classification - A Concept for In-Process Measurements”, IEEE Transactions on Software Engineering, Vol. 18, No. 11, pp. 943-956, 1992.

[Christmansson *et al.*, 1998] Christmansson, J., Hiller, M. and Rimén, M., “An Experimental Comparison of Fault and Error Injection”, International Symposium on Software Reliability Engineering, pp. 369-378, 1998.

[Claesson, 2002] Claesson, V., “Efficient and Reliable Communication in Distributed Embedded Systems”, Ph.D. thesis, Technical Report No. 6D, Department of Computer Engineering, Chalmers University of Technology, Sweden, 2002.

[Clarke and Wing, 1996] Clarke, E. and Wing, J., “Formal Methods: State of the Art and Future Directions, ACM Computing Surveys, Vol. 28, No. 4, pp. 626-643, 1996.

[Clayton, 1992] Clayton, P., “Introduction to Electromagnetic Compatibility”, John Wiley & Sons, Inc., 1992.

[Computer User, 2003] Computer User Magazine, URL: http//www.computeruser.com, 2003.

[Constantinescu, 2002] Constantinescu, C., "Impact of Deep Submicron Technology on Dependability of VLSI Circuits", International Conference on Dependable Systems and Networks, pp. 205-209, 2002.

[Crupi *et al.*, 2003] Crupi, F., Kaczer, B., Degraeve, R., De Keersgieter, A. and Groseneken, G., “A Comparative Study of the Oxide Breakdown in Short-Channel nMOSFETs and pMOSFETs Stressed in Inversion and in Accumulation Regimes”, IEEE Transactions on Device and Materials Reliability, Vol. 3, No. 1, pp. 8-13, 2003.

[Cunha *et al.*, 2001] Cunha, J., Maia, R., Rela, M. and Silva, J., “A Study of Failure Models in Feedback Control Systems”, International Conference on Dependable Systems and Networks, pp. 314-323, 2001.

[Cunha *et al.*, 2002] Cunha, J., Rela, M. and Silva, J., “On the Use of Disaster Prediction for Failure-Tolerance in Feedback Control Systems”, International Conference on Dependable Systems and Networks, pp. 123-132, 2002.

[Dahlgren and Lidén, 1991] P. Dahlgren and P. Lidén, "Injection of Physical Transistor Faults into CMOS Processor Building Blocks using Simulations", Technical Report No. 109, Department of Computer Engineering, Chalmers University of Technology, Sweden, 1991.

[Delong *et al.*, 1996] T. Delong, B. Johnson and J. Profeta III, "A Fault Injection Technique for VHDL-Behavioral-Level Models", IEEE Design & Test of Computers, Vol. 13, pp. 24-33, Winter 1996.

[Dimmer, 1985] Dimmer, C., “The Tandem Non-Stop System”, Resilient Computing Systems, 1985.

[Diefendorff 1999] Diefendorff, K., “Power4 Focuses on Memory Bandwidth”, Microprocessor Report, Vol. 13, No. 13, pp. 1-8, 1999.

[Douglass, 2001] Douglass, B., “Doing Hard Time Developing Real-Time Systems with UML, Objects, Frameworks, and Patterns”, Addison-Wesley, 2001.

[Driscoll *et al.*, 2003] Driscoll, K., Hall, B., Sivencrona, H. and Zumsteg, P., “Byzantine Fault Tolerance from Theory to Reality”, submitted to the International Conference on Computer Safety, Reliability and Security, 2003.

[Eckerbert, 2002] Eckerbert, D., “Deep Submicron Issues in RTL Power Estimation”, Licentiate thesis, Technical Report No. 7L, Department of Computer Engineering, Chalmers University of Technology, Sweden, 2002.

[Elliott and Sayers, 1990] Elliott, I., and Sayers, I., "Implementation of 32-bit RISC processor incorporating hardware concurrent error detection and correction", IEE Proceedings Computers and Digital Techniques, Vol. 137, No. 1, pp. 88-102, 1990.

[Eure *et al.*, 2001] Eure, K., Belcastro, C. and Weinstein, B., “Monitoring Functional integrity in Critical Control Computers Subjected to Electromagnetic Disturbances During Laboratory Tests”, System Readiness Technology Conference, pp. 730-738, 2001.

[Feick *et al.*, 2000] Feick, S., et al. "Steer-by-Wire as a Mechatronic Implementation", SAE World Congress, 2000.

[Feldt, 1999] Feldt, R., “Genetic Programming as an Explorative Tool in Early Software Development Phases, International Workshop on Soft Computing Applied to Software Engineering, pp. 11-21, 1999.

[Ferrandi *et al.*, 2002] F. Ferrandi, F. Fummi and D. Sciuto, "Test Generation and Testability Alternatives Exploration of Critical Algorithms for Embedded Applications", IEEE Transactions on Computers, Vol. 51, No. 2, pp. 200-215, 2002.

[Fischer *et al.*, 2000] Fischer, A., Abel, A., Lepper, M., Zitzelsberger, A. and von Glasow, A., “Experimental Data and Statistical Models for Bimodal EM Failures”, International Interconnect Technology Conference, pp. 359-363, 2000.

[Fischer *et al.*, 2002] Fischer, A., von Glasow, A., Penka, S. and Ungar, F., “Electromigration failure mechanism studies on copper interconnects”, International Interconnect Technology Conference, pp. 139-141, 2002.

[Folkesson, 1999] Folkesson, P., “Assessment and Comparison of Physical Fault Injection Techniques”, Ph.D. thesis, Technical Report No. 377, Department of Computer Engineering, Chalmers University of Technology, Sweden, 1999.

[Frank 1997] Frank, P. and Ding, X., “Survey of Robust Residual Generation and Evaluation Methods in Observer-Based Fault Detection Systems”, Journal of Process Control, No. 6, pp. 403-424, 1997.

[Gaisler, 2002] Gaisler, J., “A Portable and Fault-Tolerant Microprocessor Based on the SPARC V8 Architecture”, International Conference on Dependable Systems and Networks, pp. 409-415, 2002.

[Geppert, 2002] Geppert, L., “The Amazing Vanishing Transistor Act”, IEEE Spectrum, Vol. 39, Issue 10, pp. 28-33, 2002.

[Gil *et al.*, 2002] Gil, P., Arlat, J., Madeira, H., Crouzet, Y., Jarboui, T., Kanoun, K., Marteau, T., Durães, J., Viera, M., Gil, D., Baraza, J.-C. and Gracia, J., “Fault Representativeness”, DBench Project Report, ETIE2, 2002.

[Gracia *et al.*, 2001] J. Gracia, J. Baraza, D. Gil and P. Gil, "Comparison and Application of different VHDL-Based Fault Injection Techniques", International Symposium on Defect and Fault Tolerance in VLSI Systems, pp. 233 -241, 2001.

[Grundman *et al.*, 2003] Grundman B., Galivanche, R. and Kundu, S., “Circuit and Platform Design Challenges in Technologies beyond 90nm”, Conference and Exhibition on Design, Automation and Test in Europe, pp. 44-47, 2003.

[Gupta and Pradhan, 1996] Gupta, S. and Pradhan, D., “Utilization of On-Line (Concurrent) Checkers during Built-In Self-Test and Vice-Versa”, IEEE Transactions on Computers, Vol. 45, No. 1, pp. 63-73, 1996.

[Gäfvert *et al.*, 2003] Gäfvert, M., Wittenmark B. and Askerdal, Ö., "On the Effect of Transient Data Errors in Controller Implementations", to appear at American Control Conference, 2003.

[Hamilton, 2003] Hamilton, S., “Intel research expands Moore’s law”, IEEE Computer, Vol. 36, Issue 1, pp. 31-40, 2003.

[Hammond *et al.*, 1997] Hammond, L., Nayfeh, B. and Olukotun, K., “A Single-Chip Multiprocessor”, IEEE Computer, Vol. 30 Issue 9, pp. 79-85, 1997.

[Hanselmann, 1987] Hanselmann, H., “Implementation of Digital Controllers – A Survey”, Automatica, 23(1), pp. 7-32, 1987.

[Hawkins *et al.*, 1999] Hawkins, C., Baker, K., Butler, K., Figuera, J., Nicoladis, M., Rao, V., Roy, R. and Welsher T., “IC Reliability and Test: What Will Deep Submicron Bring?”, IEEE Design & Test of Computers, Vol. 16, Issue 2, pp. 84-91, 1999.

[Hawkins *et al.*, 1994] Hawkins, C., Soden, J., Righter, A. and Ferguson, J., “Defect Classes - An Overdue Paradigm for CMOS IC Testing”, International Test Conference, pp. 413-425, 1994.

[Hazucha, 2000] Hazucha, P., “Background Radiation and Soft Errors in CMOS circuits”, Dissertation No. 638, Dept. of Physics and Measurement Technology, Linköpings Universitet, Sweden, 2000.

[Hennessy and Patterson, 1996] Hennessy, J. and Patterson, D., "Computer Architecture A Quantitative Approach", second edition, Morgan Kaufmann Publishers, Inc., 1996.

[Henson *et al.*, 1999] Henson, B., McDonald, P. and Stapor, W., “SDRAM Space Radiation Effects Measurements and Analysis”, IEEE Radiation Effects Data Workshop, pp. 15-23, 1999.

[Hiller, 2000] Hiller, M., “Executable Assertions for Detecting Data Errors in Embedded Control Systems”, International Conference on Dependable Systems and Networks, pp. 24-33, 2000.

[Hiller *et al.*, 2001] Hiller, M., Jhumka, A. and Suri, N., “An Approach for Analysing the Propagation of Data Errors in Software, International Conference on Dependable Systems and Networks, pp. 161-170, 2001.

[Hiller *et al.*, 2002] Hiller, M., Jhumka, A. and Suri, N., “On the Placement of Software Mechanisms for Detection of Data Errors, International Conference on Dependable Systems and Networks, pp. 135-144, 2002.

[Huh *et al.*, 1998] Huh, Y., Lee, M., Lee, J., Jung, H., Li, T., Song, D., Lee, Y, Hwang, J., Sung, Y. and Kang, M., “A Study of ESD-Induced latent Damage in CMOS Integrated Circuits”, International Reliability Physics Symposium, pp. 279-283, 1998.

[IEEE, 1985] “IEEE Standard for Binary Floating-Point Arithmetic”, ANSI/IEEE Std 754-1985, 1985.

[Intel, 2003(A)] “New Transistors for 2005 and Beyond”, Intel Research & Development, URL:
http://developer.intel.com/research/silicon/micron.htm#micronpub, 2003.

[Intel, 2003(B)] “The Intel Pentium 4 Processor”, Intel Research & Development, URL:
http://www.intel.com/design/Pentium4/prodbref/index.htm, 2003.

[ITRS, 2002] “International Technology Roadmap for Semiconductors”, URL:
http://public.itrs.net, update 2002.

[Iyer and Tang, 1996] Iyer, R. and Tang, F., "Experimental Analysis of Computer System Dependability”, in book: “Fault-Tolerant Computer System Design”, editor Pradhan, D., Prentice Hall, 1996.

[Jenn *et al.*, 1994] Jenn, E., Arlat, J., Rimén, M., Ohlsson, J. and Karlsson, J., "Fault injection into VHDL models: the MEFISTO tool", International Symposium on Fault-Tolerant Computing, pp. 66 -75, 1994.

[Jhumka *et al.*, 2002] Jhumka, A., Hiller, M., Claesson, V. and Suri, N., “On Systematic Design of Globally Consistent Executable Assertions in Embedded Software”, ACM Joint Conference - Languages, Compilers and Tools for Embedded Systems/Software and Compilers for Embedded Systems, pp. 74-83, 2002.

[Jochim, 2002] Jochim, M., “Detecting Processor Hardware Faults by Means of Automatically Generated Virtual Duplex Systems”, International Conference on Dependable Systems and Networks, pp. 399-408, 2002.

[Johannessen, 2001] Johannessen, P., “Design Methods for Safety Critical Automotive Architectures”, Licentiate thesis, Technical Report No. 407L, Department of Computer Engineering, Chalmers University of Technology, Sweden, 2001.

[Johnson *et al.*, 1991] B. Johnson, T. Quarles, A. Newton, D. Pederson and A. Sangiovanni-Vincentelli, "SPICE3 version 3e user’s manual, "Department of Electrical Engineering and Computer Sciences, Univiversity of California, Berkley, April, 1991.

[Johnson, 1989] Johnson, B., “Design and Analysis of Fault-Tolerant Digital Systems”, Addison-Wessley, 1989.

[Jones and Hayes, 1999] Jones J. and Hayes J., “A Comparison of Electronic-Reliability Prediction Models”, IEEE Transactions on reliability, Vol. 48, No. 2, pp. 127-134, 1999.

[Kaczer *et al.*, 2002] Kaczer B., Crupi, R., Roussel, Ph., Ciofi, C. and Groseneken, G., “Observation of hot-carrier-induced nFET gate-oxide breakdown in dynamically stressed CMOS circuits”, International Electron Devices Meeting, pp. 171-174, 2002.

[Kalbarczyk *et al.*, 1999] Z. Kalbarczyk, R. Iyer, G. Ries, J. Patel, M. Lee and Y. Xiao, "Hierarchical Simulation Approach to Accurate Fault Modeling for System Dependability Evaluation", IEEE Transactions on Software Engineering, Vol. 25, No. 5, pp. 619-632, 1999.

[Kam *et al.*, 2000] Kam, T., Rawat, S., Kirkpatrick, D., Roy, R., Spirakis, G., Sherwani, N. and Peterson, C., “EDA Challenges Facing Future Microprocessor Design”, IEEE Transactions on Computer Aided Design of Integrated Circuits and Systems, Vol. 19, No. 12, 2000.

[Karlsson *et al.*, 1995] Karlsson, J., Folkesson, P., Arlat, J., Crouzet, Y., Leber G. and Reisinger, J., "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture", International Working Conference on Dependable Computing for Critical Applications, pp.150-161, 1995.

[Karnik *et al.*, 2002] Karnik, T., Vangal, S., Veeramachaneni, V., Hazucha, P., Erraguntla, V. and Borkar, S., “Selective Node Engineering for Chip-level Soft Error Rate Improvement”, Symposium on VLSI Circuits, pp. 204-205, 2002.

[Ker *et al.*, 2002] Ker, M.-D., Peng, J.-J. and Jiang, H.-C., “Failure Analysis of ESD Damage in a High-Voltage Driver IC and the Effective ESD Protection Solution”, International Symposium on Physical and Failure Analysis on Integrated Circuits, pp. 81-86, 2002.

[Kim and Shin, 1994] Kim, H. and Shin, K., “On the Maximum Feedback Delay in a Linear/Nonlinear Control System With Input Disturbances Caused by Controller-Computer Failures”, IEEE Transactions on Control Systems Technology, Vol. 2, No. 2, pp. 110-122, 1994.

[Kim *et al.*, 2000] Kim, H., White, A. and Shin, K., “Effects of Electromagnetical Interference on Controller-Computer Upsets and System Stability”, IEEE Transactions on Control Systems Technology, Vol 8, No. 2, pp. 351-357, 2000.

[Kopetz and Grunsteidl, 1994] Kopetz, H. and Grunsteidl, G., "TTP - a protocol for fault-tolerant real-time systems", IEEE Computer, pp. 14-23, 1994.

[Kundu *et al.*, 2001] Kundu, S., Zachariah, S., Sengupta, S. and Galivanche, R., "Test Challenges in Nanometer Technologies", Journal of Electronic Testing: Theory and Applications, Vol. 17, Issue 3-4, pp. 209-218, 2001.

[Kundu and Galivanche, 2001] Kundu, S. and Galivanche, R., "Test Challenges in Nanometer Technologies", Tutorial held at the European Test Workshop, 2001.

[Lai *et al.*, 2002] Lai, S.-C., Lu, S.-L., Lai, K. and Peir, J.-K., “Ditto Processor”, International Conference on Dependable Systems and Networks, pp. 525-534, 2002.

[Laprie, 1992] Laprie J, Ed. "Dependability: Basic Concepts and Terminology, Dependable Computing and Fault-Tolerant Systems", Vol.5, Springer-Verlag 1992.

[Lee *et al.*, 2000] Lee, Y., Wu, K., Linton, T., Mielke, N., Hu, S. and Wallace, B., “Channel-Width Dependent Hot-Carrier Degradation of Thin-Gate pMOSFETs”, IEEE International Reliability Physics Symposium, pp. 77-82, 2000.

[Leon, 2003] LEON SPARC processor, URL: http://www.gaisler.com/leonmain.html, 2003.

[Leung and Whitehead, 1988] Leung, J. and Whitehead, J., “On the Complexity of Fixed-Priority Scheduling of Periodic, Real-Time Tasks”, Performance Evaluation, pp. 237-250, December 1988.

[Leveugle and Hadjiat, 2000] R. Leveugle and K. Hadjiat, "Optimized Generation of VHDL Mutants for Injection of Transition Errors", Symposium on Integrated Circuits and Systems Design, pp. 243 -248, 2000.

[Lewin 2002] Lewin, D., “DNA Computing”, IEEE, Computing in Science & Engineering, Vol. 4, Issue 3, pp. 5-8, 2002.

[Lienig *et al.*, 2002] Lienig, J., Jerke, G. and Adler, T., “Electromigration Avoidance in Analog Circuits: Two Methodologies for Current-Driven Routing”, International Conference on VLSI Design, pp. 464-469, 2002.

[Lions, 1996] Lions, J., chairman of the inquiry board, “ARIANE 5, Flight 501 Failure Report by the Inquiry Board”, URL: http://java.sun.com/people/jag/Ariane5.html, 1996.

[Lisenker and Mitnick, 2000] Lisenker, B. and Mitnick, Y., “Reliability Assessment through Defect Based Testing”, International Reliability Physics Symposium, pp. 407-412, 2000.

[Littlewood and Miller, 1989] Littlewood, B. and Miller, D., “Conceptual Modeling of Coincident Failures in Multiversion Software”, IEEE Transactions on Software Engineering, Vol. 15, No. 12, pp. 1596-1614, 1989.

[Lo *et al.*, 1992] Lo, J.-C., Thanawastien, S., Rao, T., R., N., Nicoladis, M., "An SFS Berger Check Prediction ALU and Its Application to Self-Checking Processor Designs", IEEE Transactions on Computer-Aided Design, Vol.11, No. 4, pp. 525-540, 1992.

[Lo *et al.*, 1997] Lo, J., Eggers, S., Emer, J., Levy, H., Stamm, R. and Tullsen, D., “Converting Thread-Level Parallelism to Instruction-Level Parallelism via Simultaneous Multithreading”, ACM Transactions on Computer Systems, Vol. 15, No. 3, pp. 322-354, 1997.

[Lovric 1996] Lovric, T., "Detecting hardware faults with systematic and design diversity: Experimental results", Computer Systems Science and Engineering, Vol. 11, No. 2, pp. 83-92, 1996.

[Lönn, 1999] Lönn, H., “Synchronization and Communication Results in Safety-Critical Real-Time Systems”, Ph.D. thesis, Technical Report No. 374, Department of Computer Engineering, Chalmers University of Technology, Sweden, 1999.

[Maamar and Russel, 1998] Maamar, A., and Russel, G., "A32-Bit RISC Processor With Concurrent Error Detection", EUROMICRO Conference, Vol. 1, pp. 461-467, 1998.

[Mahmood and McCluskey, 1988] Mahmood, A. and McCluskey, E., "Concurrent Error Detection Using Watchdog Processors- A survey", IEEE Transactions on Computers, Vol. 37, No. 2, pp. 160-174, 1988.

[Makabe *et al.*, 2000] Makabe, M., Kubota, T. and Kitano, T., “Bias-temperature degradation of pMOSFETs: mechanism and suppression”, International Reliability Physics Symposium, pp. 205-209, 2000.

[Massengill *et al.*, 2000] Massengill, L., Baranski, A., Van Nort, D., Meng, J. and Bhuva, B., “Analysis of Single-Event Effects in Combinational logic - Simulation of the AM2901 Bitslice Processor”, IEEE Transactions on Nuclear Science, Vol. 47, No. 6, pp. 2609-2615, 2000.

[McVittie, 1996] McVittie, J., “Plasma Charging Damage: An Overview”, International Symposium on Plasma-Induced Damage, pp. 7-10, 1996.

[Meindl, 2001] Meindl, J., ”Special issue on limits of semiconductor technology”, Proceedings of the IEEE, Vol. 89, Issue 3, pp. 223-226, 2001.

[Meindl, 2003] Meindl, J., ”Beyond Moore’s law: The Interconnect Era”, IEEE Computing in Science & Engineering”, Vol. 5, Issue 1, pp. 20-24, 2003.

[Mendelson and Suri, 2000] Mendelson, A. and Suri, N., “Designing High-Performance & Reliable Superscalar Architectures - The Out of Order Reliable Superscalar (O3RS) Approach”, International Conference on Dependable Systems and Networks, pp. 473-481, 2000.

[Mentor Graphics 1998] Mentor Graphics (Model Technology) "ModelSim EE/PLUS Reference Manual", 1998.

[Miremadi *et al.*, 1995] Miremadi, G., et. al, "Use of time and Address Signatures for Control Flow Checking", International Working Conference on Dependable Computing for Critical Applications, 1995.

[Moore, 1965] Moore, G., “Cramming More Components onto Integrated Circuits”, Electronics, Vol. 38, No. 8, pp. 114-117, 1965.

[Mule *et al.*, 2002] Mule, A., Bakir, M., Jayachandran, J., Villalaz, R., Reed, H., Agrawal, N., Ponoth, S., Plawsky, J., Persans, P., Kohl, P., Martin, K., Glytsis, E., Gaylord, T. and Meindl, J., “Optical Waveguides with Embedded Air-gap Cladding Integrated Within a Sea-of-Leads (SoL) Wafer-level Package”, IEEE International Interconnect Technology Conference, pp. 122-124, 2002.

[Nicoladis, 1995] Nicolaidis, M., "Efficient UBIST Implementation for Microprocessor Sequencing Parts", Journal of Electronic Testing-Theory and Applications, Vol. 6, No. 3, pp. 295-312, 1995.

[Nicolaidis and Zorian, 1998] Nicolaidis, M., and Zorian, Y., "On-Line Testing for VLSI - A Compendium of Approaches", Journal of Electronic Testing - Theory and Applications Vol.12, No. 1-2, pp. 7-20, 1998.

[Nicollian *et al.*, 1999] Nicollian, P., Rodder, M., Grider, D., Chen, P., Wallace, R. and Hattangady, S., “Low Voltage Stress-Induced-Leakage-Current in Ultra-thin Gate Oxides”, International Reliability Physics Symposium, pp. 400-404, 1999.

[Oh *et al.*, 2002(A)] Oh, N., Shirvani, P. and McCluskey, E., “Control-Flow Checking by Software Signatures”, IEEE Transactions on Reliability, Vol. 51, No. 2, pp. 111-122, 2002.

[Oh *et al.*, 2002(B)] Oh, N., Shirvani, P. and McCluskey, E., “Error Detection by Duplicated Instructions in Super-Scalar Processors”, IEEE Transactions on Reliability, Vol. 51, No. 1, pp. 63-75, 2002.

[Ohlsson and Rimén, 1995] Ohlsson, J. and Rimén, M., "Implicit Signature Checking", International Symposium on Fault-Tolerant Computing, pp. 218-227, 1995.

[Ohlsson, 1995] Ohlsson, J., "Application Signature Checking", Technical Report No. 238, Department of Computer Engineering, Chalmers University of Technology, Sweden, 1995.

[Oldiges *et al.*, 2002] Oldiges, P., Bernstein, K., Heidel, D., Klaasen, B., Cannon, E., Dennard, R., Tang, H., Ieong, M. and Wong, H.-S., “Soft Error Rate Scaling for Emerging SOI Technology Options”, Symposium on VLSI Technology, pp. 46-47, 2002.

[Oskin *et al.*, 2002] Oskin, M., Chong, F. and Chuang, I., “A Practical Architecture for Reliable Quantum Computers”, IEEE Computer, Vol. 35, Issue 1, pp. 79-87, 2002.

[Otten *et al.*, 2002] Otten R., Camposano, R. and Groeneveld, P., “Design Automation for Deepsubmicron: present and future”, Conference and Exhibition on Design, Automation and Test in Europe, pp.650-657, 2002.

[Paschalis *et al.*, 2001] Paschalis, A., Gizopoulos, D., Kranitis, N., Psarakis, M. and Zorian, Y., “Deterministic software-based self-testing of embedded processor cores”, Conference and Exhibition on Design, Automation and Test in Europe, pp. 92-96, 2001.

[Parotta *et al.*, 2000] Parotta, B., Rebaudengo, M., Reorda, S. and Violante, M., “New Techniques for Accelerating Fault Injection in VHDL descriptions”, IEEE International On-Line Testing Workshop, pp. 61-66, 2000.

[Peters and Pedrycz, 1999] Peter, J. and Pedrycz, W., “Software Engineering an Engineering approach”, John Wiley & Sons, Inc., 1999.

[Powell *et al.*, 1988] Powell, D., Bonn, G., Seaton, D., Verissimo, P. and Waeselynck, F., “The Delta-4 approach to dependability in open distributed computing systems”, International Symposium on Fault-Tolerant Computing, pp. 246-251, 1988.

[Pradhan, 1996] Pradhan, D., “Fault-Tolerant Computer Systems Design”. Prentice Hall PTR, 1996.

[Prata and Silva, 1999] Prata, P. and Silva, G., "Algorithm Based Fault Tolerance Versus Result-Checking for Matrix Computations", International Symposium on Fault-Tolerant Computing, pp. 4-11, 1999.

[Radecka *et al.*, 1997] Radecka, K., et al. "Arithmetic Built-In Self-Test for DSP Cores", IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol. 16, No. 11, pp. 1358-1369, 1997.

[Rashid *et al.*, 2000] Rashid, F., Saluja, K. and Ramanathan, P., “Fault Tolerance Through Re-execution in Multiscalar Architecture”, International Conference on Dependable Systems and Networks, pp. 482-491, 2000.

[Rela *et al.*, 1996] Rela, Z., Madeira, H. and Silva, J., "Experimental Evaluation of the Fail-Silent Behaviour in Programs with Consistency Checks", International Symposium on Fault-Tolerant Computing, pp. 394-403, 1996.

[Riesgo and Uceda, 1996] T. Riesgo and J. Uceda, "A Fault Model for VHDL Descriptions at the Register Transfer Level", Conference and Exhibition on Design, Automation and Test in Europe, pp. 462 -467, 1996.

[Rimén, 1995] Rimén, M., "Use of Data Signatures and Double Execution for Data Error Detection", Technical Report No. 247, Department of Computer Engineering, Chalmers University of Technology, Sweden, 1995.

[Riordan *et al.*, 1999] Riordan, W., Miller, R., Sherman, J. and Hicks, J., “Microprocessor Reliability Performance as a Function of Die Location for a 0.25μ, Five Layer Metal CMOS Logic Process”, International Reliability Physics Symposium, pp. 1-11, 1999.

[Rotenberg, 1999] Rotenberg E., "AR-SMT: A Microarchitectural Approach to Fault Tolerance in Microprocessors", International Symposium on Fault-Tolerant Computing, pp. 84-91, 1999.

[Saab Ericsson Space, 1999] Saab Ericsson Space AB, "Red Hard Thor Microprocessor Description", Document No. P-TOR-NOT-004-SE, 1999.

[Scobie *et al.*, 2000] Scobie, J., et al. "A Cost Efficient Fault Tolerant Brake By Wire Architecture", SAE World Congress, pp. 51-60, 2000.

[Segura *et al.*, 1995] Segura, J., De Benito, C., Rubio, A. and Hawkins C., “A Detailed Analysis of GOS Defects in MOS Transistors: Testing Implications at Circuit Level”, International Test Conference, pp. 544-551, 1995.

[Shaw *et al.*, 2001] D. Shaw, D. Al-Khalili and C. Rozon, "Accurate CMOS Bridge Fault Modeling With Neural Network-Based VHDL-Saboteurs", International Conference on Computer Aided Design, pp. 531-536, 2001.

[Shen and Abraham, 1998] Shen, J. and Abraham J., "Native Mode Functional Test Generation for Processors with Applications to Self-Test and Design Validation", International Test Conference, pp. 990-999, 1998.

[Shivakumar *et al.*, 2002] Shivakumar, P., Kistler, M., Keckler, S., Burger, D. and Alvisi, L., “Modeling the Effect of Technology Trends on the Soft Error Rate of Combinational Logic”, International Conference on Dependable Systems and Networks, pp. 389-398, 2002.

[Sieh et al., 1997] V. Sieh, O. Tschäche and F. Balbach, "VERIFY: Evaluation of Reliability Using VHDL-Models with Embedded Fault Descriptions", International Symposium on Fault-Tolerant Computing, pp. 32 -36, 1997.

[Sieworek and Swarz, 1992] Sieworek, D. and Swarz, R., "Reliable Computer Systems, Design and Evaluation", second edition, Digital Press, 1992.

[Silva *et al.*, 1998] Silva, J., Prata, P., Rela, M. and Madeira, H., “Practical issues in the Use of ABFT and a New Failure Model”, International Symposium on Fault-Tolerant Computing, pp. 26-35, 1998.

[Soden and Hawkins, 1995] Soden, J. and Hawkins, C., “I_{DDQ} Testing and Defect Classes - A Tutorial”, Custom Integrated Circuits Conference, pp. 633-642, 1995.

[Steininger and Scherrer, 1999] Steininger A. and Scherrer C., "On the Necessity of On-line BIST in Safety-Critical Applications- A Case Study", International Symposium on Fault-Tolerant Computing, pp. 208-215, 1999.

[Steininger 2000] Steininger, A., "Testing and Built-in-Self-Test - A Survey", Journal of Systems Architecture, Elsevier Science Publishers, Vol. 46, pp. 721-747, 2000.

[Steininger and Temple, 1999] Steininger, A. and Temple, C., "Economic Online Self-Test in the Time-Triggered Architecture”, IEEE Design & Test of Computers, Vol. 16, Issue 3, pp.81-89, 1999.

[Steininger and Vilanek, 2002] Steininger, A. and Vilanek, J., "Using Offline and Online BIST to Improve System Dependability - The TTPC-C Example”, International Conference on Computer Design, pp. 277-280, 2002.

[Steininger and Scherrer, 2002] Steininger A. and Scherrer C., "Identifying Efficient Combinations of Error Detection Mechanisms Based on Results of Fault Injection Experiments”, IEEE Transactions on Computers, Vol. 51, No. 2, pp. 235-239, 2002.

[Stroph and Clarke, 1999] Stroph R. and Clarke T., "Dynamic Fault Detection Approaches", American Control Conference, pp. 627-631, 1999.

[Sullivan *et al.*, 1995] Sullivan, G., et al. "Certification of Computational Results", IEEE Transactions on Computers, Vol. 44, No. 7, pp. 833-847, 1995.

[Talkhan *et al.*, 1989] Talkhan E.-S., Ahmed, A. and Salama, A., “Microprocessors Functional Testing Techniques”, IEEE Transactions on Computer-Aided Design, Vol. 8, No. 3, pp. 316-318, 1989.

[Temple, 1998] Temple C., "Avoiding the babbling-idiot failure in a time-triggered communication system", IEEE International Symposium on Fault-Tolerant Computing, pp.218-227, 1998.

[Theis, 2003] Theis, T., “Beyond the Silicon Transistor: Personal Observations”, IEEE, Computing in Science & Engineering, Vol. 5, Issue 1, pp. 25-29, 2003.

[Theobald *et al.*, 1999] Theobald, K., Gao, G. and Sterling, T., “Superconducting Processors for HTMT: Issues and Challenges”, Symposium on Frontiers of Massively Parallel Computation, pp. 260-267, 1999.

[Torin, 1999] Torin, J., ”The Evolution of Microelectronics and its impacts on Avionics”, Space Technology, Vol. 19, No. 3-4, pp. 199-204, 1999.

[Turmon and Granat, 2000] Turmon, M. and Granat, R., "Algorithm-Based Fault Tolerance for Spaceborn Computing: Basis and Implementations", Aerospace Conference, pp. 411-420, 2000.

[US Department of Defense, 1991] "Military Handbook - Reliability Prediction of Electronic Equipment, MIL-HDBK-217F", US Department of Defense, 1991.

[Vado *et al.*, 2000] P. Vado, Y. Savaria, Y. Zoccarato and C. Robach, "A Methodology for Validating Digital Circuits with Mutation Testing", International Symposium on Circuits and Systems, pp. 343 -346, 2000.

[Vargas *et al.*, 1998] F. Vargas, E. Bezerra, A. Terroso and D. Barros, "Reliability Verification of Fault-Tolerant Systems Design Based on Mutation Analysis", Brazilian Symposium on Integrated Circuit Design, pp. 55 -58, 1998.

[Vinson and Liou, 2000] Vinson, J., Liou, J., “Electrostatic Discharge in Semiconductor Devices:Overview of Circuit Protection Techniques”, Electron Devices Meeting, pp. 5-8, 2000.

[Vinter *et al.*, 2001] Vinter, J., Aidemark, J., Folkesson, P. and Karlsson, J., “Reducing Critical Failures for Control Algorithms Using Executable Assertions and Best Effort Recovery”, International Conference on Dependable Systems and Networks, pp. 347-356, 2001.

[Vinter *et al.*, 2002] Vinter, J., Johansson, A., Folkesson, P. and Karlsson, J., “On the Design of Robust Integrators for Fail-Bounded Control Systems”, to appear at International Conference on Dependable Systems and Networks, 2003.

[Wells, 2003] Wells, D., “Extreme Programming: A Gentle Introduction”, URL: http://www.extremeprogramming.org, 2003.

[White and Kim, 1996] White, A. and Kim, H., ”Designing Experiments for Controller Perturbation Theories - an Example”, Aerospace Application Conference, Vol. 1 pp. 265-278, 1996.

[Wiklund, 2000] K. Wiklund, "A gate-level fault simulation toolkit", Technical Report No. 00-17, Department of Computer Engineering, Chalmers University of Technology, Sweden, 2000.

[Wilken and Shen, 1990] Wilken K. and Shen, J., "Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors", IEEE Transactions on Computer-Aided Design, Vol. 9, No. 6, pp. 629-641, 1990.

[Wittenmark *et al.*, 1995] Wittenmark, B., Nilsson, J. and Törngren, M., "Timing Problems in Real-time Control Systems: Problem Formulation", American Control Conference, pp. 2000-2004, 1995.

[Yeh, 1996] Yeh, Y., "Triple-triple redundant 777 primary flight computer", Aerospace Applications Conference, pp. 293-307, 1996.

[Yount and Siewiorek, 1996] C. Yount and D. Siewiorek, "A Methodology for the Rapid Injection of Transient Hardware Errors", IEEE Transactions on Computers, Vol. 45, No. 8, pp. 881-891, 1996.

[Yu *et al.*, 2002] Yu, Q., Kikuchi, H., Ikeda, S., Shiratori, M., Kakino, M. and Fujiwara, N., “Dynamic Behavior of Electronics Package and Impact Reliability of BGA Solder Joints”, Intersociety Conference on Thermal Phenomena in Electronic Circuits, pp. 953-960, 2002.

[Zhang *et al.*, 2001] Zhang, J., Sii, H., Groseneken, G. and Degraeve, R., “Generation of hole traps in silicon dioxide”, Physical and Failure Analysis of Integrated Circuits, pp. 50-54, 2001.

[Zhang *et al.*, 2002] Zhang, W., Zhang, J., Lalor, M., Burton, D., Groseneken, G. and Degraeve, R., "Two Types of Neutral Electron Traps Generated in the Gate Silicon Dioxide", IEEE Transactions on Electron Devices, Vol. 49, No. 11, pp. 1868-1875, 2002.

[Zhou and Doyle, 1998] Zhou K. and Doyle, J.C. "Essentials of robust control", Prentice Hall, Upper Saddle River, New Jersey, 1998.

[Åström and Wittenmark, 1997] Åström, K. and Wittenmark, B., "Computer-Controlled Systems", Prentice Hall, third edition, 1997.

[Årzén *et al.*, 1990] Årzén K-E., "A Simple Event-Based PID Controller", Reprints 14th World Congress of IFAC, 1999.

Appendix A. A Survey of Error Detection Techniques

As mentioned in the introduction of this thesis, faults can be tolerated, either by fault masking (N-Modular Redundancy (NMR)), or error detection and recovery. For products with strict requirements on recurring costs, fault masking is generally not efficient as it requires replication (at least triplication) of the hardware, but the error detection and recovery strategy is more suited even if such solutions may be more complex. This appendix focuses on surveying different error detection techniques and giving qualitative efficiency estimates. Therefore, first, what is meant with efficient error detection is discussed.

As described in the introduction (Chapter 1), error detection is generally based on some sort, or a mix, of redundancy (spatial, temporal or information). This means that error detection is associated with an overhead (additional hardware, time-consumption, memory-utilization, etc.). The type of redundancy used also determines the type of errors that can be detected. As it generally is required to handle many different types of errors, techniques detecting several error types are in that perspective advantageous. However, as it generally is more difficult to determine the exact cause for triggering of these techniques, fault diagnosis is generally more complicated.

Another important aspect of detection techniques, related to the detection coverage, is the probability for the technique to be triggered in the absence of errors, i.e., the false alarm rate. Moreover, some techniques detect errors independently on which impact they have on the system (so-called systematic techniques), whereas others detect those errors that have the greatest impact on the system (so-called system dependent techniques). To only detect those errors that can cause damage to the system is advantageous as unnecessarily detection of errors consumes resources and increases the risk to incorrectly handle errors. However, the design cost is generally higher for system-dependent techniques, as these has to be design or at least adjusted for each system. Furthermore, it is not always possible to detect all errors that can

cause damage without using systematic techniques.

An additional important issue is how long time it takes from that an error occur until it is triggered by the detection technique, i.e., the error detection latency.

In the following sections, different error detection techniques are described and discussed. At the end of this appendix, we give a short summary and present a table with simple and rough qualitative estimates of the previous described properties.

A.1 Functional Redundancy

Functional redundancy (or model based fault diagnosis) is a error detection technique based on estimation of signal values, through correlated signals and models of the system, and comparing it with the measured signal value, i.e. forming a residual. The residual should be chosen so that it is highly sensitive to malfunctions. The estimation could utilize present values of signals and/or previous values (earlier sampled values). Functional redundancy is often designed to detect errors in sensors and actuators, but it may detect errors in the computer node as well.

One important aspect of error detection through the use of residuals is to decide the border-value for the residual, i.e., a so called threshold, when it should be considered that a fault has occurred. The lower this value is set, the more sensible it will be to faults and the recovery can be started faster, but the number of false alarms will be larger (due to model inaccuracies, transient disturbances, the dynamics of the system, etc.). So this value has to be set with care.

Functional redundancy could be based on parity space methods, filter based methods, neural networks, fuzzy logic, frequency based methods, expert systems applications, pattern recognition, statistical classification and limit and trend check or knowledge based approaches, [Betta and Pietrosanto, 2000], [Frank 1997]. Some of these approaches do not rely on models of the system (for instance neural networks, fuzzy logic, pattern recognition etc.), and thus, can be used when such models do not exist, or are too costly to develop.

With model-based approaches, a system model is used to predict the values of the states and outputs. Then residuals are formed from these estimates and compared with the values of the real system. Residuals can be generated in different ways: parity equations, observer-based generation and parameter estimation.

Functional redundancy has mainly target sensor, actuator, and process faults, but some ideas have been proposed for detection of computer/controller errors. In [Abdelhay and Simeu, 2000] it is used to detect faults in linear digital systems, generally consisting of processing elements such as adders, multipliers, shifters and register which cooperate with each other to achieve a complex function. Functional

redundancy can also be used to detect faults in computer nodes when sufficient information can be obtained [Ahlström and Torin 2001], [Belcastro, 1998], e.g., by running a model of the controller on another computer.

To summarize, this technique is mainly used for detection of errors caused by sensor, actuator and process faults. In many cases it is possible to derive signals for which faults can be identified, i.e., improve fault diagnosis. as the data used for the estimation has passed the complete control loop, detected errors could be originated from a fault in any component, i.e., it is possible to also detect errors in the controller. However, if the controller and the residual generation is executed on the same computer, it is possible that one fault interfere with both computations, and thus, the error cannot be handled.

The overhead due to applying functional redundancy is mainly the required computations (increased execution time) and the memory storage for the program task executing the estimation.

A.2 Assertions

This technique utilizes the information about signals provided in the specification to verify that the system is working correctly. Many signals have some limit in what values they can attain or how much it can differ from one measurement to the next (the signals derivative).

Executable Assertions

An executable assertion is software code that checks the validity of a predicate (generally concerning the value of a signal/variable) or a set of properties, in order to find errors either during system development or operation. This technique can detect errors caused by faults in sensors, actuators as well as internal signals in the computer node. In [Rela *et al.*, 1996], assertions based on the specification, the regularity of used data structures, the produced results, and the internal structure of the code, are presented.

In [Hiller, 2000], a classification of signals and suitable assertions were made. To evaluate the efficiency of different executable assertions, several signals of an arresting aircraft control system were monitored with assertions. In further work [Hiller *et al.*, 2001], [Hiller *et al.*, 2002] the propagation of data errors were analyzed using assertions and guidance on which signals to monitor with assertions provided. In [Jhumka *et al.*, 2002] an algorithm for designing consistent assertions, i.e., designing assertions of different signals that depend on each other so that they

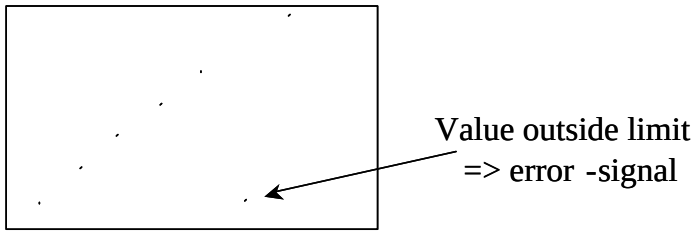


Figure A.1: Detection of a monitored property value outside the valid data space of the applied assertion.

have consistent value spaces, is proposed. This removes the possibility that the assertions will contradict each other so that one assertion accept the value of a specific signal where as another assertion do not accept it.

Some work has considered how to chose the value space for assertions. In the text below describing these approaches, we will use the term threshold to denote the borders for the acceptable value space of the executable assertions.

In [Stroph and Clarke, 1999], the threshold for the acceptable value space were computed using bounds on the properties for open-loop systems. Both assertions based on bounds for the magnitude and the rate of change for signals were developed both as static thresholds as well as dynamic thresholds. Dynamic means that the assertion has different thresholds for different operating points. It was shown in [Vinter *et al.*, 2001] that simpler checks based on the bounds of the control signal and system states together with a simple recovery technique (if an error was detected by an assertion, the old value of the signal was reused) could reach high detection coverage for transient faults. The integrator state was also identified as the most sensitive part of the system (which was confirmed in [Askerdal *et al.*, 2002]).

In order to protect the integrator, [Vinter *et al.*, 2002] implemented a robust integrator based on double execution of the integrator. Another approach was taken in [Gäfvert *et al.*, 2003] was to modify the anti-windup function. Anti-windup is generally used to reduce the influence of actuator saturation when an integrator is used (else the integrator could lock the control signal). However, instead of using the actuator saturation point for activating the anti-windup, the maximum control signal for the closed-loop system was computed, based on bounds on the reference value, which reduces the impacts of both actuator saturations as well as data errors.

In [Cunha *et al.*, 2001], the impact of data errors on the control of an inverted pendulum was investigated. In order to avoid failures, it was suggested to place a robust assertion checking the control error (the difference between the reference

(desired) value of a controlled physical process property and the actual value of this property). A robust assertion means that the assertion is executed twice and a simple code (called "magic number") is applied that both assertions have been executed (for more details on robust assertions, see [Silva *et al.*, 1998]). As assertions are simple, the overhead for making them robust, considering the compile system, is low even if the software code compared to simple assertions is duplicated.

The same authors proposed, in [Cunha *et al.*, 2002], to compute the impact of each control value on a model of the system online. This computation is termed oracle as it predicts the impact of the next computed control value on the system performance. If it was seen that the control value could make the system fail within a number of iterations, the output was blocked and the control handed over to a spare computer. In order to protect the oracle it was put in a robust assertion block.

Executable assertions can be used for faults resulting in erroneous signal values. The coverage that can be reached is very dependent on how predictable the checked signal is [Hiller, 2000]. The error detection latency will be the time from that the fault occurs until the signal, which an error caused by the fault propagated to, is checked. The ratio of unnecessary alarms should be low (false alarms should be 0) if the checked limits are derived from the specification, but to decrease the detection coverage, they may be set in other ways (e.g. based on measurements of the value of the signal) and than the ratio may become higher. The overhead for executable assertions is comparable low, but some additional computations (time) and memory are required. Quite accurate diagnosis can be achieved if the executable assertions are placed in an intelligent way.

Algorithm Based Fault Tolerance (ABFT)

ABFT is in some ways similar to executable assertions, but ABFT utilize information known about the algorithm of the calculation rather than information from the specification. Thus, for ABFT to be efficient, the algorithm should be static in the sense that the same computations are performed, more or less, independently on the input data. Applications where ABFT has been used so far are matrix multiplications, LU-factorizations, QR-decompositions, Fast Fourier Transforms and iterative solvers for partial differential equations.

In [Turmon and Granat, 2000] different implementations of ABFT for matrix multiplication is discussed. A solution on how to secure the inputs and outputs and correct execution flow of the ABFT is described in [Silva *et al.*, 1998]. In [Prata and Silva, 1999] ABFT is compared with another similar technique caller result-checking. Result-Checking utilizes that it is often simpler to check a result

than to compute it and thus the overhead for checking can be tolerated.

Assertions can also be combined with ABFT to increase the detection coverage for microprocessor errors, [Rela *et al.*, 1996].

For the applications where ABFT can be used, high detection coverage can be reached for errors affecting the calculations. The detection latency will be the time from that the error is generated until the calculation is checked. No distinction of which errors that are detected is usually made, so the ratio of unnecessary detections may be rather high. ABFT requires additional computations (time) and memory. ABFT does not seem to provide much information that can be used for fault diagnosis.

Certification

With certification, first the original algorithm is performed, but during the calculations some data is stored as a certification trail, that is used after the completion of the original algorithm, to speed-up the verification of the result. In [Sullivan *et al.*, 1995] and [Bright and Sullivan, 1995] it is described how certification can be performed for some usual problems, like for instance, sorting of elements in a list. They also compare certification with other techniques.

This technique can detect faults that affect the execution of the algorithm and the verification differently, i.e. the detection coverage is dependent on how diverse the verification is from the execution of the algorithm. The error detection latency will be the time from that the fault occur until the verification has finished and the number of unnecessarily detected faults is also dependent on how the verification is performed. This technique does not offer information that can be used for accurate diagnosis. The overhead is the additional computations (time) and memory required.

A.3 Signature Checking

Signatures are used to trace the control flow of the program execution. In most cases, the signatures are instructions inserted in the program by the compiler to indicate legal control flows. At run time, a monitor (implemented either in hardware or software) calculates signatures and compares them with the stored signatures in the program. If they are different. an error has been detected.

The earliest suggested signature techniques were based on compiler-hardware solutions (e.g, in [Mahmood and McCluskey, 1988] different designs of signature checking are surveyed, [Wilken and Shen, 1990] derives the coverage-dependence

on the implementation., and, [Ohlsson and Rimén, 1995] describes one possible implementation). The problems with these solutions is that the monitor is implemented in hardware, thus the recurring cost is increased. Furthermore, it can be hard to handle modern architectures including out-of-order execution, cache memories, branch prediction etc.

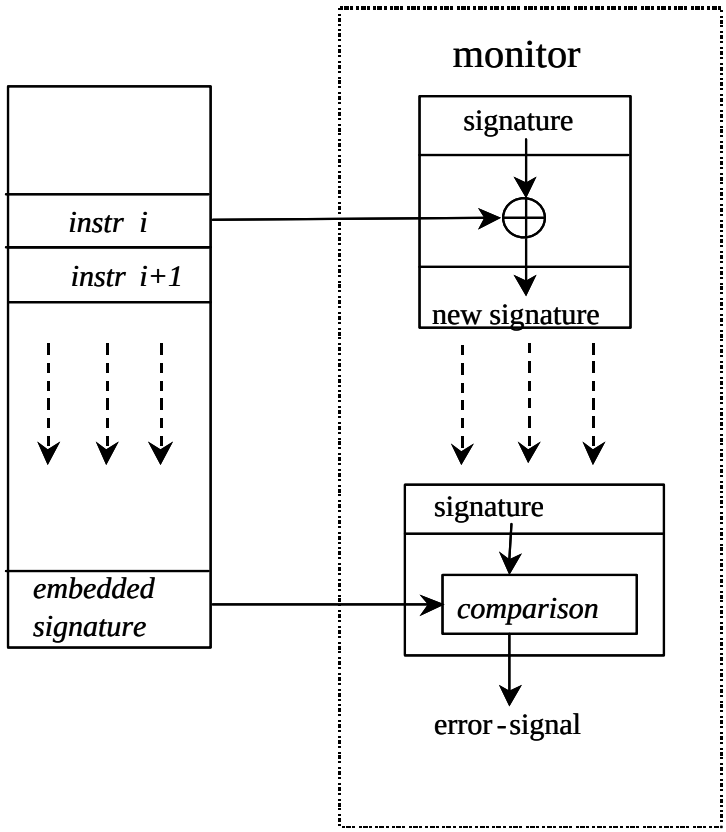


Figure A.2: Example of error detection using signature checking.

However, recent solutions have been compiler-software solutions. These solutions are not affected by the hardware architecture, e.g., [Alkhalifa *et al.*, 1999], [Bagchi *et al.*, 2001], [Oh *et al.*, 2002(A)].

Another complication is interrupts, i.e., events forcing the microprocessor to halt the present execution and execute specific interrupt routines. This mean that the control-flow is abruptly and significantly changed. Also here software-based solu-

tions seem to have an advantage as each task will have individual signatures which only are updated when the task is executed, i.e., which are not affected if other instructions are executed. However, one related drawback with software-implemented assertions is that if an error occurs, it cannot be guaranteed that the signature is checked, i.e., the error could result in that the signature checks never are executed, and thus, the error is not detected.

The detection coverage for this technique depends on how the signatures are calculated. If the signature is calculated based on the instruction codes, the coverage will be very high, but this requires specific hardware since commercial components does not utilize this. Software implementations generally have reduced coverage since the checking has to be based on more sparse information.

A.4 Self-Tests

Self-tests are tests implemented in hardware and/or software to test some specific component of the system. Traditionally they have been used for detecting developmental faults after production, but self-tests could also be used for detection of faults during system start-up and operation. However, in such cases where they are used during system operation, they must be designed so that they will not affect the normal operation. Therefore, it is hard to implement tests for all sensors and actuators, but it may be cost-efficient to use tests for microprocessors and/or communication controllers that are not fully utilized and run them during their idle time.

In [Steininger 2000], hardware BISTs are surveyed. The detection of dormant faults through BISTs is analyzed, in [Steininger and Scherrer, 1999]. The design of BISTs for TTP ([Kopetz and Grunsteidl, 1994]) controllers is discussed in [Steininger and Temple, 1999] and [Steininger and Vilanek, 2002].

Self-tests for digital signal processors (DSP) cores using the existing hardware of the DSP is suggested in [Radecka *et al.*, 1997]. In [Nicoladis, 1995] BIST and self-testing are merged to give high coverage with a decreased hardware overhead. Similar ideas are presented in [Gupta and Pradhan, 1996] where the BIST is utilized for implementing on-line checkers.

Software-implemented design of a self-test for microprocessors is described in [Shen and Abraham, 1998]. In [Batcher and Papachristou, 1999] a similar method is described. In [Talkhan *et al.*, 1989], instructions are divided into classes after which errors they will detect. This means that not all instructions need to be executed when testing the processor. Also [Paschalis *et al.*, 2001] and [Chen and Dey, 2001] presents methods for testing microprocessors with software implemented tests. De-

sign of Software-Implemented Self-Tests for on-line detection of errors is also discussed in Appendix C.

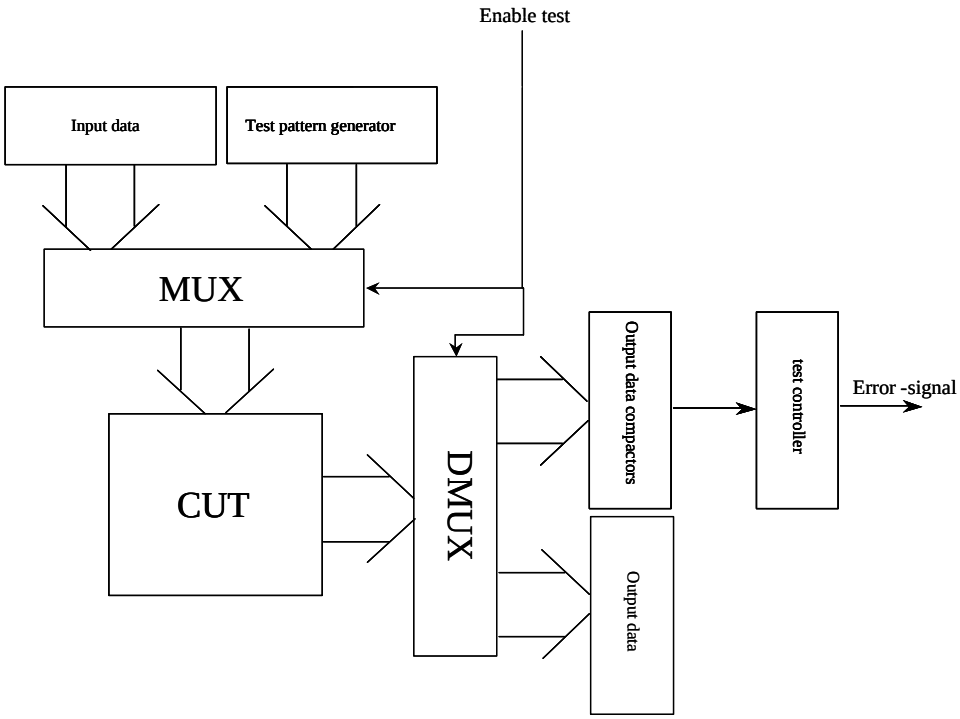


Figure A.3: Error detection using Built-In Self-Tests (BISTs).

This technique is used for detection of persistent faults in the computer nodes and the coverage is dependent on how long the test is, i.e. how many test-vectors that are included. The error detection latency is dependent on how often the test is run and the overhead is the time required for running the test and the memory for storing it. If the circuit already contains a hardware Built-In Self-Test (BIST) for detection of developmental faults, the overhead may be reduced. The diagnosis accuracy can be high (especially for BIST) since it is possible to control which parts of the hardware that should be tested.

A.5 Double Execution

Double execution means that a program is executed twice with the same input data. Then the results from the executions are compared. Different implementations are surveyed in [Askerdal *et al.*, 2000]. In [Aidemark *et al.*, 2003] analysis of which errors can be detected are performed (some of the analysis can be found in Appendix B). In [Rimén, 1995] and [Ohlsson and Rimén, 1995] double execution is implemented and the error detection coverage evaluated through fault injection simulations.

In [Johnson, 1989], different time redundancy approaches for detection of errors caused by both transient and permanent faults are surveyed. The techniques for detection of errors caused by permanent faults are based on introducing diversity by computing the second execution with the complement of the input data, with shifted operands, or with swapped operands, etc. Diverse programs are created in [Lovric 1996] by the use of different compilers, mathematical rules, force of different register usage, etc. Then, both versions are run after each other and the results are compared. The purpose of using diverse versions is to increase the error detection coverage for permanent faults.

In [Aidemark *et al.*, 2002], tasks are double executed, but if an error is detected and no hard deadlines are violated, the task is executed a third time and a majority decision on which result to deliver is taken.

In [Rotenberg, 1999], a modern simultaneous multithreading microprocessor running double execution as two threads separated with a certain time interval, is described. It also utilizes that information about branch prediction etc. from the first thread can be reused be the second thread to reduce the execution time. This technique has been followed up in a number of different papers. A survey of these can be found in [Lai *et al.*, 2002].

A technique where the instructions are duplicated by the compiler is described in [Oh *et al.*, 2002(B)]. Here no hardware modifications are necessary, but the instructions can still be scheduled to utilize the architecture of super-scalar processors to reduce the time overhead.

This technique detects errors that affects the two executions differently (i.e. mostly transient faults) and that not prevent the result comparison. The overhead is the additional execution time caused by the second execution (not necessarily > 100% [Rotenberg, 1999]) and the comparison and the error detection latency is the time from that a fault occur until next comparison, i.e. dependent on how often the task is interrupted to start the second execution. If a fault is detected it can be assumed that the fault occurred in the microprocessor or the communication controller,

but better diagnosis accuracy than so is hard to achieve.

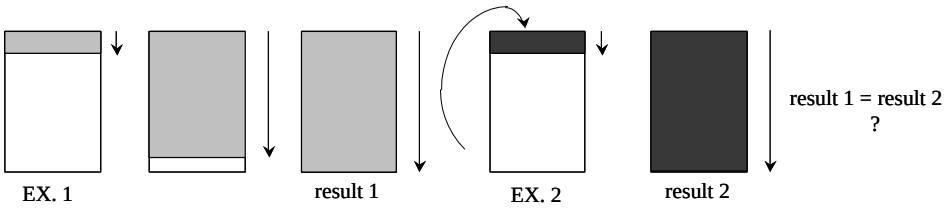


Figure A.4: Error detection using double execution.

A.6 Diversity

That faults can be detected through diversity has been known for a long time, but [Avižienis and Chen, 1977] was one of the first attempts to design diverse software in order to tolerate software faults. The main idea is to divide different (diverse) programs that should provide the same function. Assuming that these fail independently, errors can be detected by comparing their results. However, research has indicated (e.g., [Littlewood and Miller, 1989], [Feldt, 1999]) that software fail dependently even if it has been designed independently. Moreover, as the software overhead is large, this technique may significantly increase the fault intensity. This also means that the time overhead is large.

One way to reduce the software overhead is to achieve diversity through the input data rather than the software itself, i.e., use diverse input data. This technique, called data diversity is discussed in [Ammann and Knight, 1988], [Bishop, 1993], and [Christmansson *et al.*, 1998]. Generally, in these studies, several errors were detected, but the coverage is very application and fault dependent. It is also hard to compare results produced from different input data.

As described in Appendix A.5, diversity has also been applied to detect errors caused by hardware faults.

A.7 Hardware Replication

By comparing results from replicated units, errors caused by any faults can be detected, if not all units are affected by the dame fault (for instance developmental faults). It is important that the replicated components are synchronized and fed with

exactly the same input data to be able to perform a correct comparison (error masking through voting, N-Modular Redundancy (NMR), [Pradhan, 1996], [Yeh, 1996], has the same problem). Another problem is how to determine which of the units that is erroneous and how the comparison should be performed.

In [Scobie *et al.*, 2000], a fault tolerant design of a brake by wire system using hardware replication, is described. For fault diagnosis (isolation), duplicated systems (DUPLEX) need some additional test to determine which of the units that was erroneous. In [Feick *et al.*, 2000] a similar steer-by-wire system is presented.

In order to detect crash failures, on single oscillator circuits, the "I'm alive" message protocol, see [Pradhan, 1996], [Dimmer, 1985], was introduced. With this technique messages are transmitted between different computer nodes with even, predetermined time intervals. If a message do not arrive punctual, an error is assumed to occurred.

The overhead is the hardware for replication of components and the comparer and the time to compare. Even if the approach is capable of detecting all types of faults with high coverage, the technique is often considered to be expensive due to the cost for the additional hardware. The error detection latency is the time from the fault occurs until the results from the two units have been compared.

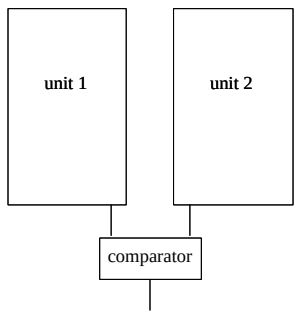


Figure A.5: Error detection using hardware replication.

A.8 Watchdog Timers

When the execution time of an event (e.g. task) is known, timers can be used to monitor it, i.e. so called watchdog timers. For instance, consider an event, i, that will take place at least every 12ms. To check this property a timer could be set for 12ms (an additional safety marginal may be added) and reset when the event takes place. If the timer reaches 12ms an interrupt is generated to indicate that an error

has occurred. This technique is often used to detect incorrect execution order inside the microprocessor and/or the communication interface of the computer node.

One important issue to consider is that if the monitored component and the timer itself is clocked with the same oscillator, they will have a common failure point. Often oscillator faults are considered to be very unlikely, but if the fault intensity is to high, replicated oscillators can be used or timers on another computer node (and thus running with another oscillator) can be used by the transmission of so called "I'm alive" messages, see Chapter A.7.

In [Miremadi *et al.*, 1995], a technique based on watchdog timers for monitoring the execution order of a microprocessor is described. A watchdog timer is implemented as a "bus guardian" in [Temple, 1998] to secure that the communication controller does not transmit data on the communication bus at incorrect times.

The detection coverage is very dependent on how predictable the execution times of the monitored tasks are. The detection latency depends on the time interval between the events. The overhead is the required execution time for setting the timer that should be low (watchdog timers often exist on-chip so no additional hardware is required). The number of unnecessary detected errors is low since most control-flow faults are harmful and the false alarm rate will generally be low. It is hard to achieve diagnosis with high accuracy.

A.9 Coding

Coding means adding redundant bits for representing data values. Just adding bits would result in that the bit combinations in the interval [max value+1, max bit pattern] would be invalid, i.e., never naturally arise. However, by intelligently transforming all combinations, the invalid combinations can be spread so that faults for instance resulting in bit-flips will with high probability result in an invalid combination and therefore be detectable.

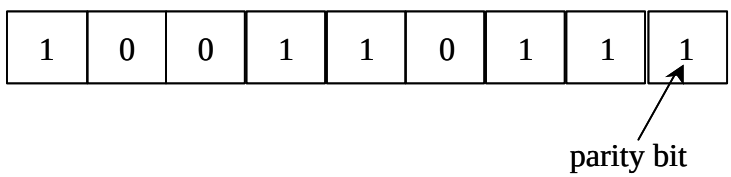


Figure A.6: Even parity bit coding.

Coding can be used to detect any errors affecting data. However, coding of

data transformations (i.e., arithmetic and logic operations) is often very costly, and thus, it is in most cases only used to secure stored or transferred data (e.g. on the communication bus).

One of the simplest and most used coding techniques is parity coding, see Fig A.6. Parity coding adds one bit (the parity bit) to the data value to indicate if the number of ones in the data value is even or odd. The code word is validated by checking that the parity bit and the number of ones (odd or even) agree. This code detects all single errors.

The dual-rail code takes the information bits and takes the complement of them and adds the complement as check bits, and thus, detects all errors with exception for those that booth changes an information bit and the corresponding check bit. This code will need more than a duplication of the hardware.

The Berger code is an optimal separable unordered codes, i.e. the information and check bits can be separated and a code word can not be transformed to another code word by changing some of its bits from $0 \rightarrow 1$ or by changing some of it bits from $1 \rightarrow 0$. So the only way a code word can be transformed to another code word is by having at least one bit flip from $0 \rightarrow 1$ and at least one from $1 \rightarrow 0$. This means that this code will detect all unidirectional errors.

Arithmetic codes with base A are built by either taking the check part equal to the modulo A of the information bits (separable code) or equal to the product of A and the information bits (non-separable code). The advantage with arithmetic codes is that they are preserved through arithmetic operations.

Error detecting and correcting codes such as for instance SEC/DED Hamming codes, Reed-Solomon codes, BHC codes, etc. are often used for error correction in memory systems and transmission channels. Cyclic codes, for instance CRC, are often used for message transmissions in communication systems, [Pradhan, 1996].

In [Elliott and Sayers, 1990] a 32-bit RISC, using a Hamming code for error detection and correction was designed. All single faults with exception for faults on the input latches to the ALU were detected. The hardware overhead needed was 112% and the processor had a performance-loss of 50% compared to the same processor without the error detection capability.

In [Lo *et al.*, 1992] a technique for designing a strongly fault secure ALU using the Berger code is presented. This article also demonstrates that a self-checking processor can be designed by just using the Berger code. If the ALU operation can be overlapped with the check symbol evaluation of the last instruction the extra delay time for using this type of ALU is negligible. The hardware overhead is less than 100% for the total self-checking processor.

In [Maamar and Russel, 1998] an implementation of a 32-bit RISC processor

using Dong's code (a modified Berger code) for error detection is described. The approach presented in [Lo *et al.*, 1992] is used. The processor is able to detect many unidirectional errors. As before the extra time delay can be neglected if it is allowed to pipeline the checking part with the next instruction's ALU operation. The hardware overhead is said to qualitatively be much less than duplication.

The error detection latency is dependent on how often the codes are checked, but is in most cases low. The diagnosis accuracy can be rather high and one specific attractive property of coding is that not only can errors be detected, but also corrected, which reduces the number errors that must be handled with other methods.

A.10 On-Line Monitoring of Reliability Indicators

One way to detect errors in the components is to monitor the current consumed. Many faults cause abnormal power consumption and can therefor be detected by either external sensors or Built-In Current Sensors (BICS). External sensors are much slower than BICS so for many applications BICS are the only possible solution. However there are many problems left to be solved for BICS such as, they have to be as fast as the monitored circuit and still offer good resolution, they must not have a negative effect on the circuit's performance and the circuits clock must be slowed done so that the steady state of the current is measured. To avoid these problems, periodic monitoring of the current could be used, but this decreases the detection coverage for transient errors which then must be detected using other methods.

Other parameters can also be used for error detection, such as temperature, voltage-level for logic circuits, output signal activity and total dose of radiation [Nicolaidis and Zorian, 1998]. However, these are more difficult to measure efficient, so they can so far only be used as complement to other methods.

This technique detects faults in the computer nodes and the coverage is dependent on where at the circuit the faults occur. The coverage that can be reached is generally not high enough for safety-critical systems so it can only be used as a complementary technique. However, the number of unnecessarily detected errors is usually low. The error detection latency is the time from the fault occurrence until a high enough erroneous current is produced so that an interrupt is generated. This time is generally low. The overhead is the sensor for measuring the current. It is hard to achieve diagnosis with high accuracy using this technique.

A.11 Summary

In this appendix, different error detection techniques have been described. To summarize, it is not only the coverage of an error detection technique that is of interest, but also detection latency, number of unnecessary alarms, diagnosis accuracy and overhead. Table A.1 shows approximate quality estimates of these properties for the different described techniques. It should be emphasized that these estimates are rough and are very dependent on the implementation of the error detection technique.

Table A.1: Rough qualities of different error detection techniques.

Technique	*Targeted Errors*	*Overhead*	*Coverage*	*Unnecessary Alarms*	*Latency*	*Diagnosis Accuracy*
Functional Redundancy	Data Errors	Time, Memory	High	Few-Several	Short-Long	Bad-Medium
Executable Assertions	Data Errors	Low Time, Low Memory	Medium-High	Few-Several	Short	Medium-Good
ABFT	Data Errors	Time, Memory	High	Few-Several	Short	Bad
Certification	Data Errors	Time, Memory	Medium-High	Few	Short-Long	Bad
Signature Checking	Control-Flow Errors	HW, Low Memory or Time Memory	Medium-High	Few	Short-Long	Bad-Medium
Self-Tests	Data Errors	Time, HW or Memory, Time	Medium-High	Few-Several	Short-Long	Medium-Good
Double Execution	Data Errors	Long Time, Memory	High	Few-Several	Short-Long	Bad
Design Diversity	Data (Design) Errors	Long Time, Large Memory	High	Few-Several	Short-Long	Bad
Data Diversity	Data (Design) Errors	Time	Medium-High	Few-Several	Short-Long	Bad
Hardware Replication	All Types Except Design	High HW	High	Few-Several	Short-Long	Bad-Medium
Watchdog Timers	Crash Failures and Control Flow Errors	Short Time, Low Memory	Medium-High	Few-Several	Short-Long	Bad
Coding	Data Errors	Low Memory	High	Few	Short	Medium-Good
Monitoring of Reliability Indicators	Data Errors	HW, Low Memory	Medium	Few	Short-Long	Bad

Appendix B. Analysis of Double Execution

Double execution, i.e., executing a task twice and comparing the results of the two executions, is a commonly proposed error detection technique based on time redundancy. Most previous research has only considered which transient errors that can be detected. However, in this appendix an attempt is made to estimate the detection coverage for errors caused with varied duration. The reasons for this investigation are:

- The occurrence rate of persistent faults can be expected to increase (see, Chapter 2).
- Double execution is a cost-efficient error detection technique considering recurring costs, i.e., no additional hardware is necessary, which makes it desirable to use for many systems.
- For fault diagnosis it is important to know which faults that can generate errors that are detected by a certain error detection technique.

A first approach for estimation can be found in [Askerdal *et al.*, 2000] which was developed further in [Aidemark *et al.*, 2003] and [Aidemark and Askerdal, 2003] from which the rest of this appendix is fetched.

Double execution detects all faults that will make the results of the two executions differ, i.e., generates a data error in the result. Considering a transient fault, such a fault will only generate a data error that can affect one of the two executions unless it occurs in a signal/variable/etc. that is stored and not updated between the executions. This means that most data errors generated by transient faults will be detected. Moreover, this fact is valid for all faults with shorter duration than the time to execute the task once[1].

[1] Tasks can be divided into smaller parts of which the first part first is executed twice and the results compared. Then is the second part double executed and so on until all parts have been double executed. For this division, all faults with shorter duration than the time to execute one part once will give different results, and thus, be detected.

It should be noted that transient faults can generate errors in internal states that not directly affect the result. However, if the states are updated separately for the two executions, these errors will eventually be detected when they affect the result (if the result never is affected by the error, the error is not likely to have any significant effect on the system either). To shorten the detection latency of such errors, internal states etc. can also be compared between the executions.

Whether the two executions will generate different results for persistent faults is dependent on where in the execution the fault occur, and when (if) it is activated. The probability for a data error, caused by a persistent fault, to be detected will be analyzed in the next section.

B.1 Detection of Data Errors Generated by Persistent Faults

In this section, the probability for errors caused by persistent faults to be detected for a processor executing a single task first will be analyzed. Then, the analysis will be extended to execution of several, non-preemptive, periodic tasks, and finally are preemptive and sporadic tasks discussed. The assumptions for which the analysis are valid are:

- **A1:** The probability that a fault occurs is evenly distributed over the Least Common Multiple time (LCM) of the periodic tasks, i.e., the time until the task schedule is repeated. As the Least Common Multiple time often is comparably low compared with the Mean Time To Failure (MTTF), this assumption is generally valid.
- **A2:** The time to execute the one instruction is considered negligible compared to LCM. This assumption is used in order to in the analysis neglect the instruction executing when the fault occurs.
- **A3:** If an error is generated, it will result in an incorrect result. This implies that no error masking is considered. In [Aidemark *et al.*, 2003], errors were seldom masked. However, only a simple workload was investigated so whether this assumption is valid also for more complex workloads needs to be further verified.

B.2 A Single Double Executed Task

As stated in the previous section, double execution detects errors if the two executions produce different results. This can only happen for a persistent fault it occurs

after the first execution and before the second execution has finished. This implies that errors caused by developmental faults will not be detected (if identically the same task is re-executed).

An example of a single double executed task, T_A, is shown in Figure B.1. The time redundant replicas are denoted $T_{A,1}$ and $T_{A,2}$ respectively. When the task has been executed twice, the results are compared and then the microprocessor is idle until the next invocation of the task.

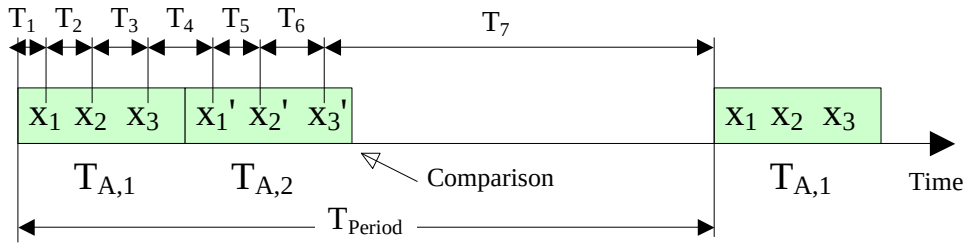


Figure B.1: A single double executed periodic task.

The x_1, x_2 and x_3 represent one specific instruction that is executed three times in the task. The x_1', x_2' and x_3' is the repeated instructions in the second execution of the task. The time intervals T_1 to T_7 represent the intervals between the executions of the instructions. The time to execute the actual instruction is considered negligible. The probability that an error is detected depends on the point in time a fault occurs and the probability that a fault is activated.

First, let P_i denote the probability that a fault occur in a certain time interval T_i. Using assumption **A1**, P_i can be computed as:

$$P_i = \frac{k \cdot T_i}{k \cdot LCM} = \frac{T_i}{LCM} \tag{B.1}$$

where, T_i is the length of time interval i, k is the total number of periods, and LCM is in this case is equal to the period time of the task.

Second, let Px_j denote the probability that a fault is activated when a certain instruction x_j is executed. Px_j can then be calculated as:

$$Px_j = \frac{\sum_{k=1}^{M} Input_{act,k}}{M} \tag{B.2}$$

where M is the number of possible inputs for the specific instruction and $Input_{act,k}$ is 0 for the inputs that did not activate the fault and 1 for the inputs that activated the

fault. Note that different subsets of the M possible inputs may be used for different executions of x. The use of each subset must then be considered as a different instruction.

Let Px, tot be the total probability that a fault is activated by any of several executions of a specific instruction x. The probability Px, tot that the fault is activated can be derived for two subsequent executions of instruction x, x_1 and x_2, as:

$$Px, tot = P(At least one of the instructions\, x_1 \, or \, x_2 \, activates \, the \, fault)$$
$$= 1 - P(x_1 \, does \, not \, activate \, the \, fault \, AND \, x_2 \, does \, not \, activate \, the \, fault)$$

Since we assume **A3**, i.e., that no errors is masked, the probability of activating a fault in instruction x_1 is independent of the probability of activating a fault in instruction x_2, Px, tot , and can be expressed as:

$$Px, tot = 1 - (1 - P(x_1 \, activates \, the \, fault) \cdot (1 - (P(x_2 \, activates \, the \, fault))$$

which can be generalized to:

$$Px, tot = 1 - (1 - Px_1) \cdot (1 - Px_2) ... \cdot (1 - Px_i) \tag{B.3}$$

where i is the number of repeated executions of instruction x.

Using equations B.1 through B.3, the probability for detecting an error in the example given in Figure B.1 can be derived: If a fault occurs during time interval T_1 or T_7 it can never be detected since it will always affect $T_{A,1}$ and $T_{A,2}$ in the same way. However, if the fault occurs during time interval T_2, errors will be detected if the first instruction of $T_{A,2}$ (x'_1 in Figure B.1) activates the fault. Thus the probability that an error is detected, $P(D)$ (where D is the set of detected errors), can be computed as $P(D) = P_2 \cdot Px'_1$, where P_2 is the probability that a fault occurs in time interval T_2, and Px'_1 is the probability that a fault is activated when executing instruction x'_1.

Deriving $P(D)$ for the remaining time interval T_3 or T_6 in the example in Figure B.1 is done in the same way, i.e.: If the fault occurs during time T_3, the error will be detected if either the first and/or the second instruction of $T_{A,2}$ (x'_1 or x'_2 in Figure B.1) activate the fault which gives $P(D) = P_3 \cdot (1 - (1 - Px'_1) \cdot (1 - Px'_2))$. For T_4 there are three possibilities that the fault can be activated (by x'_1, x'_2 or x'_3 in Figure B.1), $P(D) = P_4 \cdot (1 - (1 - Px'_1) \cdot (1 - Px'_2) \cdot (1 - Px'_3))$. At T_5 there are again only two possibilities that the fault is activated differently between $T_{A,1}$ and $T_{A,2}$ (by x'_2 or x'_3 in Figure B.1), $P(D) = P_5 \cdot (1 - (1 - Px'_2) \cdot (1 - Px'_3))$, and during T_6 there is only one (x'_3 in Figure B.1) , $P(D) = P_6 \cdot Px'_3$.

We assume that the probability that an instruction x activates a fault is constant for all executions of the instruction, i.e. $Px = Px_1 = Px_2 = Px_3$. This is reasonable if the same subset of the M possible inputs is used for each execution of x (see equation B.2). The total probability, $P(D_x)$, of detecting an error generated by a persistent fault for one task can then be computed as:

$$\begin{aligned} P(D_x) &= P_2 \cdot P_x + P_3 \cdot (1 - (1 - P_x)^2) + P_4 \cdot (1 - (1 - P_x)^3) \\ &+ P_5(1 - (1 - P_x)^2) + P_6 \cdot P_x \end{aligned} \tag{B.4}$$

Equation B.4 can be generalized for an arbitrary number of time intervals to:

$$\begin{aligned} P(D_x) &= \left(\sum_{i=1}^{(N/2)-1.5} [P_{i+1} + P_{N-i}] \cdot [1 - (1 - P_x)^i] \right) \\ &+ P_{N/2+0.5} \cdot [1 - (1 - P_x)^{N/2-0.5}] \end{aligned} \tag{B.5}$$

where N is the number of intervals (in this case 7 for T_1 to T_7). Note that the number of intervals N is always odd since the number of instructions always is even as the tasks are executed twice.

If a set, L, of different instructions each activate a certain fault, the probability of detecting the fault, $P(D)$ is the probability of the union of the detected faults for each instruction $l \in L$:

$$P(D) = P\left(\bigcup_{l=1}^{L} D_l \right) \tag{B.6}$$

This can be explained by the following example. Consider a fault that can be activated by two different instructions x and y and that a task containing one instruction of each is executed. The probability for activating a fault with each instruction is Px, and Py respectively. The instructions are distributed in the task according to Figure B.2.

Using equation B.5, the detection probability for each instruction, $P(D_x)$ and $P(D_y)$, can be computed as:

$$P(D_x) = P_2 \cdot P_x + P_3 \cdot P_x \tag{B.7}$$

$$P(D_y) = P_3 \cdot P_y + P_4 \cdot P_y \tag{B.8}$$

As we assume that all generated errors will result in an incorrect result, **A3**, the probability of activating a fault in instruction x is independent of the probability of

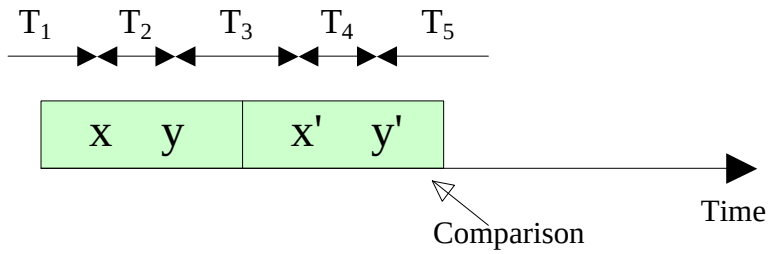


Figure B.2: A double executed task containing two multiply instructions.

activating a faults in instruction y. This implies that the total probability, $P(D)$, can be computed according to equation B.4 as:

$$P(D) = P_2 \cdot P_x + P_3 \cdot (1 - (1 - P_x) \cdot (1 - P_y)) + P_4 \cdot P_y \tag{B.9}$$

Inserting equation B.6 and B.7 in B.8 gives:

$$P(D) = P(D_x) + P(D_y) - P_3 \cdot P_x \cdot P_y \tag{B.10}$$

Since:

$$P\left(D_x \bigcup D_y\right) = P(D_x) + P(D_y) - P\left(D_x \bigcap D_y\right) \tag{B.11}$$

and $P(D_x)$ and $P(D_y)$ are independent i.e.:

$$P\left(D_x \bigcap D_y\right) = P(D_x) \cdot P(D_y) \tag{B.12}$$

Then we need to show that:

$$P(D_x) \cdot P(D_y) = P_3 \cdot P_x \cdot P_y \tag{B.13}$$

Inserting equation B.7 and B.8 in the left hand side of equation B.12 and using the information that P_2, P_3 and P_4 are mutually exclusive, i.e. (the time intervals T_2, T_3 and T_4 do not overlap), thus $P_2 \cdot P_3 = 0$, $P_3 \cdot P_4 = 0$, $P_2 \cdot P_4 = 0$ and that $P(i \mid i) = 1$, thus, informally, $P_i \cdot P_i = P_i$, the right hand expression of equation B.12 corresponds to the left: $P(D_x) \cdot P(D_y) = (P_2 \cdot P_x + P_3 \cdot P_x) \cdot (P_3 \cdot P_y + P_4 \cdot P_x) = P_3 \cdot P_x \cdot P_y$

B.2.1 Multiple Periodic Tasks

There may be several double executed tasks in a schedule. The probability of detecting errors generated by persistent faults can thereby be estimated based on all double executed tasks in the interval where all periodic tasks are invoked at least once. This interval is called the Least Common Multiple time (LCM) [Leung and Whitehead, 1988]. This means that for such tasks, the detection probability for all tasks in the LCM interval can be computed from the equations in the last section as:

$$P(D_{LCM,x}) = \frac{1}{LCM} \sum_{k=1}^{Q} T_{Period,k} \cdot P(D_{x,k}) \tag{B.14}$$

where Q is the number of double executed tasks within LCM.

B.2.2 Preemptive and Sporadic Tasks

Some systems allow preemption of tasks, i.e., higher priority tasks can interrupt lower priority tasks and after the higher priority task is finished, the lower priority task can resume its execution. This means that the time between the instructions in the lower priority task can increase (a time interval T_i can increase), and thus, the probability of detecting errors will be higher (according to equation B.1 and B.5). Therefore, the lowest error detection probability is reached when no tasks are preempted.

If sporadic tasks are considered, there must be enough resources to handle them. This is often achieved by assuming that a sporadic task cannot be invoked more often than a certain period time. This means that the LCM time must encompass the additional time for executing sporadic tasks. The lowest detection probability is reached when a sporadic task is not executed during the LCM time, then the processor idles and no errors can be detected during this time period.

B.3 Verification of the Analysis

Verification of some of the assumptions for this estimation approach can be found in [Aidemark *et al.*, 2003], where fault injection experiments were performed for the multiplication instruction. The results supported that:

1. Faults occurring after the double execution of a task has finished and before the double execution of the next task has started, will not be detected.

2. If an error occurs, it is seldom masked by further instructions.

Further verifications are necessary. However, the most important reason for adding double execution seems to be for detection of errors generated by faults in the functional units (ALU, multiplier, etc.) which this initial verification targeted.

B.4 Summary and Discussion

In this appendix, which errors that can be detected with double execution has been analyzed. As double execution will detect data errors that generate different results of the two executions, all faults with shorter duration than the time to execute the task once, generating errors in the result, will be detected.

For faults with longer duration, equations for estimating the detection coverage were derived. What these equations basically say is that the probability that an error is detected is binomially distributed $\sim Bin(n, p)$, where n is the number of times an instruction is executed that may activate the fault, and p, the activation probability for the specific fault within a certain time-interval.

Even if it can be shown that the detection probability of data errors generated by persistent faults can be increased by using the equations for scheduling of periodic tasks, as was shown in [Askerdal *et al.*, 2000], [Aidemark *et al.*, 2003]. However, this scheduling policy may not always be adaptable (it may for instance increase the detection latency and response time), and even so, the coverage may not be sufficient for faults with low activation probability. Even if such faults generally have less impacts than faults that are activated often, they may still be required to be detected. Thus, other detection techniques may need to be applied for detection of errors caused by persistent faults.

Another merit with this analysis is that it shows if an error is detected by double execution, it cannot automatically be assumed that the cause of the error must have been a transient fault. Thus, more advanced diagnosis methods are required.

Appendix C. Self-Test Tasks

In this appendix, the possibility of detecting data errors caused by persistent faults with software-implemented self-tests is discussed. The idea is only outlined and evaluations are needed to estimate the efficiencies of the proposed tests.

In Chapter2, the functional units of modern processors were identified to be one of the most vulnerable spots of a microprocessor, and thus, need to be protected. As hardware replication is costly for systems that are manufactured in large numbers, and it is complicated to protect these units with coding, other solutions would be preferable.

For mass-market products, it is important to restrict the recurrence costs. In that sense, error detection techniques based on time redundancy are preferable as they do not generally require any additional hardware, (in Appendix A error detection techniques based on different types of redundancy are surveyed). For detection of data errors caused by transient faults, double execution is an efficient technique, but for faults with longer duration (persistent faults), double execution may not be so efficient (see the analysis in Appendix B) if not diverse programs are used or the program executed on diverse units. However, diversity introduce some problems:

- The number of generated errors may increase due to the diversity.
- It can be hard to diagnose which type of fault that caused the error. That is, how can errors generated by transient faults be separated from errors caused by persistent faults?
- The fixed cost may be increased, since several versions of the program need to be developed.
- It can be hard to estimate which detection coverage that can be reached, as the efficiency of diversity generally is dependent on the specific hardware platform.
- The solutions may not be portable.

Due to these reasons, we propose to use software-implemented self-tests instead.

Previous online test approaches has mainly been based on on-line Built-In Self-Tests (BISTs), since software self-tests has been considered too slow. However, we challenge this viewpoint due to the following reasons:

- The tests do not need to detect persistent faults in the entire microprocessor, but can be focused to the functional units (see Chapter 3). This reduces the required test length drastically.
- Unlike manufacturing testing, on-line testing is not aimed to detect all existing faults, but to avoid data errors to be activated in such rate that a system failure could occur (for instance, it was discussed and shown in Chapter 4 that many transient data errors are tolerated by control systems).
- BIST solutions require support from the manufacturing company and implies a switch between normal execution and executing the BIST which is a potential safety risk. This is not the case with software-implemented tests as these are executed as normal tasks.

Thus, in the continuation of this appendix, ideas on how to design software-implemented self-tests are developed.

C.1 Design of Software-Implemented Self-Tests

We visualize a software-implemented self-test to consist of a certain number of different instructions with different input data. Therefore, designing a test implies defining which and how many instructions, that should be included, which input data to use for these instructions, and how often the test should be executed. The more often the test is executed and the more combinations of instructions and input data that are used, the higher will the error detection coverage be. On the other hand, the time-overhead for executing the test will increase.

One way of balancing the coverage and overhead is to split the execution of the test in parts in between which the “normal” system tasks can be executed. In such case, the tests can be run during the processor idle time. However, dividing the tests in parts requires that partial results are stored which increases the overhead.

Now, different ideas on how to select the instructions, the number of instructions (the length), input data, and execution interval for the test will be discussed. Finally the possibility to auto-generate tests is discussed.

C.1.1 Test Instructions

The instructions of a program determine which operations that are performed. Therefore, by determining which type of instructions that should be used, the test can be focused to certain operations, i.e., specific parts of the hardware.

A major difference between testing and fault tolerance is that testing is aimed to detect as many faults as possible, whereas fault tolerance, in order to save resources, should only detect those faults that could harm the system. Therefore, as a first selection, it seems to make sense to only use instructions in the test that are also used by the safety-critical tasks of the system, i.e., the test is focused to the operations that are used by the application.

Another selection strategy that we propose is to only use instructions utilizing the functional units. The reason for this is that, as discussed in Chapter 3.6, persistent faults in other parts of the microprocessor will in most cases behave as corresponding transient faults, and thus, be detected by the same error techniques. However, the behavior of faults in the functional units are dependent on their duration (see, Chapter 5.2.1). Furthermore, these units are some of the least protected parts of a microprocessor, [Mendelson and Suri, 2000].

To summarize, we propose to only use instructions that utilize the functional units, and that also are used by the safety-critical tasks of the system.

C.1.2 Input Data

Most instructions require input data, i.e., on which data the operation of the instruction should be performed. For instructions utilizing the functional units, which we in the previous section identified to be the most suitable target for self-tests, input data is generally numbers.

The overhead for using a large number of input data is not only the execution time, but also the memory for storing the input data and the predetermined (golden) results for comparison with the online computed results. One way to reduce the data that need to be stored is to link instructions so that the output data from one instruction is the input data for the next instruction and so on. In this way, ideally, only one pair of input data and one result need to be stored. However, this makes it harder to divide the test in several parts and there is also the possibility that faults that are activated once, are masked by a later executed instruction.

As was discussed in the previous section it is preferable to only test operations that are performed by the real system tasks, and thus, input data outside the specified value space of the system should be avoided. However, for many systems, the valid value space is large.

One way to adapt the test to the data used by the application is to use the same data. The problem with this is that the result of an operation cannot be determined in advance, i.e., off-line. One solution is to let the test consist of executing the application program backwards. In this way, the test starts with the output data of the application and computes the input data which then can be compared with the original input data. As an example of this, take a simple task (e.g., most control algorithms consist of simple computations) that computes: $(a \cdot b + c) \cdot d = e$. Then the test program would take e as input data and compute: $((e \cdot (1/d)) - c) \cdot (1/b) = a^*$, and compare the computed a^* with the original input data a.

Another idea is to use diversity as suggested in Ref. [Lovric 1996]. An example of this is to compare computations with shifted data as for instance that $a + b$ should equal $(a \cdot 2 + b \cdot 2)/2$.

C.1.3 Test Length

The test length is the total number of instructions of the test. In order to determine this, first, which instructions that are necessary to include should be determined, for instance as was done in Chapter C.1.1. Second, it is necessary to determine how many times each instruction should be included in the test. Assuming that faults activated once will not be masked by further test instructions, and that the probability to activate a specific fault is uniformly distributed over the input data space. Then, the probability of detecting a fault (i.e., activating a fault) with the test will be binomially distributed $\sim Bin(n, p)$ where n is the number of instructions that can activate the fault included in the test, and p is the estimated average probability that the fault is activated when one such instruction is executed once. This means that the probability, P_{tot} for detecting the fault when the instruction is executed (with different input data) i times is: $P_{tot} = 1 - (1 - p)^i$.

To illustrate which coverage that can be reached under these assumptions for different faults (different p) and different test lengths (different i), the function has been plotted in Figure C.1. As can be seen, quite high coverage can be reached with short tests for faults with high activation probability. To detect faults with low activation probability, longer tests are needed. However, it is important to note, *the impact of faults with low activation probability may not be severe. Therefore, they may not be necessary to detect.* (methods for estimating the impact of faults with different activation probability were presented in Chapter 4 and 5).

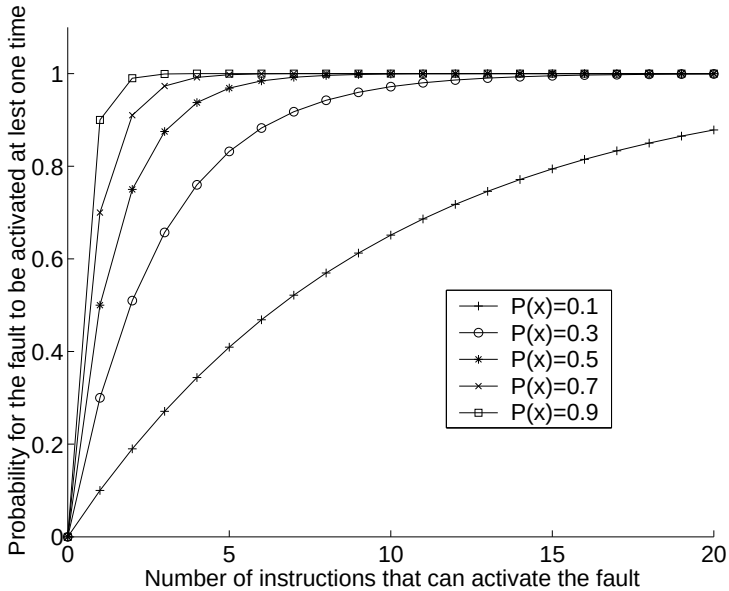


Figure C.1: The relationship between test length and error detection coverage of self-tests.

C.1.4 Test Interval

As was discussed previously, in order to reduce the time overhead for the test, it can be desirable to divide it into several parts, where a part ideally is executed when the processor else would be idling. However, dividing the test can increase the detection latency and the number of data that need to be stored.

For tests that execute application tasks backwards and tests based on diversity, more time is required to spend on comparing the results. However, such tests are generally easier to divide, and thus, are simpler to fit into applications with really hard deadlines.

C.1.5 False and Unnecessary Alarms

As has been mentioned several times already, the test can be focused to those instructions and input data that are used by the safety-critical tasks in order to reduce the overhead. However, this strategy will also imply that less faults with no or limited impact will be detected. This is valuable in order to utilize the resources of the system in the best and safest way.

To reduce the unnecessary alarms even more, and false alarms (for instance due to round-off errors), one can allow small differences between the in advance computed (golden) result and the result computed on-line. However, if the test sequences before comparison is long, it may be hard to find a suitable acceptance level.

Moreover, to differentiate errors caused by transient faults affecting only the test execution, from errors caused by persistent faults possibly affecting the system, the test can be double executed, i.e., executed two times and the results compared. However, this will significantly increase the time overhead.

C.1.6 Summary

In this appendix, we have proposed to use software-implemented self-tests for detection of persistent faults in the functional units. These tests should be designed to only contain instructions that are used by the functional units and by the safety-critical tasks of the system. Furthermore, the amount of data that need to be stored for the tests can be reduced by linking the instructions so that the output of one instruction is used as the input to the next, and so on. Another alternative is to use the same data as the application tasks by either computing the application task backwards, or to use tests based on diversity.

We also suggest that the detection coverage of the test can be estimated through the binomial distribution with the activation probability of the specific fault, and the test length (i.e., the number of instructions) as parameters. By considering that faults with high activation probability are more likely to be detected and that faults with low activation probability generally have lower impacts on systems, it should be possible to find a suitable test length.

In order to reduce the time overhead for executing the test, the test can be divided into parts, and if possible, executed when the processor else would be idling.

As the test is focused to the parts that are utilized by the safety-critical tasks, the risk of unnecessary alarms should be low. Specifically, the test can be double executed to distinguish between errors caused by transient and persistent faults.

Looking at the design process of self-test tasks, it should be possible to automate. First, which instructions the test should consist of can be determined by compiling the safety-critical tasks and identifying all used instructions utilizing the functional units. Second, when the desired coverage is specified, the number of instructions needed can be computed through the binomial distribution. Third, the test can be divided into parts with suitable sizes and scheduled in the system as any other task.

Appendix D. Control Theory

D.1 The Closed-Loop System

Let

$$\begin{aligned} x(k+1) &= \Phi_p x(k) + \Gamma_p u(k) \\ y(k) &= C_p x(k) \end{aligned} \tag{D.1}$$

with $x \in \mathbb{R}^{n_x}$, $\Phi_p \in \mathbb{R}^{n_x \times n_x}$, $\Gamma_p \in \mathbb{R}^{n_x \times 1}$, and $C_p \in \mathbb{R}^{1 \times n_x}$ be an arbitrary state realization of the process subject to control. Combined with the controller (4.2) this yields the closed-loop system

$$\begin{aligned} \xi(k+1) &= \Phi_{cl}\xi(k) + \Gamma_{cl}^{u_c} u_c(k) + \\ &\quad \sum_z \Gamma_{cl}^{\eta_z} \eta_z(k) + \Gamma_{cl}^{\eta_u} \eta_u(k) && \text{(D.2)} \\ y(k) &= C_{cl}^{y} \xi(k) && \text{(D.3)} \\ u(k) &= C_{cl}^{u} \xi(k) + D^{u_c} u_c(k) + \eta_u(k) && \text{(D.4)} \end{aligned}$$

with $\xi = (x; z)$ and

$$\begin{aligned} \Phi_{cl} &= \begin{pmatrix} \Phi_p + \Gamma_p D^y C_p & \Gamma_p C \\ \Gamma^y C_p & \Phi \end{pmatrix} \qquad \Gamma_{cl}^{u_c} = \begin{pmatrix} \Gamma_p D^{u_c} \\ \Gamma^{u_c} \end{pmatrix} \\ \Gamma_{cl}^{\eta_z} &= \begin{pmatrix} 0 \\ e_z \end{pmatrix} \qquad \Gamma_{cl}^{\eta_u} = \begin{pmatrix} \Gamma_p \\ 0 \end{pmatrix} \\ C_{cl}^{y} &= \begin{pmatrix} C_p & 0 \end{pmatrix} \qquad C_{cl}^{u} = \begin{pmatrix} D^y C_p & C \end{pmatrix} \end{aligned} \tag{D.5}$$

where e_z is the z:th column of the unit matrix $I_{n_z \times n_z}$. The impulse-responses from the computation-errors $\eta_z(k)$ and $\eta_u(k)$ to the process output $y(k)$ are

$$\begin{aligned} h_z(k) &= \begin{cases} 0, & k \leq 0 \\ C_{cl}^y \Phi_{cl}^{k-1} \Gamma_{cl}^{\eta_z}, & k > 0 \end{cases} \\ h_u(k) &= \begin{cases} 0, & k \leq 0 \\ C_{cl}^y \Phi_{cl}^{k-1} \Gamma_{cl}^{\eta_u}, & k > 0 \end{cases} \end{aligned} \tag{D.6}$$

The process output-responses to the disturbances η_z and η_u are now described by

$$\begin{aligned} y(k) &= \sum_{i=0}^{\infty} h_z(i) \eta_z(k-i) \\ y(k) &= \sum_{i=0}^{\infty} h_u(i) \eta_u(k-i) \end{aligned} \tag{D.7}$$

respectively. The sensitivity functions

$$\begin{aligned} H_z(z) &= C_{cl}^y (zI - \Phi_{cl})^{-1} \Gamma_{cl}^{\eta_z} \\ H_u(z) &= C_{cl}^y (zI - \Phi_{cl})^{-1} \Gamma_{cl}^{\eta_u} \end{aligned} \tag{D.8}$$

are the $\mathcal{Z}$-transforms of the impulse responses, and describe the frequency responses from η_z and η_u to the output y. Evaluation of the sensitivity functions at $z = e^{i\omega t_h}$, where t_h is the sampling time, gives the stationary response of the closed-loop system for pure sinusoidal inputs $\eta_z(k) = \sin(k\omega t_h)$ and $\eta_u(k) = \sin(k\omega t_h)$ as $y(k) = |H_z(e^{i\omega t_h})| \sin(k\omega t_h + \arg(H_z(e^{i\omega t_h})))$ and $y(k) = |H_u(e^{i\omega t_h})| \sin(k\omega t_h + \arg(H_u(e^{i\omega t_h})))$ respectively. Please refer to [Åström and Wittenmark, 1997] for more details.

D.2 The Brake-Slip Controller

The wheel-slip is the normalized relative velocity between the rotating wheel and ground: $\lambda = (r\omega - v)/v$, where r is the wheel radius, ω the wheel angular velocity, and v the speed of the car. The braking force of a wheel is proportional to the wheel slip for $0 \leq \lambda \lesssim 0.1$, as $F_b = F_z C_\lambda \lambda$, where F_z is the normal load on the tire and C_λ is a tire-stiffness parameter. A simple quarter-car model of the slip dynamics is

$$v/\alpha \frac{d\lambda(t)}{dt} + \lambda(t) = \beta/\alpha \tau_b(t) \tag{D.9}$$

with $\alpha = mgr^2 C_\lambda / 4J$, $\beta = r/J$, where m is the car mass, and J is the wheel inertia. The braking torque $\tau_b(t)$ is described by the actuator dynamics

$$T_a \frac{d\tau_b(t)}{dt} + \tau_b(t) = K_a u(t) \tag{D.10}$$

Combining Eq. (D.9) and Eq. (D.10), and discretizing with zero-order-hold (ZOH) sampling with period t_h results in the second order open-loop system Eq. (4.7).

Numerical values used in the example are $v = 30$ m/s, $m = 2000$ kg, $J = 16$ kgm^2, $r = 0.4$ m, $C_\lambda = 5$, $F_z = 5000$ N, $g = 9.81$ kgm/s^2, $T_a = 0.05$ s, and $K_a = 1000$.

A RST-controller [Åström and Wittenmark, 1997] is designed to obtain a closed-loop system with poles in a Butterworth pattern with $\omega_{cl} = 30$ rad/s and opening angle 45°. The controller includes integral action. An observer pole is introduced at $2\omega_{cl}$. A modal form state-realization of the controller is found in Eq. (4.8).

Appendix E.
The VHDL-code for a Saboteur

```
library ieee;
use ieee.all;
type v1 is array (1 to 2) of real;
type v2 is array (1 to 32) of real;

ENTITY saboteur IS
   PORT (
   insmul1: IN BIT_VECTOR(15 DOWNTO 0);
   insmul2: IN BIT_VECTOR(15 DOWNTO 0);
   outmul: IN BIT_VECTOR(31 DOWNTO 0);
   outsab: OUT BIT_VECTOR(31 DOWNTO 0);
   sab_act: IN BIT;
   prop_prob: IN real;
   fault_mod: IN integer;
   seed: IN v1;
   xfault_prob: IN v2;
   bit_prob: IN v2
   );
-- insmul1, insmul2 and outmul are the inputs and
-- outputs to/from the multiplier.
-- outsab is the output signal from the saboteur.
-- sab_act is the activation signal for the saboteur
-- (0 => off, 1 => on).
-- prop_prob is the set fault activation (i.e., error
-- genaration) probability for the simulated fault.
-- fault_mod determines how signals are activated
-- (0 => stuck-at-0, 1 => stuck-at-1, 2 => bit-flip).
-- seed contains the inital seeds for the
```

```
-- random number generator in the saboteur.
-- xfault_prob is the set probability for multiple
-- bit errors.
-- bit_prob is the set probabilties for an error
-- should occur in a certain bit.

END saboteur;


ARCHITECTURE behaviour OF saboteur IS
BEGIN

PROCESS(outmul,sab_act)
FUNCTION bit2int(bv: in bit_vector) return integer is
-- This function converts bit_vectors to integers.

VARIABLE result:integer:=0;
BEGIN
   FOR index in bv'range LOOP
      IF bit'pos(bv(index))=1 THEN
         result:=result*2+1;
      ELSIF bit'pos(bv(index))=0 THEN
         result:=result*2;
      ELSE
         report "not a bit_vector";
      END IF;
   end LOOP;
   RETURN result;
END FUNCTION bit2int;

CONSTANT inmax:real:=real(2**16);
CONSTANT ranmax:real:=real(2147483397);
CONSTANT ranmin:integer:=1;
-- inmax is the maximum value of each of the
-- inputs of the multiplier.
-- ranmax and ranmin is the max respectively min
-- seeds that can be used for the random generator.
```

```
VARIABLE slump,slump2,slump3,fr1,fr2,test1,test2:real;
VARIABLE s1,s2:integer;
VARIABLE outtemp:bit_vector(31 downto 0);
VARIABLE counter:integer;
VARIABLE c2:integer;
VARIABLE i:integer;
VARIABLE hit:bit_vector(1 to 32);

BEGIN
 IF sab_act ='1' THEN
-- If the saboteur is not activated (sab_act='0')
-- the output from the multiplier is just fed through.

  fr1:=(ranmax/inmax)*seed(1)*real((bit2int(insmul1)+1));
  fr2:=(ranmax/inmax)*seed(2)*real((bit2int(insmul2)+1));
  s1:=integer(math_real.floor(fr1))+ranmin;
  s2:=integer(math_real.floor(fr2))+ranmin;
  math_real.uniform(s1,s2,slump);
-- The saboteur is activated so a random number between
-- 0 and 1 is generated using the multpliers inputs
-- and the experiment specific numbers (seed) as seeds.

  outtemp:=outmul;
  IF slump > prop_prob THEN
-- Determines whether the error should propagate or not.
-- If the random number, slump, is bigger than the set
-- control signal prop_proban an error should be
-- generated.

     counter:=1;
     i:=1;
     hit:="00000000000000000000000000000000";
     test1:=xfault_prob(counter);
     math_real.uniform(s1,s2,slump2);
     WHILE slump2 > test1 LOOP
        counter:=counter+1;
        test1:=test1+xfault_prob(counter);
     END LOOP;
```

```
-- Determines in how many bits of the output the error
-- should manifest itself in according to the
-- distribution given by the set control signal:
-- xfault_prob

     WHILE i <= counter LOOP
        math_real.uniform(s1,s2,slump3);
        c2:=1;
        test2:=bit_prob(c2);
        WHILE slump3 > test2 LOOP
           c2:=c2+1;
           test2:=test2+bit_prob(c2);
        END LOOP;
        IF hit(c2)='0' THEN
           hit(c2):='1';
           i:=i+1;
        END IF;
     END LOOP;
-- Determines in which bits of the output the error
-- should manifest itself in according to the
-- probability distribution given by the set
-- control signal: bit_prob

     CASE fault_mod is
     WHEN 0 =>
        outtemp:= outtemp and (not hit);
     WHEN 1 =>
        outtemp:= outtemp or hit;
     WHEN 2 =>
        outtemp:= outtemp xor hit;
     WHEN others =>
          report "not a fault mode";
     END CASE;
  END IF;
-- Determines how the error should manifest.
-- Fault mode 0 => stuck-at-0
-- Fault mode 1 => stuck-at-1
-- Fault mode 2 => bit-flip
```

```
  outsab <= outtemp;
  else
  outsab <= outmul;
 END IF;
END PROCESS;
END behavior;
```