University of West Bohemia

Faculty of Applied Sciences

DOCTORAL THESIS

2013 **Peter Raab, M.Eng.**

University of West Bohemia in Pilsen
Faculty of Applied Sciences

MODEL-BASED RELIABILITY EVALUATION OF DATA PROCESSING IN HW-FAULT-TOLERANT PROCESSOR SYSTEMS

Peter Raab

Doctoral Thesis
in partial fulfillment of the requirements for the degree of
Doctor of Philosophy
in specialization
Computer Science and Engineering

Supervisor: Doc. Ing. Stanislav Racek, CSc.

Department: Department of Computer Science and Engineering

Pilsen 2013

Západočeská univerzita v Plzni
Fakulta aplikovaných věd

MODELOVĚ-ORIENTOVANÉ VYHODNOCENÍ ZPRACOVÁNÍ DAT V PROCESOROVÉM SYSTÉMU ODOLNÉM PROTI PORUCHÁM

Peter Raab

Disertační práce
k získání akademického titulu doktor
v oboru Informatika a výpočetní technika

Školitel: Doc. Ing. Stanislav Racek, CSc.

Katedra: Katedra informatiky a výpočetní techniky

Plzeň 2013

Abstract

The increasing complexity and functionality is a continuous process of recent software-intensive systems especially in the embedded domain. In addition, changing manufacturing processes of semiconductor devices cause higher vulnerability to external upsets. Therefore, all these conditions influence the dependability of safety-critical applications. The cost pressure often causes the manufacturers to use less expensive commercial components instead of high reliable specialized hardware. This lack of reliability in standard hardware can be compensated by several techniques that have been developed since there are computer systems. Here, the trend of a pure software solution is particularly remarkable. Error detecting codes which are known from transmission systems are one important aspect of software-based hardware fault tolerance. The *coded processing* approach also makes use of such codes for error detection in arithmetic operations. Related works on coded processing often verifies the effectivity of these codes by experimental methods. However, mathematical error models, which are required for analytical evaluations, are not available. In general, analytical approaches have the advantage of less complexity and shorter evaluation duration compared to experimental or simulation approaches. Thus, this thesis deals with the challenge to create error models to describe the error susceptibility of arithmetic operations in a computer system. Furthermore, these error models can provide the reliability or the residual error probability in case of coded processing as an important metric for the effectivity of a given error detecting code. And finally, the composition of several error models will lead to a reliability model of a given data flow in future.

Keywords: coding theory, channel model, error detecting and correcting codes, Safely Embedded Software (SES), coded data processing, data flow, error compensation, residual error probability, continuous Markov process, discrete Markov chain

Abstrakt

Narůstající složitost a komplexnost funkce je stále probíhající proces ve vývoji soudobých SW systémů, zejména pak v oblasti vestavěných počítačových systémů. Navíc neustálé změny ve výrobním procesu polovodičových zařízení způsobují jejich narůstající citlivost na vnější podněty. Uvedené skutečnosti ovlivňují návrh bezpečnostně kritických aplikací. Naproti tomu cenový tlak nutí výrobce užívat komerční součástky namísto vysoce spolehlivého specializovaného HW. Nedostatek spolehlivosti standardního HW může být kompenzován různými způsoby, které byly vyvinuty v celém průběhu vývoje počítačových systémů. V tomto vývoji lze vysledovat tendenci čistě softwarového řešení. Detekční kódy pro odhalení a opravu chyb, které jsou známé ze systémů pro přenos informace, jsou také jedním z důležitých prvků softwarově realizované odolnosti proti vlivu hardwarových chyb. Přístup *kódovaného zpracování* pak dává možnost využít tyto kódy pro detekci chyb aritmetických operací. Publikované výzkumy kódovaného zpracování často ověřují efektivitu příslušných kódů experimentálními metodami. Naproti tomu matematické modely chyb, které jsou vyžadovány analytickými metodami, nejsou dosud dostupné. Obecně, analytické postupy mají výhodu menší složitosti a kratšího času vyhodnocení ve srovnání s experimentálními nebo simulačními přístupy. Předložená disertační práce reaguje na výzvu vytvořit chybové modely schopné popsat výskyt chyb v aritmetických operacích v počítačových systémech. Tyto chybové modely mohou umožnit výpočet spolehlivosti aritmetických výpočtů, nebo výpočet pravděpodobnosti zbytkové chyby kódovaného zpracování jako důležitého ukazatele efektivnosti využití příslušného detekčního kódu. A nakonec, kompozice několika chybových modelů může v budoucnosti vést k vytvoření spolehlivostního modelu toku dat v uvažovaném systému.

Klíčová slova: teorie kódování, model komunikačního kanálu, kódy pro detekci a korekci chyb, bezpečně vestavěný SW (SES), zpracování kódovaných dat, reakce na chybu, zbytková pravděpodobnost chyby, Markovské procesy ve spojitém a diskrétním čase.

Zusammenfassung

Die steigende Komplexität und Funktionalität ist ein kontinuierlicher Process in modernen Softwaresystemen insbesondere im embedded Bereich. Zusätzlich führen Veränderungen im Herstellungsprozess von Halbleiterbauelementen zu erhöhter Anfälligkeit gegenüber externen Störungen. All diese Bedingungen beinflussen somit die Verlässlichkeit sicherheitskritischer Anwendungen. Wohingegen die Hersteller aus Kostengründen anstelle von hochzuverlässiger Spezialhardware oft günstigere Standardkomponenten verwenden. Dieser Mangel an Zuverlässigkeit von Standardhardware kann durch verschiedene Techniken kompensiert werden, an denen schon gearbeitet wurde seit es Computer gibt. Dabei ist der Trend reiner Softwarelösungen besonders bemerkenswert. Ein wichtiger Aspekt von softwarebasierter Hardwarefehlertoleranz sind die fehlererkennenden Codes, die man aus Übertragungssystemen kennt. Der Ansatz der codierten Verarbeitung verwendet ebenfalls solche Codes zur Fehlererkennung in aritmetischen Operationen. Arbeiten auf diesem Gebiet überprüfen die Effektivität dieser Codes häufig durch experimentelle Methoden. Jedoch sind mathematische Modelle, die für eine analytische Evaluation notwendig sind, kaum bekannt. Ein analytischer Ansatz hat aber den Vorteil gegenüber experimentellen oder simulationsbasierten Verfahren, dass sie weniger komplex und in der Regel kürzere Berechungszeiten haben. Deshalb behandelt diese Arbeit die herausfordernde Aufgabe, Fehlermodelle zu erstellen mit denen es möglich sein wird, die Fehleranfälligkeit arithmetischer Operationen in einem Computersystem zu beschreiben. Außerdem können diese Fehlermodelle Zuverlässigkeiten oder in Falle von codierter Verarbeitung eine Restfehlerwahrscheinlichkeit liefern, die eine wichtige Kenngröße ist, die Effektivität der eingesetzten Fehlercodes zu bewerten. Und schließlich wird die Anordung mehrerer Fehlermodelle dazu führen, dass in Zukunft die Zuverlässigkeit ganzer Datenflüsse modelliert werden kann.

Keywords: Codierungstheorie, Kanalmodell, fehlererkennende- und korrigierende Codes, Safely Embedded Software (SES), Codierte Datenverarbeitung, Datenfluss, Fehlerkompensation, Restfehlerwahrscheinlichkeit, diskrete Markov-Kette, kontinuierliche Markov-Prozesse

Declaration of Authenticity

I hereby declare that this doctoral thesis is my own original and sole work.
Only sourced listed in the bibliography were used.

Čestné prohlášení

Prohlašuji tímto, že tato disertační práce je původní a vypracoval jsem jí samostatně.
Použil jsem jen citované zdroje uvedené v přehledu literatury.

In Pilsen

V Plzni dne .....

.....

The present dissertation has been submitted as a cumulative work. It is based on the publications which are listed in Appendix A.

To Sandra and Fabian

Acknowledgements

After the completion of this thesis, which I have worked on for three years, it is now time to thank all people supporting me during that time. First and foremost, I would like to thank my supervisors Doc. Ing. Stanislav Racek, CSc. from the University of West Bohemia and Prof. Dr. Jürgen Mottok from the University of Applied Sciences Regensburg for their guidance and encouragement. I am very grateful for the opportunity to participate on the PhD study program and on numerous conferences for presenting the results of my research. I also give my best thanks to Prof. Dr.-Ing. Frank Schiller for his time to comment my publications with many following fruitful discussions. Further thanks go to my colleagues at the Laboratory for Safe and Secure Systems, especially Stefan Krämer for his help, reviews and discussions. And finally, I would like to express my deepest gratitude to my family and especially to my beloved wife Sandra. Her belief in me and her continuous support have often helped me to get through the ups and downs during the last three years.

This work was written in the context of the project S^3OP (Safe Oriented Programming of Software-Intensive Embedded Systems) founded by the Bavarian State Ministry for Science, Research and Arts.

Contents

List of Figures

List of Abbreviations

ALU Arithmetic Logic Unit

BSC Binary Symmetric Channel

CoRed Combined Redundancy

COTS Commercial Off-The-Shelf

CPU Central Processing Unit

CRC Cyclic Redundancy Code

DFT Discrete Fourier Transformation

DMR Dual Modular Redundancy

DSP Digital Signal Processor

ECC Error Correction Code

EDDI Error Detection by Duplicated Instructions

EMC Electromagnetic Compatibility

FIT Failure in Time

FPGA Field Programmable Gate Array

LaS³ Laboratory for Safe and Secure Systems

MAC Multiplier-Accumulator

RAM Random-Access Memory

RESO Recomputing with Shifted Operands

RISC Reduced Instruction Set Computing

R³TOS Regensburg´s Reliable Realtime Operating System

RTOS Realtime Operating System

SEP Software Encoded Processor

SER Soft Error Rate

SES Safely Embedded Software

SEU Single Event Upset

SIHFT Software-Implemented Hardware Fault Tolerance

SIL Safety Integrity Level

SWIFT Software-Implemented Fault Tolerance

SWIFI Software-Implemented Fault Injection

S^{3}OP Safe Oriented Programming of Software-Intensive Embedded Systems

TMR Triple Modular Redundancy

VCP Vital Coded Processor

Chapter 1

Introduction

Software intensive embedded systems are widely used in our engineered society. Almost each technical product in our daily life comprises a microprocessor, where software is running on. Everyone has a personal computer or a mobile phone at home which are only the obviously ones. There are embedded microcontrollers also in television, kitchen equipments, toys, and particularly in automotive, industry and health care products. The latter ones influence our safety and healthy with high dimension and we expect high dependability of these systems. Thanks to these electronic helpers in cars and medicine products, the number of people killed by traffic accidents have been decreased the last years [Bun12].

1.1 Motivation

The dependability of systems becomes more important in recent years. Either for economic reasons or because of safety aspects, dependable systems are required to reduce costs or safe life. Nowadays, there is the trend of increasing complexity and functionality of electronic control units in several sectors of industry and in particular in the automotive domain (Figure 1.1). One effect is the higher risk for mistakes in the specification or in the implementation during the development process. So, it follows the rising chance for malfunctions during the operation phase of a system. These mistakes manifest themselves in abnormal behavior other than intended and they are systematic in general. The strict keep of rules during the development process, e.g. review, the number of those mistakes can be minimized [ELL+92]. But, it is not possible to avoid all mistakes in limited development time. Studies showed that some errors remain in the software that potentially causes failures in the future [FP98, MD00]. For this reason, current standards and norms define functional safety to minimize the risk for a person´s life or the damage to health

and environment and they provide development techniques which depend on the required safety integration level (SIL) [IEC10a, ISO11].

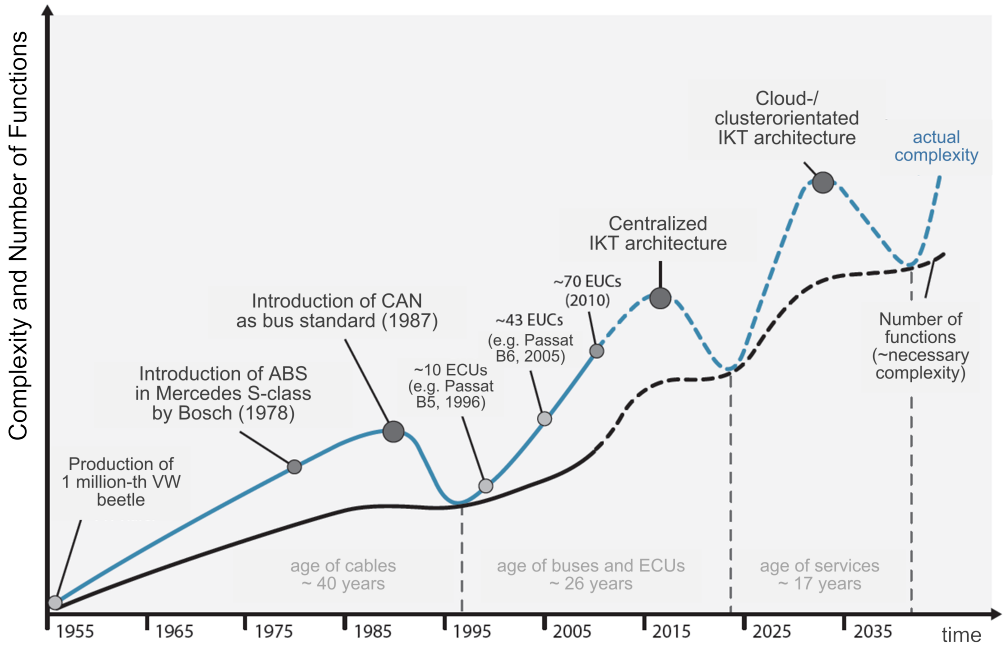

Figure 1.1: Evolution of complexity (= number of functions) in an embedded system [For11].

In contrast to the mentioned development-based faults, the main focus of this work is the reliability of the underlying hardware, where the software is executed on. The more complex a system is, the more hardware is required and the more relevant this aspect becomes. Modern manufacturing processes of present microcontrollers follow the trend of a decreasing feature size in the silicon. This leads to less reliability and arbitrary system failures are more likely during the normal operation phase [DM03]. Component defects or external disturbance are the reasons for this type of malfunction (so-called "soft errors" Figure 1.2). However in spite of this progress, industry demands a decrease in costs for electronics, while at the same time to remain competitive. The result is the use of inexpensive commodity hardware instead of highly reliable specialized hardware. The vulnerability to these arbitrary faults must be partly compensated by additional fault-tolerant techniques wherein a pure software solution is of particular interest. The *coded processing* is used to detect faults in processor systems by means of diversity and information redundancy. It is also the initial approach for this thesis which presents new error models for the reliability evaluation of data processing on unreliable hardware compared to commonly used experimental techniques. See the state-of-the-art investigation in Chapter 2 and the PhD report for further background information about fault

mechanisms, fundamentals of dependability and fault classification [13].

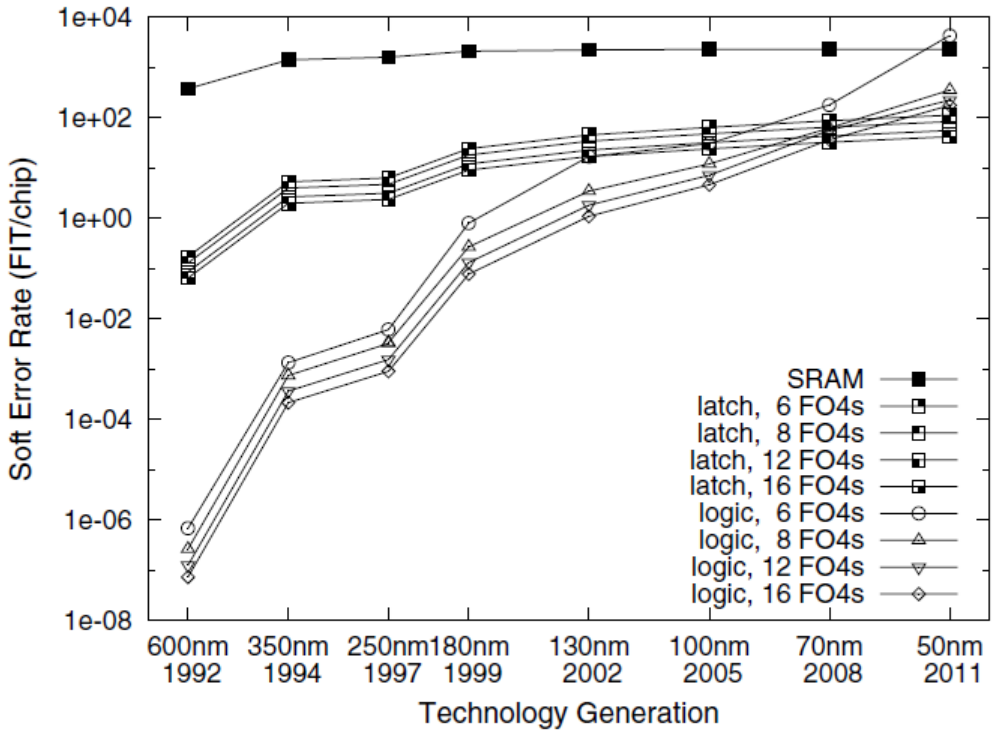

Figure 1.2: Soft error rates of individual circuits and their trend [SKK+02].

1.2 Structure of the Thesis

This PhD thesis is written in a cumulated form. It is organized by the following structure. First, the motivation for fault-tolerant data processing is introduced in Chapter 1. Further in Chapter 2, a summary of the state-of-the-art investigation is given which provides the necessary background information for this work. The goals are defined in Chapter 3 and later discussed in Chapter 5 with respect to their achievement. Finally, Chapter 4 resumes all relevant publications to discuss the contribution of this thesis. It introduces a basic concept for error models in data processing units and identifies further problems in this topic to be discussed. The thesis closes with a conclusion and a perspective for further works in Chapter 5.

Chapter 2

State of the Art

The introduction in the previous chapter highlights the motivation for software-based fault detection in dependable processor systems. It narrows the topic further down to randomly occurred hardware faults during data processing in software systems. Now, this chapter summarizes the state-of-the-art techniques and methods which are commonly used for detecting hardware faults in dependable computer systems. The text is derived from the PhD report in [13].

2.1 Software-Implemented Hardware Fault Tolerance

Because of the fact that fault-tolerant computer systems became more important in recent years, there is the trend to implement concurrent error detection capabilities in hardware [SAC+99, AYI+03, VPC02, RSS+08]. But on the other hand, the costs of current developments become a higher relevance, especially in the embedded domain where a big quantity of components is produced. Therefore, the use of redundant hardware or special customer designed hardware is not attractive for the manufacturers. To fulfill the requirements of restricted budget anyway, the manufacturers usually apply commercial standard hardware, so-called *commercial-off-the-shelf* (= COTS). As already mentioned in the introduction, these standard components are less reliable and it is assumed that by reasons of progressive changes in the manufacturing processes of semiconductor devices the reliability will further decrease. As a consequence of these strict restrictions for costs and the required reliability for safety-critical applications, a lot of pure software implemented techniques are developed. The literature mentions *redundancy* as an important aspect for dependability (e.g. [ALR03]). Only redundancy allows the detection of faults. Pradhan [Pra96] and others [Kir05, IEC10b] talk about different forms of redundancy.

These are

- hardware redundancy,
- time redundancy,
- data redundancy,
- information redundancy and
- software redundancy.

Redundancy means additional functionality being implemented in parallel to the specified function of a system. Therefore, any kind of redundancy also increases the demand for additional resources like computation time, hardware or memory. However, each kind of redundancy has an influence to certain system constraints. For example, time redundancy increases the execution time and maybe violates timing restrictions. Therefore, the use of fault-tolerant techniques must always be verified to be applicable.
In the further course of this chapter, several state-of-the-art techniques for pure *software-implemented hardware fault tolerance* (= SIHFT) is summarized. The interested reader is also referred to [GRRV06] who gives a complex and more detailed overview of modern SIHFT techniques.

2.1.1 Duplication

We know about two different effects of faults during the execution of a software program. One of them is data corruption (*data error*) which results in an erroneous outcome of a program. The corruption of data is caused either by unexpected modification in the memory (bit-flip), or by transient faults that occur in integrated circuits like arithmetic logical unit. Because of the fact that a *single event upset* (= SEU) can only occur at a certain place or time, the memory cell in another memory location is not affected, and it is unlikely that the same fault occurs in the same cell or integrated circuit twice. This leads to the idea of duplication, which can be realized in different forms [NV03, BCPT00]:

- Data duplication (data, space redundancy),
- Instruction duplication (time, space redundancy),
- Hardware duplication (space redundancy), triple-modular-redundancy (= TMR)
- Software duplication (software redundancy), N-Version Programming

For example, Nicolescu et.al. [NV03] presented an approach which replicates all variables in a program. Both copies of this variable contain the same information and the same operation is executed twice, one with the original variable and once with its replica. The dependencies of the data processing remain constant for both variables. At the end, the variables contain the final results and they are checked for consistency. Both variables must be equal. Otherwise, there was a fault either in one of the variables stored in the memory or during the calculation process in the arithmetic unit of the microcontroller. The experimental result of this approach showed that there is a factor of at least two for additional data memory and runtime compared to the original one. The code size grows even by a factor of 3.6. With a detection rate of 80 percent, the efficiency of this approach is quite good. The error rate was reduced by a factor higher than 20.

Benso et.al. [BCPT00] go a step further. They defined a *reliability-weight* for each variable in the program which is a function of the variable´s *life time* and the *dependencies* to other variables. The life period of a variable is the time between the first write (initialization) of a variable till it is read the last time before it is written again. The life time is then the sum of all periods and the longer this time is, the higher the probability of being corrupted. Variables with a high reliability-weight are usually more critical for the reliability of the application. The dependency to other variables is another criterion for reliable variables. For example in the calculation of $c = a+b$, the reliability of variable c depends on the correctness of the variables a and b. In a complex program, the dependencies of a certain variable can have a lot of descendants which are able to propagate possible faults. Thus, it is highly critical for the reliability of the final variable. The more variables are necessary for a computation the higher the risk that one of them is corrupted by an SEU. There is another important effect related to these dependencies between variables. If there are multiple faults in different but dependent variables, it is possible that the faults are compensated and the final result is not wrong anymore. Another idea of Benso was to analyze the code automatically by a compiler[1] which re-orders the code to minimize the reliability weight. In addition, the compiler modifies the code and inserts shadow variables. Consistency checks compare the value of both during the execution of the program. But Benso describes the same disadvantages of memory overhead and performance degradation.

NOTE:
The idea of using compiler-based insertions for duplicated data and instructions is widely used in current researches. There are further publications for compiler-based techniques available in [BCPT00, RRVT01, WKWY07, FSS09, SSS+11].

[1] RECCO = REliable Code COmpiler (C/C++ source-to-source compiler)

The technique of *Error Detection by Duplicated Instructions* (EDDI), introduced by Oh [Oh00], uses time redundancy for checking arithmetic, memory and also control flow errors. He duplicated the instructions at assembler level and splits the register bank and memory space into an original and a shadowed section. The original and the replicated instructions use their own registers and memory. Every time a new value is stored or a branch instruction is executed, the results of the duplicated instructions are compared with the original. In case of inequality, a fault is detected. Branching errors are directly detected by the corrupted data which the branching depends on. Other branching and control flow errors are indirectly detected, when the next comparison takes place.

NOTE:
Some of the introduced publications do not only use duplication to improve fault detection. They also describe some methods of control flow checking. One effect of faults is the change in the program flow. The pure duplication of data and execution is not enough to detect all control flow errors [ANKA99, GRSRV03, BDCDNP03].

In 2005, Reis et.al. [RCV+05] presented a similar also compiler-based ***s**oft**w**are **i**mplemented **f**ault **t**olerance approach* (= SWIFT). The SWIFT approach improves the performance of prior works[2] and implements control flow techniques in addition. Some memory devices provide *error correcting codes* (= ECC) which is often implemented in hardware. Thus, the stored data are already protected against faults and need not be duplicated any more. The memory consumption is less than in previous works. Further, they consider a special handling of function calls. In contrast to variables stored in the RAM and protected by ECC, parameters of functions are usually stored in working registers of the processor. The function is called only once, but the parameters are duplicated at the beginning of the function. At the end, the return value is verified and returned only if there are no errors detected. But, a blind spot of protection remains. Faults that occur before the parameters are duplicated by the callee are replicated and they are not detected. In general, there is a gap of protection between every validation and the next usage of validated values.

There are further publications available, which all use some kind of duplication for increasing fault tolerance. In this report, there is only a selection. The interested reader is referred to [AAN00, NVR01, NPVS03] for further study. All recent software approaches have in common that they all use some kind of duplication to realize data, space[3] or time

[2]Basically, it is a combination of EDDI, ECC and some control flow monitoring approaches.
[3]When a variable is duplicated, the copy is stored in another memory location. This can be regarded as a different peace of hardware and we can talk about space redundancy.

redundancy. In general, we can identify following advantages (+) and disadvantages (-) of duplication:

+ No additional or specialized hardware is necessary, COTS can be used.
+ Pure software solutions are independent from HW architecture.
- Permanent faults are not detected. This means that the same bit is corrupted in both variants by a common-cause.
- Faults in the comparison are not detectable.
- There is also a gap of supervising before a variable is duplicated.

2.1.2 Diversity

One disadvantage of pure data duplication is that faults are detectable, only if the outcomes of the redundant data processing are different in both variants. This is only the case with single transient faults. Permanent faults in a circuit of the *central processing unit* (= CPU) would produce the erroneous result in both data paths (e.g. stuck-at faults). The workaround for this would be again the usage of redundant hardware. Alternatively, we can use diverse data representation in the duplicated data channel. So, the probability increases that different outputs are produced with the same input. In [AK88] for example, Ammann explains this concept as data diversity. Figure 2.1 shows a simple realization of data diversity. The inputs x and y are processed twice by the algorithm A. Once in its original representation and the other in the diverse representation. A possible diverse representation of a number is the negation (2´s complement). The same algorithm, e.g. $A(x, y) = x + y$, is applied once for the original input x and y and a second time for the diverse representation $x' = -x$ and $y' = -y$. At the end, both results are compared by inverting the result of the diverse channel. Basically, this approach describes the concept of data duplication and time redundancy. But if there is a permanent fault during the algorithm A, both results will probably differ [Eng96, OMM02].

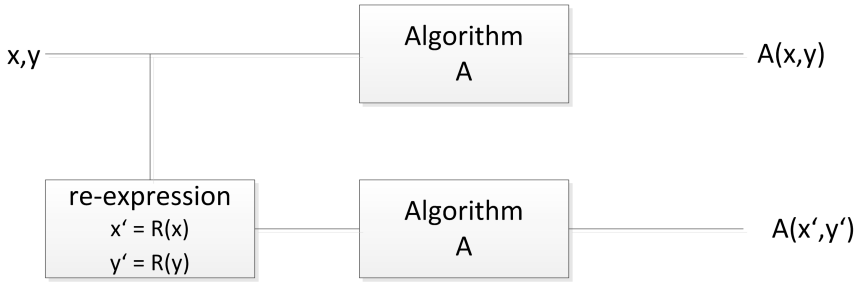

Figure 2.1: Data Diversity - The input variables x and y are processed twice (time redundancy) by the algorithm A. One channel uses the original values and the other channel re-expresses them before processing (data redundancy).

Figure 2.1 shows diverse data processing using the same algorithm A. This implies that the algorithm (e.g. addition) employs the same hardware (e.g. adder unit of CPU). A permanent fault in this hardware is propagated to both outputs and there is still a small probability that both are equal. Instead of using the same algorithm, the data re-expression can be implemented in such a way that different algorithms use different hardware (here: different operations). For example, we want to replace the addition of two inputs by another operation. Instead of the negation, the inputs are re-expressed by the logarithm $x' = log(x)$ and $y' = log(y)$ which leads to a different algorithm that must be applied:

$$x' + y' = log(x) + log(y) = log(x \cdot y) \tag{2.1}$$

The two algorithms employ different hardware of the CPU. This is a kind of space redundancy and it is less probable that both components fail and if they do, the outputs will probably differ and the fault is detected.

In [PF82], they published an approach using the described concept of data diversity for error detection in both the arithmetic and logic operations. The *Recomputing with Shifted Operands (= RESO)* approach is based on a time redundant execution of data processing with a diverse data path which is re-expressed by a shift operation. Remember, a left shift equals to a multiplication by 2. Later, Mitra tried to quantify the diversity of two data paths. In [MSM99], he defined a diversity metric to compare diverse implementations of the same function. For a given fault model, the *design diversity metric D* describes the probability that the two diverse functions produce different error patterns in response to any combination of the input and with respect to different faults in both implementations. He considered following cases of the output: (1) Both channels produce correct outputs. This case can be neglected. (2) One channel has a correct output whereas the other produces an incorrect result. Both outputs differ and the fault is detected. (3) Both data channels produce the same faulty output. The fault is not detectable and they say that the *integrity* of the system is lost. Faults that produce the same faulty output are often called common-cause faults. Further, Mitra made an interesting conclusion. He said that the reliability of duplex (= duplicated) systems is not higher for different implementations of a function compared to redundant systems with identical implementations.

The *Error Detection by Diverse Data and Duplicated Instructions* (ED^{4}I) approach presented by Oh et.al. [OMM02] is based on [Oh00] but extended by diversity. A program is executed twice (instruction duplication) and the data of the copied program is diversely represented. The diverse data are realized by a multiplication of the original data with the so-called *diversity factor k*. Furthermore, the previously introduced di-

versity metric is used to evaluate several ks with respect to the data integrity and fault detection probability of different hardware functions (e.g. adder or bus line signals). The diversity factor k determines how diverse the copied program is compared to the original data and it exists an optimum value of k for different hardware functions (e.g. $k = -2$ for an adder).

NOTE:
As we will see later, the diverse data representation which is reached by a multiplication is also important for coded processing. The multiplication with a constant value is the basic principle of the so-called AN-codes (see Chapter 2.1.3).

2.1.3 Error Detecting and Correcting Codes

A further disadvantage of data duplication as described in Chapter 2.1.1 is the demand of additional memory at least by a factor of two for the replicated data. The field of error detection and correction codes also uses redundant bits for the representation of each number. But in contrast to pure data duplication, the total size of the resulting code word is usually less than the double. These codes implement information redundancy that represents the original data in a different way. Thus, error detecting codes are also a kind of diversity and they are widely used in transmission and storage systems.

A code unambiguously assigns every element in the origin set of data $\mathbf{X}$ with cardinality $k = |\mathbf{X}|$ to an element in the coded set $\mathbf{C}$ with cardinality $n = |\mathbf{C}|$. The basic principle of error codes is the existence of more elements in the code space than in the original space. With $k < n$, there are unused code words in $\mathbf{C}$ (Figure 2.2). Only those code words which are assigned to their origin information word by the encoding function f_c are valid. All other words are invalid and indicate an error.

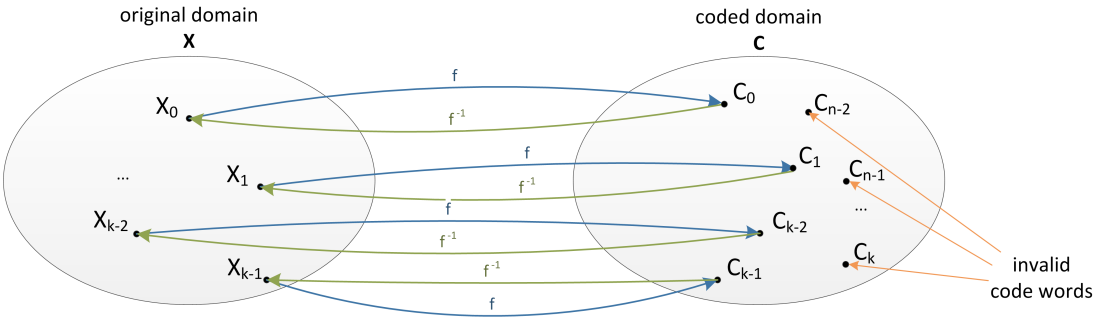

Figure 2.2: Basic principle of error detecting codes.

A formal definition of a code is given below.

Definition 1. *(Code)*

A code is an injective function (= encoding) which assigns an information word $\in \boldsymbol{X}$ *to a code word* $\in \boldsymbol{C}$ *by*

$$f_c: \quad X \to C$$

and $|C| > |X|$*. The set of all code words* $\boldsymbol{C}$ *is called the code space and* $\boldsymbol{X}$ *is the information space. The inverse function (= decoding) of* f_c *reverse the mapping from C to X.*

$$f_c^{-1} : C \to X$$

The advantage of error codes is the capability of fault detection without duplication. Because of the diverse data representation by the encoding function, it is very probable that the code word becomes invalid with the presence of the fault. However, adding redundancy in form of extra invalid code words leads to more required memory. This additional redundancy can be expressed by the so-called *code rate*. This metric only describes the blow-up of data, but it does not tell anything about the error detection capability (see residual error probability in Chapter 2.3.1).

Definition 2. *(Code Rate) [Bos98]*

Every block code with bit length c is generated by x bits of information. The ratio

$$R = \frac{x}{c} = \frac{log_2|X|}{log_2|C|}$$

is the code rate which is a metric for the redundancy $(c - x)$ *of the code.*

Algebraic Structures [Beu94, Rao74]

Algebraic structures are the background for most error detecting and correcting codes Knowing the underlying algebra of codes helps us to understand the different characteristics of codes and the selection of the optimal code for a given use case. An algebraic structure consists of a set of objects (e.g. numbers) and at least one operation regarding to this set. For example the set of integer numbers $\mathbb{Z}$ is a *group* under the operation of an ordinary addition[4] $(\mathbb{Z}, +)$, but not under multiplication. There are the necessary axioms of *closure*, *associative law*, *commutative law*, *identity* and *inverse* to form a group with respect to one operation. The extension of a *group* by a second operation leads to a further algebraic structure, for which additional rules must be fulfilled with respect to the

[4] Operations of algebraic structures need not be an ordinary addition or multiplication. There are also special operations in other structures.

second operation. For example, the set of integer numbers forms a ring under addition and multiplication $(\mathbb{Z}, +, \cdot)$ because the multiplication meets the axioms of *associative* and *distributive law*. If there is also an inverse element for the multiplication, then the algebraic structure is a field. For instance, the set of real numbers forms a field under addition and multiplication $(\mathbb{R}, +, \cdot)$ with the reciprocal number is the inverse element under the multiplication. Standard sets of numbers have infinite elements e.g. $\mathbb{N}$, $\mathbb{Q}$ or $\mathbb{C}$. But in digital data processing systems, the set of numbers is limited by the register width of the processor. With m is the width of a register in bits, $M = 2^m$ is the amount of different numbers and the finite set is then denoted by $\mathbb{Z}_M$. Finite sets are the base for *residue classes* of an integer number. The consequence of limited register width is the possible overflow in case that the result of an operation is bigger than the maximal allowed number. In a formal way, this means that the operations in computer systems are based on residue classes with the following two operations:

$$\textit{Addition:} \qquad a +_M b := (a + b) \mod M \tag{2.2}$$

$$\textit{Multiplication:} \qquad a \cdot_M b := (a \cdot b) \mod M \tag{2.3}$$

If M is a prime number, all axioms for a field are met and there is also an inverse element for the multiplication. But in digital computer systems M is surely no prime as a power of two. With this background, we know that all operations in an arithmetic unit have a ring structure over a finite set of integers $(\mathbb{Z}_M, +_M, \cdot_M)$.

Arithmetic Codes [Bro60, Rao74]
With the knowledge of error codes and the ring of integer numbers, we can define the class of so-called *arithmetic codes* whose encoding function is the multiplication of the original integer number with another constant integer number (see Equation 2.4).

$$C_{AN} := \{A \cdot X \mid A, X \in \mathbb{Z}\} \tag{2.4}$$

Introduced by Brown, arithmetic codes are better known as AN codes. This notation indicates that the code word is the product of the generator (or diversity factor) A and the information word N. The elements of this set of code words is a subset $C_{AN} \subseteq C$ of all multiples of A which is called an *ideal* and denoted as $n\mathbb{Z}_M$. The set of generated code words forms a ring over the two operations addition and multiplication $(n\mathbb{Z}_M, +_M, \cdot_M)$, where the required bit width is increased by the factor $\approx log_2 A$. The sum or the product of two code words is divisible by the generator A and therefore it is a valid code word

(see axiom of closure). But unlike the addition, the product of two code words does not correspond to the coded product of the two original information words X_1, X_2.

$$f_c(X_1) \cdot f_c(X_2) \neq f_c(X_1 \cdot X_2) \tag{2.5}$$

The multiplication does not preserve the mapping between the elements of the original and the coded set which is required for coded data processing. Using arithmetic codes for error detection in arithmetic processors, the multiplication must be replaced by an enhanced operation to follow the encoding function (see coded operations in Chapter 2.2.1).

Linear Codes [Ham50, PB61]
In contrast to the ring of integers, there is the ring of polynomials as another more complex structure. In a digital data processing system, the numbers are represented in a binary form $\mathbf{X} = (x_0, x_1, ... \; x_{m-1})$ with the elements $x_i \in \{0, 1\}$. These elements can be considered as the m scalars of an m-dimensional vector out of the vector space $\mathbf{V}^m$ with the base vectors $e_0, e_1, ... \; e_{m-1}$. Another formal but equivalent representation of vectors are so-called *polynomials*. Instead of the base vectors e_i, the scalars (or coefficients) are multiplied by the power of the undetermined variable z which corresponds to a certain digit in the binary number. The set of all possible combinations of binary coefficients forms the ring of polynomials under the polynomial multiplication and the polynomial addition (Equation 2.6).

$$\mathbb{Z}_2[z] := \{p(z) = \sum_{i=0}^{m-1} x_i \cdot z^i \mid x_i \in \{0, 1\}\}. \tag{2.6}$$

In contrast to arithmetic codes, linear codes are based on the ring of polynomials but the code generating function is also the multiplication

$$C_{CRC} := \{g(z) \odot x(z) \mid g(z), \; x(z) \in \mathbb{Z}_2[z]\}. \tag{2.7}$$

Linear codes a well-known in transmission and storage systems as *cyclic redundancy code* (= CRC). Linear codes are cyclic, if the generator polynomial $g(z)$ is a divider of the polynomial $z^n - 1$. This type of code is a special case of linear codes with the property that the cyclically rotation of the bits in a code word results in another valid code word. Cyclic codes were first introduced by Prange [Pra57] and discussed by Peterson [PB61]. Further enhancements were done by [RS60, BRC60].

2.2 Coded Processing

Coded Processing refers to a method that protects the results of operations in an arithmetic unit by means of error detection codes. The input data are encoded before being processed in an arithmetic unit and the output data are decoded again for verification (Figure 2.3). In this view, coded processing is related to channel coding as a part of the coding theory.

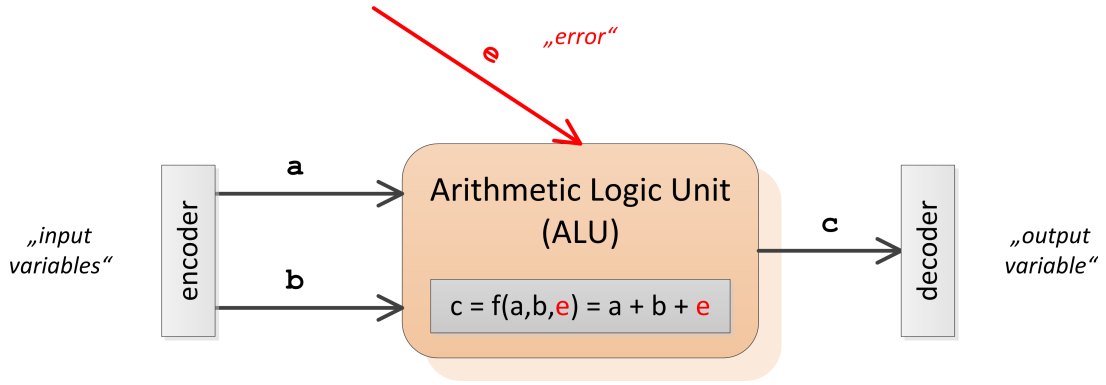

Figure 2.3: The arithmetic unit in a processor represents a channel with respect to arbitrary faults during the execution of an operation.

Channel coding describes error detection and correction codes like the Hamming code, which was originally developed for protecting data corrupted by a noisy channel like a memory or communication link. The data processing in an arithmetic unit of a microcontroller also represents a kind of channel. During the software processing, arbitrary hardware faults (permanent or transient) occur and change the value of a result. But in contrast to transmission systems, which do not process the transmitting data, an arithmetic operation has usually more than one input and the result is a function of these inputs. To decode a result, the used error code must preserve the result of the operation as a valid code word. In the past, a lot of codes were described that can be used for arithmetic processors [Bro60, Rao70, Man72, Rao74, Don84, For89].
The most important code which is commonly used for coded processing is the so-called *arithmetic code* (AN-code, see Chapter 2.1.3). In 1989, Forin made the first use of AN-codes for coded processing in a real application [For89]. He defined coded operations for most arithmetic operations and extended signatures to this kind of code to detect operation, operator and operand errors.

NOTE:
The term *coded processing* also found a place in current norms and standards. For example, the norm IEC 61508 describes it as follows [IEC10c]:

Processing units can be designed with special failure-recognizing or failure-correcting circuit techniques. So far, these techniques have been applied only to relatively simple circuits and are not widespread; however, future developments should not be excluded.

2.2.1 Coded Operations

Using code words for data processing, the operations must preserve the code for the result. This means that the coded result, processed from the coded operands, must correspond to the original result. In case of an addition, the original operation can be used. But, the multiplication of two arithmetic code words shows the necessity for corrective measures to preserve the encoding function f_c of the AN-code. Before we define coded operations as corrective measures, further background of algebraic structures is necessary. The theory of algebraic structures describes the term homomorphism as a structure preserving transformation between two algebraic structures. Or in general, a homomorphism is a mapping between two algebraic structures of the same type.

Definition 3. *(Homomorphism) [Beu94]*
Let $(G, \circ)$ and $(H, *)$ be two algebraic structures of the same type, then a homomorphism is the function $\varphi : (G, \circ) \to (H, *)$ and for all $x, y \in G$, it is

$$\varphi(x \circ y) = \varphi(x) * \varphi(y).$$

This means in case of an AN-code that the ring of integer $(\mathbb{Z}, +, \cdot)$ and all multiples $(n\mathbb{Z}, +_c, \cdot_c)$ are homomorphous, if it is

$$f_c(x + y) = f_c(x) +_c f_c(y) \qquad \text{and} \qquad f_c(x \cdot y) = f_c(x) \cdot_c f_c(y). \tag{2.8}$$

with $\varphi = f_c$. For the structure of coded data, the operations $+_c$ and $\cdot_c$ (= *coded operations*) must be defined in a way that all necessary corrective actions are done to fulfill the homomorphism.

In case of the addition, there are no further corrective actions necessary:

$$\Rightarrow +_c : (C_1, C_2) \to C_1 + C_2 \qquad \text{for all } C \in C_{AN} \tag{2.9}$$

But in case of the multiplication, there must be a correction:

$$\Rightarrow \cdot_c : \ (C_1, C_2) \rightarrow \frac{C_1 \cdot C_2}{A} \qquad \text{for all } C \in C_{AN} \tag{2.10}$$

Definition 4. *(Coded Operation)*
Let $f_c : X \rightarrow C$ be the encoding function of the code C and $\circ$ an operation that forms an algebraic structure with respect to the original set X.
Then, the operation $\circ_c$ is called "coded operation" of $\circ$ for the set of code words, if it exists the homomorphism

$$f_c(x \circ y) = f_c(x) \circ_c f_c(y).$$

2.2.2 Examples for Coded Processing

In the following, there is a summary of examples for coded data processing found in recent publications.

2.2.2.1 Vital Coded Processor

The *vital coded processor* (VCP) was probable the first application of coded processing. In 1989, Forin [For89] published an article that describes the basic concept of coded data processing for a safety-critical railway application. He made use of information redundancy in form of arithmetic codes with a signature technique to detect following types of error:

- An *arithmetic code* encodes the input variables and computational errors are detectable.

$$x_c = A \cdot x \tag{2.11}$$

- A *static (data) signature* is assigned to each variable to detect addressing errors (e.g. operand error, operator error, variable confusion).

$$x_c = A \cdot x + B_x \tag{2.12}$$

- The *dynamic (timing) signature* is updated with every processing cycle. In case of updating errors (e.g. missing or unwanted storage, incorrect number of loops), the dynamic signature of the output does not match the intended one.

$$x_c = A \cdot x + B_x + D \tag{2.13}$$

- To detect branching or sequence errors (e.g. if-then-else), a global *sequence signature* is added to each coded value. This signature must be pre-calculated for a dedicated program and is updated during the execution.

$$x_c = A \cdot x + B_x + D + G \tag{2.14}$$

In general, the VCP follows the standard sequence of a data processing unit: *capturing input data, data processing and output control* (Figure 2.4).

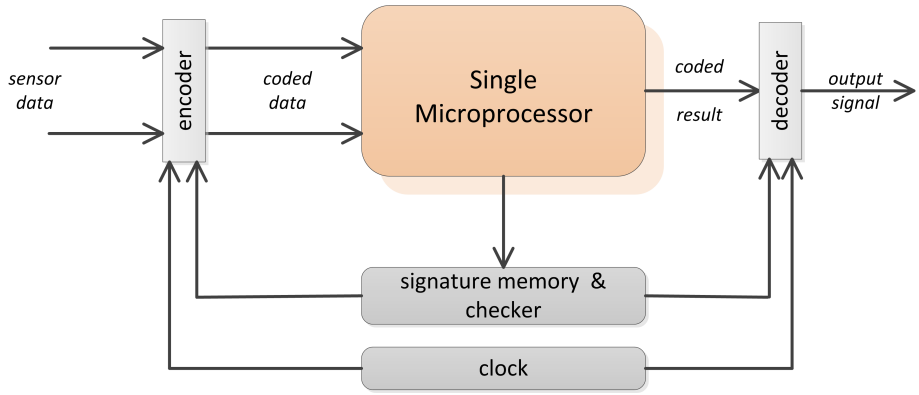

Figure 2.4: Architecture overview of the Vital Coded Processor

In Figure 2.4, the vital input data are encoded by an additional fail-safe hardware. At encoding time, the encoder hardware uses the pre-defined (e.g. during development process) constant signatures that are stored in a separated signature memory during system build. The executed software in the processor must be able to deal with the coded data by means of coded operations. These operations also process the signatures of the coded variables and calculate the value of the output. The signature of the output is verified by the dynamic check controller, which also implements the signature memory. The decoder hardware controls the output depending on the verified output variable. In case of an error, the output is set to a safe state. There are several disadvantages of the VCP approach: (1) The data flow of the program has to be analyzed before to pre-compute the valid signatures of the output. (2) The sum of all signatures (static, dynamic and global) must be smaller than A. (3) Additional fail-safe hardware is necessary to encode

the input, store the signature, verify and decode the output. (4) The publication does not tell anything about performance or loop structures.

2.2.2.2 Software Encoded Processor

The system engineering group at the University Dresden presented another approach of coded processing [WF] but without the disadvantages of Forin´s vital coded processor. The *software encoded processor* (SEP) can execute arbitrary programs on standard hardware. The basic idea is an interpreter which executes the coded variant of a program without any knowledge about the data flow and pre-computed signatures are not necessary (Figure 2.5).

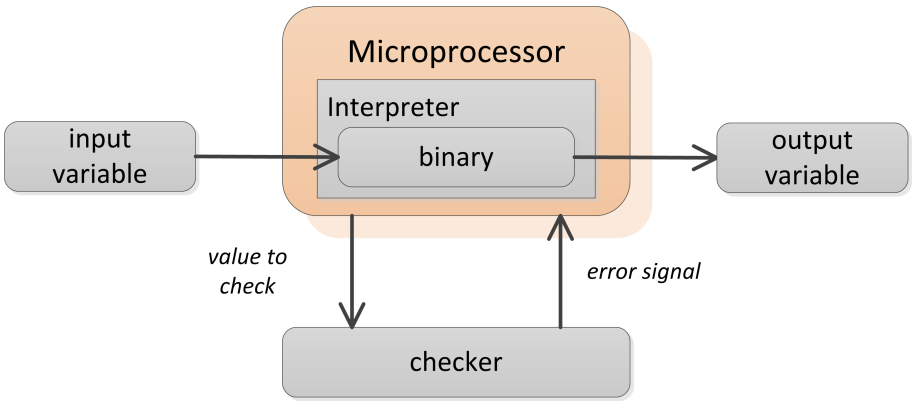

Figure 2.5: Architecture overview of the Software Encoded Processor

Instead of assigning pre-computed and stored signatures in separated memories, SEP makes use of so-called dynamic signatures that are calculated during runtime. This dynamic signature consists of a unique address for each value (variable or instruction) and a version which counts the updates. Because it is impossible to predict the data flow of unknown programs, the signature is dynamically updated during the runtime. The interpreter itself is an encoded program following the VCP approach. The SEP interpreter reads the coded instructions out of the program memory and decodes and executes it at assembler level. The output of the program is coded, as well. Similar to VCP, the coded output and the current address with version counter are sent to an external checker hardware for verification. They also reported the possibility of integrating the checker into the interpreter so that no additional hardware is required. But in case of a hardware error in this checker, it is probable that an erroneous output is not detected anymore. Compared to the VCP, this approach has the advantage that every program can be executed without the knowledge of the program flow. However, there is the big disadvantage of performance. The encoded operations, the interpretation and dynamical calculation of the signatures lead to worse performance by a factor of more than 900.

2.2.2.3 Safely Embedded Software

The Laboratory for Safe and Secure Systems at the University of Applied Sciences Regensburg (LaS^3) developed, in collaboration with the TU Munich, the *Safely Embedded Software* (= SES) technique for the programming language C to safeguard the execution of code on microprocessors [VÖ07, MSVZ07, Mot08, SMM+09, Ste09, Mot09, SMM10, Lau12].

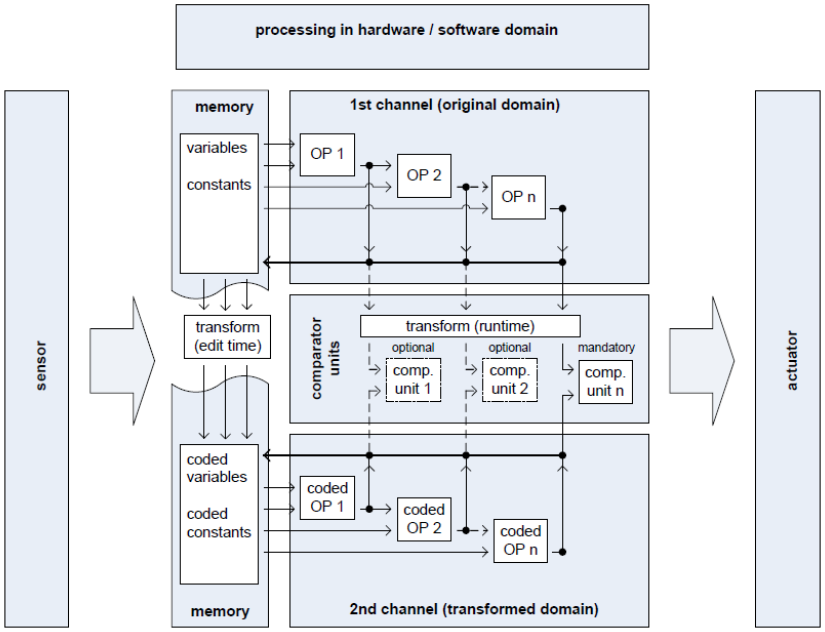

Figure 2.6: Architecture overview of the Safely Embedded Software approach by a combination of diverse data processing and duplicated instructions [VÖ07].

The SES approach uses two of the basic principles of fault detection: diversity and duplication. The original program is executed as a first channel and is transformed during development time. At source code level, the original data of the program are coded by the rules of Forin (AN-codes with static and dynamic signatures). A second channel of the program processes this diverse data path by duplicated operations. These operations must be transformed to coded operations in order to deal with the coded data. Both channels are executed in parallel by code weaving which mixes the operations of the uncoded channel with that of the coded channel (Figure 2.6). After every operation, but at least at the end, the results of both channels are compared by a comparator unit. The comparator decodes the data of the coded channel during runtime before both results can be compared. A sample implementation of SES shows a runtime increase by a factor 10 and about 8 times more of memory is needed. There is no automatism of weaving the second program channel into the existing uncoded program. Additional research effort is necessary for the development of a safety compiler that would do this automatically.

2.2.2.4 Siemens Fail Safe Automation System

Siemens published a patent in 2010 that describes an error control especially for automation systems [KSS10]. The claims comprise a method to improve the error control of a processor system with fail-safe peripheral components and a communication link between them. The main part is the check of safety-critical data and their processing in the processor system by the combination of uncoded and coded data processing (see Figure 2.7).

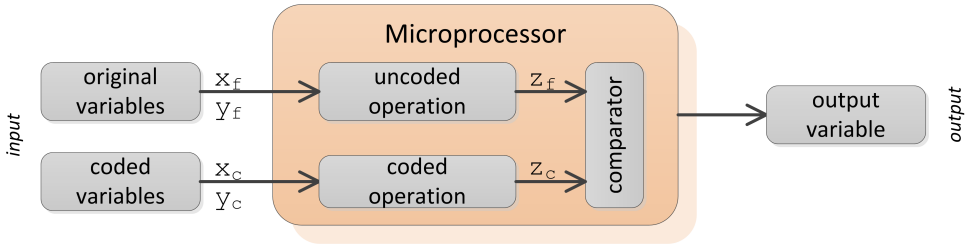

Figure 2.7: Architecture overview of Siemens´ diverse data processing

Siemens basically follows the approach of Forin´s vital coded processor, but with two diverse channels, one uses the original data and the other one is coded. They combine the advantages of both, duplication and diverse data representation. However, Siemens´ concept has some changes compared to the vital coded processor. First, the diverse processing of the same data does not verify their signatures with pre-computed signatures which are stored in an external memory device. Second, without signatures, they use the original AN-code (in contrast to SEP) and the coded operations are simpler in this case. The coded channel is automatically generated and added in the software by a so-called F-compiler (F=failsafe). The verification compares the result of both channels by coding the original data path according the same rules.

2.2.2.5 CoRed - Combined Redundancy

The *Combined Redundancy (CoRed)* approach [UHK+12] combines different software-based fault detection mechanisms. Pure replication like *TMR* [GRRV06] has the disadvantage of single points of unprotection for example input data and the voter. Although these points of interest are very short in execution and they are unlikely corrupted, these gaps must also be considered for highly safety-critical applications. Therefore, the *CoRed* approach employs a three-staged protection.
Figure 2.8 shows the concept of *CoRed* which combines mainly three software-based techniques for hardware fault detection. First, TMR is used for data acquisition and data processing. Second, the limitation of TMR, which is the transfer of sensor data to the

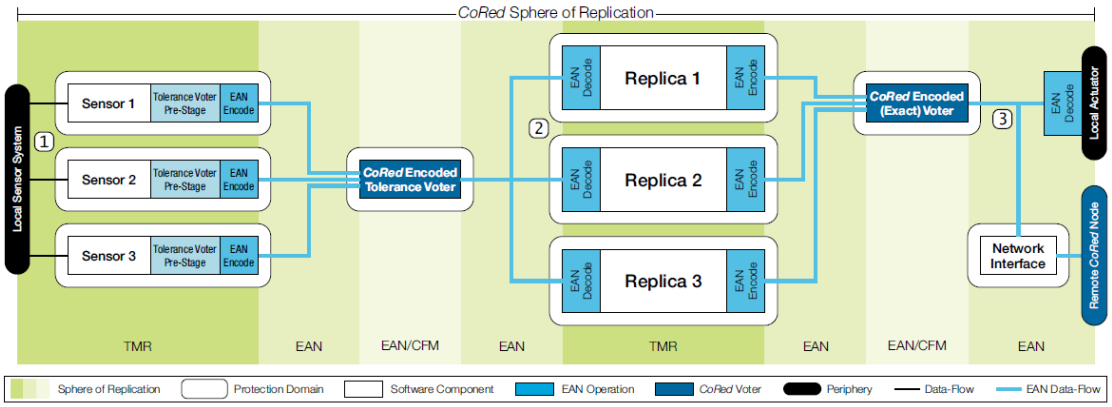

Figure 2.8: Overview of the Combined Redundancy (CoRed) approach [UHK+12].

processing unit and the subsequent voter after the multiple computations, is compensated by data flow encoding by means of arithmetic codes. And finally, the data integrity is insufficient in case there are deviations in the control flow of the voter. So, additional control flow encoding is implemented in the voter. Thereby, they achieve almost total detection of soft-errors in TMR execution and in the voter. Other parts like operating system remain unprotected. Furthermore, there is the interesting fact that TMR detects almost all injected faults, whereas the encoded data flow has a remarkable contribution to the error detection for the voter. Therefore, it could arise the assumption that TMR can be replaced equivalently by coded data processing and vice versa. But here, the voter cannot be realized by TMR because the outputs of the replicated voters must be voted again. So for the voter, coded data processing is the only way to guarantee dependability.

NOTE:
There is one remarkable issue in this publication. The implementation of *CoRed* uses C++ template classes like it was reported in [1]. The big advantage of doing so is the abstraction of the coded operations in additional software libraries. From a programmer´s point of view, the implementation is then transparent to the data types and operations.

2.3 Evaluation of Coded Processing Approaches

For each example of coded processing, they did some evaluations about the efficiency of error detection. The detection of errors during the data processing is the crucial basis for making the system more tolerant and even more safety. Therefore, the dangerous case is the non-detection of an error and the possible propagation to a system failure. But, it must be clear that every approach only increases the dependability by detection and there is still a residual risk for undetection. Of course beside the increased error detection capability, each approach requires more resources in form of memory and runtime. However, these performance issues are not part of the research in this thesis.

2.3.1 Residual Error Probability

One effect of faults is the deviation in the data path of software processing. As previously described, coded processing is a possible technique using error codes to protect the data processing against arbitrary faults in the hardware. The problem with every error detecting code is the fact that certain faults cannot be detected. The probability of undetected faults is an important metric in the theory of error detecting codes. Thus, there are a lot of publications that deal with the performance and effectivity of arithmetic codes for coded processing [Avi71, SSSF10a, MS09]. In this context, a data error is the deviation of a result caused by faults during the computation. This means that there is an error word[5] E added to the code word C. The sum of both is the corrupted result C'.

$$C' = C + E \tag{2.15}$$

With coded processing, there are now two possibilities of verification [Oze92]:

1. Check after every operation:

The coded variable is verified after each operation. An error E would transform the valid code word C into the invalid word C'. In case of AN-codes, the error is detected, if and only if it is

[5] In the context of error detecting codes, the error word is often called *syndrome*.

$$
\begin{aligned}
0 &= C' \mod A \\
0 &= (C + E) \mod A \\
0 &= \underbrace{C \mod A}_{=0} + E \mod A \\
0 &= E \mod A.
\end{aligned} \tag{2.16}
$$

If the error word E as a multiple of A, then the residue of $(C \mod A)$ is zero and the error is not detected. The code words C are a multiple of A and therefore a subset of all integers in the range $[0; A \cdot |X|]$ with $|X|$ is the number of elements to encode. Thus, there are A times more elements in the code space C than required for coding the original space X. Provided that the error words are equally distributed then the probability of being a multiple of A is [For89, Oze92, SSSF10b]

$$P_u = \frac{|X|}{|C|} = \frac{|X|}{A \cdot |X|} = \frac{1}{A}. \tag{2.17}$$

Schiffel also reported in [Sch11] a probability of undetection as a function of $\frac{1}{A} \approx \frac{1}{2^k}$ with k is the number of redundant bits determined by the generator A. She confirmed the probability by experimental evaluation by injecting random and equally distributed errors into random encoded numbers. However, this assumption is questionable whether error words follow an equally distributed probability. A flip of one bit is still more probable than two or more bit-flips. Therefore, she evaluated all combinations of n-bit faults in the error word E and checked, if it is a multiple of A. This brute force analysis leads to a probability of undetection as a function of the number of bit-flips. It seems that there is an improvement the larger the generator A is. She also showed that some non-prime As have a better performance than prime As, provided for equally distributed bit-flips. Finally, she referred to further evaluation in the future, especially for non-prime As.

2. Check of coded variable after m operations:

As we have seen in the presented approaches before, the coded data are usually not verified after each operation. Therefore, Ozello distinguished in [Oze92] a second case for the evaluation of undetected errors. The code word is not verified after each operation but only at the end of a set of operations. A possible fault E_1 during the first operation propagates the deviation to the following operation and the result remains faulty after the second operation ($= E_2$). The operation itself or other faulty variables can introduce new faults and influences the final error word. Ozello described this series of code words

by a polynomial $E_g = \{E_1, E_2, ..., E_m\}$, with m is the number of instructions until the verification of the result is done. He demonstrated that the probability of undetected faulty code words is also $\frac{1}{A}$ in this case. Additionally to the effect of the transition from a valid to an invalid code word, there is the effect that consecutive instructions compensate each other. For example, the sum of two faulty coded variables with syndrome E_1 and E_2 result in a valid code word, if the sum of both syndromes is a multiple of A again:

$$(E_1 + E_2) \mod A = 0 \tag{2.18}$$

2.3.2 Fault Injection

The validation of software-based approaches for hardware fault detection like coded data processing is an important proof of efficiency. Under normal operation, the occurrence of previously mentioned soft-errors as a root cause for system failures is a very seldom event. This makes it difficult to test a certain approach. Consequently, we have to reduce this time by artificial increase of the disturbance in the environment, which is equivalent to a higher fault probability. But, these faults still remain random and a systematic test of all types of faults is not possible. Therefore, the literature reports a lot of fault injecting approaches [HTI97, ACK+03] which are summarized in principle in this sub section. Figure 2.9 shows a typical fault injection environment which basically consists of the target system to test, the fault injector, the fault injection monitor and the controller which supervises the fault injection.

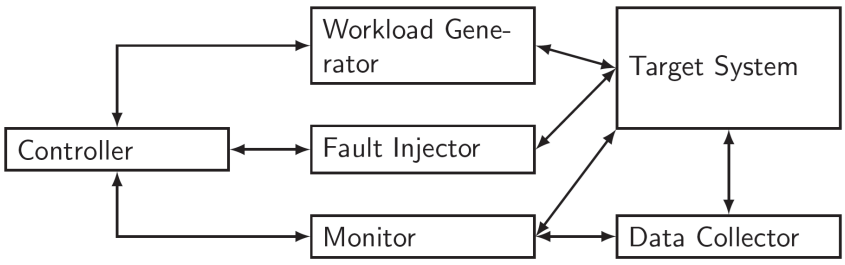

Figure 2.9: Typical fault injection environment [BP03].

In Figure 2.9, there is no mapping of the block to certain components. Depending on the injection method, the fault injection system is executed on the target system (*software-based injection*) or on additional hardware connected to the target system (*hardware-based injection*).

1. Hardware-based fault injection:
This kind of fault injection uses external hardware which injects the faults into the target system. Further, this type can be distinguished between contactless methods [KLD+94] and techniques with contact to the target system. Contactless fault injection has no direct contact but it influences the target system by some physical phenomenon such as heavy-ion radiation or electromagnetic interference. So, it simulates the original fault mechanism for soft-errors. In contrast to that, fault injection with contact uses pin-level probes to produce voltage or current changes to the hardware of the target system [MRMS94]. This is probably the most common method, which simulates other types of faults for example disturbances on the power supply or stuck-at fault.

2. Software-based fault injection:
In contrast, software-implemented fault injection (SWIFI) is an alternative to avoid expensive hardware. The fault injector in Figure 2.9 is part of the software running on the target system. This application software is extended by special code (e.g. interrupt routines) which manipulates certain parts of the processor like memory or registers [VM97]. However, this technique allows only the corruption at application level. At the machine code level, this software-based fault injection is insufficient. Therefore, more recent approaches deal with hardware instrumentation based on FPGA [FMB+12, FM12] or they use binary instrumentation [WF06] to insert faults directly into the machine code.

Fault injection has a wide area in the research of fault-tolerant systems. Thereby, the injection technique often depends on the underlying hardware. Here, this summary only makes an overview of state-of-the-art fault injection which is used to validate coded data processing. In contrast, analytical model-based methods for evaluating error codes in data processing are not reported. So, such models are promising for future researches as an alternative evaluation technique beside classical experimental-base method.

Chapter 3

Goals of the Thesis

As reported by the introduction in Chapter 1, modern computer systems become more vulnerable against arbitrary soft-errors. To be fault-tolerant, recent and future computer systems require additional redundancy in form of hardware (e.g. memory, parallel components) or in form of execution time (e.g. repeated execution). In recent years, the use of error detecting codes for arithmetic processors seems to be an important field of research, too (Chapter 2.2). Compared to classical methods of duplication, error codes offer a better capability of error detection with the same amount of memory. However, the slow-down of software execution caused by coded operations must not be underestimated. An important metric associated with error codes is the residual error probability. This metric specifies the probability of an undetected error. For coded data transmission, where the underlying channel model is known, the analytical evaluation of the residual error probability is possible. But in contrast, there are no comparable channel models available for coded data processing in an arithmetic unit because of its complexity. The determination of the residual error probability is often done by experimental methods which require the final software executed in a special environment (Chapter 2.3.2). However, an analytical approach would lead to the residual error probability much earlier in the development process. Therefore, this thesis addresses following objectives to give a possible solution to model the error behavior of a data flow in software processing units caused by arbitrary faults in the underlying hardware.

Objective 1.

Development of an error model for arithmetic operations:

In literature, a residual error probability of $\frac{1}{A}$ is often reported for AN-codes. This probability is evaluated by experimental studies or by simplifications of known error mechanisms. But, the analytical evaluation of the residual error probability requires a precise probabilistic error model of the underlying hardware. Thus, the crucial goal of this work

is an an error model to describe so-called operation errors of a certain arithmetic operation, e.g. the addition.

Objective 2.

The comparison of the residual error probability of different codes:
The residual error probability also depends on the used coding technique. Arithmetic codes are commonly used for coded operations, but there are a lot of further error detection codes available. Despite of the advantage of arithmetic codes to preserve the code after arithmetic operations, the performance of linear codes is unknown with respect to arithmetic operations. Therefore, linear codes shall be evaluated by the error model from Objective 1 as an alternative for coded processing.

Objective 3.

Reliability evaluation of a task´s data flow:
The Markov process is a formal method which is often used for the evaluation of reliability. The data flow of a task consists of a set of instructions that are consecutively executed. The reliability of the whole data flow depends on the error probability of each single instruction. A further goal of this work is to model a data flow into an applicable Markov model, whereas the residual error probability corresponds to a dedicated system state. Furthermore, the data verification cannot be done after each instruction. Multiple faulty instructions are cumulated until the final result is verified and it exists the possibility of fault compensation. This means that two faulty and dependent instructions produce the same output as the fault-free execution. This effect influences the effective reliability of the system and must be considered.

Objective 4.

Validation of correctness of derived models:
The correctness of the created error models is important for further work. Therefore, a simulation approach shall be used to evaluate the reliability of a single and also of a simple data flow.

Furthermore, following constraints are defined for this work to restrict the complexity: First, only soft-errors are considered which are assumed to become a bigger impact to the reliability of software systems Second, the processor system to be analyzed is limited to data flow instructions without any jumps and branches to reduce the complexity at the beginning.

The vision of this thesis shall be the preparation of a basic concept which offers the possibility for a model-based reliability evaluation of software in an early stage of the development process.

Chapter 4

Error Models for Reliability Evaluation of Data Processing Units

This Chapter is the cumulative part of the thesis and summarizes the contribution of all referred publications in appendix A. The publications are logically connected and commented with respect to the defined objectives in Chapter 3 and their relation to each other. Figure 4.1 shows the pictorially overview of several aspects of this work. Each aspect is discussed in a publication which (partly) fulfills a defined objective of the thesis.

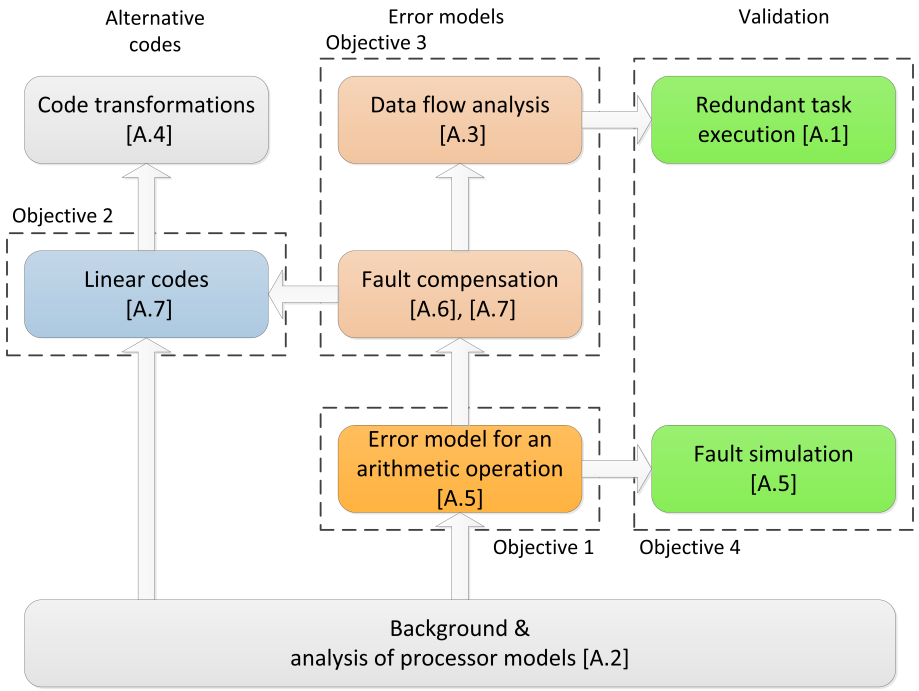

Figure 4.1: Overview of the cumulated publications (references in brackets) and their logical relation to each other.

This work and the embedded publications are the result of the research project *Safe Oriented Programming of Software-Intensive Embedded Systems* (S^3OP). This project was established and carried out at the Laboratory for Safe and Secure Systems (LaS^3), which is located at the University of Applied Sciences Regensburg, and in cooperation with the University of West Bohemia in Pilsen. Further, the S^3OP project was supported by the Bavarian State Ministry for Science, Research and Arts (Code: D2-F1116.RE/3/4).

4.1 Arithmetic Processors

The content of this section is derived from the publication in A.2.

P. Raab, S. Krämer and J. Mottok. Cyclic Codes and Error Detection during Data Processing in Embedded Software Systems. In *Proceedings of the 4th Embedded Software Engineering Congress*, pages 577-590, December 2011.

This paper makes analysis for applicable error codes in a generic processor model. This processor model consists of several error models for each component with different characteristics. The paper also contains the background of algebraic structures and error detection codes, which was summarized in Chapter 2.1.3.

4.1.1 Summary

In an embedded system, the processor is the main part which executes the software including the data processing. A deeper view inside a processor system shows several hardware units performing different tasks. The core of a processor contains the *arithmetic logical unit* (ALU) which realizes the fundamental operations like the addition or multiplication. The correctness of a result depends on the fault-free execution of all components in the chain of data processing e.g. memory, data bus and ALU. Each component is vulnerable against disturbances (see motivation in Chapter 1) and it potentially corrupts the data processing. Figure 4.2 shows a simplified model of such a data processing system. For example, the result is wrong if the memory, the internal data bus or the arithmetic operation is corrupted. Indeed, there are still further possibilities of injected faults. But all of them lead either to operand errors, operation errors or to operator errors as a basic fault model for further evaluations.

The important point of this work is that the hardware of a processor system is considered as a noisy channel as defined by the coding theory. But without following the strict definition and the required parameters of a channel (e.g. entropy, capacity ...), the interesting aspect is the erroneous behavior of a certain unit with respect to its output.

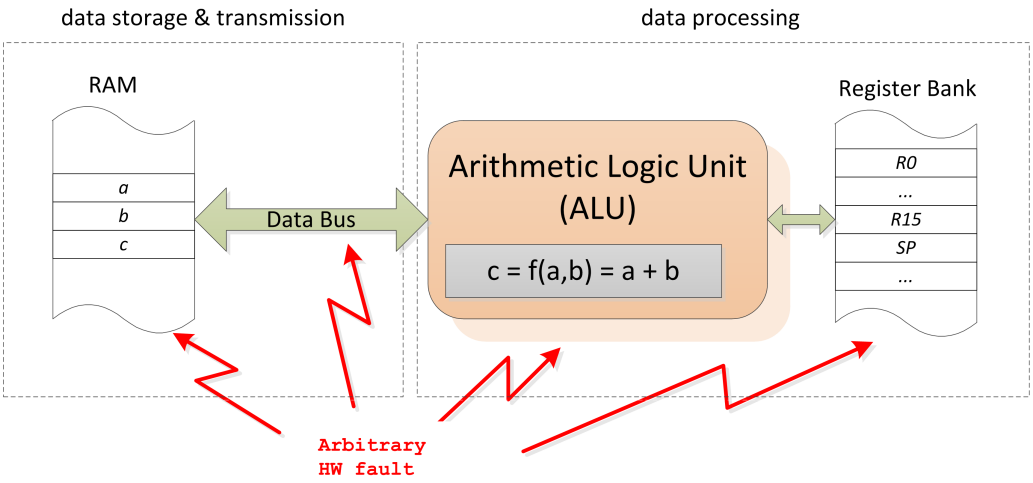

Figure 4.2: Simplified data processor model. It contains of a memory device, data transmission and the data processing unit.

The output depends on the functional realization of the underlying hardware wherein a fault in the hardware has a direct influence on the data processing and therefore to the result. A memory device, for example, consists of many vulnerable memory cells which are grouped to complete data words. For a pure storage purpose, there are no dependencies between the cells (or bits) within a data word. But the concrete realization of the memory device allows further influences in form of unintended coupling. The internal data bus as a data transmission system shows a similar behavior. In contrast to this, some arithmetic operations (e.g. addition) have a functional memory effect to the next digit caused by the iterative nature of the data processing. This means that a faulty computation of a single bit has an impact on the next cycle even if this cycle has no fault. Thus, there are mainly two different characteristics of error models to distinguish: models without a functional memory and models with a memory effect between two bits.

The same classification with respect to the dependencies between nearby bits can be done for error detecting codes, too. Whereas the operations on integer numbers have a functional influence on digits in a binary representation, there are no dependencies between the coefficients (bits) of a polynomial under the use of polynomial operations. This leads to the idea of using the optimal code which matches the characteristics of the underlying hardware. For the determination of the suitable code out of the presented ones in Chapter 2.1.3, both codes must be compared based on a given underlying channel. The residual error probability as a possible metric is a criterion for the decision of which one is better. It describes the probability that the coded output of a channel is still an element out of the set of code words in spite of a fault. The evaluation of this probability

demands the knowledge of the erroneous behavior of the channel in form of a probabilistic model. In case of a memory-less channel like it is for the memory in a processor system, the known and quite simple *binary symmetric channel model* (BSC) can be used for a first evaluation.

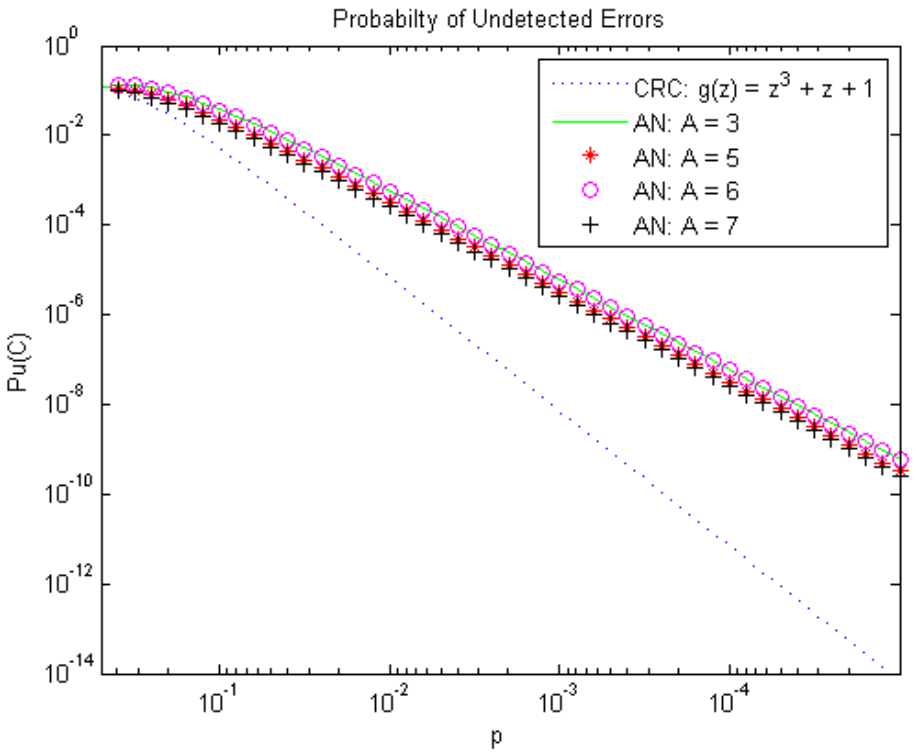

Figure 4.3: Comparison of the probabilities of undetected errors between linear codes and arithmetic codes with BSC-based channels.

Figure 4.3 shows the result of this evaluation. It compares the residual error probability p_u as a function of the basic fault probability p for several arithmetic codes with $A \in \{3, 5, 6, 7\}$ and a linear code with generator $g(z) = z^3 + z + 1$. All codes have the same code rate to be comparable. This means that the same information is encoded by the same number of bits in a binary representation. The curve of the linear code is always below all used arithmetic codes. This means that the residual error probability of this linear code is always smaller and therefore better compared to the others. This result confirms the assumption of an optimal code for a given channel as described before.

4.1.2 Discussion

The paper in A.2 provides the necessary background of error detection codes for coded data processing. In addition, it defines the crucial principle of different channels which

represent the erroneous behavior of each unit in a processor system. This is the main contribution of this paper. With diverse codes and their different characteristics, there is the assumption that one code is better than another based on a given channel. But then, the consequence is the concurrent use of different codes in a coded processing approach and the transformation from one code to the other (and vice versa). Indeed, this thesis covers also the possibility of code transformations in Chapter 4.4. The so-called *AN-codes* (and their derivatives) are state-of-the-art and commonly used in a lot of known approaches for coded data processing in arithmetic processor systems. With the consideration of multiple codes in a processor system, there is still potential for further improvements.

So far, there is only the thesis that linear codes are better than arithmetic codes in combination with memory-less channels. For other arithmetic operations, there is no such a conclusion possible because there is no applicable error model available, yet. More evaluations concerning arithmetic operations with a functional memory are further necessary steps to complete the thesis. In Chapter 4.2.1, a new probabilistic model is presented to estimate the erroneous output of an addition as an example of an arithmetic operation.

The argument of memory-less channels in a computer system is true as long no unintended coupling between single bits is considered. Such a coupling could be a problem in real applications which is often caused by environmental influences and deviations in the manufacturing process. Error models must also consider this behavior for correct estimations of the outcomes. However, the contribution of this paper is based on a simplification in a first step. These are the assumption of BSC-based error models for data storage and transmission in a processor system wherein no additional effects of coupling are regarded. Of course, the models must be enhanced by these issues in future works.

4.2 Error Models

The crucial contribution of this thesis is the development of error models to describe the faulty behavior of arithmetic operations (see Objective 1) and of a complete data flow (Objective 3). Therefore, this chapter discusses a solution for a possible model to evaluate the probability of erroneous outputs of an arithmetic operation (Chapter 4.2.1). Secondly, multiple faulty operations, which are consecutively executed, potentially compensate the faults and the final result is maybe correct again (Chapter 4.2.2). This phenomenon is important for the evaluation of the total reliability in a software´s data flow (Chapter 4.2.3).

4.2.1 Arithmetic Operation

The content of this section is derived from the publication in A.5.

> P. Raab, S. Krämer and J. Mottok. Error Model and the Reliability of Arithmetic Operations In *IEEE 2013 EUROCON - International Conference on Computer as a Tool*, pages 630-637, July 2013.

This paper achieves the goal to develop an applicable model to describe the error behavior of an arithmetic operation by means of a discrete Markov model (Objective 1). The paper also presents the verification of the model by a simulation approach which is thematically summarized in Chapter 4.5.2.

4.2.1.1 Summary

Channel models are important means in the coding theory to describe the behavior of real noisy transmission systems. Considering an arithmetic operation inside a processor system as a fault injecting channel that alters a result, such an error model is required to estimate the erroneous output of the computation. To the best of my knowledge, there is no practical error model available for arithmetic operations compared to those which are known from the coding theory.

In contrast to a quite simple error model like the so-called *binary symmetric channel* (= BSC), the addition in an arithmetic processor implies a functional memory effect to successive digits caused by the iterative process of the carry-bits (e.g. in a *ripple-carry-adder*). The literature already reports models for memory-based channels like burst errors in data transmission lines by means of a discrete Markov model. For example, the *Gilbert model* defines two randomly entered system states which describe the active disturbance of a transmission. Inspired by this approach, the memory in form of a carry-bit can also be modeled by a Markov chain. The *full-adder* as the basic 1-bit element in the addition has two outputs, the sum-bit s and the carry-bit c_{out} for the next adder element. Thus, there are four different possible outputs. In order to be independent of the actual inputs, the four possible error states are defined as the deviation to the correct output. This means that either

(1) both output bits s and c_{out} are correct (*good* state G),
(2) only the sum-bit s is wrong whereas the carry-bit c_{out} is correct (*bad* state $B1$),
(3) the sum-bit is correct but with a corrupted carry-bit(state $B2$) or
(4) both outputs are wrong (state $B3$).

The Figure 4.4 summarizes these error states in form of a discrete Markov chain.
Each transition in the model represents an iteration step of the addition. The fundamental

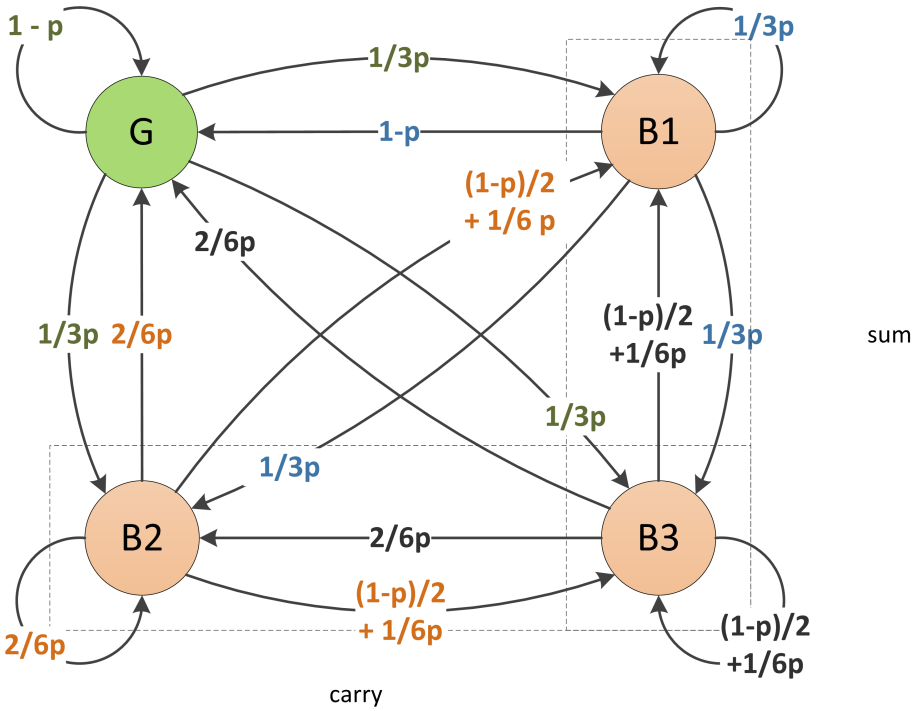

Figure 4.4: Discrete Markov chain which describes the error states of a single 1-bit adder.

fault probability p defines the corruption of a single adder element. In case of a correct carry from the previous adder (states $G/B1$), there is only a faulty transition to states Bx depending on p. But with a corrupted carry-bit (states $B2/B3$), there is also an influence on the output s without further faults (probability $1-p$). Additionally, a new fault can probably compensate the corrupted carry-bit and it causes a transition to state G.

The two states $B1$ / $B3$ represent a deviation in the sum-bit which is the only observable indication of a corruption. With a binary output, the deviation can only be a negated bit ($\neg s$) from $0 \rightarrow 1$ or $1 \rightarrow 0$. For a general notation, the *exclusive-OR* $\oplus$ is used to describe the negation of one operand only if the other operand is 1. Thus, this logical operation masks the result for corrupted bits by the formula

$$C' = (a + b) \oplus E \tag{4.1}$$

with $E = (e_{n-1} \ldots e_1\ e_0)$ is the *error mask* defining the set of corrupted bits ($e_i = 1$) of the result and a,b are the summands of the addition.

When using coded data based on error codes, a deviation in the result is detected by the verification of being a valid code word. But, there is a small probability that multiple faults corrupt the addition in a way that the result is still an element of the code space and the error is not detected. A given error code defines the set of all dangerous error masks

which lead to other valid code words. The probability of these error masks is an important metric of the code. In Figure 4.4, an n-bit addition is represented by any n consecutive transitions starting in state G. For a certain error mask however, only a subset of these transitions are allowed. Each '1' bit in the error mask represents a corrupted bit in the result and there is a transition either to state $B1$ or $B3$. Otherwise, the bit in the result is correct and the state in the model changes to G or $B2$. With two alternatives for each bit in the error mask, there are 2^n different possible sequences which can be summarized in form of a *binary tree* as shown in Figure 4.5.

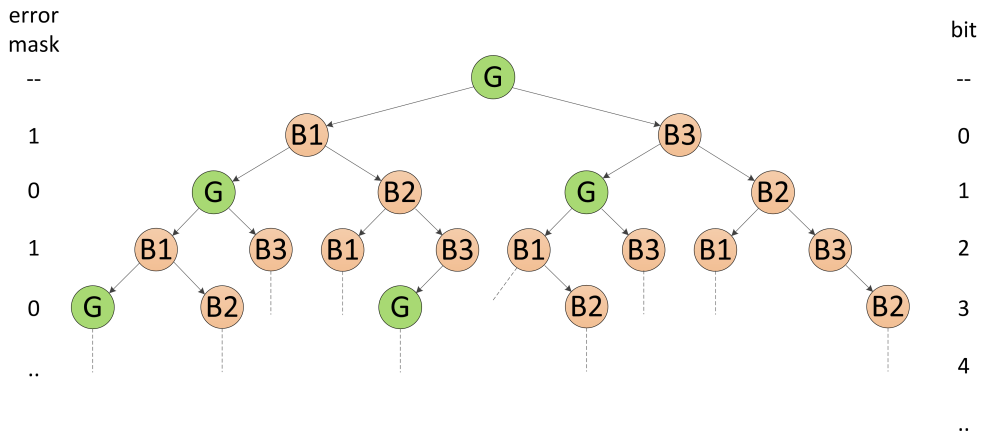

Figure 4.5: Set of possible sequences in the Markov chain in form of a binary tree, which all lead to a certain error mask.

Each path in the binary tree of Figure 4.5 from the root to any leaf is a possible sequence which corresponds to a certain error mask. The product of all single transition probabilities is the probability of this unique sequence. The sum over all 2^n different paths is the total probability of this error mask. Thus, the binary tree represents the probability of a single error mask. The summation of the probabilities over all dangerous error masks results in the total probability of an undetected error (= *residual error probability*) of the code. See A.5 for more detailed information.

4.2.1.2 Discussion

The presented error model is the first attempt to describe the erroneous behavior of an addition by a probabilistic approach. It allows the estimation of a certain outcome which is the base for further evaluations related to the residual error probability of a given code and therefore, it is the crucial part of this thesis. But, there are two important assumptions in this paper. First, the presented model is only applicable for ripple-carry-adders. Second, it is assumed that the fault probability is equally distributed to any error

state in the Markov chain (states Bx). In real processors however, the knowledge of the hardware is only available for the manufacturer. So, only he is able to specify the real fault distribution based on his realization. Indeed, there could be the case that in spite of a single fault, the output of the adder is still correct. This means a transition to state G in the Markov model. The consequence of this modified distribution is the changed transition probability matrix $\mathbf{P}$ of the discrete Markov chain with

$$\mathbf{P} = \begin{pmatrix} (1-p)+\frac{1}{4}p & \frac{1}{4}p & \frac{1}{4}p & \frac{1}{4}p \\ (1-p)+\frac{1}{4}p & \frac{1}{4}p & \frac{1}{4}p & \frac{1}{4}p \\ \frac{1}{4}p & \frac{1-p}{2}+\frac{1}{4}p & \frac{1}{4}p & \frac{1-p}{2}+\frac{1}{4}p \\ \frac{1}{4}p & \frac{1-p}{2}+\frac{1}{4}p & \frac{1}{4}p & \frac{1-p}{2}+\frac{1}{4}p \end{pmatrix}. \tag{4.2}$$

But, the evaluation of the residual error probability with a modified distribution showed that there is only a small deviation (Figure 4.6). This means that the presented Markov chain is re-usable by updating the fault distribution derived from the real hardware.

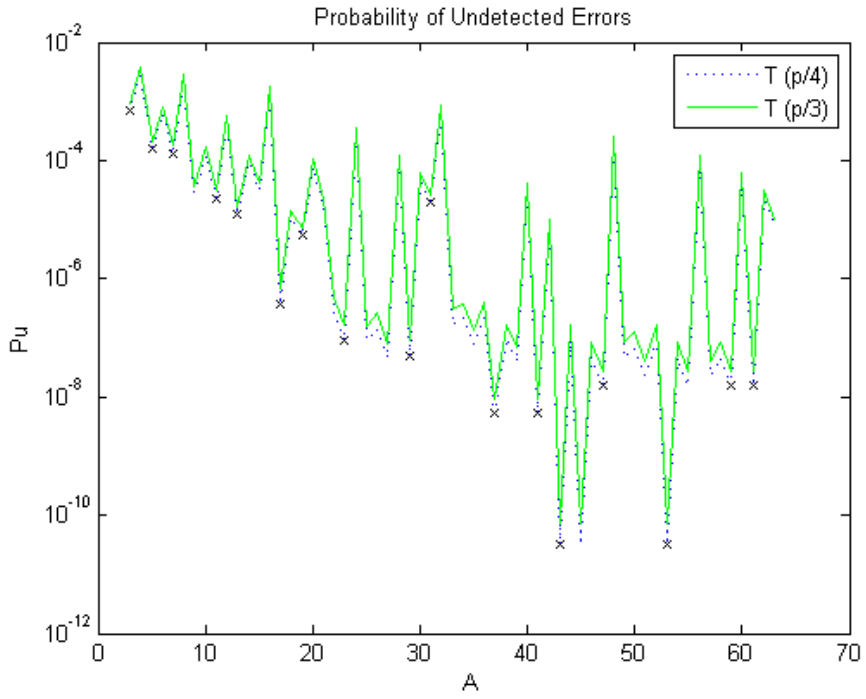

Figure 4.6: The residual error probability as a function of the generator A. The modified fault distribution causes a deviation in the results which can be ignored in this case. The x marks prime numbers for the generator A. T(p/3) means the original distribution of Publication A.5, whereas T(p/4) corresponds to the modified probability matrix P of Equation 4.2.

Nevertheless, the discussed paper describes a basic concept for modeling the error behavior of an arithmetic operation. It allows the analytical evaluation of the residual error probabilities of codes based on a given operation, here the addition of two integer numbers. For coded data processing, the so-called AN-codes are state-of-the-art. The choice of an optimal generator A is an often discussed issue in literature. Now, it is possible to search for an optimal A or even for alternative codes (Chapter 4.3) to compare them in an analytical way.

Beside the residual error probability, the *arithmetic distance* is another important metric which is often discussed along with arithmetic codes. The previously introduced error mask defines the number of corrupted bits in the result and thus, it corresponds to the *Hamming distance* between two arithmetic code words. However, the number of corrupted bits is not equal to the number of injected faults. The transitions in the Markov chain in Figure 4.4 define whether there is a fault with p or there is no fault with $1 - p$. So, a given sequence of transitions tells us the actual number of faults to get this error mask. With a lot of alternative sequences for a given error mask, there is one sequence with a minimum number of faults. Thus, the presented error model also allows us the determination of the minimum arithmetic distance by means of the Markov model.

Furthermore, the evaluation of a software´s data flow requires the error models for more than only a single operation like the addition. A data flow consists of several consecutively executed operations and each of them is vulnerable for arbitrary hardware faults. At the end, the causes for a corrupted result are increasing with the number of operations. In contrast to that, a contrary effect is observable. Two faulty operations can compensate their error if their error masks are the same. This phenomenon is discussed in detail in Chapter 4.2.2. Finally, the next goal is the evaluation of the reliability and, in case of coded processing, the residual error probability of a complete data flow according Objective 3.

NOTE:
For sake of correctness, the discussed publication contains a mistake. In page 634 right column, it is argued that the result C' is a valid code word if $E \mod A = 0$. This is not correct using the bitwise addition in $C' = C \oplus E$. As discussed later, the set of dangerous error masks are determined by the given code and they are also weighted because not every combination of C and E results in a valid code word.

4.2.2 Fault Compensation

The content of this section is derived from the publications in A.6 and partly in A.7.

P. Raab, S. Racek, S. Krämer and J. Mottok. Data Flow Analysis of Software Executed by Unreliable Hardware. In *Proceedings of the 16th Euromicro Conference on Digital System Design*, September 2013.

P. Raab, S. Krämer and J. Mottok. Reliability of Data Processing and Fault Compensation in Unreliable Arithmetic Processors. ***Submitted*** to *Microprocessors and Microsystems*, September 2013.

In these papers, it is evaluated how the data flow in given software can be analyzed with respect to the reliability of the underlying hardware. The papers present the basic principle of reliability analysis by means of simple examples.

4.2.2.1 Summary

The background of paper A.6 is the error model of a ripple-carry-adder introduced in Chapter 4.2.1. However, the presented model only covers *operation errors* of the instruction itself. Faults in the operands are not considered while these faults are important for the reliability of the final result, too. Figure 4.7 shows the concatenated execution of several instructions adding two registers which are set by additional preceding instructions.

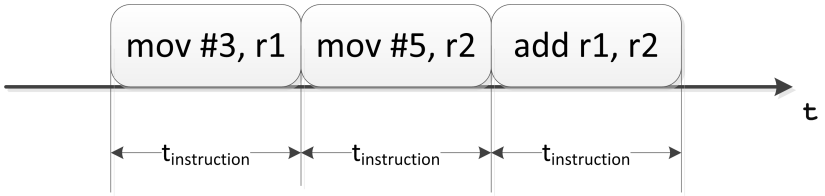

Figure 4.7: Generic machine instructions for the addition of two integer numbers.

The correctness of the final result after the addition depends on the fault-free execution of all three instructions. Or in other words, the reliability of the sum is the product $R = R_{mov} \cdot R_{mov} \cdot R_{add}$ of the single reliabilities. With increasing number of instructions and $R < 1$, the product decreases and the total reliability becomes worse. Assuming a Poisson process, the probability of one single or more faults can be estimated. But with multiple faults, there is also the possible effect of fault compensation. This means that two faults can contrary change the result of a single instruction and the final result is the same as without any faults. The longer a data flow is the higher the probability for multiple faults which can compensate each other. The consequence is that the final

result of a given data flow is more reliable than expected by the reliabilities of each single instruction. Thus, the effective reliability is increased and it can be expressed by the equation $R_{effective} = r_c \cdot \prod R_i$ with $r_c > 1$.

The given data flow in Figure 4.7 contains three instructions which are considered as independent related to their fault behavior. This means that either there is a fault in any instruction or there are two faults or maybe three faults in all instructions. In the theory of probabilities, the so-called *Venn diagram* (Figure 4.8) describes this situation more clearly.

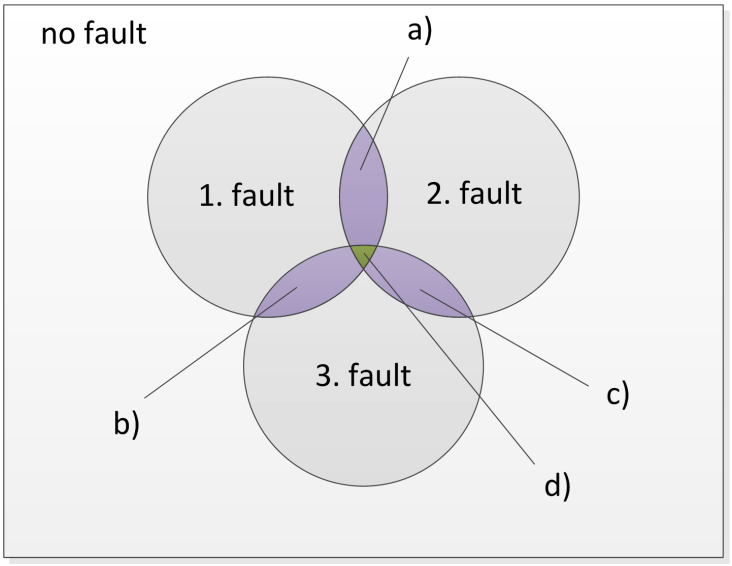

Figure 4.8: Venn diagram of three independent events. Areas a),b) and c) represent the probability of two concurrent faults whereas the area d) stands for the occurrence of three faults.

The intersections in Figure 4.8 between the areas represent the probability of two (or three) faults which concurrently occur. Assuming only two faults that compensate each other, only the intersections a), b) and c) must be considered. But further, the instructions consist of several bits depending on the data width of the processor. This means that there are different locations where the fault can occur and manipulates the result in another way. So, not every combination of faults between two instructions leads to compensation. Based on the discrete Markov chain of a ripple-carry-adder (in Publication A.5), the bit-wise calculation of the sum is extended by the corrupted operands x and y ($s = x+y$). An incorrect bit in any operand influences the sum bit but also the carry-bit and the fault is probably propagated to the next bit stage. The Markov chain in Figure 4.4 with the error states G (= no deviation), $B1$ (= only wrong sum bit), $B2$ (= only wrong carry-bit) and $B3$ (= wrong sum and carry-bit) can be re-used. But the additional faults in the operands

have an impact on the transition probabilities. Let π be the probability of any faults in any instruction with $\pi = 3p - 3p^2 + p^3$. Then the matrix 4.3 describes the Markov process of the addition with corrupted operands. See the Publication A.6 for detailed derivation of the transition probabilities.

$$\mathbf{P} = \begin{pmatrix} (1-\pi) + \frac{5}{6}p^2 + \frac{2}{3}p^3 & \frac{4}{3}p - \frac{13}{6}p^2 + \frac{13}{12}p^3 & \frac{1}{3}p + \frac{1}{2}p^2 - \frac{1}{1}p^3 & \frac{4}{3}p - \frac{13}{6}p^2 + \frac{13}{12}p^3 \\ (1-\pi) + \frac{5}{6}p^2 + \frac{2}{3}p^3 & \frac{4}{3}p - \frac{13}{6}p^2 + \frac{13}{12}p^3 & \frac{1}{3}p + \frac{1}{2}p^2 - \frac{1}{1}p^3 & \frac{4}{3}p - \frac{13}{6}p^2 + \frac{13}{12}p^3 \\ \frac{4}{3}p - \frac{7}{3}p^2 + \frac{4}{3}p^3 & \frac{1-\pi}{2} + \frac{1}{6}p + \frac{1}{6}p^2 - \frac{1}{6}p^3 & \frac{4}{3}p - 2p^2 + p^3 & \frac{1-\pi}{2} + \frac{1}{6}p + \frac{7}{6}p^2 - \frac{7}{6}p^3 \\ \frac{4}{3}p - \frac{7}{3}p^2 + \frac{4}{3}p^3 & \frac{1-\pi}{2} + \frac{1}{6}p + \frac{1}{6}p^2 - \frac{1}{6}p^3 & \frac{4}{3}p - 2p^2 + p^3 & \frac{1-\pi}{2} + \frac{1}{6}p + \frac{7}{6}p^2 - \frac{7}{6}p^3 \end{pmatrix} \tag{4.3}$$

The extended Markov chain allows us to determine the reliability of the addition of two integer numbers according Figure 4.7. The results of this reliability evaluation are shown in Figure 4.9. The figure compares the reliability derived from the model (continuous line) with that reliability derived from the serially connected reliability network[1] (dotted line). The deviation between both is mainly caused by the terms containing any power of p. These terms disappear with decreasing basic fault probability p.

4.2.2.2 Discussion

The analyses in these papers showed that fault compensation in data processing systems must not be ignored. The longer the software the higher is the chance for fault compensation. The effective reliability is higher than expected by the reliabilities of all single instructions. The given publications in A.6 and A.7 cover the effect of fault compensation by analyzing simple examples of a data flow. Indeed, the reported deviation is very small. But in real applications, there are usually more instructions forming a data flow than in the presented example. The higher the risk is for multiple faults the higher is the chance for another fault that compensates the former one. The presented publication in A.6 discusses the addition as an example of a conjunctive data flow. In contrast to a simpler linear data flow where one instruction only depends on the previous instruction, the correctness of the addition depends on both operands which are usually the result of other calculations. These previous calculations, which are now combined by the addition as a conjunctive instruction, determine the reliability of the operands. In spite of the very small chance for fault compensation in the presented example, the probability of a corrupted operand is expected to be higher in real applications because of the huge set of

[1] The total reliability of three instructions is the product of all single reliabilities, because the result depends on the fault-free execution of all three instructions.

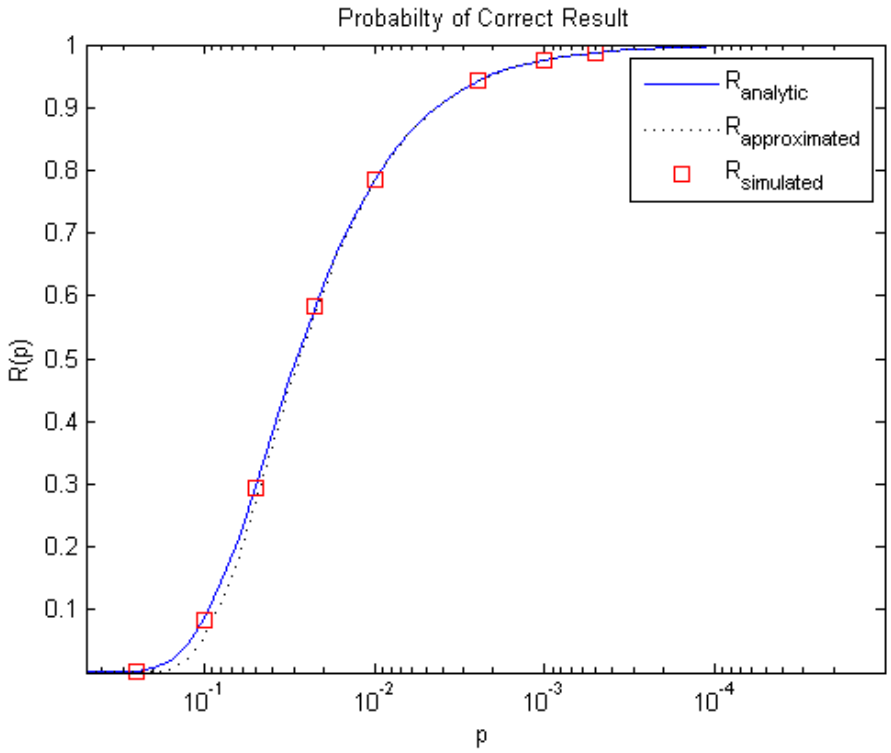

Figure 4.9: Effective reliability of a given data flow with fault compensation.

vulnerable instructions executed before to get this operand. This means that the probability of a single fault in any operand is higher than the probability of the faulty addition with $p_{op} \gg p_{add}$ instead of the assumption of equal probabilities $p_{op1} = p_{op2} = p_{add}$ in A.6. The consequence is a higher chance for compensation compared to presented evaluation because in Equation 4.3, the terms with any power of p have a bigger influence.

Of course, the error model of the addition of two integer numbers is not enough for a complete reliability evaluation of given software related to arbitrary faults in the underlying hardware. Therefore, the goal of future works should be the development of further error models for a given set of instructions. But nevertheless, the reliability (and the fault compensation) of a data flow can already be estimated with some assumptions. In Publication A.7, we investigated a general linear data flow which consists of k instructions based on a BSC model. Let p be the fault probability and n be the data width of the arithmetic processor, then the effective reliability of the data flow is greater by the factor

$$r_c = cosh(pk)^n. \tag{4.4}$$

See Publication A.7 for detailed derivation of this factor and further discussion. But based on interesting properties of the hyperbolic cosine, it is possible to make conclusions about

the steady-state behavior. With increasing number k of instructions, there is a minimum reliability with

$$R_{min} = \frac{1}{2^n}. \tag{4.5}$$

The fact of a minimum reliability also implies that fault compensation cannot be ignored. Otherwise, if there wouldn´t be fault compensation, the reliability of an infinite long data flow would be zero. This fault compensation can be regarded as an intrinsic repair inside a processor system which must be considered in further reliability evaluations (e.g. in Chapter 4.2.3).

4.2.3 Data Flow Analysis

The content of this section is derived from the publication in A.3.

> P. Raab, S. Racek, S. Krämer and J. Mottok. Reliability of Task Execution during Safe Software Processing. In *Proceedings of the 15th Euromicro Conference on Digital System Design*, pages 84-89, September 2012.

This paper presents the reliability evaluation of task execution during safe software processing by means of an enhanced continuous-time Markov model in a more abstract way.

4.2.3.1 Summary

Markov models are an often used and very powerful tool for reliability analysis. In Chapter 4.2.1, a discrete Markov model was introduced to describe the erroneous behavior of a ripple-carry-adder and to determine the probability of an erroneous result. The observed system is very small and it only covers a single arithmetic operation. Usually, Markov models describe the behavior of complete systems and the next step is the evaluation of a task system with respect to its reliability.

A task in a software system is the realization of a data flow. It consists of a set of operations, which each is vulnerable to randomly injected faults which potentially corrupt the final result. The more operations there are, the higher the risk for a wrong task output. With a constant fault rate, which can be assumed for electronic components, the time until one fault occurs is exponentially distributed. The task system changes the state from a fault-free to a faulty state. This means a failure of the task system which can only be prevented by standard techniques for fault detection. For example, the time redundant execution of the same task produces the same output if no faults have occurred. Figure 4.10 shows a (time) redundant task system which totally fails when both tasks have failed.

Figure 4.10: Parallel redundant system with two components.

In fail-safe or fail-operational systems however, it is required to detect such changes to a faulty state to avoid dangerous system failures or to recover the system. Because this detection mechanisms are usually done at defined points in time, e.g. at the end, the recovery is more regular and therefore it is not exponential distributed (Figure 4.11). The detection rate (or repair rate in case of recoverable systems) is not *markovian* and therefore the common Markov model cannot be used.

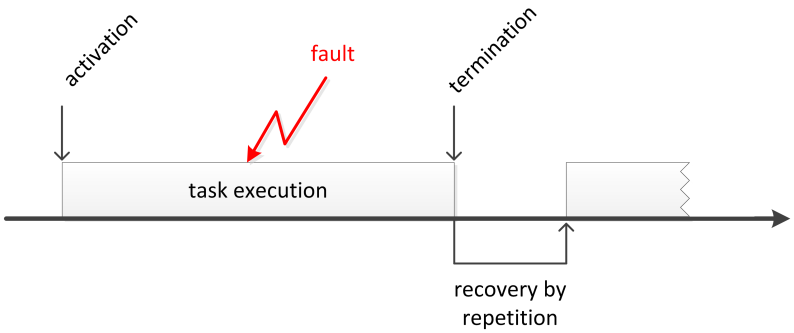

Figure 4.11: The task is repaired at a defined point and the recovery is the repetition of computation.

The *Gaussian* distribution matches the behavior of such repairs much better. Assuming detection at the end of a given task, the nearly constant runtime has a deviation which only depends on the actual execution of the program flow. The *Erlang* distribution as a composition of several exponential distributions can be used to approximate the normal distribution with an expected value for the runtime of the task. With one exponential distribution representing one single transition in a Markov model, the composition of several exponential distributions is mapped to several vertical stages. This distribution is known as the Erlang distribution from the queueing theory which connects two different probability distributions by a queue. In case of faulty tasks, the incoming faults are exponentially distributed, whereas the faults are detected following the normal distribution for example.

Figure 4.12 shows the enhanced Markov model which is based on Figure 4.10 and extended by the runtime information of the task. Every vertical stage represents a certain time interval within the task processing. The more stages there are the higher the resolution is in time. Basically, a continuous-time Markov model is described by a set of

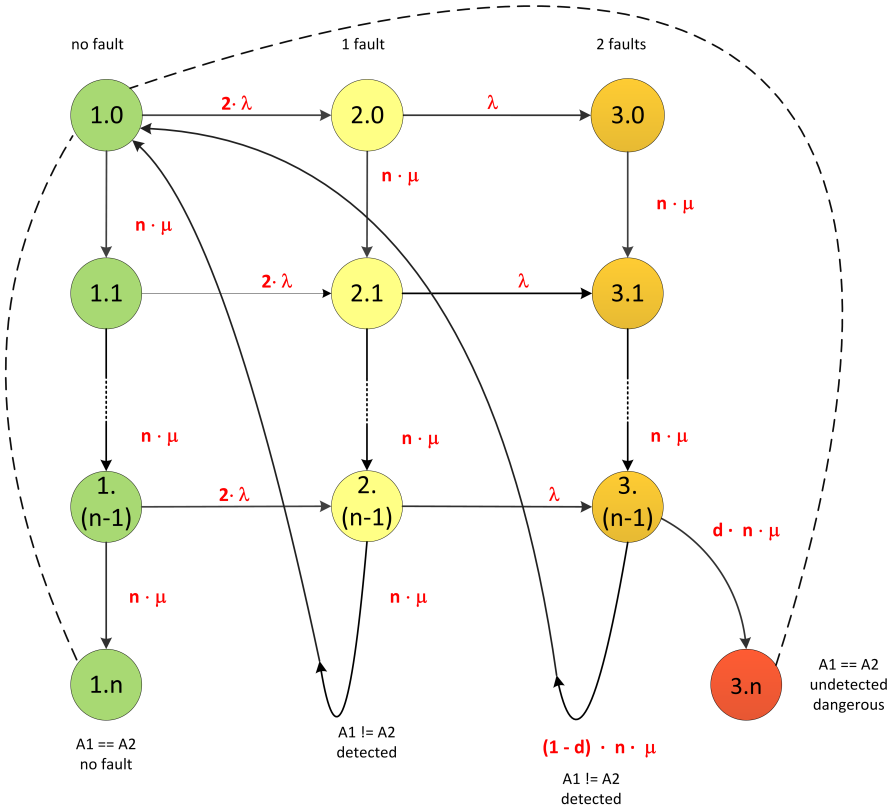

Figure 4.12: Extended Markov model of a parallel redundant task system. In case of a detected fault, the task execution is repeated (transitions to initial state 1.0).

differential equations with one equation for each state. The solution of these equations leads to the state probability as a function of time. The probability of staying in a certain state of a column is maximal just within the time interval the state stands for and it is zero outside this time interval. The last states represent the termination of the task either without any fault (state 1.n) or with an undetected fault (state 3.n). Because there are no outgoing transitions, these states are absorbing states and the probability of entering one of these states is unity in total. The computational effort for the state probabilities grows with increasing number of stages. However, the time-based state probability of each state isn´t important in many cases and a simplification is possible. When assuming non-stop execution of the task, two additional transitions from the absorbing states to the initial state are added (dotted lines in Figure 4.12). The consequence is that the Markov model becomes *ergodic* and important metrics like *availability* or *MTBF* can be determined by standard techniques like *frequency technique* (see Publication A.3 for more detailed information). This non-stop execution of a task corresponds to periodic tasks which are often used in embedded systems.

4.2.3.2 Discussion

The presented approach of Publication A.3 shows a high-level reliability model for task execution by means of an extended continuous-time Markov model. A task is defined by its runtime which is not handled by an ordinary Markov process. The presented extension allows the prediction of task termination which also includes the time for further fault recovery measures like repetition. Beside the task execution time (= row), this extended Markov model in form of a matrix identifies also the redundancy (= column) within the task. However, the adding of redundancy usually leads to more runtime which is not considered in detail. In case of a coded task, the coded operations require more runtime and also the verification of the result at the end needs additional time which must be included in the model. Depending on the redundancy, the task system is tolerant for a certain number of faults where each fault is represented by a single column in the matrix. So, it is possible to model a given task by the parameters *execution time*, *redundancy* and the *fault probability*. The fault probability is defined by the underlying hardware and it specifies the rates for new faults (= transition to the right column). But, the presented model does not show any details of the task implementation. Each single vertical stage corresponds to a certain time interval within the task. With a constant time which is required for a single instruction, a single stage can be mapped to an instruction of the task as it is shown in Figure 4.13 for example.

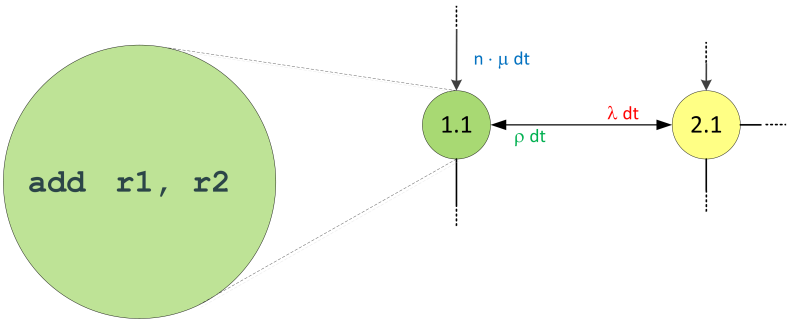

Figure 4.13: Mapping of a single instruction to a stage of the enhanced Markov model.

In the presented paper, a generic fault rate is assumed for the complete task. But this model allows different fault rates at each stage. So it is also possible to match the varying fault vulnerability of each instruction of the task.

Furthermore, we introduced the concept of fault compensation with several consecutively faulty operations in Chapter 4.2.2. In the context of the presented Markov model, compensation means a transition back to the previous column in the model, which decreases the number of active faults in the task. As we have seen in the Publications A.6

and A.7, this compensation depends on the number of previously executed instructions. Therefore, the probability of fault compensation increases with ongoing task execution. To meet the effect of compensation, there are additional transitions to specify with rate ρ in Figure 4.13.

Finally, the presented Markov model forms the basic principle of a reliability model which can be enhanced by more detailed models for single instructions (Chapter 4.2.1) or other effects as fault compensation (4.2.2). With the aspect of execution time, this Markov model can also be used as an initial point for further work e.g. scheduling analysis.

4.2.3.3 Further Work

The following text is not part of A.3 and describes a still unpublished aspect related to the presented Markov model of previous chapter. Assuming a recovery strategy of task repetition in case of a detected fault, the consequence is the increased runtime until the final correct result is available and the potential risk of violating the deadline of the task. For the reliability of safety-critical embedded real-time systems, the fulfillment of all deadlines is as important as the correct results at the end. The absorbing states of the previously introduced Markov model represent the successful or failed termination of a task. Without a repetition, the task will terminate after one single execution cycle. But in case of a detected fault, the repetition causes additional runtime and the termination is delayed. This behavior manifests itself in the time-based probability of the absorbing states 1.n and 3.n in Figure 4.14.

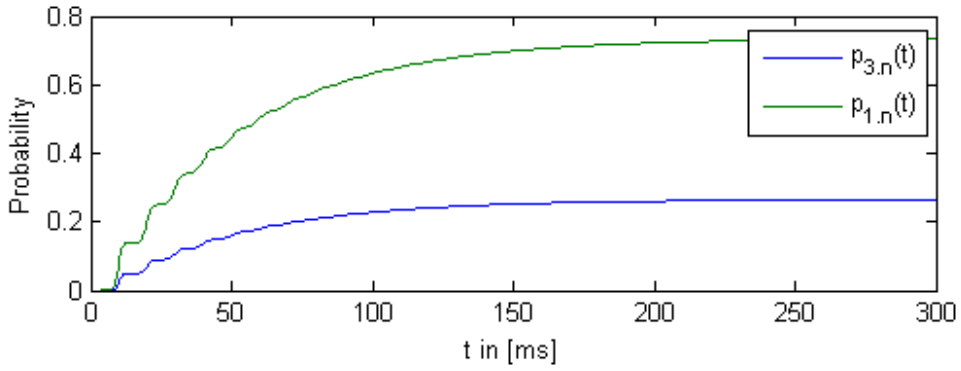

Figure 4.14: Delayed termination of a fault-tolerant task system. The steady-state probability is approached step by step with each task repetition ($t_{task} = 10ms$ and $\lambda = 0.1f/ms$).

In Figure 4.14, the repetition causes steps in the probability of the absorbing states. With each multiple of the task runtime, the probability of entering an absorbing state is

increasing until the sum of both probabilities converges to unity. For better visualization, the fault rate is dramatically increased to an unrealistic value. Otherwise, these steps cannot be observed and the steady-state probabilities are reached much faster. Note that the steady-state probabilities in Figure 4.14 have the same values as the probabilities of the availability analysis derived by the non-stop execution approach in A.3

The density function describes the normalized probability of a random variable. In this case, the interesting random variable is the total execution time of the task whose time-based probability is the sum of both absorbing states. The density function is the first derivative with respect to the time with

$$f(t) = \frac{d\big(p_{1.n}(t) + dp_{3.n}(t)\big)}{dt}. \tag{4.6}$$

The graphic chart of Equation 4.6 can be seen in Figure 4.15. It shows several decreasing peaks at multiples of the task execution time. This means that is becomes more and more unlikely for the task to be repeated multiple times.

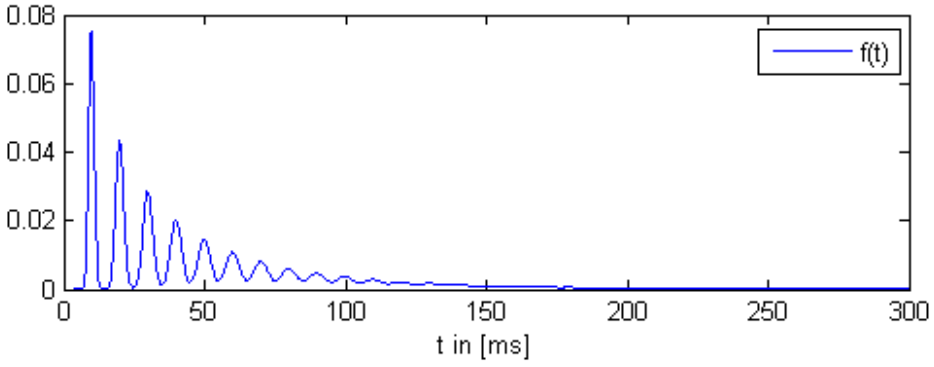

Figure 4.15: Density function of total task runtime including all repetitions (same assumed parameters like fault rate as in Figure 4.14).

The expected value is an important metric in probability theory and it describes the average value of the random variable with

$$E = \int_{-\infty}^{+\infty} t \cdot f(t)\, dt. \tag{4.7}$$

For Figure 4.15, the expected value is the average time until the task terminates either with success or with undetected fault including all repetitions. And of course, the expected value with $E \approx 50ms$ is very high in this example which means an execution of five task cycles. But it illustrates the repetition of a task with each peak in Figure 4.15 by a

detected fault. Otherwise with small p, the probability of a repetition is small and the peaks except of the first one would disappear. The repetition of a task in case of an error increases the effective runtime of this task. A task set is scheduled by an operating system in way that all deadlines are guaranteed. But now, we have a non-deterministic influence by arbitrary hardware faults and the repetition of a task could lead to a deadline violation which is not acceptable in a hard real-timed safety-critical application. Then the probability of a deadline violation is the probability that the task runtime is longer than the deadline d with

$$P_{deadline} = \int_{d}^{+\infty} f(t)\ dt. \tag{4.8}$$

This probability is important for further works considering timing constraints of safe task schedules in operating systems.

4.3 Alternative Codes for Coded Data Processing

The content of this section is derived from the submitted publication in A.7.

> P. Raab, S. Krämer and J. Mottok. Reliability of Data Processing and Fault Compensation in Unreliable Arithmetic Processors. ***Submitted*** in *Microprocessors and Microsystems*, September 2013.

This paper extends the concept of fault compensation from 4.2.2 and it investigates the usage of linear codes for coded data processing as an alternative.

4.3.1 Summary

Error detecting and correcting codes are widely used in data transmission and storage systems to protect data against arbitrary faults. As mentioned in the introduction, there is a similar influence on data processing in arithmetic units, as well. The logical circuits for arithmetic operations represent noisy channels and they randomly inject faults during the data processing. The coded processing approach usually uses arithmetic error codes (AN-codes) to protect data. However, the investigations in Publications A.2 and A.7 show that linear codes have a better error detection capability in combination with BSC-based channels. Thus, there is the question if linear codes are applicable for coded data processing, too. But, there is a crucial drawback when using linear codes for coded data processing especially for arithmetic operations like the addition. In contrast to arithmetic codes which preserve the code in the result, linear codes are based on polynomials and

the ring of polynomials has the property of closure under the polynomials operations only. The polynomial addition $\oplus$ differs from the ordinary integer addition $+$ by the missing carry-bit propagation. To keep the correct result, the missing carry-bits must be considered by an additional $\oplus$ operation with

$$y(z) = x_1(z) \oplus x_2(z) \oplus c(z). \tag{4.9}$$

Equation 4.9 describes the addition of two integer numbers $x_1(z)$ and $x_2(z)$ in a polynomial representation and the polynomial addition $\oplus$. $c(z)$ is the polynomial of the carry-bits created by an ordinary addition. Without detailed consideration of these carry-bits, the addition of two integer numbers can be replaced by two polynomial additions. The polynomial addition corresponds to the *XOR* operation in an arithmetic processor. Figure 4.16 shows the comparison wherein the residual error probability of a single addition (from A.5) is compared to that of two consecutively executed *XOR* operations.

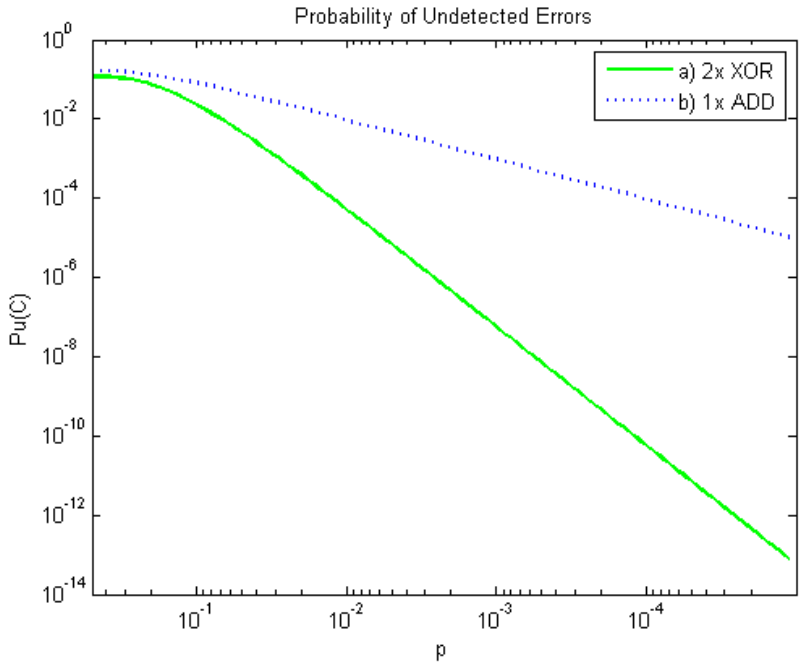

Figure 4.16: Residual error probability of the linear code ($g(z) = z^3 + z + 1$) compared with the arithmetic code ($A = 3$). Although, there are more instructions for the linear code necessary, the residual error probability of the arithmetic code is worse.

Both codes are similar related to their code rate. This means that they need the same amount of memory for information and parity. Furthermore, Figure 4.16 shows the comparison of both alternatives. First, there is the residual error probability of a single addition derived by the presented Markov chain of Chapter 4.2.1. Second, the replace-

ment of the addition by Equation 4.9 leads to a simple linear data flow which can be evaluated with respect to the reliability (see Publication A.7). The comparison shows a better error performance of linear codes in spite of the higher runtime. There are twice as much instructions using the *XOR* instructions. But keep in mind that this is only valid without any consideration of the carry-bits in order to replace the ordinary addition by Equation 4.9.

4.3.2 Discussion

The investigation of linear codes for coded data processing on arithmetic processors is one goal of this work for evaluating alternative codes (see Objective 2). The idea to use linear codes for coded data processing is also derived by the results of the Publication A.2. Here, we compared the performance of linear codes and arithmetic codes. It was shown that linear codes have a better residual error probability than arithmetic codes provided of using a BSC-based error model. But for the ordinary addition, a further corrective operation is required to replace the integer addition by the polynomial addition. This additional operation is caused by the different underlying algebra between arithmetic codes and linear codes. Instead of only one operation (*ADD*), the execution of two consecutive operations (*XOR*) results in more unreliability. But using error codes, the situation reverses and two coded *XOR* instructions have a better residual error probability than one coded *ADD* (Figure 4.16). However, this evaluation does not consider the generation of the corrective carry-bits, which introduce more unreliability into the system. It is expected that the additional unreliability will reduce this advantage of linear codes in future investigations.

But nevertheless, this analysis takes into account of a simple data flow consisting of two instructions which is used to verify the effect of fault compensation introduced in Publication A.6. In A.6, there is the possible effect of fault compensation described within a set of consecutively executed instructions. The study of compensating faults in a linear data flow provides the basics to analyze the reliability. And thus, this compensation effectively increases the reliability and it also has an influence on the calculation of the residual error probability of linear codes in this example. But at the end, the usage of linear codes is not promising to be applicable with arithmetic operations. On the other side, linear codes have advantages for data storage and transmission (see A.2). In an arithmetic processor system, there are memories and buses that can be safeguarded by linear codes. In contrast, the class of arithmetic codes is the better choice for arithmetic operations. With different codes used in a processor system, the requirement for code transformations arises (see Chapter 4.4).

4.4 Code Transformation

The content of this chapter is derived from the publication in A.4.

P. Raab, V. Vavricka, S. Krämer and J. Mottok. Isomorphism between Linear Codes and Arithmetic Codes. **Accepted** in *Computing and Informatics (CAI)*, February 2013.

This paper summarizes a possible code transformation between linear and arithmetic codes as a requirement which is derived from Chapters 4.1 and 4.3.

4.4.1 Summary

In Chapter 4.1, a processor system was separated into a set of channels which have different characteristics with respect to their error behavior. Using error codes which match the characteristic of the underlying channel, the error detection capability can be optimized by choosing the optimal code. E.g. linear codes have a better residual error probability with BSC-based channels than comparable arithmetic codes (see Chapter 4.1). In contrast, arithmetic codes are preferred for arithmetic operations (see Chapter 4.3). Thus, the requirement for code transformation becomes important to optimize the error detection capability within a given data flow when using both codes. However, there is an important constraint for this purpose. The code words of both codes must represent the same data and there must be a transformation rule between both code words (Figure 4.17).

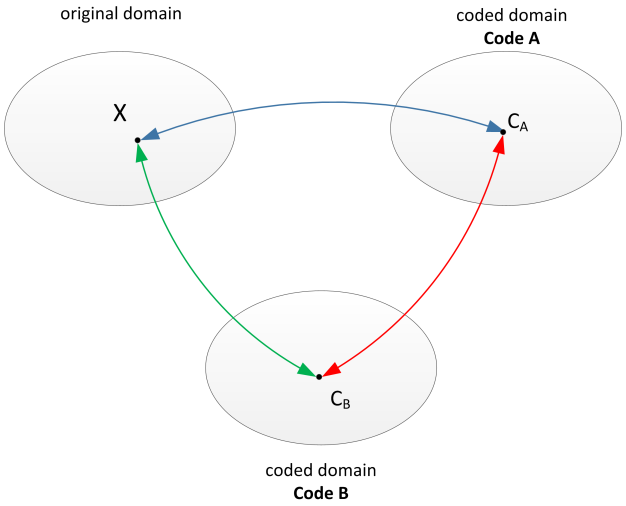

Figure 4.17: The information X can be represented by different code words C_A/C_B. Without the long way via X, there is also the possibility of direct transformation between $C_A \leftrightarrow C_B$.

A detailed view to the generation of linear and arithmetic codes shows that both codes are quite similar. The code words of both codes are the product of a constant and the data to be encoded. Whereas linear codes are based on polynomials and use the polynomial multiplication

$$c(z) = g(z) \odot x_2(z) = x_1(z) \odot z^{n-k} \oplus r(z), \tag{4.10}$$

arithmetic codes require the integer multiplication

$$C = A \cdot X_2 = X_1 \cdot 2^{n-k} + R. \tag{4.11}$$

Furthermore, there is the possibility of systematic encoding for both codes. The property of systematic codes is the separation of information and parity. This means that the information part $X_1/x_1(z)$ is shifted to the left in the binary representation by $n-k$ bits and the remaining bits are filled by the parity R or $r(z)$. This makes the decoding very simple because the original information remains part of the code word.

Let us now consider the systematic form of arithmetic codes. With $X_2 = X_1 + 1$, the Equation 4.11 can be transformed to

$$R = A - X_1 \cdot (2^{n-k} - A) \tag{4.12}$$

and with generator $A = 2^{n-k} - 1$, the remainder R of Equation 4.12 can be further simplified to

$$R = A - X_1. \tag{4.13}$$

This means that the remainder of the systematic arithmetic code is a decreasing number beginning with A and with X_1 is the original number to encode (see Table 4.1).

NOTE:
An integer of the form $A = 2^x - 1$ is a so-called *Mersenne* number. *Mersenne* numbers have the property that the binary representation only contains ones. This is an interesting property in combination with arithmetic operations like the subtraction or addition.

The goal is now to replace the remainder R of the arithmetic code in Equation 4.12 by the remainder $r(z)$ of linear codes. But, the algebra of both codes are different and the operations cannot be substitute without consideration of the carry-bits (or borrow-bits) of the arithmetic operations $+,-$ and $\cdot$. See Publication A.2 for detailed information about the algebra of linear and arithmetic codes. Without mentioning the details of the

derivation in A.4, the choice of a generator $A = 2^x - 1$ leads to some simplifications concerning these carry/borrow-bits. The Equation 4.10 can be replaced by

$$g(z) \odot x_2(z) = x_1(z) \odot z^{n-k} \oplus a(z) \odot \underbrace{\{1 \oplus x_{1_0}\}}_{=1, if\ x_{1_0}=0} \oplus r(z). \tag{4.14}$$

The term $\{1 \oplus x_{1_0}\}$ in Equation 4.14 means an inversion of the residue $r(z)$ by $a(z)$ only if the least significant bit x_{1_0} of $x_1(z)$ is zero. Remember, $a(z)$ is the polynomial representation of $A = 2^x - 1$ with all coefficients are one. The result is that every second remainder of the linear code is bitwise inverted compared to the arithmetic code word. Table 4.1 summarizes this relation between the remainders of a linear and an arithmetic code.

	X_1	R	$r(z)$
1	00001	11110	11110
2	00010	11101	00010
3	00011	11100	11100
4	00100	11011	00100
5	00101	11010	11010
6	00110	11001	00110
...	...	...	...

Table 4.1: Isomorphism between an arithmetic and linear (15,10) code with $A = 31 = 2^5 - 1$ and $g(z) = a(z) \odot z = z^5 + z^4 + z^3 + z^2 + z$. With both codes, the information is extended by $n - k = 5$ bits. Every residue $r(z)$ is inverted compared to R if the least significant bit of X_1 is one.

4.4.2 Discussion

With the presented transformation rule, it is now possible to transform linear codes to arithmetic codes and vice versa. But the generators of both codes must have a certain form which leads to a quite bad error detection capability. For example, a polynomial of the form $g(z) = \sum_{i=1}^{n-k} z^i$ generates a code with a minimum Hamming distance of two only. This means a detection of only one fault. But with the same code rate, there are more efficient codes available compared to that one. However, the arithmetic codes, which are derived by the described generator A, have the advantage of a very simple verification which avoids the runtime consuming modulo operation. The evaluation for error detection can be simplified to an addition instead of a modulo operation. Please, see Publications A.4 or [Rao74] for detailed information about this procedure.

The drawback of bad error detection capability can be compensated by other additional redundancy techniques. In Chapter 4.5.1, we will see the approach of concurrent task execution by an operating system which means time redundancy. A pure time redundant solution is able to detect a single transient fault in one copy by the deviation in the outcomes of both task instances (see Chapter 2.1). But, it is not possible to distinguish which copy is correct and the fault cannot be restored. Compared to that, the duplication of a task corresponds to a 1-fault detection system. Together with the discussed 1-fault detecting codes, each copy can be verified for correctness. The corrupted one has been found and the error can be corrected.

In Publication A.2, we described a processor system which consists of several different channels. There are parts which store data (memory) or transmit data (bus). And there are other parts that process data. But these arithmetic instructions have other characteristics related to the error behavior compared to the previously mentioned one. We also showed in A.2 that linear codes have a better error detection capability than comparable arithmetic codes given that a BSC-based model is assumed. This means that for a lot of parts in a processor system, the usage of linear codes are applicable for e.g. memories or MOV / XOR operations. In contrast to that, linear codes are not practical for other arithmetic operations (ADD operation) because of the missing carry-bit propagation as we have seen in the previous Chapter 4.3. Thus, the presented code transformation is required to take all these advantages for coded data processing. The total residual error probability of a given data flow is expected to be better using code transformations wherein the transformation itself must be considered for reliability analysis, as well. The combination of the advantages of different optimal codes, the transformation between them and the described time redundant approach promise quite good error detection capability in future applications.

4.5 Validation

The crucial contribution of this work is the analytical reliability evaluation of a given task system beside state-of-the-art experimental methods (e.g. injection techniques, Chapter 2.3.2). But, the correctness of the previously presented mathematical models is a critical issue. Therefore, the validation of the models is a further goal of this thesis (see Objective 4). In the further course of this chapter, two different approaches for validation are presented.

4.5.1 Concurrent Task Processing by an Operating System

The content of this section is derived from the publication in A.1.

> P. Raab, S. Krämer, J. Mottok, H. Meier and S. Racek. Safe Software Processing by Concurrent Execution in a Real-time Operating System. In *Proceedings of 2011 International Conference on Applied Electronics (AE)*, pages 315-319, September 2011.

This paper shows a possible evaluation platform which is based on a real-time operating system.

4.5.1.1 Summary

Redundancy is always required for fault-tolerant systems to increase the availability. Time redundancy is a special type of duplication which does not copy the component, but it executes the component several times. Thus, arbitrary transient faults are detectable, if the fault duration is shorter than the repetition cycle of the task. In case of the data flow in computer systems, the duplicated execution can be realized by existing means of a *real-time operating system* (= RTOS). Therefore, the proprietary R^{3}TOS [8, 9] is enhanced to schedule the two instances (*leading* and *trailing*) of a single safety task whereas the same program code is executed but with different task contexts. This is realized by the duplication of the task control block. Figure 4.18 shows the simplified architecture of this enhanced operating system.

The so-called *Safety Supervisor* is the main component that especially controls the synchronization of both task instances. It is comparable with the voter in a standard dual modular redundant (DMR) system. At defined points in the code, the data and also the program counter of both instances are compared. So, it is possible to detect deviations in the data flow and in the program flow (program flow monitoring) which are maybe corrupted by arbitrary hardware faults. When detected, the Safety Supervisor starts a proper recovery action. For example, this can be *backward recovery* which resets both instances back to the last safe state or even to the start. Or in the other case, the wrong instance is updated by the correct one (= *forward recovery*). However, this is only possible if each instance is able to detect the fault by itself by means of further fault detection techniques like the coded data processing approach. In contrast without coded processing, a simple re-execution of the task only leads to deviations in the outcomes which are only detected by the supervisor. But, it is not possible to determine the correct instance.

For a better usability, the coded operations are encapsulated and implemented in a separated library which is accessed by the OS via a standardized API. The encapsulation

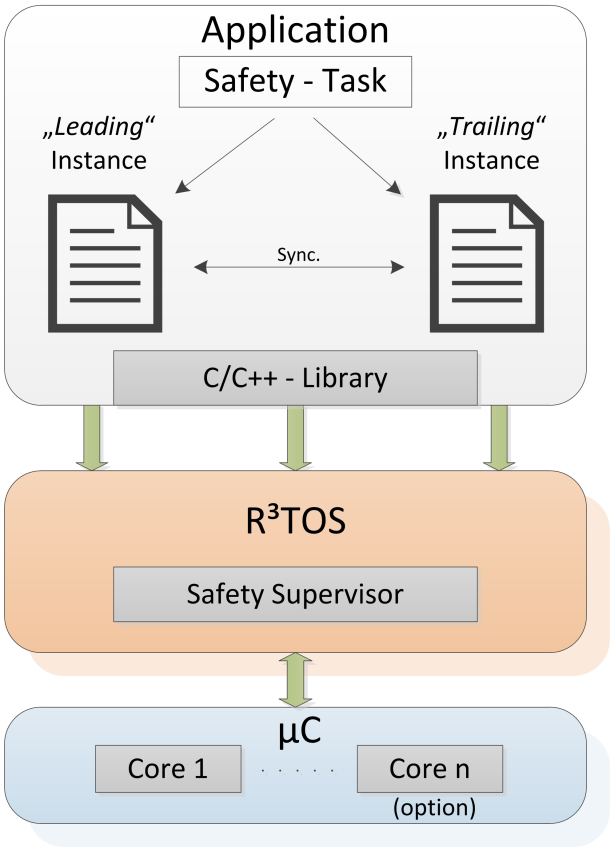

Figure 4.18: Simplified architecture of R^{3}TOS with the so-called safety-supervisor which controls the concurrent execution of two task instances.

allows a simple change of the code without changing the whole system and a fast evaluation and comparison of different codes are possible, as well.

There are also some drawbacks in this approach. Only the stack and the execution are duplicated by the operating system. The program memory is the same in both instances and it must be protected by other techniques like ECC. In general, a task has also global variables which are not located in the stack. To avoid concurrent access and for further reliability, these data must be replicated, too. But all these duplications lead to higher runtime and memory consumption by a factor of at least two. So, each new instance requires the same amount of memory (stack and global variables) and runtime as the original task if the task is uncoded. In case of coded data processing, there is additional runtime and memory to consider because of the coded data and the coded operations. First experimental results showed that a slow-down factor of about 44 must be expected compared to a single uncoded task. The extension of this evaluation platform to multi-core processors would reduce the part of the slow-down factor caused by the duplication. Each of the two task instances are executed on different cores. Instead of pure time

redundancy, there is also space redundancy and the detection of permanent faults would be possible, as well. For highly critical applications, it is then possible to remain operable also in case of one failed core. See the publications [4, 5] for further researches in the LaS^3 about scheduling in multi-core processors.

4.5.1.2 Discussion

The presented operating system in Publication A.1 can be considered as an evaluation platform which especially matches the presented Markov model in Chapter 4.2.3. The time redundant execution of the same task allows the safety-supervisor to detect a single fault in any instance and the recovery measure is the restart of the task. This is represented by the first two columns in the Markov model of Figure 4.12 and by the transition back to the initial state 1.0 from the state 2.(n-1). All intermediate states 2.x could stand for additional synchronization points during the runtime. A further fault in the other instance is detected only if both instances produce different outputs. This behavior is shown in the third column by splitting the transition to the initial state and to the termination state with undetected faults (3.n). Assuming further instances, there would be additional columns in the model for each additional task instance controlled by the enhanced operating system.

The recovery by repeated task execution requires more runtime of the task which is described by a further cycle in the presented Markov model. But this extra runtime probably violates existing deadlines that must be met for safety-critical applications. One solution could be more synchronization points to reduce the latency of fault detection. But the risk for deadline violations is still present and the probability evaluation of such an event as discussed in Chapter 4.2.3.3 is an important aspect for future works. However, the discussed approach is a stand-alone platform and cannot validate the model from Publication A.3. The experimental determination of reliability parameters like MTTF requires the arbitrary injection of faults during runtime. Therefore, the enhanced operating system in combination with a fault injection platform [FMB$^+$12, FM12] is one of the future challenges. With respect to the coded processing approach, the presented operating system is the only way to evaluate the increased runtime in case of coded tasks at the moment. This extra runtime is still ignored in the data flow model of Publication A.3.

Furthermore, the effective reliability is a function of redundancy which can be configured by two parameters in this approach. Firstly, the number of instances increases the fault tolerance by time redundancy and directly determines the number of tolerated faults. Secondly, each instance can also realize a coded data flow which explicitly detects faults by information redundancy. With respect to any risk assessment, e.g. for functional

safety (IEC61508 [IEC10a] / ISO 26262 [ISO11]), the *safety integrity level* (SIL) requires a certain reliability or residual error probability. A mapping function (Figure 4.19) would be very helpful to configure the system depending on the required safety class.

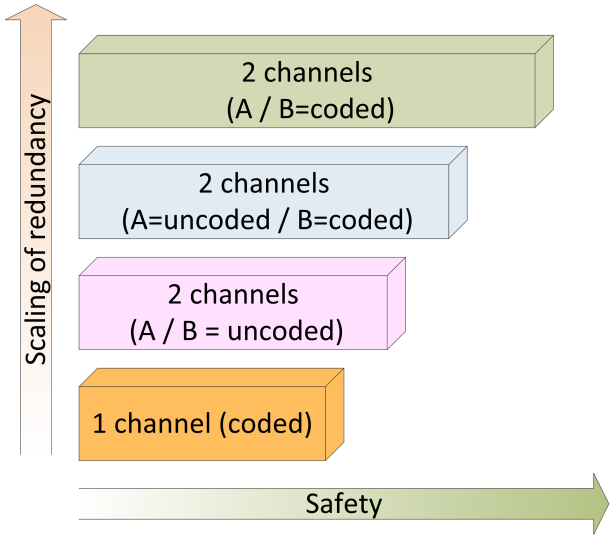

Figure 4.19: Redundancy as a function of safety. The required safety is realized by a certain configuration of redundancies.

A further problem is here whether different types of redundancy can be equally transformed to each other. Or in other words, how many tasks instances match the same fault detection as a single coded task. The idea is to create a metric which allows the comparison of different forms of redundancies.

4.5.2 Simulation Approach

The content of this section is partly derived from the publication in A.5. The already introduced paper in Chapter 4.2.1 also describes the basic principle of a simulation based verification of the presented error model.

P. Raab, S. Krämer and J. Mottok. Error Model and the Reliability of Arithmetic Operations In *IEEE 2013 EUROCON - International Conference on Computer as a Tool*, pages 630-637, July 2013.

4.5.2.1 Summary

The presented simulation approach in this paper is used for verification purposes of the discussed error model for a ripple-carry-adder (Chapter 4.2.1). The crucial part of such a simulator is the emulation of a 1-bit adder as described in A.5. This adder uniquely defines the outputs s and c_{out} as a function of the operands x, y and the carry c_{in} from the previous adder following Equations 4.15 and 4.16.

$$s_i = x_i \oplus y_i \oplus c_{i-1} \tag{4.15}$$

$$c_{i+1} = (x_i \wedge y_i) \vee (x_i \wedge c_i) \vee (y_i \wedge c_i) \tag{4.16}$$

The main task for the simulator is the random manipulation of the adder´s output according the defined distribution. This means that in case of a fault, the correct outputs are changed to any other values (see Figure 4.20).

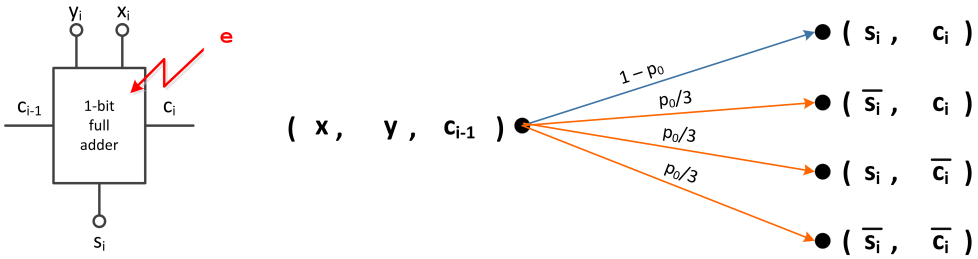

Figure 4.20: Manipulation of the outputs in a 1-bit adder by the simulator following a defined probability distribution.

The 1-bit adder in Figure 4.20 is part of the iterative process to compute the sum of two n-bit integer numbers. Thereby, each bit stage deploys the described fault mechanism independent of each other adder stage. But corrupted carry-bits propagate the error to the next stage of the adder and they probably alter the sum bit of the next stage.

In contrast to the assumed equal distribution of a fault within a single 1-bit adder, the occurrence of this fault is actually defined by the underlying hardware. In general, the exponential distribution describes the fault probability in electronic components based on a constant fault rate. Thus, a further part of the fault simulation is the generation of random numbers following the exponential distribution. This number represents the time till the fault arises and changes the output of the current stage in processing the sum. For this purpose, the existing GNU scientific library [FSF11], which offers the possibility of generating random numbers, is used for the described simulator. But, each stage of the adder requires its own random generator to fulfill the independence between the bit

stages and so, multiple faults within one addition are possible. The simulation of multiple fault is required to verify the effect of fault compensation according Chapter 4.2.2.

The investigation of the relative frequency of certain error masks of a result or the probability of undetected errors requires the execution of the simulated adder many times. With increasing number of additions with equally distributed operands, the relative frequency of undetection approximates the residual error probability (Figure 4.21).

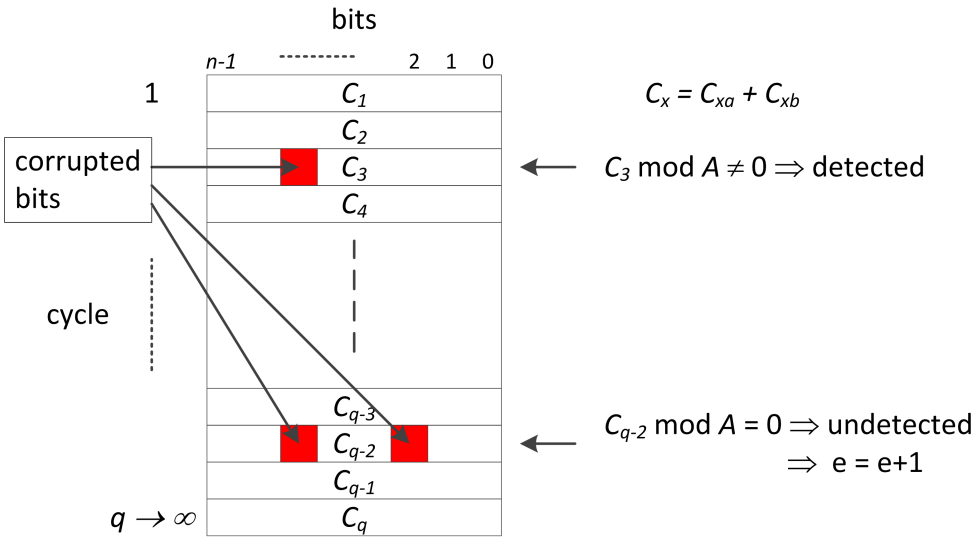

Figure 4.21: Simulation of the residual error probability by permanent additions and arbitrary injections of single faults.

With e is the number of erroneous additions in which the modulo check wrongly passes and q is the total number of additions, the residual error probability is approximated by

$$P_{u,sim} = \lim_{q \to \infty} \frac{e}{q} \tag{4.17}$$

when q is further increased.

4.5.2.2 Discussion

The crucial contribution of this thesis is the development of an error model for an arithmetic operation. This model can be considered as a starting point for further model-based reliability analysis of faulty operations and finally of a complete data flow. However, the correctness of the error model is critical for the results of the evaluation. Therefore, the introduced error model is validated by the presented simulation approach in this case. The described simulator emulates the iteration process of a ripple-carry-adder and deploys the same fault distribution as assumed in the error model. For a usable result of

the simulator, a lot of simulation runs are required to approximate the probability by the relative frequency of certain events. So, the simulation approach comes to the same result as the analytical calculation based on the error model. Thereby, the correctness is proofed.

The critical issue of the presented simulator is the underlying distribution. But also for the presented error model, both approaches assume equally distributed corrupted outcomes of a single 1-bit adder. In reality, this distribution depends on the actual realization of the adder which usually knows the manufacturer only. However, the principle concept of the error model does not change with different distributions (see also discussion in Chapter 4.2.1.2) and both, the model and the simulator can still be used.

Furthermore, the architectural concept of the developed simulator is based on a virtual class for generic operations. The example of a ripple-carry-adder is a derived specialized class. So, it is possible in a simple way to create simulation models for further operations and small data flows can be validated, as well. The presented simulator was also used to verify the results of the linear data flow in Chapter 4.3. Figure 4.22 shows the basic architecture of the used simulator by means of a simplified class diagram.

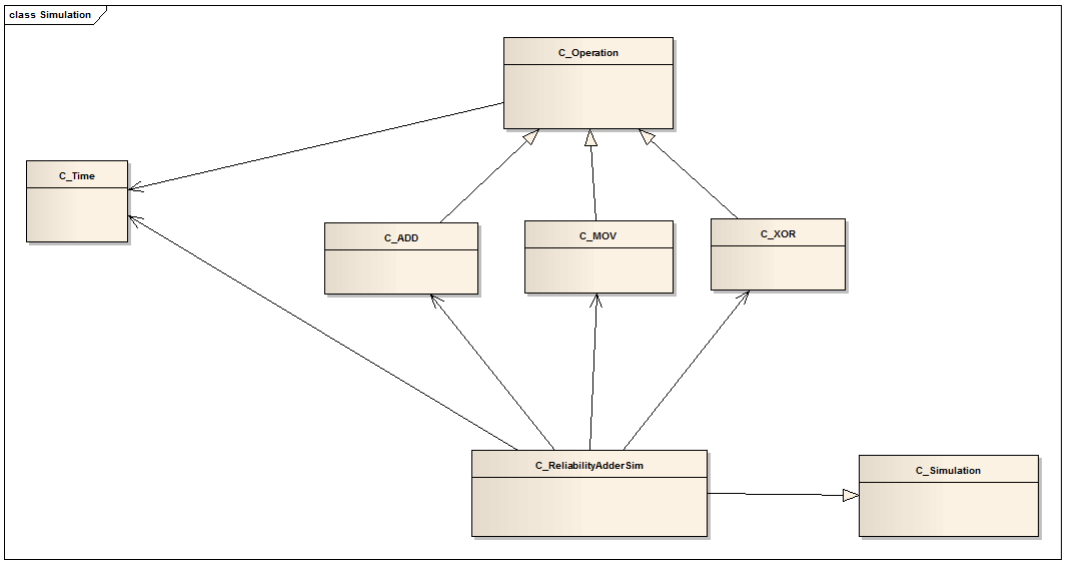

Figure 4.22: Simplified class diagram of the presented simulator.

Chapter 5

Conclusion

The system reliability becomes an important weight in modern applications, especially for safety-critical systems. Random hardware faults affect the data processing and finally the total system reliability, especially in the future with more software intensive systems [Boe06]. The study of the effects of hardware faults with respect to the software behavior is an interesting field of research. The *coded processing* approach based on Forin´s Vital Coded Monoprocessor [For89] is a state-of-the-art technique to increase the hardware fault detection in data processing units more effectively than other redundancy techniques like pure duplication of runtime or data (see Chapter 2.2). The efficiency of error detection codes in arithmetic processor is an often discussed topic which is usually evaluated by experimental fault injection methods. Formal methods are rarely described and comparable error models known from transmission systems are not available. The reason for this lies in the complexity of software with all dependencies between variables and the variety of different combinations in a given set of instructions. This work contributes the challenge to evaluate the reliability of a given data flow in an analytical way to estimate the reliability in a very early stage in the development process as soon as the first draft of a software is available.

5.1 Discussion of the Goals

This PhD thesis pursues the main goal to provide basic concepts for software reliability evaluation of hardware-based faults. Instead of a simulation, an analytical approach has generally the advantage of less computational effort. But, a detailed knowledge about the faulty behavior of arithmetic instructions is required in this case. Therefore, this thesis describes faulty data processing by several error models. Starting with the smallest unit of software - the instruction - the model is extended by dependencies between instructions

and variables. Then, the reliability of a complete software data flow can be modeled with respect to its error susceptibility. Referring to the defined objectives in Chapter 3, it follows the discussion about the achievements of these goals:

Objective 1: **Development of an error model for arithmetic operations**
The estimation of the probability of undetected faults is an important aspect of error detection codes. Therefore, the knowledge of the faulty behavior of instructions is essential for evaluating coded data processing, as well. In the Publication A.2, we discussed a processor system as a composition of several components which all influence the outcome of a given data flow. Beginning with the storage and transmission (= MOV instruction) of data, the processing within the arithmetic unit is the main focus of this objective. The Publication A.5 covers this objective by describing an error model of a single ADD instruction) instruction (ripple-carry adder) which is based on a discrete Markov process. This model can be considered as the basic principle for more models of instructions in a processor system. A further error model is described in A.7. The XOR operation is based on the BSC model and does not have functional dependencies between the bits compared to the ADD instruction.

Objective 2: **The comparison of the residual error probability of different codes**
The residual error probability depends on the used coding technique. Arithmetic codes are commonly used for coded operations, but there are a lot of further error detection codes available. In the Publication A.2, we compared the efficiency of linear codes and arithmetic codes related to their residual error probability based on data storage. In contrast, the usage of linear codes for an arithmetic operation like the addition was presented in A.7. The publications showed that there is an optimal code for a given underlying hardware. And with different channels, multiple codes are required in a processor system. Therefore in A.4, we presented a possible transformation between linear and arithmetic codes for a concurrent usage in a processor system.

Objective 3: **Reliability evaluation of a task´s data flow**
The error model of a single instruction is not enough for the evaluation of a complete data flow. With multiple faulty instructions, there is an additional possible effect of fault compensation between consecutively executed instructions which was presented in Publication A.6. A further abstraction regarding the erroneous behavior of a task was done in Publication A.3. The erroneous data flow of a given task is reduced to the parameters *fault rate, runtime and redundancy.* The effective fault rate and also the compensation rate can be derived from the previously mentioned error models. But only with the presented model in A.3, it is possible to consider aspects as runtime and repetitions for further analysis of scheduling issues.

Objective 4: Validation of derived models
The correctness of the presented error models is important for their future usage. Therefore, the validation is another part of this thesis. When the error model of the ripple-carry-adder has been introduced in Publication A.5, it was additionally verified by a simulation approach. Furthermore, the publication A.1 presents the realization of concurrent tasks which is covered by the runtime model in Publication A.3. In combination with fault injection techniques, the enhanced operation system of A.1 can be used for the validation of A.3.

The results of this thesis can be regarded as an initial point for software reliability evaluation based on hardware faults. The crucial contribution of this works is the development of error models for arithmetic instructions in a processor system. The advantage of such models is the possibility for evaluating the task reliability during the design phase of the software development. First predictions of the reliability are available before the software can be tested in a real environment and necessary changes can be done earlier to save costs during the software development process [Boe06]. Currently, the final software is necessary to determine the reliability of software processing by experimental methods. For future purpose, the set of models must be extended related to a given set of instructions. But with this set of error models as a plug-in of a compiler, the reliability evaluation is possible during the software build process and the compiler outputs the reliability model and parameters of the given software.

5.2 Further Work

In spite of the achievement of the defined goals and the publication of the result, there are still a lot of questions and open issues to be discussed in future works. This section summarizes all ideas to extend, refine and improve the presented models:

- **Program flow**:
  This thesis only presents and discusses the erroneous behavior of arithmetic operations that influence the data flow of given software. In contrast, faults in branch instructions directly affect the program flow. But also errors in the data flow have an impact on the program flow when they occur in the condition of control statements. To close this gap, the set of error models must be extended also for branch and jump instructions of a given instruction set in future works.

- **Further alternative codes**:
  The analyzed error codes in this work were restricted to linear and arithmetic codes only. But maybe, there are further usable error codes for coded data processing. For example, so-called DFT (= Discrete Fourier Transformation) codes [Bla85, Mar84] are interesting codes which can be represented as digital filters and maybe deployed in MAC units of digital signal processors (DSP). This probably means the increase of fault tolerance by further hardware redundancy.

- **Case study**:
  The presented error models are not enough for the evaluation of a complete software task. The intention is to choose a task from a real application and to model the data flow. This demands the detailed knowledge of the erroneous behavior of each instruction. The reliability and residual error probability out of the evaluation can be validated by traditional fault injection techniques, which is a further object of research in our laboratory (LaS^3). The entire reliability model can be derived from the Markov model of Chapter 4.2.3 wherein a single node could represent a single instruction in the code and the transition rates are outputs of the corresponding error model. Thus, the code of a task can be directly mapped into the stages of a Markov model.

- **Comparison between linear and arithmetic codes**:
  In A.2, we assumed that linear codes are more efficient than arithmetic codes provided that both have the same code rate and the underlying channel is based on the BSC model. But the proof for this statement is missing. The minimum Hamming distance of a given arithmetic code is maximal the minimum Hamming distance of an equivalent linear code with $d_{H,AN} \leq d_{H,CRC}$.

- **Effective reliability caused by fault compensation**:
  In Publication A.7, the effect of fault compensation leads to a minimum reliability of the given data flow. From the view of reliability analysis, the compensation of faults corresponds to some kind of repairs. With repairable systems, there is the metric of availability as the steady-state value of the reliability. The idea is now to verify the results of Publication A.7 by standard methods using a Markov model which also contains repair rates. Further, the steady-state reliability of Publication A.7 is based on a simple *XOR* instruction without any memory effect. But, how is the steady-state reliability of a more complex instructions like the *ADD*?

Bibliography

[AAN00] L. Anghel, D. Alexandrescu, and M. Nicolaidis. Evaluation of a soft error tolerance technique based on time and/or space redundancy. In *Integrated Circuits and Systems Design, 2000. Proceedings. 13th Symposium on*, pages 237–242, 2000.

[ACK+03] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G.H. Leber. Comparison of physical and software-implemented fault injection techniques. *Computers, IEEE Transactions on*, 52(9):1115–1133, 2003.

[AK88] P.E. Ammann and J.C. Knight. Data diversity: an approach to software fault tolerance. *Computers, IEEE Transactions on*, 37(4):418–425, April 1988.

[AL86] A. Avizienis and J.-C Laprie. Dependable computing: From concepts to design diversity. *Proceedings of the IEEE*, 74(5):629–638, 1986.

[ALR03] A. Avizienis, J.C. Laprie, and B. Randell. Fundamental Concepts of Dependability. Technical Report N01145, LAAS-CNRS, April 2003.

[ANKA99] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, and J.A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *Parallel and Distributed Systems, IEEE Transactions on*, 10(6):627–641, June 1999.

[Avi71] A. Avizienis. Arithmetic Error Codes: Cost and Effectiveness Studies for Application in Digital System Design. *Computers, IEEE Transactions on*, C-20(11):1322–1331, November 1971.

[Avi76] Algirdas Avižienis. Approaches to computer reliability: then and now. In *Proceedings of the June 7-10, 1976, national computer conference and exposition*, AFIPS '76, pages 401–411, New York, NY, USA, 1976. ACM.

[Avi78] A. Avizienis. Fault-tolerance: The survival attribute of digital systems. *Proceedings of the IEEE*, 66(10):1109–1125, October 1978.

[AYI+03] H. Ando, Y. Yoshida, A. Inoue, I. Sugiyama, T. Asakawa, K. Morita, T. Muta, T. Motokurumada, S. Okada, H. Yamashita, Y. Satsukawa, A. Konmoto, R. Yamashita, and H. Sugiyama. A 1.3-GHz fifth-generation SPARC64 microprocessor. *Solid-State Circuits, IEEE Journal of*, 38(11):1896–1905, November 2003.

[BA92] R. Billinton and R.N. Allan. *Reliability evaluation of engineering systems: concepts and techniques*. Plenum Press, 1992.

[Bau05] R. Baumann. Soft errors in advanced computer systems. *Design Test of Computers, IEEE*, 22(3):258–266, May-June 2005.

[BCPT00] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri. A C/C++ source-to-source compiler for dependable applications. In *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, pages 71–78, 2000.

[BDCDNP03] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto. A watchdog processor to detect data and control flow errors. In *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, pages 144–148, July 2003.

[Beu94] Albrecht Beutelspacher. *Lineare Algebra*. Vieweg, 1994.

[Bla85] R.E. Blahut. Algebraic fields, signal processing, and error control. *Proceedings of the IEEE*, 73(5):874–893, May 1985.

[BMSS05] C. Bolchini, A. Miele, F. Salice, and D. Sciuto. A model of soft error effects in generic IP processors. In *Defect and Fault Tolerance in VLSI Systems, 2005. DFT 2005. 20th IEEE International Symposium on*, pages 334–342, October 2005.

[Boe06] Barry Boehm. A view of 20th and 21st century software engineering. In *Proceedings of the 28th international conference on Software engineering*, ICSE '06, pages 12–29, New York, NY, USA, 2006. ACM.

[Bor05] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, November-December 2005.

[Bör08] J. Börcsök. *Funktionale Sicherheit: Grundzüge sicherheitstechnischer Systeme*. Hüthig, 2008.

[Bos98] M. Bossert. *Kanalcodierung*. Teubner, 1998.

[BP03] Alfredo Benso and Paolo Prinetto. *Fault injection techniques and tools for embedded systems reliability evaluation*, volume 23. Springer, 2003.

[BRC60] R.C. Bose and D.K. Ray-Chaudhuri. On a class of error correcting binary group codes. *Information and Control*, 3(1):68–79, 1960.

[Bro60] D.T. Brown. Error Detecting and Error Correcting Binary Codes for Arithmetic Operations. In *IRE Trans. Electron. Comput.*, pages 333–337. 1960.

[Bun12] Statistisches Bundesamt. Verkehr - Verkehrsunfälle, June 2012.

[CFA+11] John P. Christodouleas, Robert D. Forrest, Christopher G. Ainsley, Zelig Tochner, Stephen M. Hahn, and Eli Glatstein. Short-Term and Long-Term Health Risks of Nuclear-Power-Plant Accidents. *New England Journal of Medicine*, 364(24):2334–2341, 2011.

[Con03] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *Micro, IEEE*, 23(4):14–19, July-August 2003.

[DM03] P.E. Dodd and L.W. Massengill. Basic mechanisms and modeling of single-event upset in digital microelectronics. *Nuclear Science, IEEE Transactions on*, 50(3):583–602, June 2003.

[Don84] Hao Dong. Berger Codes for Detection of Unidirectional Errors. *Computers, IEEE Transactions on*, C-33(6):572–575, June 1984.

[ELL+92] Stephen G. Eick, Clive R. Loader, M. David Long, Lawrence G. Votta, and Scott Vander Wiel. Estimating software fault content before coding. In *Proceedings of the 14th international conference on Software engineering*, ICSE '92, pages 59–65, New York, NY, USA, 1992. ACM.

[Eng96] H. Engel. Data flow transformations to detect results which are corrupted by hardware faults. In *High-Assurance Systems Engineering Workshop, 1996. Proceedings., IEEE*, pages 279–285, October 1996.

[FM12] S. Felis, , and J. Mottok. Die fault bench for integrated incircuit injection zur verifikation von safely embedded software. In *Proceedings of the Embedded Software Engineering Congress*, December 2012.

[FMB+12] S. Felis, J Mottok, B. Bauer, D. Kohlert, D. Jantz, and M. Laumer. FBI^3 - Fehlereinspeisung auf Hardware-Ebene. In *3. Landshuter Symposium Mikrosystemtechnik*, March 2012.

[For89] P. Forin. Vital coded microprocessor principles and application for various transit systems. In *IFA-GCCT*, pages 79–84. 1989.

[For11] Mehr Software (im) Wagen: Informations- und Kommunikationstechnik (IKT) als Motor der Elektromobilität der Zukunft. Technical report, For-TISS GmbH, 2011.

[FP98] Norman E. Fenton and Shari Lawrence Pfleeger. *Software Metrics: A Rigorous and Practical Approach*. PWS Publishing Co., Boston, MA, USA, 2nd edition, 1998.

[FSF11] Inc Free Software Foundation. GSL - GNU Scientific Library, V1.15, April 2011. http://www.gnu.org/software/gsl/.

[FSS09] Christof Fetzer, Ute Schiffel, and Martin Süßkraut. AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware. In Bettina Buth, Gerd Rabe, and Till Seyfarth, editors, *Computer Safety, Reliability, and Security*, volume 5775 of *Lecture Notes in Computer Science*, pages 283–296. Springer Berlin / Heidelberg, 2009.

[GRRV06] Olga Goloubeva, Maurizia Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. *Software-Implemented Hardware Fault Tolerance*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[GRSRV03] Olga Goloubeva, Maurizio Rebaudengo, M Sonza Reorda, and Massimo Violante. Soft-error detection using control flow assertions. In *Defect and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on*, pages 581–588. IEEE, 2003.

[Ham50] R. W. Hamming. Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, (AFCRC-TN-57-103), April 1950.

[Hat00] L. Hatton. The challenges of complex it projects. *The British Computer Society*, 49(11):15–17, April 2000.

[HTI97] Mei-Chen Hsueh, T.K. Tsai, and R.K. Iyer. Fault injection techniques and tools. *Computer*, 30(4):75–82, 1997.

[IEC10a] IEC 61508-1. Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 1: General Requirements, April 2010.

[IEC10b] IEC 61508-4. Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 4: Definitions and abbreviations, April 2010.

[IEC10c] IEC 61508-7. Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 7: Overview of techniques and measures, April 2010.

[IEE90] IEEE Standard Glossary of Software Engineering Terminology, September 1990.

[ISO11] ISO 26262-5: Road vehicles - functional safety - Part 5: Product development: hardware level, November 2011.

[KH09] Ihor Kuz and Gernot Heiser. Fault Tolerance. Technical Report COMP9243, The University of New South Wales School of Computer Science and Engineering, 2009.

[Kir05] H Kirrmann. Fault Tolerant Computing in Industrial Automation. Technical Report HK050418, ABB Research Center, 2005.

[KLD+94] J. Karlsson, P. Liden, P. Dahlgren, R. Johansson, and U. Gunneflo. Using heavy-ion radiation to validate fault-handling mechanisms. *Micro, IEEE*, 14(1):8–23, 1994.

[KSS10] R. Krüger, A. Schenk, and F. Schiller. System und Verfahren zur Verbesserung von Fehlerbeherrschungsmassnahmen, insbesondere in Automatisierungssystemen, Patent, 2010.

[KWS00] H. Kim, A.L. White, and K.G. Shin. Effects of electromagnetic interference on controller-computer upsets and system stability. *Control Systems Technology, IEEE Transactions on*, 8(2):351–357, March 2000.

[Lau12] M Laumer. Determination of the diagnostic coverage of a SES implementation via fault injection. Master's thesis, University of Applied Sciences Regensburg, 2012.

[LSP82] Leslie Lamport, Robert Shostak, and Marshall Pease. The Byzantine Generals Problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[Man72] David Mandelbaum. Error Correction in Residue Arithmetic. *Computers, IEEE Transactions on*, C-21(6):538–545, June 1972.

[Mar84] Jr. Marshall, T. Coding of Real-Number Sequences for Error Correction: A Digital Signal Processing Problem. *Selected Areas in Communications, IEEE Journal on*, 2(2):381–392, March 1984.

[MD00] Y.K. Malaiya and J. Denton. Module size distribution and defect density. In *Software Reliability Engineering, 2000. ISSRE 2000. Proceedings. 11th International Symposium on*, pages 62–71, 2000.

[ME02] D.G. Mavis and P.H. Eaton. Soft error rate mitigation techniques for modern microcircuits. In *Reliability Physics Symposium Proceedings, 2002. 40th Annual*, pages 216–225, 2002.

[Mot08] J. Mottok. Safely Embedded Software - a safety framework for C++. *Embedded Software Engineering Report*, 2008.

[Mot09] J. Mottok. Safely Embedded Software (SES). *Embedded Software Engineering Report*, 2009.

[MRMS94] Henrique Madeira, Mário Rela, Francisco Moreira, and João Gabriel Silva. Rifle: A general purpose pin-level fault injector. In *Dependable Computing-EDCC-1*, pages 197–216. Springer, 1994.

[MS09] Marcel Medwed and Jörn-Marc Schmidt. Coding Schemes for Arithmetic and Logic Operations - How Robust Are They? In Heung Youm and Moti Yung, editors, *Information Security Applications*, volume 5932 of *Lecture Notes in Computer Science*, pages 51–65. Springer Berlin / Heidelberg, 2009.

[MSM99] S. Mitra, N.R. Saxena, and E.J. McCluskey. A design diversity metric and reliability analysis for redundant systems. In *Test Conference, 1999. Proceedings. International*, pages 662–671, 1999.

[MSVZ07] J. Mottok, F. Schiller, Th. Völkl, and Th. Zeitler. A Concept for a Safe Realization of a State Machine in Embedded Automotive Applications. In *26th Safecomp Conference, ISBN 978-3-540-75100-7*, pages 283–288, 2007.

[NPVS03] B. Nicolescu, P. Peronnard, R. Velazco, and Y. Savaria. Efficiency of transient bit-flips detection by software means: a complete study. In *Defect*

and Fault Tolerance in VLSI Systems, 2003. Proceedings. 18th IEEE International Symposium on, pages 377–384, November 2003.

[NV03] B. Nicolescu and R. Velazco. Detecting Soft Errors by a Purely Software Approach: Method, Tools and Experimental Results. *Design, Automation and Test in Europe Conference and Exhibition*, 2:20057, 2003.

[NVR01] B. Nicolescu, R. Velazco, and M.S. Reorda. Effectiveness and limitations of various software techniques for "soft error" detection: a comparative study. In *On-Line Testing Workshop, 2001. Proceedings. Seventh International*, pages 172–177, 2001.

[Oh00] Nahmsuk Oh. *Software Implemented Hardware Fault Tolerance*. PhD thesis, Standford University, December 2000.

[OMM02] N. Oh, S. Mitra, and E.J McCluskey. ED4I: Error Detection by Diverse Data and Duplicated Instructions. *IEEE Transactions On Computers, Vol. 51*, pages 180–199, 2002.

[Oze92] P Ozello. The Coded Microprocessor Certifcation. In *International Conference on Computer Safety, Reliability and Security*, pages 185–190. Springer Munich, 1992.

[Par97] B. Parhami. Defect, fault, error,..., or failure? *Reliability, IEEE Transactions on*, 46(4):450–451, December 1997.

[PB61] W.W. Peterson and D.T. Brown. Cyclic Codes for Error Detection. *Proceedings of the IRE*, 49(1):228–235, January 1961.

[PF82] J.H. Patel and L.Y. Fung. Concurrent Error Detection in ALU's by Recomputing with Shifted Operands. *Computers, IEEE Transactions on*, C-31(7):589–595, July 1982.

[Pra57] E. Prange. Cyclic Error-Correcting Codes in Two Symbols. Technical report, Air Force Cambridge Research Center, September 1957.

[Pra96] Dhiraj K. Pradhan, editor. *Fault-tolerant computer system design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[Rao70] T.R.N. Rao. Biresidue error-correcting codes for computer arithmetic. *Computers, IEEE Transactions on*, C-19(5):398–402, May 1970.

[Rao74] Thammavarapu R. N. Rao. *Error coding for arithmetic processors.* Electrical science series. Academic Press, New York and London, 1974.

[RCV+05] G.A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D.I. August. SWIFT: software implemented fault tolerance. In *Code Generation and Optimization, 2005. CGO 2005. International Symposium on*, pages 243–254, March 2005.

[RRVT01] M. Rebaudengo, M.S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. In *Source Code Analysis and Manipulation, 2001. Proceedings. First IEEE International Workshop on*, pages 33–42, 2001.

[RS60] I S Reed and G Solomon. Polynomial codes over certain finite fields. *Journal of the Society for Industrial and Applied Mathematics*, 8(2):300–304, 1960.

[RSS+08] K. Reick, P.N. Sanda, S. Swaney, J.W. Kellington, M.J. Mack, M.S. Floyd, and D. Henderson. Fault-Tolerant Design of the IBM Power6 Microprocessor. *Micro, IEEE*, 28(2):30–38, March-April 2008.

[SAC+99] T.J. Slegel, III Averill, R.M., M.A. Check, B.C. Giamei, B.W. Krumm, C.A. Krygowski, W.H. Li, J.S. Liptay, J.D. MacDougall, T.J. McPherson, J.A. Navarro, E.M. Schwarz, K. Shum, and C.F. Webb. IBM's S/390 G5 microprocessor design. *Micro, IEEE*, 19(2):12–23, March/April 1999.

[Sch11] Ute Schiffel. *Hardware Error Detection Using AN-Codes.* PhD thesis, Technische Universität Dresden, Germany, 2011.

[Seo09] Poong Hyun Seong. *Springer Series in Reliability Engineering.* Springer London, London, 2009.

[SKK+02] P. Shivakumar, M. Kistler, S.W. Keckler, D. Burger, and L. Alvisi. Modeling the effect of technology trends on the soft error rate of combinational logic. In *Dependable Systems and Networks, 2002. DSN 2002. Proceedings. International Conference on*, pages 389–398, 2002.

[SMM+09] M. Steindl, J. Mottok, H. Meier, F. Schiller, and M. Fruechtl. Diskussion des Einsatzes von Safely Embedded Software in FPGA-Architekturen. In Proceedings of the 2nd Embedded Software Engineering Congress. *Embedded Software Engineering Report*, 2009.

[SMM10] M. Steindl, J. Mottok, and H. Meier. SES-based framework for fault-tolerant systems. In *Intelligent Solutions in Embedded Systems (WISES), 2010 8th Workshop on*, pages 12–16, July 2010.

[SN04a] Siemens Norm SN 29500-2. Failure rate, component, expected value, general, January 2004.

[SN04b] Siemens Norm SN 29500-2. Failure rate, component, expected values for integrated circuits, January 2004.

[SSS+11] Martin Süßkraut, André Schmitt, Ute Schiffel, Marc Brünink, and Christof Fetzer. SIListra Compiler: Building Reliable Systems with Unreliable Hardware (Poster paper). In *Proceedings of The 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2011)*, 2011.

[SSSF10a] U. Schiffel, A. Schmitt, M. Susskraut, and C. Fetzer. Software-Implemented Hardware Error Detection: Costs and Gains. In *Dependability (DEPEND), 2010 Third International Conference on*, pages 51–57, July 2010.

[SSSF10b] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-encoding: Detecting hardware errors in software. In *Computer Safety, Reliability, and Security*, volume 6351 of *Lecture Notes in Computer Science*, pages 169–182. Springer Berlin / Heidelberg, 2010.

[Ste09] M. Steindl. Safely Embedded Software (SES) im Umfeld der Normen für funktionale Sicherheit. *Jahresrückblick 2009 des Bayerischen IT-Sicherheitsclusters*, 2009.

[UHK+12] P. Ulbrich, M. Hoffmann, R. Kapitza, D. Lohmann, W. Schroder-Preikschat, and R. Schmid. Eliminating single points of failure in software-based redundancy. In *Dependable Computing Conference (EDCC), 2012 Ninth European*, pages 49–60, 2012.

[VÖ07] T. Völkl. A concept for a safe realization of a state machine in embedded automotive applications. Master's thesis, University of Applied Sciences Regensburg, February 2007.

[VM97] Jeffrey M Voas and Gary McGraw. *Software fault injection: inoculating programs against errors*. John Wiley & Sons, Inc., 1997.

[VPC02] T.N. Vijaykumar, I. Pomeranz, and K. Cheng. Transient-fault recovery using simultaneous multithreading. In *Computer Architecture, 2002. Proceedings. 29th Annual International Symposium on*, pages 87–98, 2002.

[WA08] Fan Wang and V.D. Agrawal. Single Event Upset: An Embedded Tutorial. In *VLSI Design, 2008. VLSID 2008. 21st International Conference on*, pages 429–434, January 2008.

[WF] Ute Wappler and Christof Fetzer. Software encoded processing: Building dependable systems with commodity hardware. In *Computer Safety, Reliability, and Security*, Lecture Notes in Computer Science.

[WF06] Ute Wappler and Christof Fetzer. Hardware fault injection using dynamic binary instrumentation: Fitgrind. *Proceedings Supplemental, vol. EDCC-6 (October 2006)*, 2006.

[WKWY07] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. Compiler-Managed Software-based Redundant Multi-Threading for Transient Fault Detection. In *Proceedings of the International Symposium on Code Generation and Optimization*, CGO '07, pages 244–258, Washington, DC, USA, 2007. IEEE Computer Society.

[WM08] Ute Wappler and Martin Müller. Software protection mechanisms for dependable systems. In *DATE '08: Proceedings of the conference on Design, automation and test in Europe*, pages 947–952, New York, NY, USA, 2008. ACM.

Publications of the Author

[1] P. Raab, S. Krämer, J. Mottok, H. Meier and S. Racek. Safe Software Processing by Concurrent Execution in a Real-time Operating System. In *Proceedings of 2011 International Conference on Applied Electronics (AE)*, pages 315-319, September 2011.

[2] P. Raab, S. Krämer and J. Mottok. Cyclic Codes and Error Detection during Data Processing in Embedded Software Systems. In *Proceedings of the 4th Embedded Software Engineering Congress*, pages 577-590, December 2011.

[3] P. Raab, S. Racek, S. Krämer and J. Mottok. Reliability of Task Execution during Safe Software Processing. In *Proceedings of the 15th Euromicro Conference on Digital System Design*, pages 84-89, September 2012.

[4] S. Krämer, P. Raab, J. Mottok and S. Racek. Reliability Analysis of Real-time Scheduling by Means of Stochastic Simulation. In *Proceedings of 17th International Conference on Applied Electronics*, pages 151-156, September 2012.

[5] S. Krämer, P. Raab, J. Mottok and S. Racek. Simulationsbasierte Reliability-Analyse - Einfluesse von zufaelligen Fehlern auf das Echtzeit-Scheduling. In *Proceedings of the 5th Embedded Software Engineering Congress*, pages 613-622, December 2012.

[6] P. Raab, S. Krämer and J. Mottok. Error Model and the Reliability of Arithmetic Operations In *IEEE 2013 EUROCON - International Conference on Computer as a Tool*, pages 630-637, July 2013.

[7] P. Raab, S. Racek, S. Krämer and J. Mottok. Data Flow Analysis of Software Executed by Unreliable Hardware. In *Proceedings of the 16th Euromicro Conference on Digital System Design (DSD)*, pages 243-249, September 2013.

Journals

[8] P. Raab, J. Mottok and H. Meier. OSEK-RTOS für Jedermann (Teil 1). *Embedded Software Engineering Report*, pages 14-15, September 2009.

[9] P. Raab, J. Mottok and H. Meier. OSEK-RTOS für Jedermann (Teil 2). *Embedded Software Engineering Report*, pages 10-12, November 2009.

[10] P. Raab, V. Vavricka, S. Krämer and J. Mottok. Isomorphism between Linear Codes and Arithmetic Codes. **Accepted** for *Computing and Informatics (CAI)*, February 2013.

[11] P. Raab, S. Krämer and J. Mottok. Reliability of Data Processing and Fault Compensation in Unreliable Arithmetic Processors. **Submitted** to *Microprocessors and Microsystems: Embedded Hardware Design (MICPRO)*, September 2013.

Reports

[12] P. Raab. Modeling of Computer System Performance and Reliability. Report for subject *Modeling of Computer System Performance and Reliability*, Examiner: doc. Racek, University of West Bohemia, December 2011.

[13] P. Raab. Reliability Evaluation of Data Processing in Fault-Tolerant Computer Systems. Report for PhD state examination, University of West Bohemia, December 2012.

Appendix A

List of Cumulated Articles

A.1 Safe Software Processing by Concurrent Execution in a Real-time Operating System

Authors: P. Raab, S. Kraemer, J. Mottok, H. Meier and S. Racek
Published: In *Proceedings of 2011 International Conference on Applied Electronics (AE)*, pages 315-319, September 2011.
ISBN-13: 978-80-7043-987-6

Safe Software Processing by Concurrent Execution in a Real-Time Operating System

Peter Raab, Stefan Krämer, Jürgen Mottok, Hans Meier
Regensburg University of Applied Sciences
Faculty of Electronics and Information Technology
Seybothstr. 2, D-93053 Regensburg, Germany
{peter.raab, stefan.kraemer, juergen.mottok, hans.meier}@hs-regensburg.de

Stanislav Racek
University of West Bohemia
Faculty of Applied Sciences
Univerzitní 22, 306 14 Plzen,
Czech Republic
stracek@kiv.zcu.cz

Abstract—**The requirements for safety-related software systems increases rapidly. To detect arbitrary hardware faults, there are applicable coding mechanism, that add redundancy to the software. In this way it is possible to replace conventional multi-channel hardware and so reduce costs. Arithmetic codes are one possibility of coded processing and are used in this approach. A further approach to increase fault tolerance is the multiple execution of certain critical parts of software. This kind of time redundancy is easily realized by the parallel processing in an operating system. Faults in the program flow can be monitored. No special compilers, that insert additional generated code into the existing program, are required. The usage of multi-core processors would further increase the performance of such multi-channel software systems. In this paper we present the approach of program flow monitoring combined with coded processing, which is encapsulated in a library of coded data types. The program flow monitoring is indirectly realized by means of an operating system.**

I. Introduction

The complexity and functionality of electronic control units has more and more increased in several sectors of industry the last years. In addition the requirements of these systems became harder according safety, reliability and availability. In opposite to this progress the industries demand is to decrease costs for electronics and to remain competitive. The use of inexpensive commodity hardware is the result. However the development of present microcontrollers follows the trend of decreasing feature size that leads to less reliability. Arbitrary hardware faults became more and more probable [1]. Existing ECC protection units only safeguard RAM or FLASH memory but cannot detect faults on internal buses, registers, caches or the CPU itself. The consequences are bit flips that can occur during runtime and change data or even the program flow. For safety critical applications it is important to detect bit errors in time before they lead to fatal system malfunctions and the availability is not guaranteed any more. For increasing the reliability there are a lot of so-called SIHFT techniques [2] (= Software-Implemented-Hardware-Fault-Tolerance). SIHFT techniques operate with diverse approaches. Diversity is the plurality, to program and execute software in different manners. There are several kinds of diversity. All of them add redundancy to the system and increase fault tolerance in a way that is only possible with redundant hardware.

ED^{4}I [3] adds a second time and data redundant software channel to the original program. This second channel uses the same data like the original channel, but in coded form (= data diversity). [4], [5] and [6] follow similar approaches. [7] follows the approach of compiler based encoding. Different to the mentioned solutions the Laboratory for Safe and Secure Systems (LaS^3) investigates the approach of time redundant execution of a task by an operating system (see section III).

Instead of using a simple copy of data, arithmetic codes are preferred to add redundancy. So it is possible to process the coded data with arithmetic operations. Brown [8] was one of the first who researched AN-codes regarding error detection and correction. Since then AN-codes are often used for coded software processing [9], [3], [10], [11]. Safely Embedded Software (SES) [5] is an approach of LaS^3 at the University of Applied Science Regensburg. In section II we introduce our C/C++ library using the SES approach to define safe data types for safe guarding the software.

Besides data corruption, program flow errors can be another consequence of bit flips. There are several solutions for program flow monitoring. The supervision can be realized by additional hardware [12], [13] or by software [14], [15], [16], [17]. All software approaches have in common that a kind of preprocessor analyzes the program flow, assigns unique signatures for every branch-free block and adds generated code into the existing program. In opposite to this we introduce a method to monitor the program flow without additional hardware or a special preprocessor. In section III we describe how to monitor the program flow indirectly by means of an operating system.

II. Safely Embedded Software Library

As introductory mentioned AN-codes are used for providing software based fault tolerance. The computed data is transformed to the coded domain. For that reason the LaS^3 has developed a C and a C++ library which encapsulates the coded operations. Because C is widely used in the automotive domain there are two versions of the library [5]. C has the disadvantage of not offering the possibility of operator overloading, so every safe operation has to be performed by an explicit call of the corresponding library function. Whereas C++ has the possibility

of operator overloading. Hence a programmer can transparently use coded variables. In the following the description is focused on the C++ library. The basic concepts and ideas are analogue to the C-library. By declaring the variables as instances of safe data types the transformation to the coded domain is done implicitly within the safe data class. All operators are overloaded and perform the coded operation instead of the uncoded one. The original value x_f is transformed to the coded value x_c by applying the following transformation rule:

$$x_c = A * x_f + B_x + D_j \quad (1)$$

The diversity factor A is responsible for several safety characteristics e.g. the residual error probability (1/A) or the Hamming distance. With regard to the implementation the factor A has important influence to the size of the generated code word. The parameters B_x and D_j form the signature of the coded word. In the presented implementation B_x represents the static signature and is computed on base of the memory address of the coded variable. By this means the access to the correct memory address or variable can be assured. The second part of the signature is formed by the dynamic signature D_j. This signature ensures that the variable is accessed by the correct cycle (j) of a task [5].

```
T_SAFE_INT16 a = 12;
T_SAFE_INT16 b = 2;
T_SAFE_INT16 c;
c = a / b;
```

Listing 1. Usage of safe data types: Thus the coded computation based on the transformation rule (equation 1) is hidden from the user within the safe data type class (figure 1).

With the transformation rule stated above (equation 1) each mathematic operation can be mapped to the coded domain. The calculation is done completely in the coded domain without retransforming the data while performing an operation. The introduced library handles all the mechanisms mentioned above by providing overloaded coded operators. So the usage is straightforward like listing 1 shows.

The library (see figure 1) provides safe data types for corresponding standard types like int32, uint32, int16 and uint16, etc. Furthermore it provides safe pointer types and safe array types. All operations regarding pointer arithmetic and array access are implemented in the coded domain and are thereby safeguarded by the SES approach. Even without the usage of a separate channel the AN-code offers the possibility to check the correctness of a coded operation or the integrity of a memory location (by a simple modulo operation). In case of detecting errors in the coded data an error handler is triggered and the application executes an error reaction. The integration of our library into the real-time operating system - Regensburg´s Reliable Realtime Operating System (R^3TOS, see section III) - allows a simple usage of redundancy in practice. Furthermore our library can be used as a platform for future research activities of

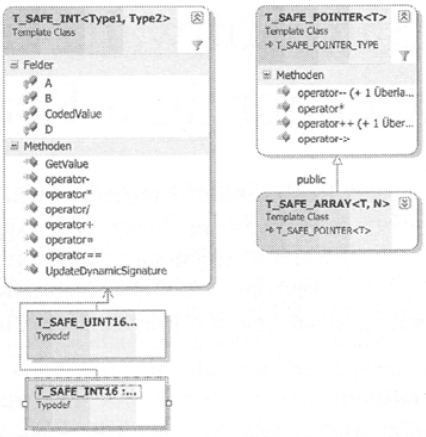

Figure 1. Class-diagram of the SES library: The class consists of overloaded operators, which perform the coded computation. The class encapsulates data fields like the coded representation of the value itself, the applied signature (B and D) and the diversity factor (A).

different coding mechanisms.

III. Time Redundancy by Concurrent Execution

The Laboratory For Safe and Secure Systems (LaS^3) develops the open-source real-time operating system R^3TOS based on the OSEK-VDX standard [18][19]. The usage of an operating system in safety-related systems leads to advantages in controlling the concurrent processing needed for time redundancy. The scheduler of an operating system determines the time when a task will be activated, executed or terminated. For handling safety critical tasks the scheduler has to be slightly adapted to fulfill the requirements. E.g. the synchronized simultaneous execution of the instances of a safety task has to be implemented. The proposed concept suggests the usage of dual-core hardware. This paper assumes single core execution, however the R^3TOS is able to deal with dual-core hardware and implements suitable scheduling algorithms [20]. The following section describes our approach of concurrent execution based on R^3TOS (Figure 2).

A. Time Redundancy

Figure 3 demonstrates the time behavior of a task executed twice The execution time doubles and the deadline must delay according the increased runtime. After termination of the second instance (= trailing task) the task will be finished and the supervisor compares the results.

The doubled execution of one task is realized by using existing services of the operating system. The scheduler handles two single instances - which form a safety task - and executes them concurrently. Both instances run the same program code. Because both tasks instances have the same priority, they are executed concurrently in two different time slots[1]. The

[1]The scheduling of tasks with same priority is often called Round Robin. This scheduling policy is also used by R^3TOS.

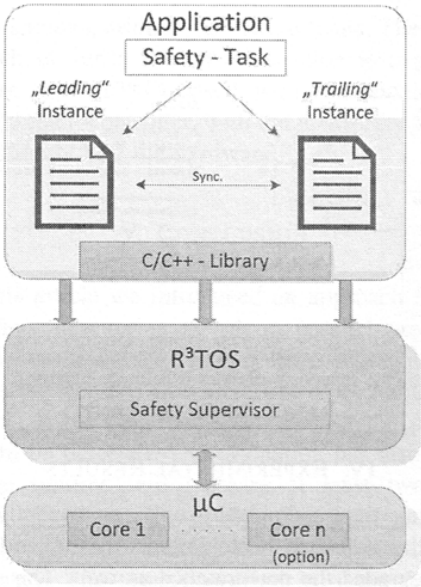

Figure 2. Simplified software architecture of R³TOS. The Safety-Supervisor as an extension of R³TOS monitors the execution of a safety-task and improves the fault-tolerance. The safety task is executed in two separate instances - the leading and the trailing task instance.

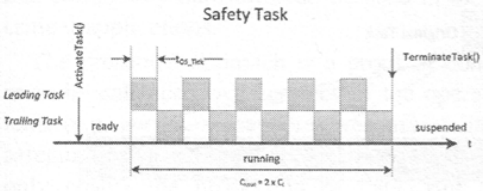

Figure 3. Time redundancy achieved by duplicated task execution

length of this time slot depends on the system tick of the operating system. The second instance is delayed compared to the first instance (= leading task) at least by one period of the system tick.

B. Data Diversity

A pure time redundant approach as described above discloses transient faults in the data memory only if the fault is not present during execution of the second instance any more. If this is not the case or one memory cell is defect permanently, the fault will remain undiscovered. To detect this error the data must be doubled. To detect systematic faults the two data channels are coded different diversity factors (see equation 1). This data diversity is partly realized by existing means of the operating system. Every task has its own stack memory used for local variables [19]. Data stored in the stack is redundant in two different memory areas. But what about global variables that are not located on the stack? To make the task redundant completely, every instance of a safety task must have its own global variables in different RAM sections. Therefore two problems must be solved:

1) During software build process the linker maps every global variable to a dedicated address. They can be accessed by unique identifiers in the source code. Both instances use the same source code and access the same memory. When declaring a global variable it has to be assigned to its safety task. During system start the operating system copies it to a defined memory section for the second channel (see figure 4).
2) Every task instance may access only its own data although both use the same source code with equal identifiers. During every operation the address of the operands have to be switched to the correct memory section that belongs to the currently running instance. The usage of C++ (see section II) allows us to overload operators for safe data types[2]. By overloading it is possible to hide the pointer arithmetics required for the correct memory access from programmer (see figure 4).

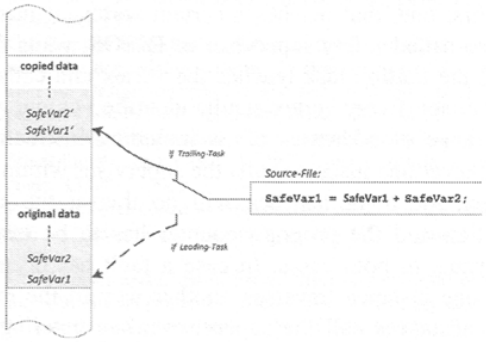

Figure 4. Access of duplicated global variables by a safety task: Original data is generated by the linker during software build process. Copied data are generated by the operating system during startup.

C. Synchronization

As described before the concurrent execution of a task can be realized by existing services of an operating system (e.g. multiple task activation). The operating system executes the same task twice. Without synchronization both task instances would be executed independently until they terminate. Both task instances must synchronize each other to detect arbitrary hardware faults (e.g. program flow error). The shorter the time between two consecutive synchronization points the shorter the latency of fault detection (figure 5). The synchronization includes two mechanisms:

- Comparison of the program counter: During control flow structures (like if-else) the program flow can diverge between the two tasks in case of a fault. Every block[3] of the program is synchronized and the program flow can be checked indirectly.
- Comparison of variables: After every calculation both instances of the safety task must have the same result in case of no error. Every calculation step can be safe-guarded by synchronizing the variable.

[2]In standard C overloading is not possible, but here you can realize this by special functions for each operation.

[3]A block is a branch-free sequence of instructions of a program [3].

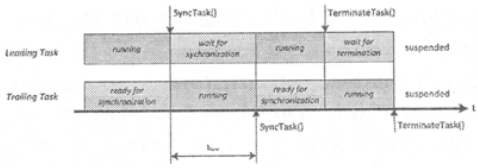

Figure 5. Synchronization of two task instances

1) Program Flow Monitoring: The described time redundancy forces a task to traverse its program graph twice. With constant conditions the path (= program flow) must always be the same. Every vertex represents a branch-free code segment and the edges define the allowed transitions from one block to another[4]. So a program can be depicted as a graph (see figure 6) where the vertices represent the moment when a task has to be synchronized. The leading task is the first one, that reaches a certain vertex. It invokes the so-called safety supervisor of R^{3}TOS, waits there until the trailing task reaches the vertex and calls the supervisor. Every vertex can be identified uniquely by the range of addresses of its including instructions. Whenever one instance calls the supervisor within one vertex, the other instance will do this at the same position and the program counter has to be exactly the same in both cases. In case a fault has occurred and one instance traverses another way in the graph both instances call the supervisor when entering the next vertex. But the program counters are not equal and a program flow error is detected. However the supervisor cannot correct this error, but it can pass the last common vertex to an error handler, where for example it is possible to perform a backward-recovery if the reaction time allows this or to initialize an organized shutdown of the system to put it in a safe state.

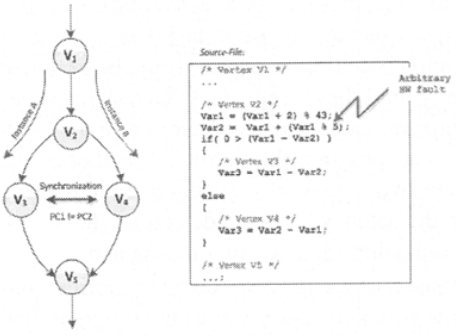

Figure 6. Example for a program graph of a task

2) Data Diversity: Every instance deals with its own data. The data diversity allows to detect faults in data memory. But for detecting arbitrary errors during calculations, the data must be coded (see SES library in section II). Similar to the program flow monitoring described before, data monitoring is also possible by this approach. After important calculations or after partial results, the supervisor synchronizes the content of safety variables. (see figure 7).

[4]if-else, switch, loops or function calls

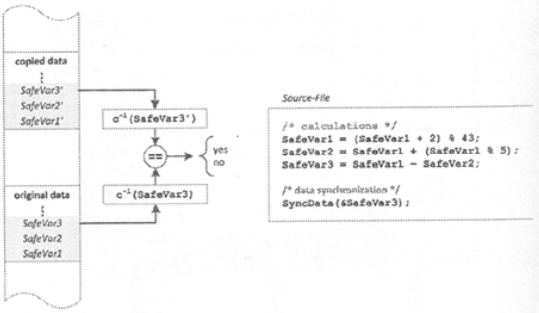

Figure 7. Synchronization of two data channels: In case of different coded data, the supervisor decodes it before comparison.

IV. Experimental Results

Adding time redundancy will consequently slow-down the software. Calculations in the coded domain degrades the performance as well. This section summarizes the results of our evaluation using the safe state machine introduced in [5]. The safe state machine, realized by if-else and switch, is applicable for evaluating our program flow monitoring and is implemented by the presented coding techniques.

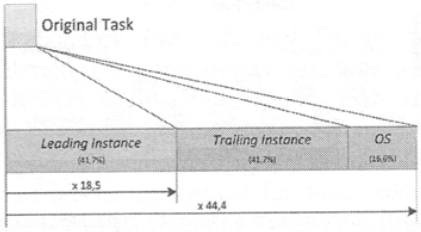

Figure 8. Slowdown of application runtime compared to its original version

Figure 8 shows the slowdown of the runtime executing one cycle of the presented safe state machine splitted into the part caused by the coding, the parallel processing and the operating system. First the coded processing has the most significant influence on the slowdown. The safe state machine using coded data types[5] is about 18 times slower compared to the uncoded state machine. Clearly the concurrent execution doubles the runtime by the second factor of two. But the impact of the overhead generated by the scheduling is low compared to the coded processing. The operating system slows down the application only by a factor of about 1,2 (or 16,6% of total reduction of performance).

Furthermore we recommend to use 32 bit controller for coded processing. The SES framework makes usage of 32 bit calculations for the coded data types (8 and 16 bit). SES on controllers with smaller data width would increase the slowdown much more.

Future work will verify the error detection mechanisms by applying fault injection. This can be realized by additional OS-Task, that corrupts the safety task data and context (program counter, stack, registers,

[5]For this evaluation the C-library is used. Similar results are expected for the C++ library.

etc.) to simulate arbitrary hardware faults. The second approach to verify the error detection is to run the software on a FPGA-based hardware emulating a modified microcontroller, with the possibility to inject faults directly into the hardware.

V. Conclusion

In this article we introduced an approach for program flow monitoring based on parallel processing done by an operating systems. In combination with the existing libraries of coded data types faults in the memory, in the operation itself and in the program flow can be detected. The evaluation has also shown that the runtime will increase at least two times because of redundant execution. But furthermore the runtime will also increase using coded data types of our library. The total slowdown is about 44 times slower compared to the original version. In applications which have long idle times and do not require fast reaction times, like elevators, smart sensors, electrical window lifters the increased computation time of the SES approach does not affect the system design and commodity hardware can be used in these safety critical applications.

The presented approach is a proof of concept and must be enhanced. All services of the operating systems (e.g. events, semaphores, resources) have to be safeguarded in a similar way. Till now our approach only covers the processing of data. But does not include the handling of input and output devices.

There are following advantages in using operating systems and our library for safety applications. The user software can be programmed independently of the underlying hardware. The hardware abstraction is part of the operating system. The presented program flow monitoring needs no additional hardware and no special preprocessors are required. Standard C or C++ compilers are sufficient. The mapping of the AN-coding into data type packed in a library makes the usage for the programmer more easily. In case of a C++ library the operations for the data types are overloaded and the source code looks like standard C code. Using operating systems for safety applications allows further methods to increase availability. For example in case of detecting faults, the called error handler can trigger reactions like backward-recovery or reject the faulty task instance. Moreover multi-core processors offer possibilities in concurrent execution of a safety task. Special schedulers allocate two instances on different cores and take the advantage of hardware redundancy. Regarding existing safety standards there is the possibility for scaling the system for different levels of safety. Continuing our researches the injection of faults into an SES protected application would verify the coding mechanism and the program flow monitoring.

This work is part of the research project S^3OP and is supported by the Bavarian State Ministry for Science, Research and Arts.

References

[1] S. Borkar. Designing reliable systems from unreliable components: the challenges of transistor variability and degradation. *Micro, IEEE*, 25(6):10–16, Nov.-Dec. 2005.

[2] Olga Goloubeva, Maurizia Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. *Software-Implemented Hardware Fault Tolerance*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[3] N. Oh, S. Mitra, and E.J McCluskey. ED4I: Error Detection by Diverse Data and Duplicated Instructions. *IEEE Transactions On Computers, Vol. 51*, pages 180–199, 2002.

[4] B. Nicolescu and R. Velazco. Detecting soft errors by a purely software approach: Method, tools and experimental results. *Design, Automation and Test in Europe Conference and Exhibition*, 2:20057, 2003.

[5] J. Mottok, F. Schiller, Th. Völkl, and Th. Zeitler. A Concept for a Safe Realization of a State Machine in Embedded Automotive Applications. In *26th Safecomp Conference, ISBN 978-3-540-75100-7*, pages 283–288, 2007.

[6] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante. Soft-error detection through software fault-tolerance techniques. In *Defect and Fault Tolerance in VLSI Systems, 1999. DFT '99. International Symposium on*, pages 210–218, November 1999.

[7] Ute Wappler and Martin Müller. Software protection mechanisms for dependable systems. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '08, pages 947–952, New York, NY, USA, 2008. ACM.

[8] D.T. Brown. Error detecting and error correcting binary codes for arithmetic operations. In *IRE Trans. Electron. Comput.*, pages 333–337. 1960.

[9] P. Forin. Vital coded microprocessor principles and application for various transit systems. In *IFA-GCCT*, pages 79–84. 1989.

[10] Ute Wappler and Christof Fetzer. Software encoded processing: Building dependable systems with commodity hardware. In Francesca Saglietti and Norbert Oster, editors, *Computer Safety, Reliability, and Security*, volume 4680 of *Lecture Notes in Computer Science*, pages 356–369. Springer Berlin / Heidelberg, 2007.

[11] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. Anb- and anbdmem-encoding: Detecting hardware errors in software. In *Computer Safety, Reliability, and Security*, volume 6351 of *Lecture Notes in Computer Science*, pages 169–182. Springer Berlin / Heidelberg, 2010.

[12] N.R. Saxena and E.J. McCluskey. Control-flow checking using watchdog assists and extended-precision checksums. *IEEE Transactions on Computers*, 39:554–559, 1990.

[13] D.J. Lu. Watchdog processors and structural integrity checking. *Computers, IEEE Transactions on*, C-31(7):681–685, July 1982.

[14] Z. Alkhalifa, V.S.S. Nair, N. Krishnamurthy, and J.A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *Parallel and Distributed Systems, IEEE Transactions on*, 10(6):627–641, June 1999.

[15] N. Oh, P.P. Shirvani, and E.J. McCluskey. Control-flow checking by software signatures. *Reliability, IEEE Transactions on*, 51(1):111–122, March 2002.

[16] O. Goloubeva, M. Rebaudengo, M. Sonza Reorda, and M. Violante. Soft-error detection using control flow assertions. *Defect and Fault-Tolerance in VLSI Systems, IEEE International Symposium on*, 0:581, 2003.

[17] R. Vemu and J.A. Abraham. Ceda: control-flow error detection through assertions. In *On-Line Testing Symposium, 2006. IOLTS 2006. 12th IEEE International*, page 6 pp., 0-0 2006.

[18] P. Raab, J. Mottok, and H. Meier. OSEK-RTOS für Jedermann (Teil 1). *Embedded Software Engineering Report*, pages 14–15, September 2009.

[19] P. Raab, J. Mottok, and H. Meier. OSEK-RTOS für Jedermann (Teil 2). *Embedded Software Engineering Report*, pages 10–12, November 2009.

[20] S. Kraemer, J. Mottok, and H. Meier. Modifikation des Taskzustandsmodells des LLREF-Schedulers auf einem Dual-Core-Prozessor. In *2nd Embedded Software Engineering Conference*, ISBN 978-3-8343-2402-3, pages 628–636, December 2009.

A.2 Cyclic Codes and Error Detection during Data Processing in Embedded Software Systems

Authors: P. Raab, S. Kraemer and J. Mottok
Published: In *Proceedings of the 4th Embedded Software Engineering Congress*, pages 577-590, December 2011.

Cyclic Codes and Error Detection during Data Processing in Embedded Software Systems

Peter Raab, Stefan Krämer and Jürgen Mottok

Laboratory for Safe and Secure Systems
Regensburg University of Applied Sciences
Faculty of Electronics and Information Technology
Seybothstr. 2, D-93053 Regensburg, Germany
{peter.raab, stefan.kraemer, juergen.mottok}@hs-regensburg.de

Cyclic codes are well-known for error detection and correction for data transmission and storage. Coded processing is a domain using error detection codes for arithmetic operations. Instead of static data for example used in broadcast systems the data in processing units change with execution of arithmetic operations and the codes must be modified in an adequate manner. Only then the code corresponds to its data and error detection is possible. Codes generated by arithmetic operations are optimal for processing of coded data. The arithmetic unit of a processor can be used for both the operation itself and the code. This paper describes cyclic linear codes as an alternative for coded processing with its advantages and disadvantages compared to state of the art AN codes. A footprint of effort and performance is given.

1. INTRODUCTION

The complexity and functionality of electronic control units have more and more increased in several sectors of industry the last years. In addition the requirements of these systems became harder according safety, reliability and availability. In opposite to this progress the industry demands to decrease costs for electronics to remain competitive. The use of inexpensive commodity hardware is the result. However the development of present microcontrollers follows the trend of decreasing feature size. That leads to less reliability and arbitrary hardware faults are more likely [1]. Increasing the fault tolerance of unreliable hardware is often a requirement in safety critical applications. [2] summarizes the state of the art techniques of Software-Implemented-Hardware-Fault-Tolerance (= SIHFT). One simple possibility of hardening data against Single Event Upsets (= SEU) is duplication (= data redundancy) and the multiple computation of data (= time redundancy). But only transient faults can be detected by pure data redundancy. Permanent faults in the CPU (e.g. stuck-at fault in the adder hardware) will generate the same erroneous result. The consequence is the use of redundant hardware or of diverse data, so that different units of the CPU are used.
There are several approaches for diverse data. Oh et al. [3] achieve data diversity by multiplication of every variable with a diversity factor. They showed that depending on this factor the computation of both data variants uses different parts of the underlying hardware. Different outputs are generated and a fault is detected. The multiplication of data with a constant factor is called AN codes, which were introduced by Brown [4] the first time. Forin [5] made use of this AN codes to protect the calculation of data in real applications. He defined coded operations like addition and multiplication to detect errors in the operator, operand and the operation itself of a single instruction in a program. A similar approach is followed by the University Dresden [6] and the University of Applied Sciences Regensburg [7], [8], [9], [10], [11]. The Laboratory for Safe and Secure Systems at the University of

Applied Sciences Regensburg developed in collaboration with the TU Munich the Safely Embedded Software (= SES) technique for the programming language C to safeguard the execution of code on microprocessors. Whereas in [12] the idea of adding extra parity bits to every variable is described. This hamming code is updated every time when the program assigns a new value to the variable. But the computation itself is not protected. Before a calculation starts, the variable is decoded and the result is coded again after computation.
This paper presents the approach of using cyclic linear codes for hardening data in unreliable hardware to detect bit flips in data storage and transportation. Chapter 2 depicts the mathematical basis for linear and arithmetic codes which are defined in chapter 3. In chapter 4 we define coded operations when using linear codes for safe software processing. Then chapter 5 makes a comparison of our approach with the state of the art technique of AN codes in respect to error detection. Finally a summary and outlook will be given in chapter 6.

2. ALGEBRA OF CODES

Algebraic structures are the background for most error detecting and correcting codes. We discuss the different algebras for two important coding mechanisms. Understanding the underlying algebra of codes helps us to describe the differences and to develop possible transformations. Interested readers are referred to [13] and [14] for detailed study of algebraic structures. An algebraic structure consists of a set of objects (e.g. numbers) and at least one operation regarding to this set. For example the set of integer numbers Z is a group under the operation of an ordinary addition (Z, +), but not under multiplication. There are the necessary axioms of closure, associative law, commutative law, identity and inverse to form a group with respect to one operation.

2.1 The Ring of Integer Numbers

A ring is a further algebraic structure beside a group. It is defined by a set of numbers under two operations. For example the set of rational numbers forms a ring under addition and multiplication (Q, +, ·). In addition to the above axioms, there are some more necessary rules. A ring consists of an additive group and the axioms of associative and distributive law under the multiplication. Note that there is also a neutral element for the multiplication when the algebraic structure is a field. For instance the set of real numbers forms a field under addition and multiplication (R, +, ·).
The set of integer numbers has infinite elements like many other sets (e.g. N, Z, Q or C). But finite sets of numbers are more important for error detecting codes in computer systems. The range of binary numbers depends on the width of the implemented registers. So the quantity of numbers is limited and the set of numbers is finite. If m is the length of a register in bits, then $M = 2^m$ is the amount of different numbers and the finite set is denoted Z_M. It can be shown that there are all axioms for a ring under addition and multiplication if m is no prime. If m is a prime there is also an inverse element for the multiplication and all axioms for a field are fulfilled [14].
With this background we know that all operations in an arithmetic unit have a ring structure over a finite set of integers (Z_M, $+_M$, $\cdot_M$). The quantity of the elements in the set is $M = 2^m$ with $m \in \{8,16,32,64\}$. These elements correspond to standard register length in computer systems and are no primes. The two operations $+_M$ and $\cdot_M$ are defined as their ordinary equivalents but with final modulo M.

Addition: $a +_M b := (a + b) \mod M$

Multiplication: $a \cdot_M b := (a \cdot b) \mod M$

2.2 The Ring of Polynomials over GF(2)

Another more complex algebraic structure is the ring of polynomials. Let us first consider an integer number in a different point of view. The m digits of a number, which are represented in a positional notation system with radix r, span a vector space $\mathbf{V}^m$ with base vectors e_0, e_1, … e_{m-1}. The m tuples of the vector $\mathbf{X} = (x_0, x_1, \ldots x_{m-1})$ with the scalars $x_i \in \{0, 1, \ldots r-1\}$ identify a unique number in the range $[0, r^m - 1]$. We know from the linear algebra that every element of the vector space can be represented as a linear combination of its base vectors. This means that a vector (or a number) can also be written as

$$\boldsymbol{X} = x_0 \cdot e_0 + x_1 \cdot e_1 + \ldots\ x_{m-1} \cdot e_{m-1}.$$

The m-dimensional vector space forms an algebraic structure of a ring under the two operations:

$$\mathbf{A} \oplus \mathbf{B} := \left\{ \left(a_k + b_k\right) \mod r \;\middle|\; \mathbf{A}, \mathbf{B} \in \mathbf{V}^m, 0 \le k < m \right\} \tag{1}$$

$$\mathbf{A} \otimes \mathbf{B} := \left\{ \left(\sum_{i=0}^{k} a_i \cdot b_{k-i} \right) \mod r \;\middle|\; \mathbf{A}, \mathbf{B} \in \mathbf{V}^m, 0 \le k < m \right\} \tag{2}$$

In computer systems the radix of the number system is r = 2 and the coefficients of the vector are in {0, 1}. The finite set consisting of two elements is called Galois Field GF(2). With above defined multiplication, the product of any two base vectors is for example

$$\begin{aligned} e_1 \otimes e_1 &= (0,1,0,\ldots) \otimes (0,1,0,\ldots) \\ &= (0,0,1,0,\ldots) \\ &= e_2 \end{aligned}$$

or in a general form

$$e_i \otimes e_j = e_{i+j}.$$

Let us now define

$$z^k := e_k \quad \text{and} \quad 1 := e_0$$

so we get the numbers in the known polynomial representation

$$R[z] := \left\{ p(z) = \sum_{i=0}^{m-1} x_i \cdot z^i \mid x_i \in GF(2) \right\}. \tag{3}$$

Where R[z] is the set of all polynomials p(z) which form a ring of polynomials over the ring of integer R and the undetermined variable z. In binary number systems it is R = GF(2). The

interested reader is referred to [13] and [14] for further detailed information in the algebra of polynomials.

2.3 Transformation of Algebraic Structures

A homomorphism in general is a mapping between two algebraic structures. This means that a set and its operation can be transformed into another set with another operation.

Definition 1. (Homomorphism) [14]

Let (G, +) and (H, ⊕) be two algebraic structures. A homomorphism is the function $\varphi: G \rightarrow H$, and for all $x, y \in G$ it is $\varphi(x + y) = \varphi(x) \oplus \varphi(y)$.
For rings and fields it is required that there is a map function for each of both operations.

An example for a homomorphism is the transformation from the ring of polynomials R[z] to the ring of integer (R, +, ·). The polynomial $p(z) \in R[z]$ is mapped to the element $X \in R$ by

$$\varphi: \quad R[z] \rightarrow (R, +, \cdot).$$

Assigning 2 to the independent variable z the result of the polynomial is the value of the number in R:

$$X = \varphi\big(p(z)\big) = p(2) = \sum_{i=0}^{m-1} x_i \cdot 2^i \qquad (4)$$

3. ERROR DETECTION AND CORRECTION CODES

In this chapter we discuss two important error detection codes which base on the prescribed algebraic structures. The first are the well-known arithmetic codes introduced by Brown [4] and later used by Forin [5] for coded software processing. The second are the linear codes which are the general case of the Hamming codes [15]. Linear codes are the basis for all important derived codes used in transmission systems [16], [17].

Definition 2. (Code)

A code is an injective function (= encoding) which assigns an information word ∈ **X** to a code word ∈ **C**

$$f_c: \quad X \rightarrow C$$

and $|C| \geq |X|$. The set of all code words **C** is called the code space and **X** is the information space. The inverse function (= decoding) of *f* reverse the mapping from **C** to **X**.

$$f_c^{-1}: \quad C \rightarrow X$$

The basic principle of error correcting codes is the existence of more elements in **C** than in **X**. But only those code words which are assigned to their origin information word by the encoding function are valid. All other words are invalid and indicate an error (see figure 1). A code is the subset of its underlying vector space $\mathbf{C} \subset \mathbf{V}^n$.

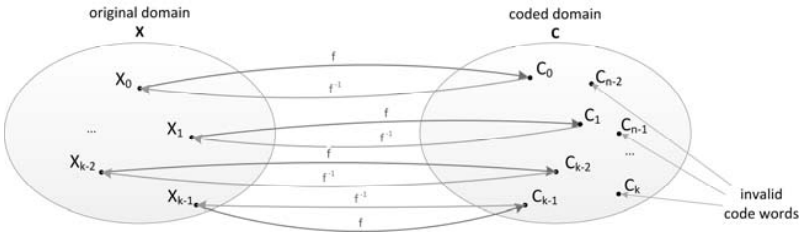

figure 1. A code unambiguously assigns every element in the origin set of cardinality K = |X| to an element in the coded set of cardinality N = |C|. With K ≤ N there are unused code words in C. An error during transmission, storage or computation can change a valid code word into an invalid one. No information word matches this code word and an error is detected.

In safety applications the information is only used in its coded representation. After transmission or computation a decoding algorithm of the dedicated code checks the received words against the original words. If there is no inverse function $f_c^{-1} : C \to X$, then an error has occurred. In the following two important error detecting codes are introduced.

3.1 Arithmetic Codes

The encoding function of an arithmetic code is the multiplication of the origin number and another integer, called the generator. Arithmetic codes are detailed discussed by Rao [18].

$$C_{AN} := \{A \cdot X \mid A,X \in Z\} \tag{5}$$

The generated code forms a ring over the set of all multiples of A less than or equal $A \cdot (2^k - 1)$ over the two operations addition and multiplication (nZ_N, $+_N$, $\cdot_N$). The sum or the product of two code words is divisible by the generator A and is therefore a valid code word (see axiom of closure). But unlike the addition, the product of two code words does not correspond to the coded product of the two original information words. Let $X_1, X_2 \in Z_K$ and the encoding function be f, then the product of the coded information words is not the code of the product:

$$f_c(X_1) \cdot f_c(X_2) \neq f(X_1 \cdot X_2) \tag{6}$$

When using arithmetic codes during software processing, a correction of the multiplication must be done. In contrast to transmission systems where the code words are not changed by the transmission itself (except errors have occurred), the output of an arithmetic unit usually differs from the inputs. In case of using coded information in an arithmetic unit, the code must be able to preserve the correspondence between the code word and its original information word. Or in other words, there must be a homomorphism that maps the calculated output of an arithmetic unit to the code words which correspond to the result of the calculation of the uncoded information words

Definition 3. (Coded Operation)

A coded operation is a homomorphism which maps the operations of the used algebraic structure to that corresponding to ordinary operations used by arithmetic units. In case of a ring there are two coded operations:

$$+_C := \varphi_+(f_c(X_1), f_c(X_2)) \tag{7}$$

$$\cdot_C := \varphi_\cdot(f_c(X_1), f_c(X_2)) \tag{8}$$

In case of an addition no correction is necessary:

$$\begin{aligned} C_1 + C_2 &= f_c(X_1) + f_c(X_2) \\ &= A \cdot X_1 + A \cdot X_2 \\ &= A \cdot (X_1 + X_2) = f_c(X_1 + X_2) \\ \Rightarrow +_C : \quad \varphi_+(C_1, C_2) &:= C_1 + C_2 \qquad \textit{for all} \quad C \in C_{AN} \end{aligned}$$

In case of a multiplication it is:

$$\begin{aligned} C_1 \cdot C_2 &= f_c(X_1) \cdot f_c(X_2) \\ &= A \cdot X_1 \cdot A \cdot X_2 \\ &= A^2 \cdot (X_1 \cdot X_2) = A \cdot f_c(X_1 \cdot X_2) \\ \Rightarrow \cdot_C : \quad \varphi_\cdot(C_1, C_2) &:= \frac{C_1 \cdot C_2}{A} \qquad \textit{for all} \quad C \in C_{AN} \end{aligned}$$

In computer systems the numbers are stored in registers which are an ordered set of k tuples. Each of them represents a binary digit of the number. Every digit can be an element from the set {0, 1}. There must be $\log_2 K$ bits to describe a number in the range from 0 till k – 1 in a binary notation. Let be K = |X|, N = |C| and K ≤ N, then there are more bits required for the code word than for the origin information word. These additional bits are needed for the code in form of the parity or check bits which represents the redundancy of the information. This redundancy is called information redundancy and is a diverse presentation of the information. A metric for this increase of bits is the code rate.

Definition 4. (Code Rate) [19]

Every block code with length n is generated by k bits of information. The ratio

$$R = \frac{k}{n} = \frac{\log_2 K}{\log_2 N} \qquad (9)$$

is the code rate and is a metric for the redundancy of the code.

3.2 Linear Codes

Linear codes [20] use another algebraic structure than arithmetic codes. We showed in chapter 2.2 that every number can be represented as a k-dimensional vector. The vector space itself forms a ring of polynomials over the finite set GF(2). Linear codes are generated by the polynomial multiplication

$$C_{CRC} := \{ g(z) \otimes x(z) \mid g(z), x(z) \in Z_2[z] \} \qquad (10)$$

Linear codes are cyclic, if the generator polynomial g(z) is a divider of the polynomial $z^n - 1$. This type of code is a special case of linear codes with the property that the cyclically rotation of the bits of a code word results in another valid code word. Cyclic codes were first introduced by Prange [21] and further discussed by Peterson [20]. As described in the previous section the set of polynomials forms a ring under the operations ⊕ and ⊗. Every sum

or product of code polynomials are valid code polynomials (= axiom of closure). Next section describes how linear codes can be used for coded software processing.

4. CODED OPERATIONS FOR SAFE SOFTWARE PROCESSING

The adder and multiplier in the arithmetic unit of a CPU implement the ordinary addition and multiplication, which correspond to the ring (Z_M, $+_M$, $\cdot_M$) described in chapter 2. Linear codes with a different algebra as for ordinary operations need transformations (= homomorphism) to be used for coded processing. In chapter 2 we showed a simple homomorphism from a polynomial to an integer (formula 4). The assignment of 2 to the undetermined variable of the polynomial results in the integer number. But the other transformation from an integer to a polynomial is also possible. The coefficient of a polynomial can be calculated bitwise by

$$\varphi^{-1}: x_i = \frac{X}{2^i} \bmod 2 \qquad (11)$$

for all $i \in [0, k-1]$. But this is exactly the binary notation of the number $X = (x_{k-1}, \ldots x_1, x_0)$ in a digital computer system. No special transformation function is needed. But what is about the operations? For a ring homomorphism there must be a function not only for the set of numbers but also for both operations. The implementation of a digital adder shows us the details, how the ordinary addition in the ring of integer works. The adder links the single bits of two distinct numbers in binary representation and the carry bit from the previous stage with an exclusive-or (XOR, $\oplus$) operation.

$$s_i = x_i \oplus y_i \oplus c_{i-1} \qquad (12)$$

Except for the carry bit c_{i-1} (delivered by the previous adder stage) the calculation of the sum of two numbers equals the addition of two polynomials.

example:

X:		0	0	1	1	1	0	1
Y:	$\oplus$	0	1	0	1	1	0	0
C:	$\oplus$	1	1	1	1	0	0	0
S:	=	*1*	*0*	*0*	*1*	*0*	*0*	*1*

Let us define the function φ_+ to transform the addition in the ring of integer to the addition in the ring of polynomials over GF(2).

$$\varphi_+: X + Y \rightarrow x(z) \oplus y(z) \oplus c(z) \qquad (13)$$

An addition in an arithmetic unit generates the carry bits represented by the polynomial c(z). Since the register of the ALU contains the numbers X and Y in binary number representation, they are practically equal to the polynomial x(z) and y(z) over the finite field GF(2). For the correct result only the carry bits c(z) must be added to the XOR sum. The carry bits are calculated by

$$c_{i+1} = (x_i \wedge y_i) \vee (x_i \wedge c_i) \vee (y_i \wedge c_i) \qquad (14)$$

or they can be derived from the adder

$$c(z) = \varphi^{-1}(X + Y) \oplus x(z) \oplus y(z). \qquad (15)$$

With this knowledge we define the coded operations in the ring of polynomial over the field GF(2).

4.1 Coded Addition

Using the algebra of polynomials over the finite field GF(2) in ordinary arithmetic units, corrective actions are required.

Definition 5. (Coded Addition)

Let $f_c(x(z)) = x(z) \otimes g(z)$ be the coding function for linear codes. The arithmetic sum of the coded integer numbers X and Y follows

$$+_c := f_c(x(z)) \oplus f_c(y(z)) \oplus f_c(c(z)). \qquad (16)$$

The coded sum of two integers X, Y is the XOR sum of the coded polynomials x(z), y(z) and the carry polynomial c(z).

Derivation: With encoding function f_c, transformation function of the addition φ_+ and the transformation from an integer to a polynomial φ^{-1} follows:

$$\begin{aligned} & f_C\left(\varphi_+(\varphi^{-1}(X), \varphi^{-1}(Y))\right) \\ &= f_C\left(\varphi_+(x(z), y(z))\right) \\ &= f_C\left(x(z) \oplus y(z) \oplus c(z)\right) \\ &= \left(x(z) \oplus y(z) \oplus c(z)\right) \otimes g(z) \\ &= x(z) \otimes g(z) \oplus y(z) \otimes g(z) \oplus c(z) \otimes g(z) \\ &= f_c(x(z)) \oplus f_c(y(z)) \oplus f_c(c(z)) \end{aligned}$$

4.2 Coded Multiplication

Similar to the addition a transformation is needed. The sum in the multiplication of polynomials (formula 2) from chapter 2 does not propagate a carry bit to the next digit. The modulo is always smaller than the radix. These missing carry bits must be added (XOR!) to the product of two polynomials so that the result is correct and matches the coded product of two integer numbers.

Definition 6. (Coded Multiplication)

Let $f_c(x(z)) = x(z) \otimes g(z)$ be the coding function for linear codes. The arithmetic product of the coded integer numbers X and Y is then

$$\cdot_c := \frac{f_c(x(z)) \otimes f_c(y(z))}{g(z)} \oplus f_c(c(z)) \qquad (17)$$

Derivation: With encoding function f_c, transformation function of the addition φ_+ and the transformation

$$
\begin{aligned}
& f_C\left(\varphi_{\cdot}(\varphi^{-1}(X),\varphi^{-1}(Y))\right) \\
&\quad = f_C\left(\varphi_{\cdot}(x(z),y(z))\right) \\
&\quad = f_C\left(x(z)\otimes y(z)\oplus c(z)\right) \\
&\quad = \left(x(z)\otimes y(z)\oplus c(z)\right)\otimes g(z) \\
&\quad = x(z)\otimes y(z)\otimes g(z)\oplus c(z)\otimes g(z) \\
&\quad = \frac{f_c(x(z))\otimes f_c(y(z))}{g(z)}\oplus f_c(c(z))
\end{aligned}
$$

The coded multiplication of polynomials contains a polynomial multiplication. This operation differs from that of ordinary multiplication in an arithmetic unit. Using coded multiplication for codes, software processing requires a different hardware for this kind of multiplication or additional software libraries. Such libraries will increase the execution time. The usage of coded operations based on arithmetic codes is preferable. But in the next section we will see other advantages of linear codes when they are used in safe software systems.

5. ERROR DETECTION PERFORMANCE

This section compares the probability for undetected errors of arithmetic codes with that of cyclic linear codes. The minimum hamming distance d_{min}, the code rate R and the probability for undetected errors P_u are important metrics which describe the performance of error detection codes. Let us compare the (7,4) AN code and a (7,4) linear code. These codes with length of n = 7 bits and n – k = 7 – 4 = 3 parity bits have $2^4 = 16$ different valid code words. The generator polynomial of degree 3 with $g(z) = z^3 + z + 1$ is primitive and generates a code with minimum hamming distance $d_{min} = 3$. An appropriate arithmetic code, which generates as many parity bits as the linear (7,4) code, must have a generator $A < 2^3 = 8$. As Brown already described in [4], an AN code like this with $A \in \{3, 5, 6, 7\}$ can only have a minimum hamming distance of 2. We can now compare the two metrics hamming distance and code rate of the given example. Both codes add the same number of parity bits to the information, the code rate $R = k / n = 4 / 7 \approx 0{,}57$ is the same. With equal code rate the linear block code has a larger minimum hamming distance than the arithmetic code. The capability of this code according error detection is better. In the following we compare the third metric of the probability for undetected errors.

5.1 Error Model

The error model according [2] regards faults affecting the hardware components in a processor based system. It can be distinguished between two types of hardware faults. Single stuck-at faults result from hardware components whose output is set to logical 1 or 0 permanently. Single bit-flips are caused by single event upsets (= SEU). The erroneous output of the hardware (bit change from 1 to 0 or vice versa) is temporally and does not have effects after reuse of the component a certain time later. Figure 2 shows the hardware components that can produce arbitrary hardware faults.
In this section we only view on the left part of figure 2 containing the data storage and transmission. As in broadcast systems the stored data is not changed until it is used by the arithmetic unit. The channel model of classical communication systems can be considered.

The term of a channel is known from the channel coding which is one part of the coding theory. Generally, a channel is a medium where data is transmitted or stored and errors are injected. A simple channel model is the binary symmetric channel (= BSC, see [22], [13]). This model describes the probability that a transmitted (or stored) bit is changed. There is no influence of bits to their neighbours. Figure 3 shows the transition probability of a bit in a BSC model.

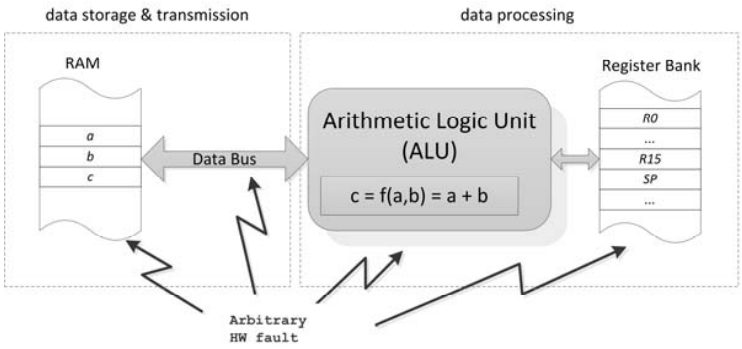

figure 2. Simplified fault model. For simple representation only faults in the data path are considered. Data errors have one cause in corrupted RAM cells or bit-flips during access via memory bus or during computation in the ALU. The processor model can be divided into one part for data storage and transmission and a second part for data processing.

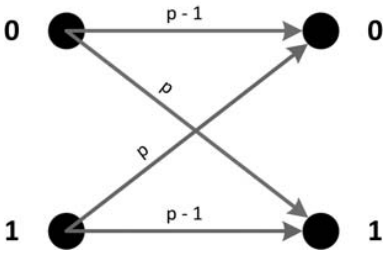

figure 3. Binary symmetric channel model describes the probability that a single bit changes its value or leaves it unchanged.

The BSC model simulates the behavior of single bits in case of an arbitrary hardware fault in the memory or the data bus in a processor system. The channel gets a single bit which is either 0 or 1 and inverts it with a probability p, or leaves it with the probability p–1. From a logical view the inversion of a single bit is the same as an exclusive-or (XOR) operation. Every bit of a code polynomial c(z) is combined with the bit of the error polynomial e(z).

for a single bit: $c_i = c_i \otimes e_i$

for the vector: $c(z) = \sum_{i=0}^{n-1} c_i \cdot z^i \otimes \sum_{i=0}^{n-1} e_i \cdot z^i$

The XOR operation of the single bits describes the property of a memoryless channel. No bit has an influence to nearby bits. The XOR does not propagate a carry to the next bit as it does an ordinary addition. The algebra of linear codes matches to the model of the BSC channel. Next the probability for undetected errors of codes through a BSC channel model will be evaluated.

5.2 Residue Error Probability

The described channel model is a simple model to derive the probability for undetected errors occurred during data storage or transmission. We follow the calculation of the probability by [22] and [20]. One axiom of the algebraic structure of rings is the property of closure (see chapter 2). This means that the addition of two elements out of the set of code words results in another code word of this set. Thus an incorrect code word is not detected in case the error vector is an element of the set of valid code words.
Let W_i be the number of valid code words with hamming weight i, then the set $\mathbf{W} = \{W_0, W_1, \ldots W_{n-1}\}$ is the weight distribution of the code. The probability that an error word has a certain hamming weight is the probability that i bits are set and n–1 bits are not set (p is the probability that a bit is inverted by the channel). For a given code there are W_i possibilities for a code word to have a weight i. The probability increases W_i times that an i-bit error results in a valid code word. Accumulation over all weights will give us the probability that the outcome of the channel is a valid code word and the faulty output is not detected. There will be no code words with weight less than the minimum hamming distance. Thus the sum of formula 18 starts at $i = d_{min}$.

$$P_u(p) = \sum_{i=d_{\min}}^{n} W_i \cdot p^i \cdot (1-p)^{n-1} \qquad (18)$$

The hamming weight for linear codes is simply the number of nonzero bits [20]. In opposite to linear codes the XOR addition of two arithmetic codes words is not a valid code word. To compute the hamming weight distribution of an arithmetic code, all XOR differences of two distinct codes words specifies all possible error words of the channel that change one valid word to another. The set of all pairs of distinct words determine the weight of all possible error words that will result in an undetected error. Applying formula 18 to the cyclic linear code with generator polynomial $g(z) = z^3 + z + 1$ and to arithmetic codes with generator integer $A \in \{3, 5, 6, 7\}$ the probability of undetected error is given by the probability of a single bit error p and is depicted in figure 4.

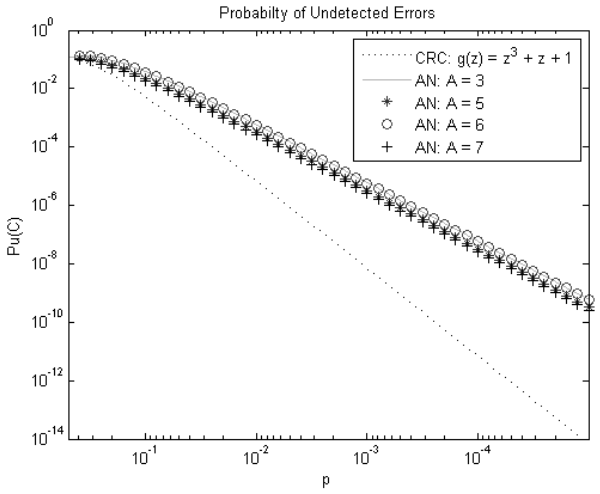

figure 4. Comparison of the probabilities for undetected errors between the cyclic linear code (CRC) and arithmetic codes (AN); both with same code rate of ≈ 0, 57.

As mentioned before all codes generated by an $A \in \{3, 5, 6, 7\}$ have a minimum hamming distance of two only. But the cyclic linear code with given polynomial has a minimum hamming distance of 3. This is reflected by the fact that in the weight distribution $W_2 > 0$ for all arithmetic codes and $W_2 = 0$ for the linear code. If $W_2 = 0$, the sum in formula 18 neglects the summand, which contains p^2. Since the probability p is significant less than 1, the i-th power of p with $i > 2$ can be ignored. Comparing both codes, one with $d_{min} = 3$ and the other with $d_{min} = 2$, shows that the residue error probability $P_u(p)$ is less for $d_{min} = 3$ than for $d_{min} = 2$ for all p with $0 < p < 0{,}5$. Note that using a generator polynomial which is not primitive, the generated cyclic code has a minimum hamming distance of 2 only. The appropriate plot for this linear code is moved to the top in figure 4. The minimum hamming distance d_{min} directly determines the residual error probability in a BSC channel.

6. CONCLUSION

The evaluation of the residual error probability of linear and arithmetic codes has shown that most cyclic linear codes (CRC) have a better property for error detection than arithmetic codes provided the BSC channel model is used and both have the same code rate. The consequence is that linear codes are more suitable for data storage and transmission than arithmetic codes. For this reason there are memory devices available which already implements error correcting codes (= ECC) based on linear codes. A CPU can be modeled as a composition of several channels. One channel was discussed in this paper and models the data storage in the memory and the transmission via the memory bus to the ALU. For other channels (like the arithmetic unit) other codes offer a better performance for error detection. There are a lot of questions that must be answered in future works. The development of an appropriate channel model of important operations in the arithmetic unit is the base for further analytical evaluation of the probability for undetected errors during arithmetic operations. The usage of divergent codes requires transformations between the codes which must be developed. Some codes, especially cyclic linear codes, also offer the possibility of error correction. A metric, which was omitted in this paper, is the probability of a decoding error. This is the case, when the decoding (correcting) algorithm of the dedicated code corrects a corrupted code word to another invalid code word. Finally, the implementation of the codes in a real application will validate this approach.

REFERENCES

[1] P.E. Dodd and L.W. Massengill. Basic mechanisms and modeling of single-event upset in digital microelectronics. Nuclear Science, IEEE Transactions on, 50(3):583 – 602, June 2003.

[2] Olga Goloubeva, Maurizia Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. Software-Implemented Hardware Fault Tolerance. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[3] N. Oh, S. Mitra, and E.J McCluskey. ED4I: Error Detection byDiverse Data and Duplicated Instructions. IEEE Transactions OnComputers, Vol. 51, pages 180–199, 2002.

[4] D.T. Brown. Error detecting and error correcting binary codes for arithmetic operations. In IRE Trans. Electron. Comput., pages 333–337. 1960.

[5] P. Forin. Vital coded microprocessor principles and application for various transit systems. In IFA-GCCT, pages 79–84. 1989.

[6] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-encoding: Detecting hardware errors in software. In Computer Safety, Reliability, and

Security, volume 6351 of Lecture Notes in Computer Science, pages 169–182. Springer Berlin / Heidelberg, 2010.
[7] J. Mottok, F. Schiller, Th. Völkl, and Th. Zeitler. A Concept for a Safe Realization of a State Machine in Embedded Automotive Applications. In 26th Safecomp Conference, ISBN 978-3-540-75100-7, pages 283–288, 2007.
[8] J. Mottok. Safely embedded software- a safety framework for c++. Embedded Software Engineering Report, 2008.
[9] H. Meier F. Schiller M. Steindl, J. Mottok and M. Fruechtl. Diskussion des Einsatzes von Safely Embedded Software in FPGA-Architekturen. In Proceedings of the 2nd Embedded Software Engineering Congress. Embedded Software Engineering Report, 2009.
[10] M. Steindl. Safely Embedded Software (SES) im Umfeld der Normen für funktionale Sicherheit. Jahresrückblick 2009 des Bayerischen IT-Sicherheitsclusters, 2009.
[11] J. Mottok. Safely Embedded Software (SES). Embedded Software Engineering Report, 2009.
[12] B. Nicolescu, R. Velazco, and M.S. Reorda. Effectiveness and limitations of various software techniques for ”soft error” detection: a comparative study. In On-Line Testing Workshop, 2001. Proceedings. Seventh International, pages 172 –177, 2001.
[13] Weldon E.J. Peterson W.W. Error Correcting Codes. MIT Press, 2nd edition, 1972.
[14] Albrecht Beutelspacher. Lineare Algebra. Vieweg, 1994.
[15] Richard. W. Hamming. Error detection and error correction codes. The Bell System Technical Journal, 26(2):147–160, 1950.
[16] I. S. Reed and G. Solomon. Polynomial codes over certain finite fields. Journal of the Society for Industrial and Applied Mathematics, 8(2):300–304, June 1960.
[17] R.C. Bose and D.K. Ray-Chaudhuri. On a class of error correcting binary group codes. Information and Control, 3(1):68 – 79, 1960.
[18] Thammavarapu R. N. Rao. Error coding for arithmetic processors. Electrical science series. Academic Press, New York and London, 1974.
[19] M. Bossert. Kanalcodierung. Teubner, 1998.
[20] W.W. Peterson and D.T. Brown. Cyclic codes for error detection. Proceedings of the IRE, 49(1):228 –235, January 1961.
[21] E. Prange. Cyclic error-correcting codes in two symbols. Technical report, Air Force Cambridge Research Center, Sept. 1957.
[22] Robert H. Morelos-Zaragoza. The Art of Error Correcting Coding. John Wiley & Sons, Ltd, 2006.

AUTHORS

M.Eng., Dipl.-Ing. (FH) Peter Raab is research assistant at the Laboratory for Safe and Secure Systems (www.las3.de) working in the research project S³OP. He is a PhD student at the University of West Bohemia Pilsen.

M.Eng., Dipl.-Ing. (FH) Stefan Krämer is research assistant at the Laboratory for Safe and Secure Systems (www.las3.de) working in the research project S³OP. He is a PhD student at the University of West Bohemia Pilsen.

Prof. Dr. rer. nat. Jürgen Mottok is professor of software engineering, computer languages, operating systems and safety at the University of Applied Sciences Regensburg. He is the head of the Laboratory for Safe and Secure Systems (www.las3.de).

A.3 Reliability of Task Execution during Safe Software Processing

Authors: P. Raab, S. Racek, S. Kraemer and J. Mottok
Published: In *Proceedings of the 15th Euromicro Conference on Digital System Design*, pages 84-89, September 2012.
ISBN-13: 978-0-7695-4798-5

2012 15th Euromicro Conference on Digital System Design

Reliability of Task Execution during Safe Software Processing

Peter Raab, Stefan Krämer, Jürgen Mottok
Faculty of Electronics and Information Technology
Regensburg University of Applied Sciences
Seybothstr. 2, D-93053 Regensburg, Germany
{*peter.raab, stefan.kraemer, juergen.mottok*}*@hs-regensburg.de*

Stansislav Raček
Faculty of Applied Sciences
University of West Bohemia
Univerzitní 22, 306 14 Plzen, Czech Republic
stracek@kiv.zcu.cz

***Abstract*—This paper presents the reliability evaluation of task execution during safe software processing. The standard method of duplication in a safety-critical application can also be applied for tasks in a software system. But in addition to this, there is also the possibility for coded task processing to increase the reliability and availability of software. The presented analysis covers the reliability analysis of a single, a duplicated and a coded task by the technique of continuous-time Markov processes. Markov processes are often used for the reliability evaluation of safety-critical systems. We introduce a method to describe the execution time of tasks by means of enhanced Markov models and their solution by numerical methods.**

Keywords: reliability analysis, continuous-time Markov process, error probability, Erlang-distribution

I. Introduction

Fault-tolerant systems have become more important in recent years. Either for economic reasons or because of safety aspects, fault-tolerant systems are required to reduce costs or save life. Achieving this aim, the correct operation of the system has to be assured and failures must be detected and repaired.

The complexity and functionality of electronic control units have increased more and more in several sectors of industry. In addition, the requirements of these systems have become more demanding in terms of safety, reliability and availability. In contrast to this progress, industry demands a decrease in costs for electronics, while at the same time remaining competitive. The use of inexpensive commodity hardware is the result. However, the development of current micro-controllers follows the trend of decreasing feature size. That leads to less reliability and arbitrary hardware faults are more likely [1]. But despite unreliable hardware, fault tolerance is a requirement of safety-critical applications. This can often be realized by software techniques in many ways [2].

For this reason, current standards and norms for functional safety summarize several state-of-the-art techniques to detect possible errors in safety-critical systems. The European standard IEC61508 - "Functional Safety of E/E/P Safety-Related Systems" [3] demands for high safety-critical applications (SIL 3 to SIL4) a maximum rate from $3 \cdot 10^7$ to $3 \cdot 10^9$ per hour of dangerous failures in the system. The norm describes the technique of coded processing and reciprocal comparison for detection of faults within the processing unit with high probability.

Today, there are a variety of publications about fault-tolerant computer systems. In [2], the authors present different state-of-the-art techniques of so-called Software-Implemented-Hardware-Fault-Tolerance (= SIHFT). They describe a simple possibility of hardening data against Single Event Upsets (= SEU) by duplication (= data redundancy) and multiple computation of data (= time redundancy).

But only transient faults can be detected by pure data and time redundancy. Permanent faults in the CPU (e.g. a stuck-at fault in the adder hardware) will generate the same erroneous result when an instruction is executed twice with the same data. The consequence is the usage of redundant hardware or of diverse data in the way that different units in a CPU are employed [4].

This paper is organized as follows. Sec. II gives an overview of the background and definitions for fault tolerance, coded software processing and reliability. The main part of this paper is Sec. III. In that section the reliability of a task in a software system is analyzed. Finally, Sec. IV summarizes the results of the analysis for further work in this research area.

II. Background

This section gives an overview of the necessary background and state-of-the-art for the reliability analysis of coded task processing.

A. Reliability and the Markov Model

Reliability is a metric to describe the ability that a system has to perform the required function correctly for a specified period of time [5]. Stochastic Markov processes are a powerful technique that is often used for the reliability analysis of a given system (see [6] for detailed description of the technique). The basic concept of Markov processes is the partition of the system into several failure states. Each state identifies a certain fault condition of the system. Thus, the Markov process can be modeled as a kind of finite automata

978-0-7695-4798-5/12 $26.00 © 2012 IEEE
DOI 10.1109/DSD.2012.110

consisting of nodes and transitions between them. The transition between two states represents a random event of a fault and its frequency is described by the transition rate. This rate can be interpreted as the reciprocal mean time of one transition. The underlying probability distribution of Markov processes is the exponential distribution with the transition rate as a parameter. Therefore, Markov processes are suitable for events that happen with exponential distributed time, like hardware faults. More regular events like repairs or the runtime of certain software tasks follow another distribution (e.g. normal distribution) and the Markov process can not be used in its original form.

In order to evaluate tasks with fixed execution time, the Markov model must be enhanced to approximate other distributions by means of the exponential distribution. The *Erlang-distribution* [7], [8], [9] is derived by the consecutive stages of exponential distributed transitions. The enhancement of a Markov process by adding further states realizes the Erlang-distribution and we can approximate other distributions like the normal distribution [10]. The evaluation in Section III makes use of this kind of Markov model to simulate the execution time of a single task.

B. Coded Software Processing

Coded Processing is the protection of calculations and their results during operations in an arithmetic unit by means of error detection codes. Channel coding, as a part of coding theory, describes error detection and correction codes like the Hamming code, which are originally developed for protecting stored or transmitted data. Another important group of error detecting codes are so-called *arithmetic codes* (AN-code) that are based on ordinary algebra like addition and multiplication. Forin made the first use of coded processing in a real application [11]. He defined coded operations for most arithmetic operations and extended signatures to this kind of code to detect operation, operator and operand errors. A detailed description of arithmetic codes for coded processing can be found in [12].

Since fault-tolerant computer systems have become more important in safety-critical applications, several institutes research this topic using error detecting codes for arithmetic operations. The Laboratory for Safe and Secure Systems at the University of Applied Sciences Regensburg developed, in collaboration with the TU Munich, the Safely Embedded Software (= SES) technique for the programming language C to safeguard the execution of code on microprocessors [13], [14]. Based on SES, an enhanced approach of safe software processing using concurrent task execution is presented in [15]. The basic idea of this approach is that the operating system generates two instances of the same task, which are executed in parallel. This time redundancy in combination with diverse coded data in each instance allows the detection of data errors by comparing both instances at certain points of time, so-called synchronization points. In addition, it is also possible to compare the program counter of both instances. Thus, program flow monitoring is realized and deviations in the program flow are detectable.

III. Reliability Evaluation of Task Processing

A linear program executes several operations consecutively and the data constantly changes. With a constant error rate, the result of an operation is less confident the longer the operation lasts. Usually, the operations of a task depend on each other. The result of one operation is the input for the next. The reliability of a complete task $R(t_{Task})$ has a serial structure and is the product of the reliabilities of the single operations R_1 and R_2.

$$\begin{aligned} R(t_{Task}) &= R_1 \cdot R_2 = e^{-\lambda \cdot t_{op1}} \cdot e^{-\lambda \cdot t_{op2}} \\ R(t_{Task}) &= e^{-\lambda \cdot (t_{op1} + t_{op2})} = e^{-\lambda \cdot t_{Task}} \end{aligned} \tag{1}$$

Equation 1 shows the analytical way to calculate the reliability of a single task and can be derived directly by a simple Markov model consisting of two states (good and failed state) and constant failure rate λ. As mentioned before, the events of hardware faults are exponential distributed in time. For a duplicated task the analytical formula [14] is extended to

$$R(t) = 2 \cdot e^{-\lambda \cdot t} - e^{-2 \cdot \lambda \cdot t}. \tag{2}$$

The problem with Equation 2 is the restrictions in the resolution of numbers in computer systems. In real applications, the error rate λ is a very small number ($\approx 10^{-9} \quad h^{-1}$). The execution time of a task is usually a small number, too. The exponent in the terms of Equation 2 goes to zero and the result of the exponentiation goes to 2 and 1. But the resolution of numbers in computer systems is limited also when using floating point numbers. With a decreasing exponent in Equation 2, the difference between both terms decreases till it is smaller than the resolution of the number system. The subtraction of both terms always leads to 1. The consequence is that an analytical method can not be applied here. In Section III-B, we present a numerical method for the evaluation of the reliability based on an n-staged continuous-time Markov model.

A. Constraints

The first constraint of the following evaluation is the assumption of only sporadic transient faults (="soft errors") that cause data errors during the software processing. In contrast to permanent faults, the recovery of transient faults is possible during the runtime of a task. On the other hand, faults that result in program flow errors can only be detected by a special flag technique [16] or the time redundant execution as described in [15]. Program flow errors and their detection are not the topic of this paper.

The rate of soft errors in integrated circuits exceeds up to 50.000 FIT in worst case (see [17]) and they are more probable than permanent errors. This means that we can assume a fault rate of

$$\lambda = \frac{50.000\ failures}{10^9\ h} = 0.00005\ h^{-1} \qquad (3)$$

for our evaluation of task reliability. This rate is supposed to be constant and the technique of Markov processes can be used. The failure state of a task system is the condition wherein the result of a computation in the task leads to incorrect values and this is not detected by the task. Otherwise, the system is in the operating state, if the result is correct or the error can be detected.
In the course of this section, we do the comparison based on the reliability of

- a single, uncoded task (Section III-B),
- a duplicated task (Section III-C) and
- a coded task for single error detection (Section III-D).

B. Uncoded Task

An uncoded task is a simple task without any additional redundancy. It is executed only once and errors in the result of the task can not be detected. This case will be the reference for later comparison with different realizations of task processing. The reliability model of a simple, uncoded task consists of only one single component that is either working (OK) or there is a fault and the task terminates with incorrect data (NOK). Because of the previously mentioned limitations in the resolution of computer arithmetic, Equations 1 and 2 can not be applied in all cases. The idea is to model the short time of task execution with several stages in the Markov model. The basic Markov model of a non-repairable system consists of two states that are enhanced by n stages. At the end of the stages, there are two absorbing states which represent the termination of the task (Figure 1). The model of Figure 1 is described by the set of $2 \cdot (n+1)$ differential equations:

$$\begin{aligned}
P'_{1.0} &= -\lambda \cdot P_{1.0} - n\mu \cdot P_{1.0} \\
P'_{2.0} &= +\lambda \cdot P_{1.0} - n\mu \cdot P_{2.0} \\
&\ldots \\
P'_{1.k} &= -\lambda \cdot P_{1.k} - n\mu \cdot P_{1.k} + n\mu \cdot P_{1.(k-1)} \\
P'_{2.k} &= +\lambda \cdot P_{1.k} + n\mu \cdot P_{2.(k-1)} - n\mu \cdot P_{2.k} \\
&\ldots \\
P'_{1.n} &= +n\mu \cdot P_{1.(n-1)} \\
P'_{2.n} &= +n\mu \cdot P_{2.(n-1)}
\end{aligned}$$

The solution of the system of differential equations results in the time-dependent probabilities of each state. With increasing complexity of the model, the equations can only be easily solved numerically by means of software tools like MATLAB. The absorbing states of a Markov model represent the state of the system in infinity. In this model with two absorbing states, the task will terminate with certainty either in one or the other state, and the functions $p_{1.n}(t)$ and $p_{2.n}(t)$ describe the probability of being in one of these states. At the beginning, when the system is in state 1.0, the probability of being in one of the final states is zero. With progressing time, the probability of being in state 1.n or 2.n increases rapidly at the end of the execution (see Figure 2).

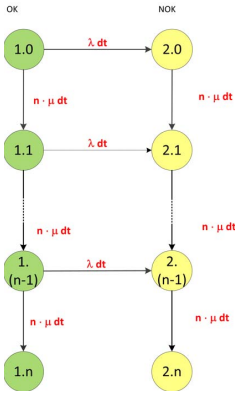

Figure 1. Enhanced Markov model that simulates the runtime of a single task with several stages. The total execution time with $t_{Task} = \frac{1}{\mu}$ is divided into n segments. The transition rate of every stage is therefore $n \cdot \mu$. The task terminates after t_{Task} either without any faults (state 1.n) or with a fault (state 2.n). This kind of task does not offer the possibility of fault detection and repair.

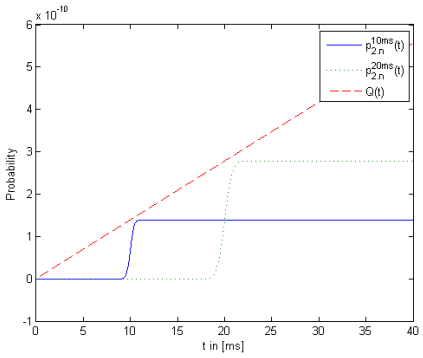

Figure 2. State 2.n represents the failure state which the system enters after termination of the task. The figure shows the probability of the failure state with two different execution times $t_1 = 10ms$ and $t_2 = 20ms$. The steady-state probability of both curves equals the probability of the failure probability calculated by $Q(t) = 1 - e^{-\lambda t}$. This means that the numerical solution by the n-staged Markov model is equivalent to the standard analytical solution.

C. Symmetric Redundant Task

The duplication of components is a common method to increase the reliability of systems. The same technique can be applied to task execution, as well. The task is executed twice either on different cores or sequentially on the same core of a micro-controller. When only transient faults are considered, the duplicated task is equivalent to a repeated execution. The second run of the task (= time redundancy) restarts with correct data like the task would do running on duplicated hardware (= space redundancy). At the end of the second task instance, the results of both are compared and a possible single fault can be detected. There are three possible outcomes for this comparison: (1) both tasks terminate without any faults and the results are equal, (2) only one task terminates with a fault and the two results differ, and (3) the results of both tasks are not correct. In the second case with different outcomes, the fault can be detected. As a result, the task is repeated and there is a low probability that after the repetition one of the tasks will terminate with a fault again. The third case is similar. If there is a different fault in each task, then the results are different, as well. But there is a small probability that both tasks will have the same faulty output and the fault is not detectable. This dangerous condition can cause a system failure in the further run of the software and defines the residue error probability of the system.

What is the probability that the two faulty results are equal? With a register width of b bits, there are b possibilities for a fault in the task result. We assume that the faults in both tasks are independent and the probability of a single fault is equal for all bits. A single fault in one result has b outcomes (one fault in one bit). With two tasks, there are b^2 different probabilities of faulty results. But only b combinations of the two results are equal. Thus, only $d = \frac{1}{b}$-th part of the failure probability results in a dangerous undetected condition.

In our case, we have two equivalent tasks and the system is regarded as failed if both tasks have the same faulty output. The reliability model consists of two parallel nodes that can fail independently. The basic Markov model with three states is enhanced by n stages in the same way as it was done for the single uncoded task.

Description of states:

- State 1.x: No errors have occurred in either task. The results of both tasks are equal.
- State 2.x: One task has a faulty output/result. The results of the two tasks differ and the fault is detectable.
- State 3.x: Both tasks have a faulty output. An error detection is only possible if the two results are different. In case both results have the same erroneous output, the fault can not be detected.

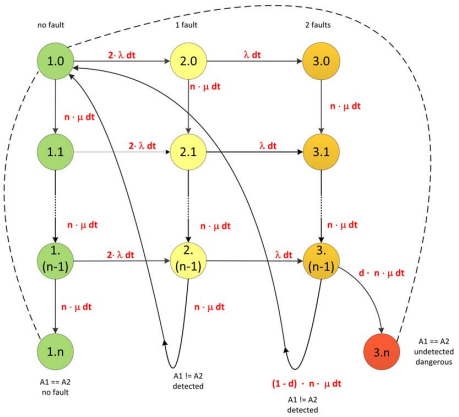

Figure 3. The basic Markov model is that of a two-component, non-repairable system with three states. It is enhanced by n stages to model the execution time of the task. The state 1.n represents the condition when the task terminates without any faults. If a faulty output is detected, then the task is repeated and the model has additional transitions to default state 1.0. There is a small probability that the erroneous output of both tasks is not detectable. In this case, the task will also terminate. This condition represents the dangerous failure state of the task system.

The absorbing states of the Markov model in Figure 3 represents the probability of the task´s outcome. The solution of the set of differential equations leads to the time-dependent probabilities of both ways the task can end:

(1) $p_1(t)$ is the probability of being in state 1.n. This state is entered when no faults have occurred during the runtime of the task. The steady-state of $p_1(t)$ represents the probability of termination in this state:

$$R(t_{Task}) = \lim_{t\to\infty} p_1(t) \tag{4}$$

(2) $p_3(t)$ is the probability of being in state 3.n, which represents the termination of the task with undetected faults. This is the dangerous condition, because the task will terminate with a fault and it could propagate a system failure.

$$Q(t_{Task}) = \lim_{t\to\infty} p_3(t) = p_{failure} \tag{5}$$

In most software applications, a task is executed many times. These periodic tasks are restarted again after their termination. With this assumption of non-stop activity, the absorbing states in Figure 3 are obsolete and the transitions into the absorbing states move to the default state 1.0 instead. The modified model allows us to compute the steady-state probabilities $p_{1.0}, p_{1.1}, ...p_{3.n-1}$ of all states. Three interesting parameters can be calculated with this modified model:

1.) Mean frequency of successful execution of the task:

$$f_1 = p_{1.n-1} \cdot n \cdot \mu \tag{6}$$

2.) Mean frequency of detected unsuccessful execution:

$$\begin{aligned} f_2 &= p_{2.n-1} \cdot n \cdot \mu + p_{3.n-1} \cdot n \cdot \mu \cdot (1-d) \\ &= \left(p_{2.n-1} + (1-d) \cdot p_{3.n-1}\right) \cdot n \cdot \mu \end{aligned} \tag{7}$$

3.) Mean frequency of undetected unsuccessful (dangerous) execution:

$$f_3 = d \cdot p_{3.n-1} \cdot n \cdot \mu \tag{8}$$

The parameter f_3 is perhaps the most interesting one. This frequency represents the average occurrence of task termination with undetected faults. This is the dangerous case and the reciprocal value of f_3 represents the average duration between two failures during non-stop task execution:

$$MTBF = \frac{1}{f_3} \hat{=} \frac{1}{\lambda_{DU}} \tag{9}$$

D. Diverse Redundant Task

Instead of task duplication, coded task processing uses arithmetic error codes for error detection. These diverse representations of data contain additional redundancy that can be used for checking the data validity. The pure time redundancy of the duplicated task is replaced by the more effective information redundancy. The model a) in Figure 4 shows the parallel structure of a duplicated task from the previous section in form of a reliability model. One task is executed twice as instance A and B, either on different cores (space redundancy) or sequentially on the same core (time redundancy). A subsequent comparator C verifies the results at the end of the two task instances. Because of the parallelism of the instances A and B, the probability of a single fault in one of the two tasks is doubled compared with a single task.

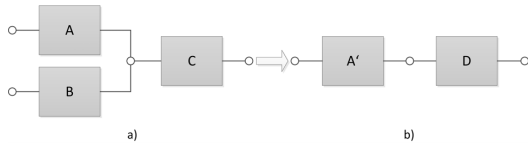

Figure 4. Change of the reliability model: The parallel structure of a duplicated task a) is replaced by the single structure of a coded task b).

In contrast to the doubled execution, the coded task is executed only once. The reliability model consists of a single component A' (see b) in Figure 4) whose probability of a single fault is half compared to the duplicated case. This means that with half execution time the fault rate of the coded task is only half compared to the duplicated case (see fault rate $2\lambda \rightarrow \lambda$ in Figure 3 and 5). But with regard to the error detection capability, both models are equivalent provided that single error detecting codes are used. This allows us a better comparison of both techniques.

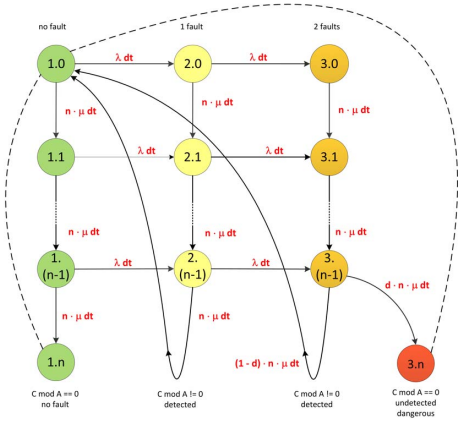

Figure 5. With the given code, all single faults are detectable. This is equivalent to a two-component, non-repairable system which detects all single faults. Because of the fact that there is only one instance of a task, the probability of a single fault is half compared to the duplicated task, whose runtime is twice as long.

Depending on the used error code, the capability of detecting errors varies enormously. We refer to [12], [11], [18], [19]. They describe a lot of codes used for detecting errors in arithmetic operations. When using an error code, there is also a residue error probability of undetection (see state 3.n in Figure 5). It is not scope of this paper to discuss this probability. Interested readers are referred to literature cited before. In this analysis, we assume the same residue error probability as for the duplicated task (for reasons of comparability).

E. Comparison

In the previous sub-sections, we described three different realizations of task execution. A single uncoded task as a reference model is now compared with a duplicated and a coded task. Using the described models in Figure 1,3 and 5, it is possible to calculate the numerical results for a) the probability of undetected failure and b) the mean time between failure (= MTBF). The comparison of these results are summarized in Table I.

The comparison does not consider the increased runtime of coded tasks with coded operations. With increasing runtime, the probability of a fault will increase [14] and of course the probability of undetected faults is higher. However, the technique with coded task is preferable, particularly with the background that error codes can provide better error detection (and correction) capabilities, instead of multiplication of the parallel structure in Figure 4 a).

	$p_{failure}$	MTBF
Simple Task:	$1.388 \cdot 10^{-10}$	$7.200 \cdot 10^{9}\ ms$
Symmetric Redundant Task:	$271.3 \cdot 10^{-21}$	$47.40 \cdot 10^{18}\ ms$
Diverse Redundant Task:	$135.6 \cdot 10^{-21}$	$94.79 \cdot 10^{18}\ ms$

Table I
COMPARISON BETWEEN DIFFERENT REALIZATIONS OF TASK PROCESSING. ($t_{task} = 10ms, n = 8, \lambda = 0.0005\ h^{1}, d = \frac{1}{8}$)

IV. CONCLUSION

The reliability analysis of software systems is becoming more important for evaluation of safety-critical applications. In this paper, we presented a technique based on an enhanced Markov model for investigating the probability of undetected failures during task processing. The analytic solution for this problem is restricted by the limited resolution of number representation in computer systems, especially for tiny failure rates and short task execution times. The presented idea is to model the runtime of a task by an n-staged Markov model to compensate for the restriction mentioned above. The solution of the system of differential equations derived by this enhanced Markov model results in the time-dependent state probability which is the basis for further investigations such as, for example, the steady-state probabilities of a possible outcome of tasks. Further enhancement of the model with the assumption of non-stop execution leads to the computation of the mean time between failure (= MTBF). The comparison of three possible realizations as presented in this paper, shows that the reliability of task processing is increased by additional redundancy, especially information redundancy in form of arithmetic codes (see Table I). In contrast to the described improvements, decreased performance must be considered as an open item. The duplicated task execution requires more resources, either additional hardware or runtime. The coded task also needs more time because of the coded operations and the decoder at the end. Further investigations have to be done for the analysis, whether there is an influence where and how many synchronization points are during the task processing as described in [15].

REFERENCES

[1] P.E. Dodd and L.W. Massengill. Basic mechanisms and modeling of single-event upset in digital microelectronics. *Nuclear Science, IEEE Transactions on*, 50(3):583 – 602, June 2003.

[2] Olga Goloubeva, Maurizia Rebaudengo, Matteo Sonza Reorda, and Massimo Violante. *Software-Implemented Hardware Fault Tolerance*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2006.

[3] Functional safety of electrical/electronic/programmable electronic safety-related systems - part 1: General requirements, April 2010.

[4] N. Oh, S. Mitra, and E.J McCluskey. ED4I: Error Detection by Diverse Data and Duplicated Instructions. *IEEE Transactions On Computers, Vol. 51*, pages 180–199, 2002.

[5] IEEE standard glossary of software engineering terminology, September 1990.

[6] R. Billinton and R.N. Allan. *Reliability evaluation of engineering systems: concepts and techniques*. Plenum Press, 1992.

[7] Magdi S. and Moustafa. Availability of k-out-of-n:g systems with m failure modes. *Microelectronics Reliability*, 36(3):385 – 388, 1996.

[8] K.D. Thies. *Elementare Einführung in die Wahrscheinlichkeitsrechnung, Informationstheorie und stochastische Prozesse*. Shaker Verlag, 2010.

[9] E. Härtter. *Wahrscheinlichkeitsrechnung, Statistik und mathematische Grundlagen: Begriffe, Definitionen und Formeln*. Vandenhoeck & Ruprecht, 1987.

[10] V. Vais and S. Racek. Experimental evaluation of regular events occurrence in continuous-time markov models. In *Informatics, 2011*, Nov. 2011.

[11] P. Forin. Vital coded microprocessor principles and application for various transit systems. In *IFA-GCCT*, pages 79–84. 1989.

[12] Thammavarapu R. N. Rao. *Error coding for arithmetic processors*. Electrical science series. Academic Press, New York and London, 1974.

[13] J. Mottok, F. Schiller, Th. Völkl, and Th. Zeitler. A Concept for a Safe Realization of a State Machine in Embedded Automotive Applications. In *26th Safecomp Conference, ISBN 978-3-540-75100-7*, pages 283–288, 2007.

[14] M. Steindl, J. Mottok, and H. Meier. SES-based framework for fault-tolerant systems. In *Intelligent Solutions in Embedded Systems (WISES), 2010 8th Workshop on*, pages 12 –16, July 2010.

[15] P. Raab, S. Kramer, J. Mottok, H. Meier, and S. Racek. Safe software processing by concurrent execution in a real-time operating system. In *Applied Electronics (AE), 2011 International Conference on*, pages 1 –5, Sept. 2011.

[16] N. Oh, P.P. Shirvani, and E.J. McCluskey. Control-flow checking by software signatures. *Reliability, IEEE Transactions on*, 51(1):111–122, March 2002.

[17] R. Baumann. Soft errors in advanced computer systems. *Design Test of Computers, IEEE*, 22(3):258 – 266, may-june 2005.

[18] P Ozello. The coded microprocessor certifcation. In *International Conference on Computer Safety, Reliability and Security*, pages 185–190. Springer Munich, 1992.

[19] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-encoding: Detecting hardware errors in software. In *Computer Safety, Reliability, and Security*, volume 6351 of *Lecture Notes in Computer Science*, pages 169–182. Springer Berlin / Heidelberg, 2010.

A.4 Isomorphism between Linear Codes and Arithmetic Codes

Authors: P. Raab, V. Vavricka, S. Kraemer and J. Mottok
Accepted : In *Computing and Informatics (CAI)*, February 2013

The following article is in the revised state after acceptance notification. It is ready for re-submission to CAI.

Computing and Informatics (accepted February 2013)

ISOMORPHISM BETWEEN LINEAR CODES AND ARITHMETIC CODES FOR SAFE DATA PROCESSING IN EMBEDDED SOFTWARE SYSTEMS

Peter Raab, Stefan Krämer, Jürgen Mottok

Laboratory for Safe and Secure Systems
Regensburg University of Applied Sciences
Faculty of Electronics and Information Technology
Seybothstr. 2, D-93053 Regensburg, Germany
e-mail: {peter.raab, stefan.kraemer, juergen.mottok}@hs-regensburg.de

Vlastimil Vavricka

University of West Bohemia
Faculty of Applied Sciences
Univerzitni 8, 306 14 Plzen, Czech Republic
e-mail: vavricka@kiv.zcu.cz

Abstract. We present a transformation rule to convert linear codes into arithmetic codes. Linear codes are usually used for error detection and correction in broadcast and storage systems. In contrast, arithmetic codes are very suitable for protection of software processing in computer systems. This paper shows how to transform linear codes protecting the data stored in a computer system into arithmetic codes safeguarding the operations built on this data. The combination of the advantages of both coding mechanisms will increase the error detection capability in safety critical applications for embedded systems by detection and correction of arbitrary hardware faults.[1]

Keywords: coding theory, linear codes, arithmetic codes, code transformation, residue error probability, Safely Embedded Software (SES)

[1] Mathematics Subject Classification 2010: 94B05, 94B40, 11T71, 14G50

1 INTRODUCTION

The complexity and functionality of electronic controlled units have increased more and more in several sectors of industry during recent years (e.g. automotive, aeronautics). In addition, the requirements of these systems have become more demanding in terms of safety, reliability and availability. In contrast to this progress, industry demands a decrease in costs for electronics, while at the same time remaining competitive. The use of inexpensive commodity hardware is the result. However, the development of present micro-controllers follows the trend of decreasing feature size that leads to less reliability; arbitrary hardware faults are more likely [1]. Increasing the fault tolerance of unreliable hardware is often a requirement in safety critical applications. The consequence is the use of redundant hardware or of diverse data [2, 3, 4]. The Laboratory for Safe and Secure Systems at the University of Applied Sciences Regensburg developed, in collaboration with the TU Munich, the Safely Embedded Software technique for the programming language C to safeguard the execution of code on microprocessors [5, 6, 7, 8].

Modern broadcast and storage systems follow similar strict requirements for reliability and safety. Disturbing influences can corrupt transmitted or stored data in the same way they do during computation in a computer system. Techniques of error detection and correction have been studied since Hamming researched codes for increasing fault tolerance in storage systems the first time [12]. Based on this, a lot of improvements for error detection and correction were made: Cyclic Linear Block Codes [13], Bose-Chaudhuri-Hocquenghem (= BCH) codes [14] and Reed-Solomon codes [15], just to mention the most important ones. But, linear codes are not the optimal solution for coded data processing because they do not preserve the code after arithmetic operations [11]. For an optimal error detection capability, code transformations can be a possible solution for further improvements also in coded data processing systems with memories and bus which have similar characteristics a storage and transmission systesm. For this reason, this article presents the required background (Section 2) for a method to transform linear codes into arithmetic codes (Section 3) and the performance in Section 4.

2 BACKGROUND

The ISO 26262 norm, "Road vehicles - Functional Safety", recommends several diagnostic techniques to detect possible occurring errors which are state-of-the-art. In Table ISO 26262-5, D.4 [16], there are only two techniques with high diagnostic coverage (DC) enumerated. Due to this norm, the highest DC is achievable only with coded processing (like the SES framework) and reciprocal comparison by software. The former executes the code in a transformed domain. Permanent errors and transient errors are detected by a check of the validity of the code words. Based on the normative approach, a closer look at coding techniques is valuable to get deeper insights.

An important metric for comparison of different codes is the residual error probability. This is the probability that a received code word is corrupted but no errors are detected by the decoding algorithm. This is the case when the erroneous bits in the received code word itself form a valid code word [20]. Based on a binary symmetric channel model (= BSC, see Figure 1), the analysis of linear block codes and arithmetic codes (so-called AN code) shows that the probability for undetected errors is greater for AN codes than for linear codes with same code rate [11].

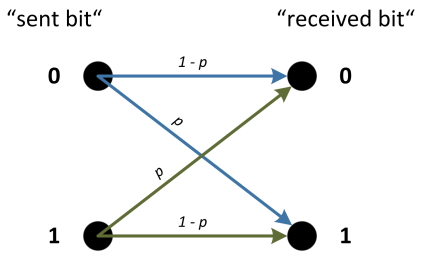

Fig. 1. The binary symmetric channel model describes the probability p that a single bit changes its value or remains unchanged $(1-p)$. There is no dependency between two nearby bits. In contrast, there are other channels with memory described in [21].

The BSC model is valid for channels where the single bits have no influence on others. Thus, linear codes are excellent for protecting single-bit errors in data storage and transmission, whereas the underlying channel model for arithmetic operations has a kind of memory. The carry bit propagation of an addition has a direct influence on nearby bits of a code word. Figure 2 shows a simple model of a computing system. It consists of a data storage unit and a transmission bus for which the BSC model is valid. Otherwise the BSC model can not be used for the *arithmetic logical unit* (= ALU). An applicable channel model for the ALU must be developed in future work.

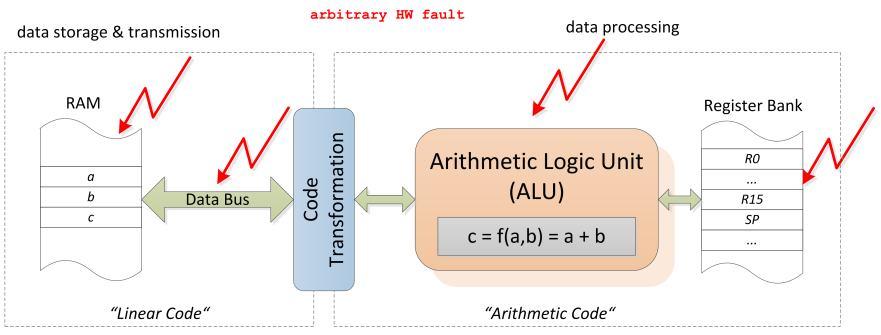

Fig. 2. Simplified CPU model that shows the hardware components propagating arbitrary hardware faults. Using an adequate code for the different channel models, a transformation of the code is required.

Remark:
IBM developed a code, which is also called *Arithmetic Code*, for lossless data compression [17]. But not to be confused, this arithmetic encoding is part of entropy encoding whereas the described AN codes are an example of channel encoding and error detecting codes.

2.1 Algebra of Codes

Algebraic structures are the background for most error detecting and correcting codes. An algebraic structure consists of a set of objects (e.g. numbers) and at least one operation applied to this set [22]. Two important codes are the linear codes and arithmetic codes, which have different algebraic structures [11]. An arithmetic code is a set of code words which are the product of two integer numbers X (= original number) and A (= constant generator).

$$C_{AN} := \{A \cdot X \mid A, X \in \mathbb{Z}\} \tag{1}$$

A finite set of numbers is more important for computer arithmetic because of the limited register width of k bits in a micro-controller. The code words are out of the set of all multiples of A, but smaller than the possible range of $M = 2^k$. Such a subset of integer numbers is called an *ideal* $A\mathbb{Z}_M$ in algebra. This finite subset forms a ring under the two operations, addition and multiplication $(A\mathbb{Z}_M, +_M, \cdot_M)$. The sum and the product of two code words are divisible by the generator A and the result is therefore a valid code word (see axiom of closure [22]). Linear codes use another algebraic structure than arithmetic codes [19]. Instead of integer numbers, the structure of linear codes consists of a more complex set of polynomials. A polynomial is a different representation of a vector in a k-dimensional vector space [22]. The coefficients of a polynomial are the digits of a number described by a positional notation system. In a computer system, the number system is based on the finite field of order two, the so-called Galois Field *GF(2)* or $\mathbb{Z}_2 := \{0, 1\}$. Thus an integer number X is represented by the set of k binary digits in a computer system

$$\vec{x} = (x_{k-1}, x_{k-2}, ...x_1, x_0)$$

with $x_i \in \mathbb{Z}_2$, or in the polynomial representation:

$$\begin{aligned} x(z) &= x_{k-1} \cdot z^{k-1} + x_{k-2} \cdot z^{k-2} ... + x_1 \cdot z^1 + x_0 \\ x(z) &= \sum_{i=0}^{k-1} x_i \cdot z^i \end{aligned} \tag{2}$$

The set of polynomials with coefficients out of the finite field $\mathbb{Z}_2$

$$\mathbb{Z}_2[z] := \{p(z) = \sum_{i=0}^{k-1} x_i \cdot z^i \mid x_i \in GF(2)\} \tag{3}$$

forms a ring of polynomials and under the two operations:

$$a(z) \oplus b(z) := c(z) \tag{4}$$
$$\text{with } c_j = (a_j + b_j) \mod 2 \qquad \text{and } 0 \leq j < k$$

$$a(z) \odot b(z) := c(z) \tag{5}$$
$$\text{with } c_j = \left(\sum_{i=0}^{j} a_i \cdot b_{j-i}\right) \mod 2 \qquad \text{and } 0 \leq j < k$$

Linear codes have a coding rule similar to that of arithmetic codes. Both codes are generated by the multiplication based on their algebraic structure. This is the ordinary integer multiplication for arithmetic codes and the polynomial multiplication for linear codes.

$$C_{CRC} := \{g(z) \odot x(z) \mid g(z),\ x(z) \in \mathbb{Z}_2[z]\} \tag{6}$$

2.2 Systematic Codes

Systematic codes are known from linear codes in communication systems. They consist of k bits of information that is separated into the $n-k$ bits of parity in their binary representation [19]. Arithmetic codes can also be in a systematic form (see Figure 3).

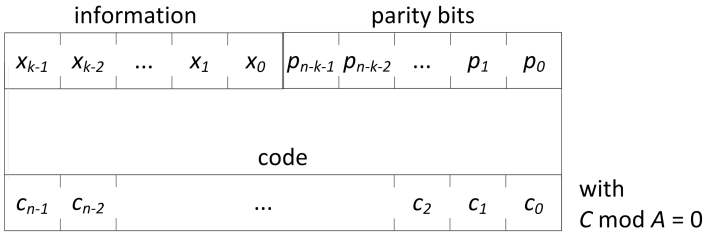

Fig. 3. The information of a systematic code word is separated into two segments. The original number can be read directly from the code word. The parity bits are added after. The code word itself remains a multiple of A.

But for systematic arithmetic codes, the code is not separable. The addition of two systematic code words propagates possible carry bits into the information part. The separated information of the result does not match the sum of both information words (see Section 2.3). Systematic encoding is the basic principle for the

code transformation described in Section 3. Rao already showed parallels between arithmetic codes and linear block codes (see Table 1) [18].

Linear Block Codes	**Arithmetic Codes**
$c(z) = g(z) \odot x_2(z) = x_1(z) \odot z^{n-k} \oplus r(z)$	$C = A \cdot X_2 = X_1 \cdot 2^{n-k} + R$
with:	*with:*
$c(z)$: code polynomial	C : coded integer
$g(z)$: generator polynomial	A : generator integer
$x_1(z)$: information polynomial	X_1 : information integer
$r(z)$: remainder polynomial	R : remainder integer

n : length of code word in bits
k : length of information word in bits
$n - k$: number of redundant bits

Table 1. Comparison between systematic linear and arithmetic codes.

When analyzing the coding rules (Table 1) for both codes, we can see that $X_1 \neq X_2$ and $x_1(z) \neq x_2(z)$. It is clear that X_2 must be greater than X_1 to fulfill the equation. When coding a number, the terms X_1 and $x_1(z)$ represent the original values. Therefore, X_2 and $x_2(z)$ are not important and need not be considered.

2.3 Coded Operations

The transformation of one algebraic structure into another one is called *homomorphism*. In general, a homomorphism maps one algebraic structure into the other. Let $(G, +)$ and $(H, \oplus)$ be two algebraic structures and φ the map function of the sets $G \rightarrow H$, then it must be $\varphi(x + y) = \varphi(x) \oplus \varphi(y)$. If there is a bijective homomorphism between $(G, +)$ and $(H, \oplus)$, then both structures are isomorphic [22]. The addition of polynomials differs from that of an ordinary addition. A polynomial addition must be enhanced in a way that the above rule for a homomorphism is satisfied. The coefficients of the polynomial are elements out of the Galois Field *GF(2)* and possible carry bits are ignored. This difference between the two operations results in different outcomes and makes a correction $c(z)$ necessary. For the addition of two integers, it is

$$\varphi_+ : \quad X + Y \rightarrow x(z) \oplus y(z) \oplus c(z). \tag{7}$$

In [11], it was shown that the coded operation which is used for coded software processing [6, 4] is a homomorphism. It defines operations of code words in such a way that the result of this operation matches the coded result of the original information word.

Example: For the addition of two arithmetic coded numbers, it is

$$+_c : \ \varphi_+(C_1, C_2) := C_1 + C_2 \tag{8}$$

for all $C \in C_{AN}$. This means that no correction is necessary in the case of the addition of two code words (in contrast to coded multiplication [11]). For systematic encoded numbers, the sum of two coded numbers is also a systematic code word.

$$\begin{aligned} C_1 + C_2 &= X_1 \cdot 2^{n-k} + R_1 + X_2 \cdot 2^{n-k} + R_2 \\ &= (X_1 + X_2) \cdot 2^{n-k} + (R_1 + R_2) \end{aligned} \tag{9}$$

With an ordinary adder unit in an ALU, there is the problem of an overflow of the sum of the remainders R_1 and R_2. The carry bit will be propagated into the information X of a systematic code. The result will be a valid code word, because the result remains a multiple of A, but this carry bit propagation changes the value of the information X. A special version of an adder must be used. Both parts, the information and the remainder, must be handled separately (Figure 4).

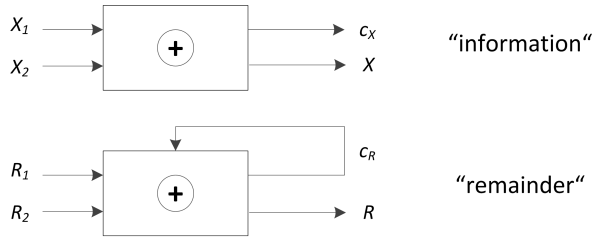

Fig. 4. Special adder unit for addition of systematic arithmetic codes.

When there is an overflow detected in the remainder unit, the carry bit must not be propagated to the information part. But then the remainder does not correspond with the resulting information. The information number is not increased by the propagated carry-bit.

Remark:
Rao and previously Garner have described another form of systematic codes. The so-called gAN code does not require special adders. For detailed information see [18].

3 CODE TRANSFORMATION

In Section 2, we saw a close similarity between linear and arithmetic codes. Obviously, there is a simple rule to transform them to each other. This section shows what the generator integer A and the polynomial $g(z)$ must look like for a transformation. Let us begin with one important theorem from Rao [18].

Theorem 1. A systematic AN code has the form $C = X_1 \cdot 2^{n-k} + R = A \cdot X_2$ if and only if $R = (-X_1 \cdot 2^{n-k}) \bmod A$ with $2^{n-k-1} < A < 2^{n-k}$.

Following Theorem 1, the remainder of a systematic AN code can be generally expressed as

$$\begin{aligned} C &= X_1 \cdot 2^{n-k} + R = A \cdot X_2 \\ \Rightarrow R &= A \cdot X_2 - X_1 \cdot 2^{n-k}. \end{aligned} \tag{10}$$

With $X_2 = X_1 + 1$, the remainder R of a systematic code is

$$R = A - X_1 \cdot (2^{n-k} - A). \tag{11}$$

The term $2^{n-k} - A$ in Equation 11 is 1, if it is $A = 2^{n-k} - 1$. The remainder can be simplified to

$$R = A - X_1. \tag{12}$$

The remainder is always positive ($R \geqq 0$). Consequently, it is $X_1 \leqq A$ and the range of X_1 depends on A. If the generator A is of the form $2^{n-k} - 1$ (all bits are 1), the remainder of the systematic code words are decreasing numbers beginning with A (see example in Table 3).

Because of the homomorphism between the algebra of arithmetic codes and the algebra of linear codes (see Section 2), the integer numbers and the operations of Equation 11 are substituted as seen in Table 2:

R	$\rightarrow$	$r(z):$	polynomial of remainder
X_1	$\rightarrow$	$x_1(z):$	polynomial of information word
A	$\rightarrow$	$a(z):$	polynomial of generator
$X_1 - X_2$	$\rightarrow$	$x_1(z) \oplus x_2(z) \oplus c(z):$	subtraction of polynomials with $c(z)$ describes the borrow bits
$X_1 \cdot X_2$	$\rightarrow$	$x_1(z) \odot x_2(z) \oplus c(z):$	multiplication of polynomials with $c(z)$ describes the carry bits

Table 2. Transformation rules from integer numbers to polynomials

The three terms of Equation 11 are substituted step by step now:

1. The subtraction $(2^{n-k} - A)$ causes a borrow bit in every consecutive digit. The borrow bits $c(z)$ are the same as $a(z)$ but shifted by one to the left. The term $c(z)$ can be replaced by $a(z) \odot z$.
2. The multiplication $X_1 \cdot (..)$ can be simplified. The subtraction within the brackets always results in 1. A multiplication of a polynomial with 1 does not generate any carry bits. The term $c(z)$ for this multiplication can be ignored.
3. The last subtraction $A - X_1 \cdot (..)$ is also a special case. If $A = 2^{n-k} - 1$, then all $n - k$ coefficients of the polynomial $a(z)$ are equal to 1. The subtraction of any polynomials of the same or smaller order (that is, $X \leqq A$, see definition above) does not result in any borrow bits. Consequently, c(z) can be ignored, too.

It follows

$$\Rightarrow r(z) = a(z) \oplus x_1(z) \odot \{z^{n-k} \oplus a(z) \oplus a(z) \odot z\}. \tag{13}$$

Expanding the brackets and reordering Equation 13 leads to:

$$\begin{aligned} r(z) &= a(z) \oplus x_1(z) \odot z^{n-k} \oplus x_1(z) \odot a(z) \oplus x_1(z) \odot a(z) \odot z \\ r(z) &= a(z) \oplus x_1(z) \odot z^{n-k} \oplus a(z) \odot z \odot \{x_1(z) \odot z^{-1} \oplus x_1(z)\} \end{aligned} \tag{14}$$

The term $x_1(z) \odot z^{-1} \oplus x_1(z)$ of Equation (14) is the binary XOR operation of $x_1(z)$ with itself but shifted by one bit to the right. If $x_2(z) = x_1(z) \odot z^{-1} \oplus x_1(z)$ and x_{1_0} is the least significant bit of $x_1(z)$, then the term in brackets of Equation 14 can be replaced by

$$x_1(z) \odot z^{-1} \oplus x_1(z) = x_2(z) \oplus x_{1_0} \odot z^{-1}. \tag{15}$$

The right shift of $x_1(z)$ removes the least significant bit out of the integer number. But it must not be ignored and it follows

$$\begin{aligned} r(z) &= a(z) \oplus x_1(z) \odot z^{n-k} \oplus a(z) \odot z \odot x_2(z) \oplus a(z) \odot x_{1_0} \\ &\quad \text{with} \qquad g(z) = a(z) \odot z \\ \Rightarrow r(z) &= a(z)\{1 \oplus x_{1_0}\} \oplus x_1(z) \odot z^{n-k} \oplus g(z) \odot x_2(z) \\ \Rightarrow c(z) &= g(z) \odot x_2(z) \\ &= x_1(z) \odot z^{n-k} \oplus a(z) \odot \underbrace{\{1 \oplus x_{1_0}\}}_{=1, if\ x_{1_0}=0} \oplus r(z) \end{aligned} \tag{16}$$

If the least significant bit $x_{1_0} = 0$, then $a(z)$ is not reduced and it inverts the remainder polynomial $r(z)$ compared to R. Table 3 shows an example of this effect of the isomorphism between linear block codes and arithmetic codes.

	X_1	R	$r(z)$
1	00001	11110	11110
2	00010	11101	00010
3	00011	11100	11100
4	00100	11011	00100
5	00101	11010	11010
6	00110	11001	00110
...	...	...	...

Table 3. Isomorphism between an arithmetic and linear (15,10) code with $A = 31 = 2^5 - 1$ and $g(z) = a(z) \odot z = z^5 + z^4 + z^3 + z^2 + z$. Every second remainder R is inverted compared to $r(z)$. With both codes, the information is extended by $n - k = 5$ bits.

4 PERFORMANCE ESTIMATION

In the following part of the paper, an estimation for resource consumption and error performance of the given approach is presented.

4.1 Error Detection Capability

The error detection capability describes the performance of the code to detect bit flips. The presented linear block code that is generated by the polynomial of the form $g(z) = a(z) \odot z = \sum_{i=0}^{n-k-1} 1 \cdot z^{i+1}$ results in a code with a minimum hamming distance of only two. This means that only single bit errors can be detected but not corrected. The polynomial of the described form is indeed an irreducible polynomial, but not primitive [19]. Only primitive polynomials generate codes with a minimum hamming distance greater than two.

The range of the information X is limited to A. The required bits for the remainder R are the same as for the information part of the code. The code rate, which is a metric for the redundancy of a code [23], is

$$R = \frac{k}{n} = \frac{5}{10} = 0.5 \tag{17}$$

for the example in Table 3. This means that only half of every code word represents any information and the rest is redundancy. A comparison with rates of other codes shows that the efficiency of this code is not ideal. There are codes with a rate of 0.5 which have a minimum hamming distance of more than two, and an error correction is possible in this case.

4.2 Runtime Evaluation of Check

The verification of a code word is the evaluation of the remainder

$$R = C \mod A$$

which must be zero in the case of no errors. The division operation in a microcontroller consumes a lot of runtime. But Rao [18] described a class of useful codes for error detection when selecting $A = 2^{n-k} - 1$. In this case, the evaluation of the remainder can be simplified and the modulo operation is substituted by a more simple addition.
Let $\vec{c} = \{c_{n-1}, c_{n-2}, ...c_1, c_0\}$ be the binary representation of a code word. These bits are partitioned into l segments of length $n - k$. If these segments are $B_{l-1}...B_1, B_0$, then the modulo A of the sum of all segments B_j equals the modulo of the code word C with

$$C \mod A = \sum_{j=0}^{l-1} B_j \mod A. \tag{18}$$

The length of one segment B_j is $n-k$ bits. The sum is also restricted to $n-k$ bits. A possible overflow must be added to the sum.

EXAMPLE:

Let $C = 219$ be the corrupted systematic code word for $X = 6$ with $A = 2^5 - 1 = 31$. The binary representation for C is 0011011011_b and it follows that there are two segments with $B_0 = 6 = 00110_b$ and $B_1 = 27 = 11011_b$. If b is the overflow bit of the sum of both segments $B_0 + B_1$ of size $n-k$ bits each, then $B_0 + B_1 + b = 2 = 219 \mod 31$. The remainder is not equal 0 and an error is detected. Let C now be 217 without errors. Then $B_0 = 6 = 00110_b$ and $B_1 = 25 = 11001_b$. The sum is $B_0 + B_1 = 6 + 25 = 31$ and equals the generator A. In this case, the modulo is always zero and no errors have occurred. The evaluation for errors is reduced to the sum $\sum_{j=0}^{l-1} B_j$ and a following comparison with 0 or A.

5 CONCLUSION

We saw that the generator A is of the form $2^c - 1$, a simple transformation between linear and arithmetic codes and vice versa is possible (see Section 3). Secondly, the evaluation of error occurrence can be simplified to an addition instead of a division. This makes the evaluation for an error more efficient compared to the standard method with a modulo operation. But on the other hand, there are the disadvantages of the error correction probability and the code rate. The approach of a transformation between linear and arithmetic codes introduced here results in a code rate of 0.5 and a minimum hamming distance of only two. However, this type of code required for the isomorphism is not as efficient as other codes. An improved error performance is only possible when the same software is concurrently executed in two redundant coded channels. As described in [10], one single fault can also be corrected.

Furthermore, the transformation of codes can be useful in coded data processing. According [11], the simplified processor model consists of several channels with different characteristics. BSC based operations like *MOV* or *XOR* match linear codes and have a better residual error probability than comparable arithmetic codes (see also [11]). In contrast, linear codes are not practical for arithmetic operations like the addition (*ADD*) because of the missing carry-bit propagation [24]. Thus, the presented code transformation has advantages for coded data processing if different underlying error models are considered.

REFERENCES

[1] Dodd, P. E.—Massengill, L. W.: Basic mechanisms and modeling of single-event upsets in digital microelectronics. *Nuclear Science, IEEE Transactions on*, Vol. 50, June 2003, No. 3, pp. 583 - 602.

[2] Forin, P.: Vital coded microprocessor principles and application for various transit systems. In *IFA-GCCT*, pp. 79 - 84, 1989.

[3] Oh, N.—Mitra, S.—McCluskey, E. J.: ED4I: Error Detection by Diverse Data and Duplicated Instructions. *IEEE Transactions On Computers*, Vol. 51, 2002, pp. 180 - 199.

[4] Schiffel, U.—Schmitt, A.—Süsskraut, M.—Fetzer, C: ANB- and ANBDmem-encoding: Detecting hardware errors in software. In *Computer Safety, Reliability, and Security*, Volume 6351 of Lecture Notes in Computer Science, pp. 169 - 182. Springer Berlin / Heidelberg, 2010.

[5] Mottok, J.—Schiller, F—Völkl, Th.—Zeitler, Th.: A Concept for a Safe Realization of a State Machine in Embedded Automotive Applications. In *26th Safecomp Conference, ISBN 978-3-540-75100-7*, pp. 283–288, 2007.

[6] Mottok, J.:Safely Embedded Software - A Safety Framework for C++. In *Embedded Software Engineering Report*, 2008.

[7] Meier, H.—Schiller, F—Steindl, M—Mottok, J.—Früchtl, M.: Diskussion des Einsatzes von Safely Embedded Software in FPGA-Architekturen. In *Proceedings of the 2nd Embedded Software Engineering Congress*, 2009.

[8] Steindl, M.: Safely Embedded Software (SES) im Umfeld der Normen für funktionale Sicherheit. *Jahresrückblick 2009 des Bayerischen IT-Sicherheitsclusters*, 2009.

[9] Brown, D. T.: Error Detecting and Error Correcting Binary Codes for Arithmetic Operations. In *IRE Trans. Electron. Comput.*, pp. 333 - 337, 1960.

[10] Raab, P.—Krämer, S—Mottok, J.—Meier, H.—Racek, S.: Safe software processing by concurrent execution in a real-time operating system. In *Proceedings of 16th International Conference on Applied Electronics*, pp. 315 - 319, September 2011.

[11] Raab, P.—Krämer, S—Mottok, J.: Cyclic Codes and Error Detection during Data Processing in Embedded Software Systems. In *Proceedings of the 4rd Embedded Software Engineering Congress*. pages 577-590. December 2011.

[12] Hamming, R. W.: Error Detecting and Error Correcting Codes. *Bell System Technical Journal*, No. AFCRC-TN-57-103, April 1950.

[13] Prange, E.: Cyclic Error-Correcting Codes in Two Symbols. *Technical Report, Air Force Cambridge Research Center*, September 1957.

[14] Bose, R. C.—Ray-Chaudhuri, D. K.: On a class of error correcting binary group codes. In *Information and Control*, Vol. 3, No. 1, pp. 68 - 79, 1960.

[15] Reed, I. S.—Solomon, G.: Polynomial codes over certain finite fields. In *Journal of the Society for Industrial and Applied Mathematics*, Vol. 8, No. 2, pp. 300 - 304, 1960.

[16] ISO: ISO/DIS 26262-5: road vehicles - functional safety - part 5: Product development: hardware level, 2009.

[17] Rissanen J. J.: Generalized kraft inequality and arithmetic coding. In *IBM Journal of Research and Development*, 20(3):198–203, May 1976.

[18] Rao, T. R. N.: Error coding for arithmetic processors. *Electrical Science Series.* Academic Press, New York and London, 1974.

[19] Peterson, W. W.—Brown, D.T.: Cyclic codes for error detection. In *Proceedings of the IRE*, Vol. 49, No. 1, pp. 228 -235, January 1961.

[20] Robert H. Morelos-Zaragoza: The Art of Error Correcting Coding. John Wiley & Sons, Ltd, 2006.

[21] Osmann, C.: Bewertung von Codierverfahren für einen störungssicheren Datentransfer - Evaluation of error-correcting codes used for a reliable data transfer. PhD thesis Universitaet Duisburg-Essen, Campus Duisburg, 2001.

[22] Beutelspacher, A.: Lineare Algebra, Vieweg, 1994.

[23] Bossert, M.: Kanalcodierung, Teubner, 1998.

[24] Raab, P.—Krämer, S—Mottok: Reliability of Data Processing and Fault Compensation in Unreliable Arithmetic Processors. ***Submitted*** in *Microprocessors and Microsystems: Embedded Hardware Design (MICPRO).*

A.5 Error Model and the Reliability of Arithmetic Operations

Authors: P. Raab, S. Kraemer and J. Mottok
Published: In *IEEE 2013 EUROCON - International Conference on Computer as a Tool*, pages 630-637, July 2013.
ISBN-13: 978-1-4673-2231-7

EuroCon 2013 • 1-4 July 2013 • Zagreb, Croatia

Error Model and the Reliability of Arithmetic Operations

Peter Raab [1], Stefan Krämer [2], Jürgen Mottok [3]

Laboratory for Safe and Secure Systems,
Faculty of Electronics and Information Technology,
Regensburg University of Applied Sciences
Seybothstr. 2, D-93053 Regensburg, Germany
[1] peter.raab@hs-regensburg.de
[2] stefan.kraemer@hs-regensburg.de
[3] juergen.mottok@hs-regensburg.de

***Abstract*—Error detecting and correcting codes are widely used in data transmission, storage systems and also for data processing. In logical circuits like arithmetic operations, arbitrary faults can cause errors in the result. However in safety critical applications, it is important to avoid those errors which would lead to system failures. Several approaches are known to protect the result of operations during software processing. In the same way like transmission systems, coded processing uses codes for fault detection. But in contrast to transmission systems, there is no adequate channel model available which makes it possible to evaluate the residue error probability of an arithmetic operation in an analytical way. This paper tries to close the gap of arithmetic error models by the development of a model for an ordinary addition in a computer system. Thus, the reliability of an addition´s result can be analytically evaluated.**

***Keywords*—addition, channel model, coded processing, fault simulation, Markov model,residue error probability**

I. Introduction

The complexity and functionality of electronic control units have increased more and more in several sectors of industry. In addition, the requirements of these systems have become more demanding in terms of safety, reliability and availability. In contrast to this progress, industry demands a decrease in costs for electronics, while at the same time remaining competitive. The use of inexpensive commodity hardware is the result. However, the development of current micro-controllers follows the trend of decreasing feature size. That leads to less reliability and arbitrary hardware faults are more likely [1]. But despite unreliable hardware, fault tolerance is a requirement of safety-critical applications. Reference [2] summarizes the state of the art techniques of Software-Implemented-Hardware-Fault-Tolerance (= SIHFT). One simple possibility of hardening the data against *single event upset* (= SEU [3]) is the duplication (= data redundancy) and the multiple computation of these data (= time redundancy). But only transient faults can be detected by pure data redundancy. Permanent faults in the *central processing unit* (= CPU), e.g. stuck-at fault in the adder hardware, will generate the same erroneous result. The consequence is the use of redundant hardware or of diverse data, so that different units of the CPU are used. A lot of approaches for *coded processing* use coded data, which is simply the multiplication with a constant factor. This idea bases on so-called arithmetic codes (or AN-codes) which are introduced by Brown the first time [4]. Forin made use of this AN-codes to protect the calculation of data in real applications [5]. He defined coded operation like addition and multiplication to detect errors in the operator, operand and the operation itself of a single instruction in a program. References [6] and [7] follow similar approaches. The usage of codes cannot guarantee that every fault can be detected. It is known from transmission systems that there is a small chance for each code not detecting an error. This residue error probability is either derived from the hardware network [8] or is evaluated by experiments [9]. Until now, all known approaches of coded processing report residue error probabilities that are evaluated by simulated fault injection techniques [10], [11], [2]. Because of the missing error models of arithmetic operations, it was not possible to determine a probability of undetection [12]. Thus, this paper presents the approach of determining an error model for one arithmetic operation in Section III and their verification by simulation in Section V. The paper intends to show a possible way for the analytical evaluation of the residue error probability of a single arithmetic operation using arithmetic codes (Section IV).

II. Background

A channel model is an important mean in the information theory to approximate the behavior of real noisy transmission channels by a probabilistic approach. For example, a simple channel model is the so-called *binary symmetric channel* (= BSC) in Figure 1.

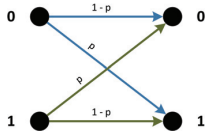

Fig. 1. The binary symmetric channel model describes the probability p that a single bit changes its value or remains unchanged $(1 - p)$.

978-1-4673-2232-4/13/$31.00 ©2013 IEEE

This model allows us to calculate the residue error probability of error detecting codes used for a simple data transmission [13], [14]. The residue error probability describes the chance for a received code word to be corrupted in a way that it is again a valid code word after reception. The error in a transmitted code word is not detected in this case.

The data processing in the arithmetic unit of a microcontroller also represents a kind of channel. During the software processing, arbitrary hardware faults (permanent or transient) occur and change the value of a result. Similar to transmission systems, a lot of error detecting and correcting codes are presented for arithmetic operations [4], [15], [16]. But in contrast to BSC model, an arithmetic operation is usually more complex with at least one input and the result is a function of these inputs (Figure 2). The used error code must preserve the result of the operation as a valid code word.

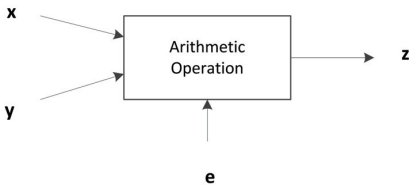

Fig. 2. An arithmetic operation usually processes at least one input and generates an output as a function of the inputs $\big(z = f(x, y, e)\big)$.

III. Error Model of an Adder

The literature does not report any channel models for processors because of its complexity. Only the manufacturers would be able to develop an appropriate model for their processors [12]. But with a certain level of abstraction, it is possible to derive a probabilistic model for any block in a micro-controller (e.g. memory, bus line or adder). For example, the probability of incorrect outputs of a hardware block in a processor system was evaluated by [8]. Thereby, they assume equally distributed input patterns of the hardware and evaluate all possible outcomes with respect to faults that can occur in the considered gate. This approach requires a broad knowledge of the hardware which is not always available in detail. But the example of the binary symmetric channel for transmission systems shows that a simplified error model describes the behavior of a system without the knowledge of the detailed mechanisms of the upsets. The upset in a transmission channel depends on different parameters like distance, position etc. which are not handled by the BSC model. It is merely a probabilistic model that describes the chance of a bit-flip derived by empirical data. A further approach is known from the *discrete memoryless multiple-access channel* (DMMAC)[1] [17], [18]. Basically, this channel model describes the concurrent transmission of several transmitters to one common receiver via a noisy channel. However, this model can be adapted because it corresponds to most logical gates that maps several inputs to one (or several) output(s).

[1]Sometimes called as *binary adder model*

A. State Transition of an Addition

The *ripple carry adder* as an often used operation is an iterative network that consists of several cascaded elements[2]. The full adder as the smallest logical element of the addition receives the the carry-bit from the previous adder c_{in} and calculates the result out of c_{in} and two summands x and y as inputs. The output of a full adder is the sum-bit s and the carry-bit c_{out} for the next. Figure 3 shows an n-bit adder built-up from single adder elements, which are also called stages in the further course of this paper.

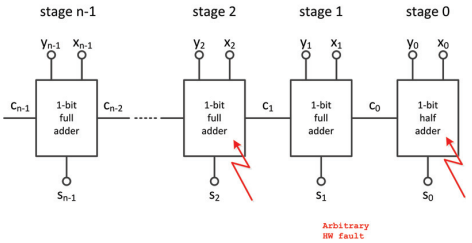

Fig. 3. A standard network for the addition of two n-bit numbers X and Y consists of n consecutive adder elements. (1 half adder / n-1 full adders)

The behavior of a single full adder can be characterized by the states of all inputs and outputs. Every input and output can take one value out of its alphabet:

- Input alphabet:
  $\mathbf{X} = \{0, 1\}$, $\mathbf{Y} = \{0, 1\}$, $\mathbf{C_{in}} = \{0, 1\}$
- Output alphabet:
  $\mathbf{S} = \{0, 1\}$, $\mathbf{C_{out}} = \{0, 1\}$

The input alphabet $\mathbf{X}, \mathbf{Y}, \mathbf{C_{in}}$ defines the finite number of input states with

$$m_{in} = |\mathbf{X}| \cdot |\mathbf{Y}| \cdot |\mathbf{C_{in}}| = 8,$$

whereas the output alphabet $\mathbf{S}, \mathbf{C_{out}}$ defines the number of output states with

$$m_{out} = |\mathbf{S}| \cdot |\mathbf{C_{out}}| = 4.$$

In general, there are $m_{in} \cdot m_{out}$ possible combinations of all states. But the behavior of the addition only allows a unique subset of them. So, there is only one correct result which is mapped to a certain input (= valid assignment). However, several different inputs can lead to the same result (= surjective mapping [19], see Figure 4).

In case of a fault with probability p, the result of the addition probably changes and becomes incorrect (= invalid

[2]This is the simplest form of an adder as an example. In real applications, there are more different networks for an adder available (e.g. carry-look-ahead).

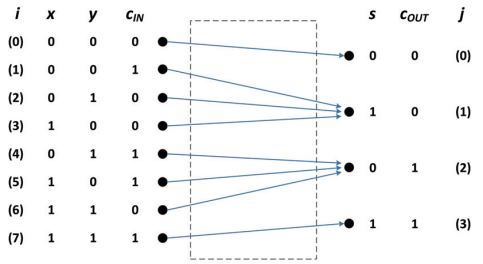

Fig. 4. Every input state has only one assignment to one correct output state.

assignment). If we assume that all wrong results related to one input pattern are equally distributed, the $m_{in} \times m_{out}$ matrix $\mathbf{T}$ (Equation 2) describes the conditional probabilities that the adder produces any result with a given input pattern

$$t_{i,j} = p_{S,C_{out}|X,Y,C_{in}}\left(s_j, c_{out,j} | x_i, y_i, c_{in,i}\right) \tag{1}$$

and $x_i \in \mathbf{X}$, $y_i \in \mathbf{Y}$, $c_{in,i} \in \mathbf{C_{in}}$, $s_j \in \mathbf{S}$ and $c_{out,j} \in \mathbf{C_{out}}$.

$$\mathbf{T} = \begin{pmatrix} 1-p & \frac{1}{3}p & \frac{1}{3}p & \frac{1}{3}p \\ \frac{1}{3}p & 1-p & \frac{1}{3}p & \frac{1}{3}p \\ \frac{1}{3}p & 1-p & \frac{1}{3}p & \frac{1}{3}p \\ \frac{1}{3}p & 1-p & \frac{1}{3}p & \frac{1}{3}p \\ \frac{1}{3}p & \frac{1}{3}p & 1-p & \frac{1}{3}p \\ \frac{1}{3}p & \frac{1}{3}p & 1-p & \frac{1}{3}p \\ \frac{1}{3}p & \frac{1}{3}p & 1-p & \frac{1}{3}p \\ \frac{1}{3}p & \frac{1}{3}p & \frac{1}{3}p & 1-p \end{pmatrix} \tag{2}$$

B. Propagation of a Fault

The iterative nature of an addition leads to the propagation of a fault by the corrupted carry-bit. This propagation is the main difference to the previously mentioned BSC model which has no dependencies to nearby bits. The dependency between two consecutive bits represents a kind of memory with respect to the fault. If there is a fault in one stage which generates a corrupted carry-bit, then the result of the consecutive stage is also wrong. Referring to [20], Osmann summarizes several channel models with memory for transmission systems. For example, the Gilbert model is based on a Markov process that models burst errors in transmission system [21], where a single upset has an influence to one or several following bits. The main idea is to adapt the Gilbert model to simulate the memory of arithmetic adders. The possible output states of an adder (see Figure 4) can be generalized in the following form which is independent on the input:

- *State **G** (good)*:
  The output state is correct with respect to the inputs.
- *State **B1** (bad)*:
  The output state is not correct with respect to the inputs. The sum-bit is corrupted with probability $\frac{p}{3}$.
- *State **B2** (bad)*:
  The output state is not correct with respect to the inputs. The carry-bit is corrupted with probability $\frac{p}{3}$.
- *State **B3** (bad)*:
  The output state is not correct with respect to the inputs. Both bits are corrupted with probability $\frac{p}{3}$.

For each adder stage, there are three possible fault conditions. (1) the-sum bit is not correct, (2) the carry-bit is not correct or (3) both bits are corrupted. If the carry-bit is incorrect, then there is the chance that the fault is propagated to the next adder stage and corrupts the following sum-bit. But there is also the probability that the next stage is also corrupted and compensates the wrong input c_{in}. Figure 5 shows the discrete Markov model of an addition with all described states.

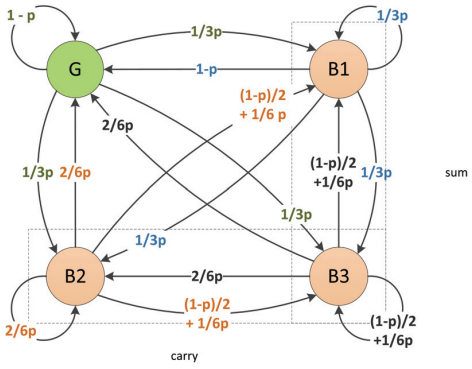

Fig. 5. Discrete Markov process which models the memory of an adder caused by the carry propagation.

Each transition in the discrete Markov chain represents a single iterative stage of an addition. Beginning with the fault-free default state G, each stage probably injects a fault which is maybe propagated to the next bit. The next section describes the behavior of the states.

C. Probability of a Corrupted Bit

How is the probability that a corrupted carry-bit will change the result (s and $c_{out}bit$) in the following stage? With a deeper insight into the logic of a standard adder (Equations 3, 4 and Figure 4), we can evaluate the chance of propagating a corrupted carry-bit to consecutive adder stages.

$$s = x \oplus y \oplus c_{in} \tag{3}$$

$$c_{out} = (y \wedge c_{in}) \vee (x \wedge c_{in}) \vee (x \wedge y) \tag{4}$$

- A corrupted input carry-bit c_{in} will surely cause a change in the sum of the following stage because of the XOR sum of all inputs (Equation 3).
- But there is the probability of $\frac{1}{2}$ that the corrupted carry-bit is propagated again. There are four different possible

combinations of two inputs x and y. With two of them being equal ($x = y$), the carry-out c_{out} is not changed with corrupted input carry c_{in}. If $x = y = 1$, then $x \wedge y = 1$ and thus $c_{out} = 1$ independent on c_{in}. If $x = y = 0$, then $x \wedge y = 0$, but $y \wedge c_{in} = 0$ and $x \wedge c_{in} = 0$ independent on c_{in}. Otherwise the input signals are not equal ($x \neq y$) and $x \wedge y$ is always 0. One of the terms $y \wedge c_{in}$ and $x \wedge c_{in}$ are always changed with inverted input carry (Equation 4).

Mathematically, the Markov process can be expressed by an stochastic transitional probability matrix $\mathbf{P}$. The elements of $\mathbf{P}$ describes the probability of the transition from one state to the other.

State G:

The default state is **G** (= no faults). In case of a fault, it is probable that the output switches to one of the three erroneous output states. With a equally distributed probability, the chance for a certain faulty state is $\frac{1}{3}$ (see Matrix $\mathbf{T}$, Equation 2).

State B1:

The state **B1** represents the state when the sum of the current adder stage is corrupted, but the carry-bit is correct. With a correct carry-bit, there is no influence to the subsequent stage and the same probabilities as for state G can be assumed.

State B2:

The state **B2** differs from the previous states. The carry-bit of the preceding stage c_{in} is corrupted. For the current stage, the sum-bit is always wrong and the carry-bit c_{out} is false with a probability $\frac{1}{2}$. Without any further faults, there are the transition probabilities:

$$B2 \rightarrow B1: \ (1-p) \cap \frac{1}{2} = \frac{1-p}{2}$$
$$B2 \rightarrow B3: \ (1-p) \cap \frac{1}{2} = \frac{1-p}{2}$$

But, if there is an additional fault in the current stage, the corrupted input carry is ignored and the faulty sum, caused by the wrong input, can be compensated. An incorrect input carry c_{in} always corrupts the sum of the current stage. But the new fault corrupts the sum-bit s with probability $\frac{2}{3}$ and corrects it again. The probability of compensating the sum s is $p_A = \frac{2}{3}$ (Event A).

An incorrect input carry c_{in} does not only change the sum s but the output carry c_{out}, too. But different to s, the output carry c_{out} is only changed with probability $\frac{1}{2}$ by c_{in}. There are two conditions to consider:

1) The output carry c_{out} is corrupted (Event $\bar{B}$),
either if it is corrupted by the current fault, whereas the input carry c_{in} does not influence the adder

$$p_{\bar{B}1} = p_{fault} \cap \bar{p}_{carry} = \frac{2}{3} \cdot \frac{1}{2} = \frac{2}{6}, \tag{5}$$

or if the fault has no influence and the input carry c_{in} corrupts the adder

$$p_{\bar{B}2} = \bar{p}_{fault} \cap p_{carry} = \frac{1}{3} \cdot \frac{1}{2} = \frac{1}{6}. \tag{6}$$

With rule of *independent and mutually exclusive events* [22], the probability that the output carry c_{out} remains corrupted is the sum of both probabilities

$$p_{\bar{B}} = p_{\bar{B}1} \cup p_{\bar{B}2} = \frac{2}{6} + \frac{1}{6} = \frac{1}{2}. \tag{7}$$

2) The output carry c_{out} is correct (Event B),
either if the fault and the input carry c_{in} corrupt the output carry c_{out}

$$p_{B1} = p_{fault} \cap p_{carry} = \frac{2}{3} \cdot \frac{1}{2} = \frac{2}{6}, \tag{8}$$

or if the fault and input carry c_{in} does not change the output carry c_{out}

$$p_{B2} = \bar{p}_{fault} \cap \bar{p}_{carry} = \frac{1}{3} \cdot \frac{1}{2} = \frac{1}{6}. \tag{9}$$

Both possibilities results in a correct output carry-bit c_{out} in the current adder stage in spite of a wrong input carry-bit and a fault in the calculation. This means that at least one of the two events leads to correct c_{out}.

$$p_B = p_{B1} \cup p_{B2} = \frac{2}{6} + \frac{1}{6} = \frac{1}{2}. \tag{10}$$

With the knowledge of the behavior in case of a corrupted carry-bit, the transition probabilities from the state $B2$ can be calculated.

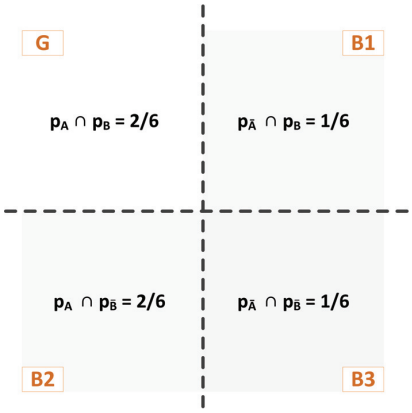

Fig. 6. Transition probabilities in case the c_{in} is corrupted (state $B2$ and $B3$) with $p_A = \frac{2}{3}$ and $p_B = \frac{1}{2}$.

Figure 6 shows all four combinations of the event A (= compensation of s) and event B (= compensation of c_{out}). If

the sum-bit is compensated, there is a transition either to state G or state $B2$. In case the carry-bit is compensated, the state changes either to G or $B1$. The final probability for any state is then the product of both compensating events, because both events must occur.

State B3:

The Markov state B3 represents the condition that the sum-bit s and the output carry c_{out} is corrupted. For the following stage, the current sum can be ignored and only the carry-bit can propagate the fault. The same transitions as for state B2 can be considered.

Now, we have all transition probabilities to construct the stochastic transitional probability matrix $\mathbf{P}$ for the discrete Markov process of an addition.

$$\mathbf{P} = \begin{pmatrix} 1-p & \frac{1}{3}p & \frac{1}{3}p & \frac{1}{3}p \\ 1-p & \frac{1}{3}p & \frac{1}{3}p & \frac{1}{3}p \\ \frac{2}{6}p & \frac{1-p}{2}+\frac{1}{6}p & \frac{2}{6}p & \frac{1-p}{2}+\frac{1}{6}p \\ \frac{2}{6}p & \frac{1-p}{2}+\frac{1}{6}p & \frac{2}{6}p & \frac{1-p}{2}+\frac{1}{6}p \end{pmatrix} \tag{11}$$

A discrete Markov process describes the time dependent state probabilities of a system. The time is quantized in a discrete unit of time, which is in our case one stage in the calculation of a bit in the sum. The multiplication of the matrix $\mathbf{P}$ with itself gives the state probabilities after the second stage. Or in general, the state probabilities after the r-th stage (= r-th bit position in the result) is according [23]

$$P(r) = \mathbf{P}^r. \tag{12}$$

The states B1 and B3 represent an error in the sum-bit s. The probability $p_s(r)$ of a corrupted bit s_r in the result is the sum of the two states B1 and B3 in each stage:

$$p_s(r) = P(r)_{1,2} + P(r)_{1,4} \tag{13}$$

IV. Coded Addition

With the knowledge of the error model for an addition (see Section III), the probabilities for single bit flips in the sum of two numbers can be evaluated. A common method for detecting such deviations in the result is the use of coded numbers. Arithmetic codes, for example, are usually used for coded data processing and for detecting faults in the result of arithmetic operations [15], [6], [24], [25]. The code word C of an arithmetic code is the product of any integer X with the constant generator integer number A

$$C = A \cdot X. \tag{14}$$

The verification of a code word is then the check, if it is a multiple of A, or the residue is zero

$$C \mod A = 0. \tag{15}$$

Faults during the execution of the addition leads to bit flips in the result. These bit flips are described by the *error mask* E and the final result is the bitwise exclusive OR disjunction

$$C' = a_c + b_c = C \oplus E. \tag{16}$$

But certain combinations of bit flips are not detectable in the result. The randomly generated error mask changes the result in a way that it is still valid and Equation 15 is satisfied, if E is a multiple of A:

$$\begin{gathered} C' = C \oplus E \\ C' \bmod A = C \bmod A \ \oplus \ E \bmod A \\ 0 = 0 \ \oplus \ E \bmod A \end{gathered}$$

This is the dangerous case because the corrupted result is assumed to be correct and the error can cause further failures in the system. It is not possible to avoid these undetected faults. But for the purpose of reliability evaluation, the probability of undetection is an important metric with respect to safety-critical aspects.

A. Probability of Error Masks

An *error mask* describes the changed bits in the result of an operation. A corrupted bit in the result is caused by a fault in the current bit stage or by the carry-bit from the previous bit stage as modeled by the Markov process in Figure 5. If the Markov model is considered as a graph, a certain error mask corresponds to a defined path in the graph. The transitions from one state to the next equates to the probability traversing this state. And the product of all transitions is the total probability of the complete path describing the error mask. In Figure 5, the two right states (B1 and B3) stand for wrong bits in the result and the left states for correct bits. Traversing the error mask beginning with the least significant bit, every stage matches either states G/B1 when zero or the states B2/B3 when one. Calculating the probability of a certain error mask leads to the basic problem of finding all paths in the graph that match a given error mask.

Example:
Let $E = 00000101_b$ be the error mask of a result after an 8-bit addition. The bits of the error mask define the transitions from G/B2 to B1/B3 and vice versa at each stage of the adder. Beginning in the default state G and the least significant bit of the error mask, there are two possible transitions from state G. The first bit in the result is corrupted and follows a transition either to state B1 or B3. With each successive stage, there are always two possible transitions and all possible paths can be represented by a binary tree as shown in Figure 7.

The number of nodes in the binary tree depends on the path length n, which equates the bit length of the adder:

$$N_{nodes} = \sum_{k=0}^{n} 2^k \tag{17}$$

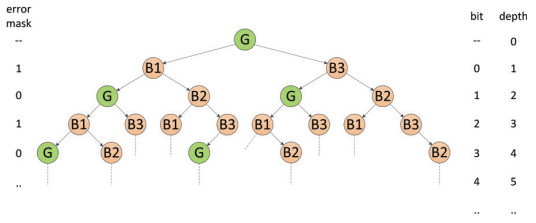

Fig. 7. The binary tree represents all possible paths for a certain error mask.

Each leaf of this tree is the end of a path beginning in state G. Thus, there are

$$N_{paths} = 2^n \tag{18}$$

leaves or different paths. The product of all transition probabilities is the probability for a certain path. The error mask defines all possible paths in the Markov chain. A "1" in the error mask represents a corrupted bit and there is a path segment to the states $B1$ or $B3$. Otherwise, there is a correct bit and the path traverses to G or $B2$.

Let $\mathbb{P}$ be the set of all paths of length n starting in state G. Then, a single path $\Pi \in \mathbb{P}$ can be expressed by the set of transitions

$$\Pi = \{e_1, e_2, ...e_n\}. \tag{19}$$

A transition e_k is defined by the 3-tuple (S_i, S_j, p_i) which describes the probability $prob(e_k) = \mathbf{P}_{i,j}$ for the transition from state S_i to the destination node S_j, which is defined by the stochastic transitional probability matrix of Equation 11. The set of allowed states are defined by the Markov model in Figure 5 with $S \in \{G, B1, B2, B3\}$. Finally, the total probability for a certain path is the product of all edges from the root to the leaf of the tree:

$$p_{path} = \prod_{k=1}^{n} prob(e_k) \quad \text{with } e_k \in \Pi \tag{20}$$

A given error mask defines a unique binary tree with several paths as shown in Figure 7. Each path represents a possible way with a certain combination of faults to generate the error mask in the result word. Thus, the total probability of this error mask is the sum of all possible paths with

$$p_E = \sum_{P \in \mathbb{P}} p_{path}. \tag{21}$$

B. Residue Error Probability of a Code

Using code words for arithmetic operations, there is still the probability for undetected faults. In the previous section, we defined so-called *error masks* that change a valid code word into another one and we showed the way to evaluate their probabilities. The evaluation of a given code consequently requires the evaluation of all possible error masks which contribute a part of the total residue error probability of the code.

Let $\mathbb{C}_A$ be the set of all arithmetic code words C defined by the generator A

$$\mathbb{C}_A := \{A \cdot X \mid A, X \in \mathbb{Z}\}. \tag{22}$$

Then, $\mathbb{E}$ is the minimum set of all error masks that can change valid code words to each other:

$$\mathbb{E} := \{C_a \oplus C_b \mid a \neq b \wedge C_a, C_b \in \mathbb{C}_A\}. \tag{23}$$

But not every combination $\mathbb{C}_A \times \mathbb{E}$ leads to a valid code word. The weight w_E reduces the probability for a certain error mask $E \in \mathbb{E}$ to change a valid code word.

$$w_E := \frac{\sum_{i=0}^{|\mathbb{C}_A|} w_i}{|\mathbb{C}_A|} \quad \text{with} \quad w_i = \begin{cases} 1 \text{ if } (C_i \oplus E) \in \mathbb{C}_A \\ 0 \text{ else} \end{cases} \tag{24}$$

Example:
Let $\mathbb{C}_A$ be the set of code words generated by $A = 3$. The error mask $E = 3$ changes the code word $C_1 = 15$ to the corrupted code word $C_1' = C_1 \oplus E = 12$, which is a multiple of 3 and therefore valid. But, the code word $C_2 = 6$ is corrupted by E to $C_2' = C_2 \oplus E = 5$, which is no multiple of 3.

The total residue error probability P_u of the code $\mathbb{C}_A$ is the sum over all weighted error masks $E \in \mathbb{E}$.

$$P_u = \sum_{\mathbb{E}} p_E \cdot w_E \tag{25}$$

Equation 25 describes the probability that the result of an addition is a valid code word in spite of the occurrence of a fault. In the next section, the results of this probability evaluation is discussed and verified by a simulation approach

V. Simulation and Discussion

For verification of the presented approach, a fault simulation is used. The basic concept of this simulator is the permanent addition of equally distributed code words (Figure 8). With a huge number of additions, the arbitrary corruption of an addition leads to the average number of undetected errors in the results. For this simulation, the adder is emulated by software according the iterative network shown in Figure 3, where a bitwise fault injection is realized for each stage. The GNU scientific library [26] offers the functionality of random number generators which can be easily integrated in our program. The generated random numbers follow an exponential distribution and represent the time of the occurrence of a fault, when the output state of one adder stage changes from the correct one to a wrong one.

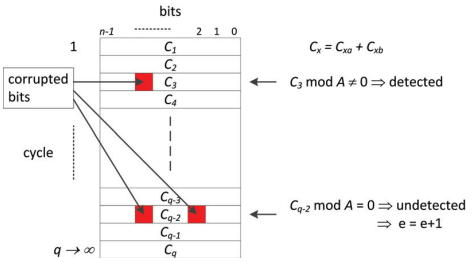

Fig. 8. Simulation of the residue error probability by permanent addition and arbitrary injection of faults.

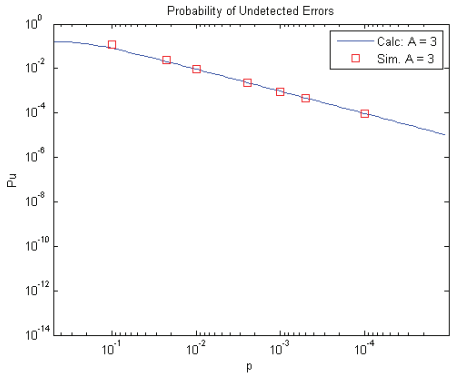

Fig. 9. Residue error probability of a given code as a function of probability. The squares are simulated samples for a certain probability p.

The simultaneously and independently injection of a fault in each stage of the adder allows multiple faults in the addition and the resulting error mask makes it possible to have a valid code word again in spite of an error. But with the knowledge of each fault injection, the number of these undetected errors can be evaluated by the verification of the result by means of the modulo check $C \mod A$. With q additions, there are $q \cdot n$ bits in the result words. But the fault probability p leads to $q \cdot n \cdot p$ corrupted bit in average. Sometimes, the error mask of the current addition is one of that which preserves the code and the error is not detectable. The evaluation of all corrupted results, which the modulo check passes for, leads to the residue error probability

$$P_{u,sim} = \frac{e}{q} \tag{26}$$

with

e is the number of undetected erroneous additions and
q is the total number of additions.

The output of some simulation runs can be seen in Table V and Figure 9. Whereas the figure shows the residue error probability as a function of the fault probability p once from the the error model and some simulation results for comparison. The table shows the probability of single error masks with $p = 0.001$ and the generator $A = 3$. Both, the simulation and the model verify the correctness of the model by supplying comparable outputs within a given tolerance.

Table I
PROBABILITIES OF ERROR MASKS

Error Mask	Simulation	Model	min. Faults
0x02	$4.973 \cdot 10^{-4}$	$4.969 \cdot 10^{-4}$	1
0x03	$1.662 \cdot 10^{-4}$	$1.658 \cdot 10^{-4}$	1
0x0A	$3.140 \cdot 10^{-7}$	$3.317 \cdot 10^{-7}$	2
0x05	$1.999 \cdot 10^{-7}$	$2.211 \cdot 10^{-7}$	2
0x09	$1.632 \cdot 10^{-7}$	$1.659 \cdot 10^{-7}$	2
0x29	n.a.	$1.107 \cdot 10^{-10}$	3

The probability of a given error mask in Table V doesn´t show the minimum number of faults which are required to get the mask. For example, one single fault is enough for the mask 0x03 because the fault in the least significant stage can also corrupt the next stage by an incorrect carry-bit. In contrast, the mask 0x0A requires at least two faults because of the gap in the error mask between bit 3 and bit 1. The propagation by the carry-bits would corrupt all consecutive bits. The transitions in the Markov chain (Figure 5) represent either a transition caused by a fault (term p) or a transition caused by a fault-free stage (term $1 - p$). So, it is possible to determine the number of faults of each path in Figure 7. But, the probability of no fault is higher than a fault and therefore, the path with the minimum number of faults is the most important one which can be compared with the *arithmetic distance* [15], [27].

In many publications, the choice of the generator A is an often discussed topic [6], [9] to be a prime number. Now with the presented approach, the metric of P_u is calculated by means of an error model derived from the Markov process. Different codes can be compared in an easy way. Thus, Figure 10 compares the residue error probability of different AN-codes with $A < 64$.

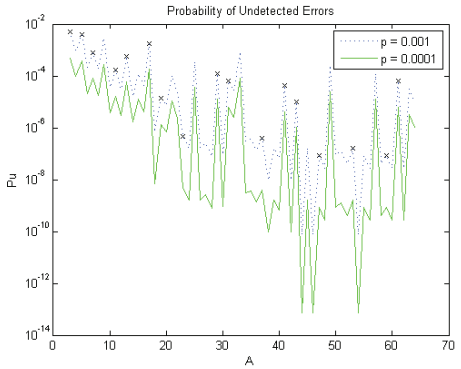

Fig. 10. Comparison of the residue error probability of three different AN-codes with $A < 64$. The x marks prime numbers.

It seems that there is no obvious rule for the right choice. Even though, for example, $A = 31$ is a prime number and greater than $A = 29$, which is also a prime number, the residue error probability is worse than for $A = 33$, which is no prime. Whereas the code with a smaller prime is better than the codes with greater generators

$$P_u(A = 29) < P_u(A = 33) < P_u(A = 31).$$

The results of Figure 10 requires a more detailed analysis of the generator in future works.

VI. Conclusion

The presented error model shows us the possibility for the analytical evaluation of the residue error probability of a given code with respect to the addition. The ripple-carry adder as a basic arithmetic operation is considered as a noisy channel. Inspired by the Gilbert channel, the memory effect between bits inside an addition is modeled by a discrete Markov chain, where each transition corresponds to a single stage of the iterative adder network. In contrast to the approach in [8], the Markov approach is more abstract and does not model the details of the adder realization. But, it allows the computation of corrupted bits in the sum and their probability which is the most important advantage of the presented approach. Instead of simulation and experimental approaches, the presented error model offers the possibility to evaluate erroneous outputs in an analytical way with less computational effort than with simulation.

This is important especially for coded processing, where errors in the result don´t probably preserve the code. The state-of-the-art of coded processing uses the group of so-called AN-codes which are generated by the multiplication of integer numbers with the constant number A. In future works, the presented error model is used to research the dependency between the error detection capability of a given code and the generator and an optimum can be found.

Furthermore, the presented HW model of a ripple-carry adder is a simple example and there are a lot of different realizations in real applications. The basic concept of an assignment matrix (Equation 2 in Section III) provides an easy adaption of the distribution in other adder implementations. So, the presented error model can be considered as the principle model for other arithmetic operations.

With a set which contains more error models than described here, the reliability of a computation can be calculated as long the data flow of the computation is known. The software reliability with respect to unreliable hardware is an important metric for safety-critical applications. For this, further error models will be provided in future.

References

[1] P. Dodd and L. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 583 – 602, June 2003.

[2] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[3] F. Wang and V. Agrawal, "Single Event Upset: An Embedded Tutorial," in *Proc. 21st International Conference on VLSI Design*, January 2008, pp. 429 – 434.

[4] D. Brown, "Error detecting and error correcting binary codes for arithmetic operations," in *IRE Trans. Electron. Comput.*, 1960, pp. 333–337.

[5] P. Forin, "Vital coded microprocessor principles and application for various transit systems," in *IFA-GCCT*, 1989, pp. 79–84.

[6] J. Mottok, F. Schiller, T. Völkl, and T. Zeitler, "A Concept for a Safe Realization of a State Machine in Embedded Automotive Applications," in *26th Safecomp Conference*, ser. Lecture Notes in Computer Science, vol. 4680. Springer, 2007, pp. 283–288.

[7] U. Schiffel, A. Schmitt, M. Süsskraut, and C. Fetzer, "ANB- and ANBDmem-encoding: Detecting hardware errors in software," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 6351, pp. 169–182.

[8] N. Oh, S. Mitra, and E. McCluskey, "ED4I: Error Detection by Diverse Data and Duplicated Instructions," *IEEE Transactions On Computers*, vol. 51, pp. 180–199, 2002.

[9] U. Schiffel, "Hardware Error Detection Using AN-Codes," Ph.D. dissertation, Technische Universität Dresden, Germany, 2011. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-69872

[10] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante, "Soft-error detection through software fault-tolerance techniques," in *International Symposium on Defect and Fault Tolerance in VLSI Systems*, November 1999, pp. 210–218.

[11] B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: Method, tools and experimental results," *Design, Automation and Test in Europe Conference and Exhibition*, vol. 2, p. 20057, 2003.

[12] P. Ozello, "The coded microprocessor certifcation." in *International Conference on Computer Safety, Reliability and Security*. Springer Munich, 1992, pp. 185–190.

[13] M. Bossert, *Kanalcodierung*, 2nd ed. Teubner, 1998.

[14] R. H. Morelos-Zaragoza, *The Art of Error Correcting Coding*. John Wiley & Sons, Ltd, 2006.

[15] T. R. N. Rao, *Error coding for arithmetic processors*, ser. Electrical science series, H. G. Booker and N. DeClaris, Eds. Academic Press, New York and London, 1974.

[16] A. Avizienis, "Arithmetic error codes: Cost and effectiveness studies for application in digital system design," *IEEE Transactions on Computers*, vol. C-20, no. 11, pp. 1322 – 1331, Nov. 1971.

[17] T. Ericson, "The noncooperative binary adder channel," *IEEE Transactions on Information Theory*, vol. 32, no. 3, pp. 365 – 374, May 1986.

[18] T. Kasami and S. Lin, "Coding for a multiple-access channel," *Information Theory, IEEE Transactions on*, vol. 22, no. 2, pp. 129 – 137, Mar. 1976.

[19] K. Fritzsche, *Grundkurs Analysis 1: Differentiation und Integration in einer Veränderlichen*, ser. Für Bachelor und Diplom. Spektrum Akademischer Verlag, 2008.

[20] C. Osmann, "Bewertung von Codierverfahren für einen störungssicheren Datentransfer - Evaluation of error-correcting codes used for a reliable data transfer," Ph.D. dissertation, Universitaet Duisburg-Essen, 2001. [Online]. Available: http://www.ub.uni-duisburg.de/ETD-db/theses/available/duett-05302001-111522/

[21] M. Mushkin and I. Bar-David, "Capacity and coding for the gilbert-elliott channels," *IEEE Transactions on Information Theory*, vol. 35, no. 6, pp. 1277 –1290, Nov. 1989.

[22] D. Applebaum, *Probability and information: An integrated approach*, 2nd ed. Cambridge: Cambridge University Press, 2008.

[23] R. Billinton and R. Allan, *Reliability evaluation of engineering systems: concepts and techniques*. Plenum Press, 1992.

[24] U. Wappler and C. Fetzer, "Software encoded processing: Building dependable systems with commodity hardware," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2007, vol. 4680, pp. 356–369.

[25] R. Krüger, A. Schenk, and F. Schiller, "System und Verfahren zur Verbesserung von Fehlerbeherrschungsmassnahmen, insbesondere in Automatisierungssystemen," Patent DE10 219 501B1, 2010.

[26] (2011, April) GSL - GNU Scientific Library. Free Software Foundation, Inc. [Online]. Available: http://www.gnu.org/software/gsl/

[27] W. E. Peterson W.W., *Error Correcting Codes*, 2nd ed. MIT Press, 1972.

978-1-4673-2232-4/13/$31.00 ©2013 IEEE

A.6 Data Flow Analysis of Software Executed by Unreliable Hardware

Authors: P. Raab, S. Racek, S. Kraemer and J. Mottok
Published: In *Proceedings of the 16th Euromicro Conference on Digital System Design*, pages 243-249, September 2013.
ISBN-13: 978-0-7695-5074-9

2013 16th Euromicro Conference on Digital System Design

Data Flow Analysis of Software Executed by Unreliable Hardware

Peter Raab, Stanislav Racek

University of West Bohemia
Faculty of Applied Sciences
Univerzitní 8, 306 14 Plzen, Czech Republic
{praab, stracek}@kiv.zcu.cz

Stefan Krämer, Jürgen Mottok

Laboratory for Safe and Secure Systems
Regensburg University of Applied Sciences
Faculty of Electronics and Information Technology
Seybothstr. 2, D-93053 Regensburg, Germany
{stefan.kraemer, juergen.mottok}@hs-regensburg.de

***Abstract*—The data flow is a crucial part of software execution in recent applications. It depends on the concrete implementation of the realized algorithm and it influences the correctness of a result in case of hardware faults during the calculation. In logical circuits, like arithmetic operations in a processor system, arbitrary faults become a more tremendous aspect in future. With modern manufacturing processes, the probability of such faults will increase and the result of a software´s data flow will be more vulnerable. This paper shows a principle evaluation method for the reliability of a software´s data flow with arbitrary soft errors also with the concept of fault compensation. This evaluation is discussed by means of a simple example based on an addition.**

Keywords: data flow, error probability, fault compensation, reliability analysis, software-implemented-hardware-fault-tolerance (SIHFT)

I. Introduction

The complexity and functionality of electronic control units have increased more and more in several sectors of industry. In addition, the requirements of these systems have become more demanding in terms of safety, reliability and availability. In contrast to this progress, industry demands a decrease in costs for electronics, while at the same time remaining competitive. The use of inexpensive commodity hardware is the result. However, the development of current micro-controllers follows the trend of decreasing feature size. That leads to less reliability and arbitrary hardware faults are more likely [1]. The impact of so-called *Single Event Upsets* (SEU) [2], [3] are bit flips in logical and memory circuits of a processor based system. The consequence is a deviation in software processing like *operator error, operation error, operand error* and *lost updates* [4]. These errors finally produce data flow or program flow errors [5], [6] and probably lead to fatal system failures at the end.

In literature, a lot of pure software based techniques are described that tolerate faults and compensate the lack of reliability in present commodity hardware [6]. All of those techniques have in common that they increase the reliability by additional redundancy. Transient faults can be detected by pure duplication either of the data or of the task execution. But in contrast, permanent faults will generate the same erroneous result. Therefore, only the use of diverse data allows the detection of permanent faults by employing different units of the *central processing unit* (CPU) [7], [8]. The concept of diversity leads to the approach of coded data processing [9], [4], [10], [11], which uses encoded variables for protecting the data flow against faults. In [8], they describe the ED^4I as a similar approach compared to coded data processing using diverse data. They duplicate the program whereas the diverse data of the copy is a simple multiplication with a constant factor. The evaluation of an iterative network such as a ripple-carry-adder shows that the *diversity factor* has an influence on the fault detection probability.

Further, the fault detection probability is an important metric for the comparison of different fault-tolerant techniques. As presented in [8], this probability is determined by the analysis of the circuitry of the underlying hardware. In contrast to that, there are a lot of examples reported in the literature which derives the fault detection probability by experimental methods. Whereas in [12] a practical evaluation leads to the probability of undetected errors, there is also the possibility of fault injection techniques [13], [14], [6], [15], [16]. But none of those experimental methods analytically evaluate the reliability either of a single instruction or the complete data flow. In fact, there is an analytical way presented in [8] but they don´t consider the effect of fault compensation which occurs with the execution of several consecutive and dependent faulty instructions. Thus, this paper starts closing this gap and presents the basic concept of reliability evaluation including fault compensation during the data processing.

For this, the structure is as follows: Section II summarizes the needed background of reliability analysis which is extended for a data flow in Section III. Further in Section IV, we show the influence of fault compensation during data flow processing by a simple example. Finally, the paper ends with a discussion of the model and an outlook to further works in Sections V and VI.

II. Background

Dependability defines the term *reliability* as the conditional probability that a component is not failing for a period of time, given the component has not failed at the beginning [17]. Compared to processor systems, the reliability of the data flow is the probability of the correct outcome in case the input data of the computation

`/13 $26.00 © 2013 IEEE
DOI 10.1109/DSD.2013.35

was correct. During the execution of software, a lot of faults with a given rate influence the data flow and likely corrupt the outcome of the computation and a system fails in worst case. The probability of a corrupted result determines the reliability of the software and finally the reliability of the whole system. In dependable applications, the goal is the improvement of reliability by the detection of faults to avoid system failures.

The reliability of a computed result depends on the correct execution of a set of instructions on the underlying unreliable hardware. Indeed, there are only these two behavioral effects in a software, which are corrupted by hardware faults: data flow errors and program flow errors [6]. In a complex program, the dependency between variables with lot of descendants propagates possible faults to consecutive results and has a big impact on their reliability. The more variables and computation steps are necessary for the computation of a final one, the higher the risk that one of them is corrupted by an SEU [18]. But, there is another interesting effect in a corrupted data flow. If there are several faults in different but dependent variables or instructions, it is possible that the faults compensate each other [19]. The final result is again correct or in case of coded processing, the fault is not detected.

The literature reports a lot of probabilistic models for bit faults in transmission systems [20], [21], [22], but not for arithmetic operations in computer systems because of its complexity. So-called channel models are an important mean in the information theory to approximate the behavior of real noisy transmission channels by a probabilistic approach. Therefore, we presented an approach for an error model describing the behavior of a faulty addition in [23]. Because of the carry-bit propagation in a ripple-carry-adder, there is a kind of memory effect between two consecutive bits which is modeled by the Markov chain in Figure 1.

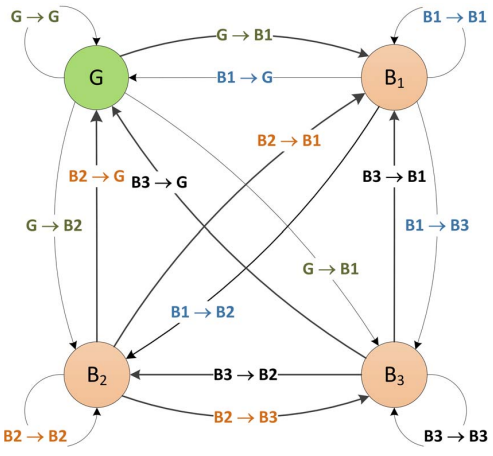

Figure 1. Discrete Markov process modelling the memory of an adder [23].

Each state transition in Figure 1 corresponds to a computation of a single bit a sum by a full-adder element with the two outputs s and c_{out}. Thus, there are four different states for the outputs which are defined to be independent on the inputs (x, y and c_{in}). The default state G is the correct output of any bit stage. But a fault in the computation of a bit position will change the result in a way that either the sum bit (state $B1$), the carry-bit (state $B2$) or maybe both bits (state $B3$) are corrupted. If only the calculation of the sum is wrong, the result contains an inverted bit. But in case of an incorrect carry-bit, the following bit in the result can be corrupted as well. However, a further fault in the next bit stage could compensate a wrong carry-bit and the output is correct again. The complete derivation of the transition probability matrix (Equation 1) describing the Markov process for operation errors in an adder can be found in [23].

$$P = \begin{pmatrix} 1-p & \frac{1}{3}p & \frac{1}{3}p & \frac{1}{3}p \\ 1-p & \frac{1}{3}p & \frac{1}{3}p & \frac{1}{3}p \\ \frac{2}{6}p & \frac{1-p}{2}+\frac{1}{6}p & \frac{2}{6}p & \frac{1-p}{2}+\frac{1}{6}p \\ \frac{2}{6}p & \frac{1-p}{2}+\frac{1}{6}p & \frac{2}{6}p & \frac{1-p}{2}+\frac{1}{6}p \end{pmatrix} \quad (1)$$

This model assumes that a single fault in an adder element causes a transition to any error states *Bx*. Indeed, this is a simplification because the transitions depends on the realization of the adder which is usually not known. So it is possible that the model remains in state G in spite of an active fault. But, the basic concept of the given model is the same and only the values of the transitions must be adapted according the hardware implementation. Further in Section IV, this model is extended for faulty operands and the associated fault compensation.

III. Data Flow

The data flow of a software describes the dependencies of variables and the order of their processing. On the other hand, the reliability of the final result depends on the correct execution of each instruction and the storage of data in the memory. In the course of this section, it is shown that the probability of an erroneous result is higher the more data and the more instructions depends on these data. A simple example of a data flow is the addition of two integer numbers corresponding to the pseudo assembler code in Listing 1.

```
mov   #3, r1        ; r1 = 3
mov   #5, r2        ; r2 = 5
add   r1, r2, r3    ; r3 = r1 + r2
```

Listing 1. Pseudo assembler code for the addition of two integer numbers.

It shows a set of assembler instructions, which copy two integer numbers into the working registers $r1$ / $r2$, add them and store the sum into the register $r3$. The reliability of the sum depends on the fault-free execution of each single instruction. This can be illustrated by the reliability network of serially connected components. Or in other words, the result is only correct, if the first mov AND the second mov AND the add instruction work correctly. Thus, the total reliability of this serial connection is the product of all single reliabilities with $R = R_{mov} \cdot R_{mov} \cdot R_{add}$. The complement of the reliability is the probability of an erroneous output which is the

case if at least one instruction produces an error. This probability increases the more instructions are involved for the final result.

The mov instruction is a simple operation, which copies a constant number or the content of a register to another register or memory location. In contrast to the addition, the mov has no functional dependencies between each bit stage. Each bit, which is copied by the mov instruction, represents an independent event and the more bits the register has, the higher the probability for an error in the output. Generally, the probability of a single faulty bit follows the exponential distribution and the reliability is

$$R(t) = e^{-\lambda \cdot t}. \tag{2}$$

With an execution time t_{mov} and a constant fault rate λ for a single bit, the reliability can also be expressed by the fault probability p

$$R_{bit} = R(t_{mov}) = e^{-\lambda \cdot t_{mov}} = e^{-p}. \tag{3}$$

The n bits of a register word represent independent components. So, the total reliability of the complete mov instruction is the product of all bit reliabilities with

$$R_{mov} = (R_{bit})^n = e^{-p \cdot n}. \tag{4}$$

Further, the exponential term in Equation 4 can be re-expressed by the binomial distribution (see Chapter 6.6.3 in [24])

$$R_{mov} = (1-p)^n. \tag{5}$$

The term in Equation 5 corresponds to the probability that all n bits of a register are not corrupted and it shows that the probability of an error also increases with the bit size of the data and the time the data is stored. But, the data flow of a complete software task is usually more complex with a lot of dependencies and variables. The total data flow can be splitted into several and more manageable nodes for simplifications. With respect to a given instruction set, there are basically two categories to distinguish:

1) Linear Node: The output of one instruction is the only input of the following. A possible fault compensation happens between two instructions in the linear data flow. The *mov* instruction is an example for a linear data flow.

2) Conjunctive Node: A conjunctive instruction has more than one input. At least three elements (two operands and the instruction) influence the data flow. A possible fault compensation is either between the two inputs, or one input and the conjunctive instruction. The *add* instruction is an example for a conjunctive data flow.

These basic nodes allow us the reliability evaluation of any data flow by concatenating them. But in general, there are further specializations in terms of the instruction itself in detail. This means that the error behavior of each instruction influences the model of a single node. In future works, a complete set of error models must be developed to determine the reliability of a complete data flow. For now, we present the concept of reliability evaluation based on the simple data flow in Listing 1. The addition as an example of a conjunctive node depends on the correctness of the instruction itself and both operands. Further, we evaluate the effect of fault compensating based on this example in the next section.

IV. Fault Compensation

Fault compensation describes the effect that after a given data flow the outcome is correct in spite of faults. However, fault compensation is only possible with at least two contrary faults and the longer the data flow, the higher the number of instructions and the higher the chance for this effect. A fact is that the effective reliability R_e of a compensated data flow is higher than it is expected by the multiplication of all single reliabilities. The Equation 6 describes this effect by the factor $r_c > 1$ increasing the basic reliability with

$$R_e = r_c \cdot \prod R_i. \tag{6}$$

A given software defines the data flow and the set of executed instructions which has to be analyzed with respect to their reliability. In this paper, we want to show only the basic principle to do this. Because of the fact that there are no models of a complete instruction set available except of [23], the effect of fault compensation is discussed by the simple data flow of Listing 1. This simple example represents a conjunctive node where either an operand or the *add* instruction is faulty. The total reliability of the data flow depends on the reliability of each single part. Thus, the outcome of the addition is correct, if both operands and the addition itself is correct.

There are three elements that can propagate a fault to an erroneous outcome of the data flow. But, it is also possible that two faults can compensate each other and the effective reliability is higher than expected. The combinatoric of these three faulty elements is manageable and there are only four cases to distinguish. Let the faults in any element be independent events with the fault probability $p_1 = p_2 = p_3 = p$ and $\bar{p}$ is the probability of no fault in any element. Because both probabilities are mutually exclusive, it is $\bar{p} + p = 1$ and all possible combinations of faults can be described by the binomial formula

$$(\bar{p}+p)^3 = \underbrace{\bar{p}^3}_{\text{no faults}} + \underbrace{3 \cdot \bar{p}^2 \cdot p}_{\text{one fault}} + \underbrace{3 \cdot \bar{p} \cdot p^2}_{\text{two faults}} + \underbrace{p^3}_{\text{three faults}}. \tag{7}$$

There are following four cases:

(1) No fault in any element with probability:

$$\bar{p}^3 = (1-p)^3 = 1 - 3p + 3p^2 - p^3 \tag{8}$$

(2) One fault in any element with probability:

$$3 \cdot \bar{p}^2 \cdot p = 3p - 6p^2 + 3p^3 \tag{9}$$

This case exactly describes one fault, while the other two elements are fault-free.

(3) Two faults in any two elements with probability:

$$3 \cdot \bar{p} \cdot p^2 = 3p^2 - 3p^3 \tag{10}$$

This case describes the probability of any two faults, whereas the other remaining element is fault-free.

(4) All three elements are faulty with a probability p^3.

The Equations 8 - 10 are important to describe the fault behavior of the data flow assuming only single faults in a single element. First, let us assume only a single fault in any element (operand or adder). This means, there is no fault compensation possible and the fault is propagated to the output. Because of the adder as the basic element in the data flow, we use the Markov chain from [23] to model the memory effect between the bits. Basically, we have to distinguish the two cases:

1) The carry-bit is correct (states G / $B1$). There is no influence to the next stage.
2) The carry-bit is wrong (states $B2$ / $B3$). There is a possible influence to the next stage.

G / B1 - carry-bit is correct

Assuming a correct carry-bit from the previous adder stage, a fault in the current stage means a transition from the state $G/B1$ to any other state of the Markov chain in Figure 1. But, there is a difference in the behavior if an operand or the addition is faulty. Let us now assume a correct adder and evaluate the behavior in case of a corruption in the operands. The following example shall explain the basic principle of this evaluation procedure.

Example:
Let the input of an adder stage be $x = 0$, $y = 0$ and $c_{in} = 0$, then the correct output is $s = 0$ and $c_{out} = 0$. Assuming a single fault in either x or y, the sum changes to $s = 1$, whereas the carry remains unchanged $c_{out} = 0$. Only s is incorrect and there is a change to state $B1$ in the Markov model. Evaluating all 8 possible input states with only one faulty operand, the sum bit of the addition is always corrupted and the carry-bit has a chance of 50% to be corrupted (see Table I). This means a transition either to state $B1$ (wrong s / correct c_{out}) or to state $B3$ (wrong s and c_{out}).

Table I summarizes the error behavior of an adder with a single corrupted operand. The first column shows all fault-free transition from possible input patterns to correct outputs. The second column shows all possible corruptions of any operand and the third column shows what output is changed (X) or remains unchanged (-) with given fault in the operand. With Table I, we have the distribution of transitions to states $B1/B3$ in case of a single corrupted operand. But, there could be a fault in the adder as well. Equation 9 defines the probability of a single fault in any element of the given data flow with only two thirds which are related to a fault in the operands. This means that there is a transition to the states $B1$ or $B3$ caused by a faulty operand with probability of $2(p - 2p^2 + p^3)$, while the remaining probability of $p - 2p^2 + p^3$ defines the fault transition to the state $B1$, $B2$ or $B3$ according the matrix in Equation 1. Both fault

Table I
ERROR BEHAVIOR OF AN ADDITION WITH CORRUPTED OPERAND

fault-free			one fault			error	
x y c		s c	x'y'c		s' c'	s	c
0 0 0	→	0 0	0 1 0	→	1 0	X	-
			1 0 0	→	1 0	X	-
0 0 1	→	1 0	0 1 1	→	0 1	X	X
			1 0 1	→	0 1	X	X
0 1 0	→	1 0	0 0 0	→	1 0	X	-
			1 1 0	→	0 1	X	X
0 1 1	→	0 1	0 0 1	→	1 0	X	X
			1 1 1	→	1 1	X	-
1 0 0	→	1 0	1 1 0	→	0 1	X	X
			0 0 0	→	0 0	X	-
1 0 1	→	0 1	1 1 1	→	1 1	X	-
			0 0 1	→	1 0	X	X
1 1 0	→	0 1	1 0 0	→	1 0	X	X
			0 1 0	→	1 0	X	X
1 1 1	→	1 1	1 0 1	→	0 1	X	-
			0 1 1	→	0 1	X	-

scenarios are summarized in the first column of Table II. With the original state of G or $B1$, the total transition probability to any state is the sum of all combinations. For example, there is a transition from G to $B1$ either with one fault in any operand, or with a fault in the adder. The sum of all probabilities in the first column is the probability of exact one fault in any element according Equation 9. For a complete evaluation of all transition probabilities from states $G/B1$, the cases with two or three faults must be considered, too:

- Two faults in both operands have the probability of $p^2 - p^3$ according Equation 10. A similar procedure of evaluating all possible corrupted input patterns as in Table I shows that two corrupted operands compensate each other (state G) or only the carry-bit is wrong (state $B2$) each with the same frequency (see Table III).

Table III
ERROR BEHAVIOR OF AN ADDITION WITH TWO CORRUPTED OPERANDS

x y c		s c	x' y' c		s' c'	s	c
000	→	0 0	110	→	0 1	-	X
001	→	1 0	111	→	1 1	-	X
010	→	1 0	100	→	1 0	-	-
011	→	0 1	101	→	0 1	-	-
100	→	1 0	010	→	1 0	-	-
101	→	0 1	011	→	0 1	-	-
110	→	0 1	000	→	0 0	-	X
111	→	1 1	001	→	1 0	-	X

- Derived from Equation 10, two faults in any operand and the adder stage have the probability of $2(p^2-p^3)$. But here, a special behavior in the data flow must be considered. The fault in the adder "overwrites" the fault in the operand and probably compensates it. A faulty adder usually makes a transition to $B1$, $B2$ or $B3$ given that the state was G or $B1$ before. But in case of a second fault in any operand, there is already a transition to state B1 or B3 (see before) and the faulty addition has one of these states as an initial state. Consequently, the probability of this case is

Table II
TRANSITION PROBABILITIES FROM $G/B1$

initial state	number of faults			
$G/B1$	1 fault	2 faults	3 faults	total transition probability
G		O: $\frac{1}{2}(p^2-p^3)$ A2: $\frac{2}{6}(p^2-p^3)$	A2: $\frac{2}{6}\frac{p^3}{2}$	$\frac{5}{6}p^2-\frac{2}{3}p^3$
B1	O: $\frac{1}{2}(2p-4p^2+2p^3)$ A: $\frac{1}{3}(p-2p^2+p^3)$	A1: $\frac{1}{3}(p^2-p^3)$ A2: $\frac{1}{6}(p^2-p^3)$	A1: $\frac{1}{3}\frac{p^3}{2}$ A2: $\frac{1}{6}\frac{p^3}{2}$	$\frac{4}{3}p-\frac{13}{6}p^2+\frac{13}{12}p^3$
B2	A: $\frac{1}{3}(p-2p^2+p^3)$	O: $\frac{1}{2}(p^2-p^3)$ A1: $\frac{1}{3}(p^2-p^3)$ A2: $\frac{2}{6}(p^2-p^3)$	A1: $\frac{1}{3}\frac{p^3}{2}$ A2: $\frac{2}{6}\frac{p^3}{2}$	$\frac{1}{3}p+\frac{1}{2}p^2-\frac{1}{2}p^3$
B3	O: $\frac{1}{2}(2p-4p^2+2p^3)$ A: $\frac{1}{3}(p-2p^2+p^3)$	A1: $\frac{1}{3}(p^2-p^3)$ A2: $\frac{1}{6}(p^2-p^3)$	A1: $\frac{1}{3}\frac{p^3}{2}$ A2: $\frac{1}{6}\frac{p^3}{2}$	$\frac{4}{3}p-\frac{13}{6}p^2+\frac{13}{12}p^3$
sum of column	$\mathbf{3p-6p^2+3p^3}$	$\mathbf{3p^2-3p^3}$	$\mathbf{p^3}$	$\mathbf{3p-3p^2+p^3}$

splitted to both initial states of the faulty adder:
(1) Initial state caused by the operand is $B1$: There is a transition to $B1$, $B2$ or $B3$.
(2) Otherwise, the initial state is $B3$ and the transition is to any other state.
But with different initial states, the transition probabilities of the corrupted adder also differs (see Equation 1).

- The evaluation of three faults in all components requires a similar consideration as in the item before. Two faulty operands lead to either state G or $B2$, but the third fault in the adder "overwrites" these states again. With G is the initial state, the faulty adder results in the states $B1$, $B2$ or $B3$. Otherwise with $B2$ is the initial state, there is a transition to the states G, $B1$, $B2$ or $B3$.

Table II summarizes all transition probabilities caused by faults in any element of the given data flow in Listing 1. All single probabilities are based on the binomial distribution of Equations 8 - 10 and are differentiated in detail depending on the cause. **O** means here a transition which is caused by a faulty operand, whereas **Ax** represents a transition caused by a corrupted adder with the further distinction of the initial state caused by the corrupted operand. The last rows contains the sum of all probabilities in each column. This verifies the derived probabilities and must be equal to Equations 8 - 10. The total probabilities in the last column describes the transitions of the complete data flow to any state and the overall sum of $3p-3p^2+p^3$ corresponds to the complementary probability of no faults in Equation 8. But the Table II does not show the case of no faults. With a probability of $1-3p+3p^2-p^3$, the data flow remains in state G.

B2 / B3 - carry-bit is wrong

The important characteristic of the discussed adder is the propagation of the carry-bit as a kind of a memory effect. This means that a fault in one bit stage influences the next bit even though there is no further fault. Corrupted carry-bits are described by the states $B2$ and $B3$ in the Markov model of Figure 1. A similar procedure as for the initial states G / $B1$ are required for the evaluation of these transition probabilities. But now, a corrupted carry-bit from the previous stage is assumed.

- In spite of no additional fault, the incorrect carry-bit propagates the fault to an erroneous sum and also to a wrong carry-bit to the next stage. This means a transition to the states $B1$ or $B3$. In Table IV, the occurrence of these transitions are evaluated in case of no faults.

Table IV
BEHAVIOR OF AN ADDER WITH ONLY A CORRUPTED CARRY-BIT

x y c		s c	x y c'		s' c'	s	c
000	→	0 0	001	→	1 0	X	-
001	→	1 0	000	→	0 0	X	-
010	→	1 0	011	→	0 1	X	X
011	→	0 1	010	→	1 0	X	X
100	→	1 0	101	→	0 1	X	X
101	→	0 1	100	→	1 0	X	X
110	→	0 1	111	→	1 1	X	-
111	→	1 1	110	→	0 1	X	-

- Is there one fault, this is either in any operand or in the addition. In case the fault is in one operand, the outcome of the addition in combination with a corrupted carry-bit changes to a further corrupted carry-bit in halve of the cases (see Table V).

Table V
BEHAVIOR OF AN ADDER WITH CORRUPTED CARRY AND OPERAND

x y c		s c	x' y' c		s' c'	s	c
000	→	0 0	011	→	0 1	-	X
			101	→	0 1	-	X
001	→	1 0	010	→	1 0	-	-
			100	→	1 0	-	-
010	→	1 0	001	→	1 0	-	-
			111	→	1 1	-	X
011	→	0 1	000	→	0 0	-	X
			110	→	0 1	-	-
100	→	1 0	111	→	1 1	-	X
			001	→	1 0	-	-
101	→	0 1	110	→	0 1	-	-
			000	→	0 0	-	X
110	→	0 1	101	→	0 1	-	-
			011	→	0 1	-	-
111	→	1 1	100	→	1 0	-	X
			010	→	1 0	-	X

Table VI
TRANSITION PROBABILITIES FROM $B2/B3$

initial state $B2/B3$	number of faults: 1 fault	2 faults	3 faults	total transition probability
G	O: $p-2p^2+p^3$ A: $\frac{2}{6}(p-2p^2+p^3)$	A2: $\frac{2}{6}(p^2-p^3)$	A: $\frac{2}{6}p^3$	$\frac{4}{3}p-\frac{7}{3}p^2+\frac{4}{3}p^3$
B1	A: $\frac{1}{6}(p-2p^2+p^3)$	A1: $\frac{1}{3}(p^2-p^3)$ A2: $\frac{1}{6}(p^2-p^3)$	A: $\frac{1}{6}p^3$	$\frac{1}{6}p+\frac{1}{6}p^2-\frac{1}{6}p^3$
B2	O: $p-2p^2+p^3$ A: $\frac{2}{6}(p-2p^2+p^3)$	A1: $\frac{1}{3}(p^2-p^3)$ A2: $\frac{2}{6}(p^2-p^3)$	A: $\frac{2}{6}p^3$	$\frac{4}{3}p-2p^2+p^3$
B3	A: $\frac{1}{6}(p-2p^2+p^3)$	O: p^2-p^3 A1: $\frac{1}{3}(p^2-p^3)$ A2: $\frac{1}{6}(p^2-p^3)$	A: $\frac{1}{6}p^3$	$\frac{1}{6}p+\frac{7}{6}p^2-\frac{7}{6}p^3$
sum of column	$\mathbf{3p-6p^2+3p^3}$	$\mathbf{3p^2-3p^3}$	$\mathbf{p^3}$	$\mathbf{3p-3p^2+p^3}$

This means a transition to state G or $B2$. Otherwise, the single fault is caused by the adder itself and the transitions are defined by the original Markov chain (Equation 1).

- Two faults in both operands definitely lead to corrupted sum and carry-bit which corresponds to state $B3$ (see Table VII).

Table VII
ERROR BEHAVIOR OF AN ADDITION WITH CORRUPTED CARRY-BIT AND TWO FURTHER FAULTS

x y c		s c	x' y' c'		s' c'	s	c
000	→	0 0	111	→	1 1	X	X
001	→	1 0	110	→	0 1	X	X
010	→	1 0	101	→	0 1	X	X
011	→	0 1	100	→	1 0	X	X
100	→	1 0	011	→	0 1	X	X
101	→	0 1	010	→	1 0	X	X
110	→	0 1	001	→	1 0	X	X
111	→	1 1	000	→	0 0	X	X

However, there is also the combination of one fault in an operand and the other fault in the adder. The fault in the adder anyhow "overwrites" the fault in the operand. One fault in any operand leads to state G or $B2$ (see before) and these states are the initial states for the faulty addition. (see transition probabilities marked with A1 and A2 in third column of Table VI).

- Three faults in all elements of the data flow means that there are two faulty operands which lead to the initial state $B3$ for the faulty addition. The result is now a transition to any state according Equation 1.

All results of previous evaluation are summarized in Table VI. **O** marks the probability for an error caused by an operand and **Ax** marks the probability for an error caused by the adder with distinction of the initial state.

V. DISCUSSION OF THE MODEL

The presented model in the previous section allows the estimation of fault compensation in a corrupted conjunctive data flow which consists of a simple addition of two operands. Both, the operands and the adder are vulnerable for faults and influence the validity of the result. The effect of fault compensation is very small in the presented case study as shown in Figure 2.

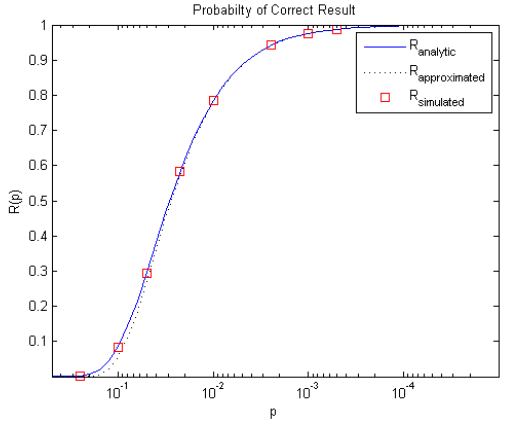

Figure 2. Effect of fault compensation in a given data flow model.

The figure shows the reliability of the data flow as a function of the basic fault probability p. It compares the reliability which is derived from the enhanced Markov chain presented in the previous section (continuous line) with the approximated reliability which ignores the terms p^2 and p^3 (dotted line) in the transition probabilities in Table II and VI. The approximation can be also interpreted as three serially connected elements. But this doesn´t consider the effect of fault compensation and there is the observable deviation between both curves. In this example of a simple data flow, the deviation is obviously negligible. But in real applications, there are usually more instructions forming a data flow. The longer the history of the data flow, the higher the risk for faults and the higher the chance for another fault that compensates the former one. Thus, the effect of fault compensation cannot be ignored in a real data flow with a long history of processing data. It is expected to be more relevant in longer data flow models of real software applications. Furthermore, the Figure 2 shows some sample values which come from simulations depicted as small squares and verifies the correctness of the presented model.

VI. Conclusion

The presented paper shows the fundamental technique of evaluating the reliability of data which is processed on unreliable hardware. The data flow is broken down into several instructions which have dependencies on each other. The total reliability is the composition of all that single reliabilities which possibly influence the correctness of the final result. But with increasing number of instructions, the effect of fault compensation has a bigger impact on this reliability analysis. This means that multiple faults in a given data flow correct each other with a small chance. This effect must be considered to avoid deviations in the total reliability and the more dependencies and instructions the data flow has, the more probably a fault compensating event occurs. Based on the error model of an adder in [23], the Markov chain is extended by faulty operands which have a similar influence on the sum like the faulty addition itself.

In future works the set of error models must be extended for more instructions to analyze the reliability of a real data flow. In addition, permanent faults and the control flow during the software execution still remains uncovered. The pure reliability analysis describes the probability of a correct outcome with any faults. But a further interesting goal is the evaluation of certain outcomes. Using the coded processing approach means that only valid code words are possible outcomes. A fault in the data flow corrupts the result and it is probable no valid code word any more. But what about the seldom case that the result is another valid code word? The verification of the result doesn´t detect the fault. This residual error probability is one important metric for fault tolerant systems.

In Reference [25], they introduced a method to evaluate the reliability of data flow based on extended Markov models on a more abstract level. But there is no information about single elements of the data flow, only the execution time of the data flow is known. With the more detailed knowledge of the data flow concerning the reliability, the transition probabilities in [25] can be adapted and the model is more realistic also with respect to the introduced concept of fault compensation. In addition, the set of reliability models can be also used by a compiler environment for data flow analysis in future. An enhanced compiler knows the data flow and it is possible to evaluate the reliability of a given data flow just after the build process of a software development automatically. So, necessary changes in the data flow to increase the reliability can be done very early in the development process.

References

[1] P. Dodd and L. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Transactions on Nuclear Science*, vol. 50, no. 3, pp. 583 – 602, June 2003.

[2] F. Wang and V. Agrawal, "Single Event Upset: An Embedded Tutorial," in *Proc. 21st International Conference on VLSI Design*, January 2008, pp. 429 – 434.

[3] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, "Static analysis of seu effects on software applications," in *Test Conference, 2002. Proceedings. International*, 2002, pp. 500 – 508.

[4] P. Forin, "Vital coded microprocessor principles and application for various transit systems," in *IFA-GCCT*, 1989, pp. 79–84.

[5] A. Benso, S. Di Carlo, G. Di Natale, and P. Prinetto, "A watchdog processor to detect data and control flow errors," in *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE*, July 2003, pp. 144 – 148.

[6] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, *Software-Implemented Hardware Fault Tolerance*. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.

[7] H. Engel, "Data flow transformations to detect results which are corrupted by hardware faults," in *High-Assurance Systems Engineering Workshop, 1996. Proceedings., IEEE*, October 1996, pp. 279 –285.

[8] N. Oh, S. Mitra, and E. McCluskey, "ED4I: Error Detection by Diverse Data and Duplicated Instructions," *IEEE Transactions On Computers*, vol. 51, pp. 180–199, 2002.

[9] T. R. N. Rao, *Error coding for arithmetic processors*, ser. Electrical science series, H. G. Booker and N. DeClaris, Eds. Academic Press, New York and London, 1974.

[10] J. Mottok, F. Schiller, T. Völkl, and T. Zeitler, "A Concept for a Safe Realization of a State Machine in Embedded Automotive Applications," in *26th Safecomp Conference*, ser. Lecture Notes in Computer Science, vol. 4680. Springer, 2007, pp. 283–288.

[11] U. Schiffel, A. Schmitt, M. Süsskraut, and C. Fetzer, "ANB- and ANBDmem-encoding: Detecting hardware errors in software," in *Computer Safety, Reliability, and Security*, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 6351, pp. 169–182.

[12] U. Schiffel, "Hardware Error Detection Using AN-Codes," Ph.D. dissertation, Technische Universität Dresden, Germany, 2011. [Online]. Available: http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-69872

[13] M. Rebaudengo, M. Sonza Reorda, M. Torchiano, and M. Violante, "Soft-error detection through software fault-tolerance techniques," in *International Symposium on Defect and Fault Tolerance in VLSI Systems*, November 1999, pp. 210–218.

[14] B. Nicolescu and R. Velazco, "Detecting soft errors by a purely software approach: Method, tools and experimental results," *Design, Automation and Test in Europe Conference and Exhibition*, vol. 2, p. 20057, 2003.

[15] P. Gawkowski and J. Sosnowski, "Developing fault injection environment for complex experiments," *11th IEEE International On-Line Testing Symposium*, vol. 0, pp. 179–181, 2008.

[16] S. Felis, J. Mottok, B. Bauer, D. Kohlert, D. Jantz, and M. Laumer, "FBI3 - Fehlereinspeisung auf Hardware-Ebene," in *3. Landshuter Symposium Mikrosystemtechnik*, March 2012, pp. 210–218.

[17] "Functional safety of electrical/electronic/programmable electronic safety-related systems - part 4: Definitions and abbreviations," April 2010.

[18] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri, "A C/C++ source-to-source compiler for dependable applications," in *Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on*, 2000, pp. 71 –78.

[19] P. Ozello, "The coded microprocessor certifcation." in *International Conference on Computer Safety, Reliability and Security*. Springer Munich, 1992, pp. 185–190.

[20] M. Bossert, *Kanalcodierung*, 2nd ed. Teubner, 1998.

[21] C. Osmann, "Bewertung von Codierverfahren für einen störungssicheren Datentransfer - Evaluation of error-correcting codes used for a reliable data transfer," Ph.D. dissertation, Universitaet Duisburg-Essen, 2001. [Online]. Available: http://www.ub.uni-duisburg.de/ETD-db/theses/available/duett-05302001-111522/

[22] R. H. Morelos-Zaragoza, *The Art of Error Correcting Coding*. John Wiley & Sons, Ltd, 2006.

[23] P. Raab, S. Krämer, and J. Mottok, "Error Model and the Reliability of Arithmetic Operations," in *Proceedings of 2013 IEEE EUROCON - International Conference on Computer as a Tool*, July 2013.

[24] R. Billinton and R. Allan, *Reliability evaluation of engineering systems: concepts and techniques*. Plenum Press, 1992.

[25] P. Raab, S. Racek, S. Krämer, and J. Mottok, "Reliability of Task Execution during Safe Software Processing," in *Proceedings of the 15th Euromicro Conference on Digital System Design*, September 2012, pp. 84–89.

A.7 Reliability of Data Processing and Fault Compensation in Unreliable Arithmetic Processors

Authors: P. Raab, S. Kraemer and J. Mottok
Submitted: In *Microprocessors and Microsystems*, September 2013.

The following manuscript was submitted to the journal.

Reliability of Data Processing and Fault Compensation in Unreliable Arithmetic Processors

Peter Raab

Faculty of Applied Science
University of West Bohemia
Univerzitní 8, 306 14 Plzen,
Czech Republic
praab@kiv.zcu.cz

Stefan Krämer, Jürgen Mottok

Faculty of Electronics and Information Technology
Regensburg University of Applied Science
Seybothstr. 2, D-93953 Regensburg,
Germany
{stefan.kraemer,juergen.mottok}@hs-regensburg.de

Abstract

In logical circuits, like arithmetic operations in a processor system, arbitrary faults become a more tremendous aspect in future. Modern manufacturing processes lead to less reliability and higher vulnerability of software execution to soft-errors. The correctness of certain results is important especially for safety-critical applications whose reliability depends on the fault-free execution of each single instruction and the dependencies between them. The more complex a software is the more unreliable the outcome is. But, there is a contrary effect. If the probability for multiple faults increases, there is also the chance that two faults compensate each other and the result is correct again. This paper presents the basic ideas for such a reliability evaluation of a software´s data flow with arbitrary soft-errors and the effect of fault compensation. Further, this evaluation provides a possibility to compare different implementations of a data flow with respect to the reliability. This is shown by the comparison of two different error codes as alternatives for coded data processing.

Keywords: data flow, error probability, fault compensation, reliability analysis, software-implemented hardware fault tolerance (SIHFT)

1 Introduction

The complexity and functionality of electronic control units have more and more increased in several sectors of industry the last years. In addition, the requirements of these systems have become more demanding in terms of safety, reliability and availability. In contrast to this progress, industry demands a decrease in costs for electronics, while at the same time remaining competitive. The use of inexpensive commodity hardware is the result. However, the development of present microcontrollers follows the trend of decreasing feature size in silicon. This leads to less reliability and arbitrary hardware faults are more likely [1]. But despite unreliable hardware, fault tolerance is a requirement of safety-critical applications [2]. This can often be realized by *Software-Implemented Hardware Fault Tolerance* (= SIHFT) in many ways [3]. One simple possibility of hardening the data against *SEU* (= Single Event Upset [5]) is the duplication (= data redundancy) and the multiple computation of data (= time redundancy) [3], [4], [6]. But, only transient faults can be detected by pure data redundancy. Permanent faults in the CPU (e.g. stuck-at fault in the adder hardware) will generate the same erroneous result. The consequence is the use of redundant hardware or of diverse data so that different units of the CPU are used [6]. An example of diverse data is coded data processing which is considered as an important aspect for software-based hardware fault detection in recent applications.

The structure of this paper is as follows:
Section 2 summarizes the related works in the domain of software-based hardware fault detection and coded data processing. Further, Section 3 repeats the necessary background of coded processing and reliability evaluation for better understanding. In Section 4, we introduce the reliability evaluation of a

given data flow and investigate linear codes as an alternative for coded processing based on the previously defined evaluation. The paper proceeds with a discussion of the results in Section 5 and ends with a conclusion for further works in Section 6.

2 Related Works

In literature, they report a lot of pure software-based fault-tolerant approaches which are diverging in the effectivity of fault detection.

The approach of *coded processing* refers to special error detecting or error correcting techniques [29]. But this approach is not limited only to circuits. There are pure software methods available that protect the results of operations in an arithmetic unit by means of error detection codes, as well. The input data are encoded before being processed in an arithmetic unit and the output data are decoded again for verification at the end (Figure 1). With this view, coded processing is related to channel coding as a part of the coding theory.

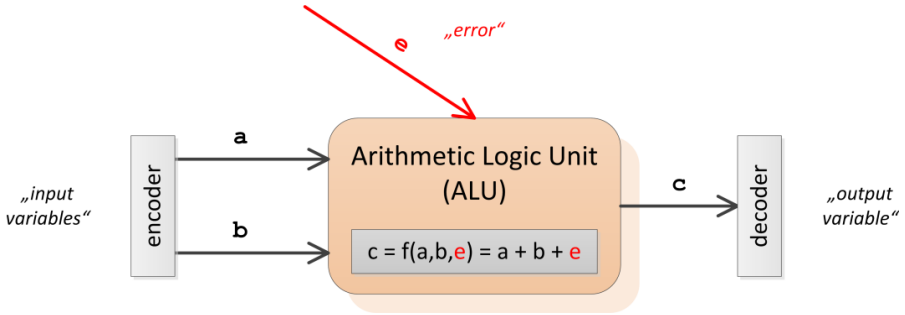

Figure 1: Simplified processor model: The arithmetic unit in a processor represents a channel with respect to arbitrary (transient or permanent) faults which change the value of a result during the execution of an operation.

To be applicable for arithmetic operations, the used error code must preserve the result of the operation as a valid code word. In the past, a lot of codes with this property were described that can be used for arithmetic processors [7], [8], [9], [10], [11], [12]. The most important code which is commonly used for coded processing is the so-called *arithmetic code* (AN-code) whose code words are the product of the constant generator *A* and the information word (Equation 1).

$$\mathbb{C}_{AN} := \{A \cdot X \mid A, X \in \mathbb{Z}\} \tag{1}$$

AN-codes are based on ordinary algebra and preserve the code word with respect to the addition of two code words. This means that the sum of two code words is still a multiple of A and thus it is an element out of the set of code words.

$$C_1 + C_2 = A \cdot X_1 + A \cdot X_2 = A \cdot (X_1 + X_2) \tag{2}$$

But the product of two code words does not match to the coded product of the originals and further corrections would be required.

$$C_1 \cdot C_2 = A \cdot X_1 \cdot A \cdot X_2 = A^2 \cdot (X_1 \cdot X_2) \neq A \cdot (X_1 \cdot X_2) \tag{3}$$

In 1989, Forin made use of AN-codes for coded processing in a real application the first time [9]. He defined coded operations (including additional corrective actions) for most arithmetic operations and he extended signatures to detect operation, operator and operand errors, as well. Furthermore in [13], Ozello discussed the probability of undetection of coded processing. He distinguished between the case where each instruction is verified and the second case when the verification is done after *m*

operations. The latter case is more important for real applications, because the verification of the coded result is usually done at certain points within a task [14]. A possible fault E_1 during the first operation propagates the deviation in the code word with $C_1' = C_1 + E_1$ to the following operation and the result remains faulty also after the second operation ($C_2' = C_2 + E_2$). However, the operation itself or other faulty variables can introduce new faults and influence the final error word. He described this series of deviations by a polynomial $E_g = \{E_1, E_2, \ldots, E_m\}$ with m is the number of operations until the verification of the result is done. His evaluation is independent of the underlying error model of the processor. But with the assumption that the elements of E_g are equally distributed and not zero, he demonstrated that the probability of non-detected faulty code words is $1/A$. This simplification is questionable for real operations and it does not consider the concrete realization of the underlying hardware. Additionally to the effect of the transition from a valid to an invalid code word, there is the effect that consecutive instructions with new error words compensate each other. For example, the sum of two faulty coded variables with deviations E_1 and E_2 results in a valid code word, if the sum of both errors is a multiple of A again:

$$(E_1 + E_2)\ mod\ A = 0 \tag{4}$$

Ozello further remarked that the longer the software the greater the probability to have a polynomial identical to zero which means no deviation in the result. But only programs with more than 10000 lines show this effect.

The ED^4I (= Error Detection by Diverse Data and Duplicated Instructions) approach presented by Oh et.al. [6] is basically a standard method of duplication. A program is executed twice (= instruction duplication) and the data of the copied program are diversely represented. These diverse data are generated by the multiplication of the original data with the so-called diversity factor. For verification, the coded result is compared with the original data at the end. Furthermore, they defined a diversity metric to evaluate several diversity factors with respect to the data integrity and the fault detection probability of different hardware functions (e.g. adder or bus line signals). The diversity factor k determines how diverse the copied program is compared to the original program. They also evaluated several optimal values of k for different hardware functions (e.g. $k = -2$ for an adder). Basically, this approach is a simple example for coded data processing where they used coded data instead of the original. In addition, they defined mathematic models to evaluate and compare different diversity factors with respect to the data integrity and detection probability.

Moreover, Benso et.al. [15] introduced a reliability-weight for each variable in a program which is a function of the variable´s life time and the dependencies to other variables. The life period of a variable is the time between the first write (initialization) of a variable till it is read the last time before it is written again. The life time is then the sum of all life periods and the longer this time is the higher the probability of being corrupted. Variables with a high reliability-weight are usually more critical for the reliability of the application. Another criterion for reliable variables is the dependencies to other variables. For example in the calculation $c = a + b$, the variable c depends on the variables a and b. In a complex program, the dependencies of a certain variable can have a lot of descendants which are able to propagate possible faults. Thus, it is highly critical for the reliability of the final variable. The more variables are necessary for the computation of a final result the higher the risk that it is corrupted by an SEU.

The validation of software-based approaches for hardware fault detection like coded data processing is an important proof of efficiency. Under normal operation, the occurrence of hardware faults as a root cause for system failures is a very seldom event. This makes it difficult to test a given approach. Consequently, the disturbance by the environment must be artificially increased to reduce the time until a fault happens. So, this is equivalent to a higher fault probability. But, these faults still remain random and a systematic test of all types of faults is not possible. Therefore, the literature reports a lot of fault injecting approaches [16],[17],[18] whereas analytical methods are not described. Thus, the main motivation for this work is the development of certain error models to describe the erroneous output of a given arithmetic operation as an alternative for state-of-the-art fault injection techniques. But in contrast to [6], our error models determine the basic reliability of a single hardware module without any fault detection methods (like duplication) in a first step. Thus, these error models are comparable to so-called *channel models* which are known from coding theory (see Chapter 3).

3 Background of theory

A channel model is an important mean in information theory to approximate the behavior of real noisy transmission channels by a probabilistic approach. For example, a simple channel model is the so-called *binary symmetric channel* (= BSC) in Figure 2. This BSC model allows us to calculate the probability of faulty bits and further the residual error probability of error detecting codes based on a simple transmission line [19], [21]. The residual error probability describes the chance for a received code word to be corrupted in a way that it is again a valid code word after reception. The error in a transmitted code word is not detected in this case.

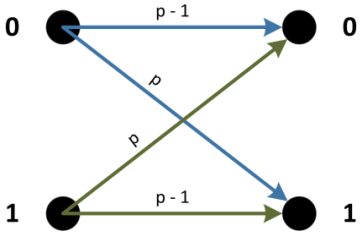

Figure 2: The binary symmetric channel model describes the probability p that a single bit changes its value or remains unchanged (1 - p).

The data processing in arithmetic units of a microcontroller uses hardware based operations, which also represent a kind of channel. During the software processing, arbitrary, permanent or transient hardware faults can occur and they probably change the value of the result. But in contrast to transmission systems, an arithmetic operation is usually more complex with at least one input and the output is a function of these inputs (Figure 3).

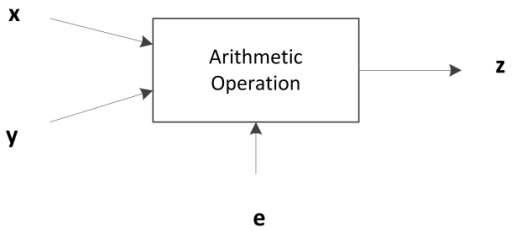

Figure 3: An arithmetic operation usually processes at least one input and generates an output as a function of the inputs and the arbitrary error (z = f(x, y, e)).

In a processor system, there is a set of arithmetic operations and other components which all take part in data processing and are vulnerable to arbitrary faults. There is for example a memory which stores data and instructions which deal with that data (Figure 4).

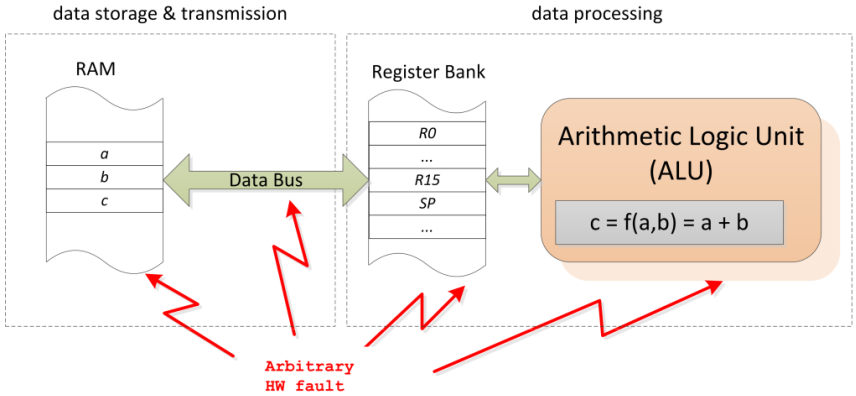

Figure 4: Simplified fault model of a processor [26].

Data errors have their cause either in corrupted memory or in bit-flips during accessing the memory bus or during computation in the ALU (= arithmetic logic unit). In contrast to data transmission and storage systems, where channel models are state-of-the-art for error estimation, comparable models are not available for data processing channels [13].

A first attempt of such an error model for an arithmetic operation was done in [20]. The ripple-carry-adder as an iterative operation is described by the discrete Markov chain in Figure 5 to model the faulty behavior caused by the dependencies between the carry-bits and the following digits.

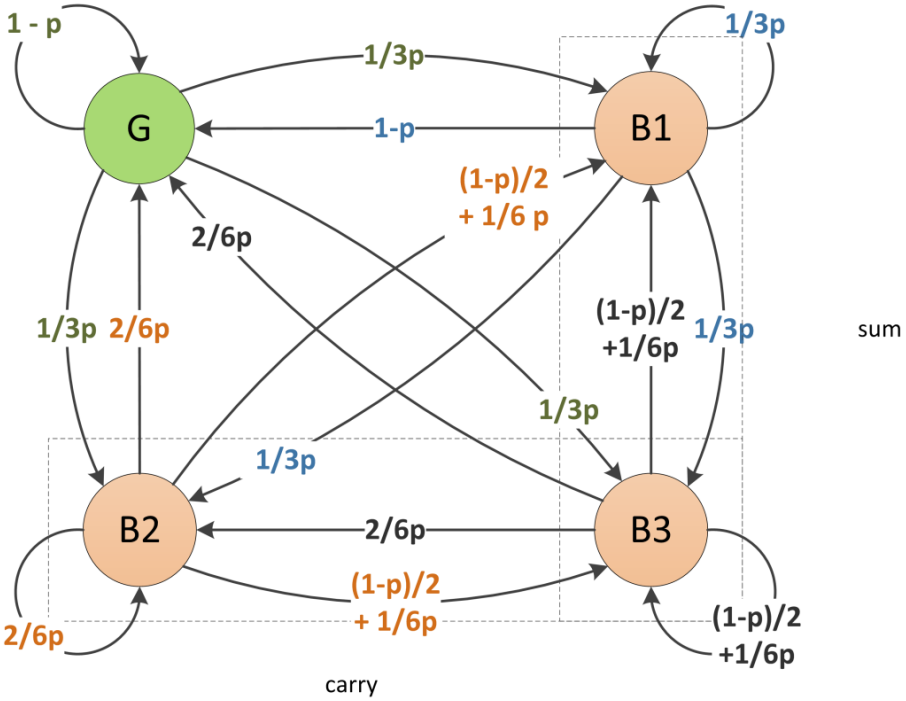

Figure 5: Discrete Markov chain which models the iterative process of a ripple-carry-adder [20].

Figure 5 describes the error states of the iterative addition process. Referred to a 1-bit adder, there are four error states which are defined by the two outputs sum s and carry c_{out}. Either no fault has occurred and both outputs are correct (state G) or there is a fault and at least one output is wrong (B1 - only sum is wrong; B2 – only the carry is wrong; B3 – both are wrong). In [20], it is assumed that in case of a fault, the adder changes to one of the error states Bx with an equal probability. This is still a simplification and the distribution must be adapted for the concrete realization of the adder. But by means of this model, the reliability of each bit in the sum can be calculated and further the probability of undetection in case of coded data.

For reliability evaluations, Markov models are often used and a very powerful tool. The example of an arithmetic instruction in [20] is a small system and only covers a single step in a given data flow. The reliability of the outcome of a complete task depends on all executed instructions. So, an evaluation of the total reliability requires all error models for each instruction in the data flow. However, this set of error models is not yet available and the reliability of a data flow must be determined in a more abstract manner. Therefore, the data flow is modeled by means of a continuous-time Markov model, in which the runtime is an important characteristic [27]. The longer the runtime which means the more instructions are executed, the higher the risk for an incorrect result at the end. But the basic concept of Markov models cannot be used to describe more regular events like task runtime. In contrast, continuous-time Markov models describe events that randomly occur with a constant rate (= exponential distribution). However, the *Erlang* distribution [22], [23] can be considered as a composition of several exponential distributed events. In this way, it approximates more regular distributions like the normal distribution which matches to a task´s runtime better. The result of this consideration leads to the multi-staged Markov model in Figure 6. Each vertical state represents a certain time interval and the last state is entered with the termination of the task. The

columns represent the number of active faults. Depending on the amount and type of redundancy, there is a maximum number of surely detectable faults. In case of a detected error, a possible recovery is the repetition of the whole task to remain operational. In Figure 6, this repetition is marked by the transitions back to the initial state 1.0.

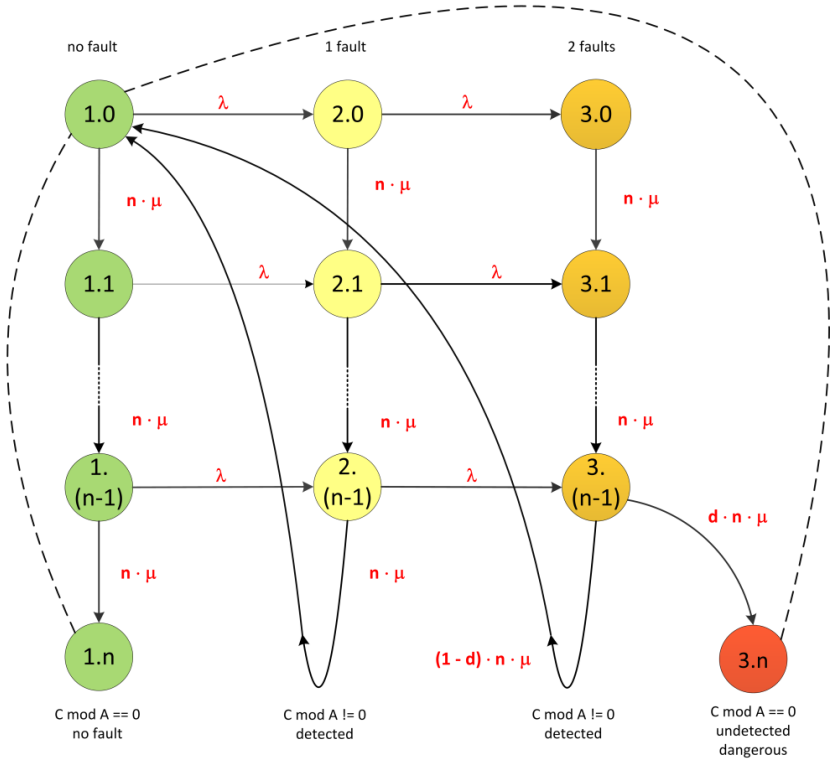

Figure 6: Extended Markov model for a coded data flow [27].

But the presented approach in [27] does not show any details of the task implementation. Each single vertical stage corresponds to a certain time interval within the data flow. With a constant time of an instruction, a single stage can be considered as an instruction which is defined by the incoming transition. So, it is possible to map all instructions into the Markov model with an individual fault probability (horizontal transitions) and runtime (vertical transitions).

4 Results

Based on the presented approaches for error models from the previous background section, we now discuss reliability related issues in data processing like residual error probability, fault compensation and alternative codes for coded data processing.

4.1 Reliability of Data Processing

The data flow of software describes the dependencies of variables and the order of their processing. In other words, the reliability of the final result depends on the correct execution of each previous instruction and the storage in the memory. A simple example of a data flow is the addition of two integer numbers corresponding to the pseudo assembler code in Listing 1.

```
mov   #3 , r1          ; r1 = 3
mov   #5 , r2          ; r2 = 5
add   r1 , r2 , r3     ; r3 = r1 + r2
```

Listing 1: Pseudo assembler code for the addition of two integer numbers.

Listing 1 shows a set of assembler instructions which adds two integer numbers stored in the working registers *r1* and *r2*. The reliability of the sum depends on the fault-free execution of each single instruction. With respect to the total reliability, the given data flow can be illustrated by three serially connected components and the outcome is only correct, if the first *MOV* instruction and the second *MOV* instruction and the final *ADD* instruction work correctly. The total reliability is then the product of all single reliabilities:

$$R_0 = R_{mov} \cdot R_{mov} \cdot R_{add} \tag{5}$$

Generally, a single instruction depends on the register width of an arithmetic processor. Each digit potentially injects a single fault into the instruction. In the theory of probability, this means *n* independent events with a common fault probability *p*. But the correctness of the instruction relies on the fault-free execution of each digit. With the complementary probability *(1 - p)* for no fault, the binomial distribution describes the probability of *r* faults out of *n* bits.

$$p_r = \binom{n}{r} \cdot p^r \cdot (1-p)^{n-r} \tag{6}$$

With no faults (*r = 0*), the Equation 6 is reduced to

$$p_0 = (1-p)^n \tag{7}$$

and it represents the reliability of a single *n*-bit instruction. Assuming the same reliability for each instruction ($R_{mov} = R_{add}$), the total reliability R_0 of Listing 1 is then

$$R_0 = p_0^3 = (1-p)^{3n}. \tag{8}$$

Generally for any number of instructions *k*, it is

$$R_0 = (1-p)^{kn}. \tag{9}$$

The Equation 9 shows that the probability of a correct result decreases with increasing number of consecutive instructions. Actually, the reliability converges to zero with an infinite number of instructions. However, the Equation 9 does not consider additional effects like fault compensation. As we will see in Section 4.2, fault compensation cannot be ignored in huge software systems and high fault probabilities.

As a further note, the previous investigation has a simple view on the given data flow. In [20], we showed that the digits of an addition are not independent because of the carry-bit propagation. This is a very important issue if we want to estimate the probability of a certain erroneous result. Furthermore, we extended the error model of an addition by faulty operands in [28]. But in the course of this article, we leave the assumption of independent bits and we investigate the effect of fault compensation in data processing in a general manner.

4.2 Fault Compensation

Fault compensation is the effect that the final result after a set of consecutively executed instructions is correct despite of faults. But this effect only works with an even number of contrary faults. The longer the data flow the higher the number of instructions and the higher the chance for such an effect. In other words, the fault compensation effectively increases the reliability with increasing number of instructions. This effective reliability R_e is higher than it is expected by the multiplication of all single reliabilities. With the compensation factor $r_c > 1$, the effective reliability is then

$$R_e = r_c \cdot \prod_k R_k. \tag{10}$$

In [20], we introduced the concept of *error masks* as the bit-wise covering of a result C with faults described by the exclusive-OR:

$$C' = C \oplus E \tag{11}$$

Assuming an independent sequence of faulty instructions, the error mask after the k-th instruction is the superposition of all previous error masks by

$$E = E_1 \oplus E_2 \ldots \oplus E_k. \tag{12}$$

Thus, the deviation of a single bit in the result depends on all previous faults in the same digit at which only an odd number of faults causes a deviation. In general, the probability of a correct bit p_{E0} in the result is the total probability of an even number of faults in k instructions:

$$p_{E0} = \sum_{\substack{r=2n \\ n \in \mathbb{N}_0}}^{k} \binom{k}{r} \cdot p^r \cdot (1-p)^{k-r} \tag{13}$$

With the relation between the binomial and the Poisson distribution [30], the Equation 13 can be transformed to

$$p_{E0} = \sum_{\substack{r=2n \\ n \in \mathbb{N}_0}}^{k} \frac{pk^r}{r!} \cdot e^{-pk}. \tag{14}$$

The same applies for the probability of corrupted bits, but with an odd number of faults:

$$p_{E1} = \sum_{\substack{r=2n+1 \\ n \in \mathbb{N}_0}}^{k} \frac{pk^r}{r!} \cdot e^{-pk} \tag{15}$$

Further, Equation 14 contains the power series of the *hyperbolic cosine* [31]

$$\cosh(x) = 1 + \frac{x^2}{2!} + \frac{x^4}{4!} + \cdots \tag{16}$$

with $x = pk$. Thus, it follows

$$p_{E0} = \cosh(pk) \cdot e^{-pk} \tag{17}$$

for a single bit and the effective reliability is

$$R_e = p_{E0}^n = \cosh(pk)^n \cdot e^{-pkn} \tag{18}$$

for all *n* digits of a processor word. The term e^{-pkn} is identical to the binomial expression $(1-p)^{kn}$ for small *pk* and corresponds to the basic reliability R_0 of Equation 9 (see also Chapter 6.6.3 in [30]).

$$R_e = \cosh(pk)^n \cdot R_o \tag{19}$$

The hyperbolic cosine has the property of $\cosh(x) > 1$ for all $|x| > 0$. This means that with a growing number of consecutive instructions *k* or higher fault probability *p*, the effective reliability R_e is higher by more compensating faults than the basic reliability R_0 (*Figure 7*).

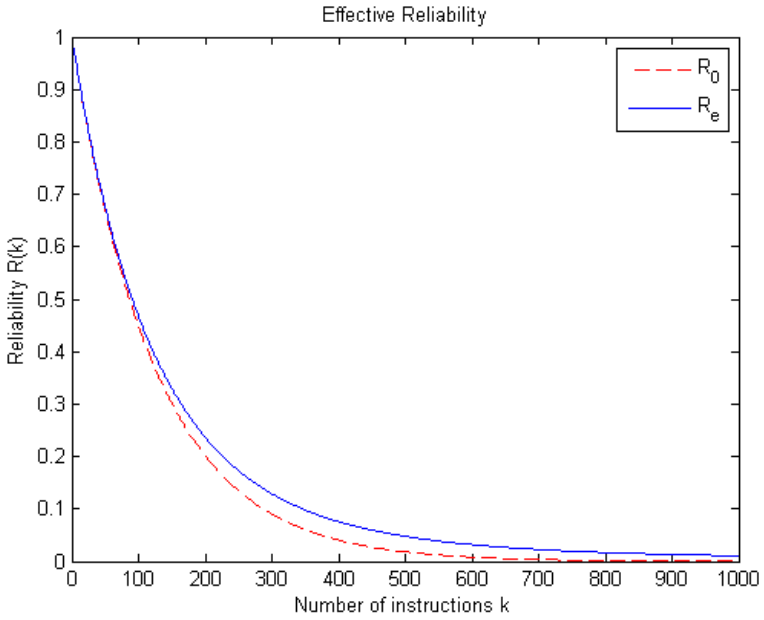

Figure 7: The effective reliability R_e is higher than the basic reliability R_0 without considering fault compensation.

Assuming an infinite number of instructions in Equation 17, we can evaluate the steady-state probability of a single bit. The hyperbolic cosine has the characteristic that it converges to

$$\lim_{x\to\infty} \cosh(x) \approx \frac{1}{2} \cdot e^x \tag{20}$$

with increasing argument *x*. This means that the limit of the probability of a correct bit in a processor word after infinite instruction is

$$\lim_{k\to\infty} p_{E0} \approx \frac{1}{2} \cdot e^{pk} \cdot e^{-pk} = \frac{1}{2}. \tag{21}$$

Or in other words, there is a minimum reliability of a single bit or even of the whole n-bit result of

$$R_{min} = \frac{1}{2^n}. \tag{22}$$

This minimum reliability manifests itself as a horizontal asymptote of curve R_e unequal to zero.

4.3 Linear Codes for Coded Data Processing

The previous sections presented a reliability evaluation of a complete data flow. One advantage of such an analysis is the comparison of different realizations of a given data flow or even the comparison of different codes in case of coded processing. Now, we can investigate the effectivity of an error code based on a certain data flow.

AN-codes are usually used for coded data processing in arithmetic operations. But what is about other classes of codes. Linear codes are commonly known from transmission systems (e.g. Hamming codes, cyclic redundancy codes/CRC). Similar to arithmetic codes, the linear code words are generated by a multiplication (Equation 23).

$$\mathbb{C}_{CRC} := \{g(z) \odot x(z) \,|\, g(z), x(z) \in \mathbb{Z}_2[z]\} \tag{23}$$

In contrast to arithmetic codes, linear codes are based on the algebra of polynomials, whereas arithmetic codes use an ordinary addition upon integer numbers. The addition of polynomials differs from that of integer numbers by the missing carry-bit propagation. When using linear codes instead of arithmetic codes, additional corrective actions must be done (Equation 24).

$$X + Y = x(z) \oplus y(z) \oplus c(z) \tag{24}$$

Based on the BSC model, linear codes have a better residual error probability than arithmetic codes [26]. The *XOR* operation $\oplus$ in Equation 24 matches the characteristic of the BSC model because of the missing influence between the bits in contrast to the addition (ripple-carry-adder). The reliability model in Section 4.1 and the model of fault compensation in Section 4.2 can be used. With $k = 2$, Equation 19 determines the reliability of the result after two consecutively executed *XOR* operations. This reliability is shown as a function of p in Figure 8 (continuous line). Furthermore, the figure also shows the reliability of a single adder (dotted line) derived by the adder model in [20] for comparison. Clearly, one single operation is more reliable than two because of the longer data flow.

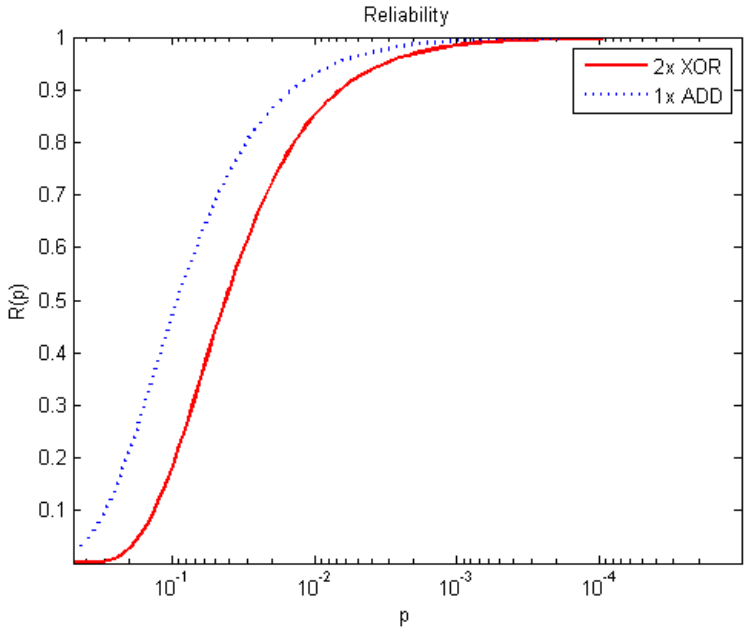

Figure 8: Comparison of the reliability: (1) two consecutive XOR instructions (2) one single ADD instruction.

Using linear codes, the result after the two *XOR* operations in Equation 24 is a valid code word in case no faults have occurred. However, there is the probability that certain faults lead to another code word

and the error is not detected. A given linear code is characterized by the distribution of the *Hamming weight* W_i over all valid code words [24], [25]. The Hamming weight of a code word is the number of non-zero bits [21]. By means of the binomial distribution, the probability of a certain Hamming weight can be determined. Thus, the total probability of getting any valid code word (= *residual error probability*) is

$$P_u(p) = \sum_{r=d_{min}}^{n} W_r \cdot p^r \cdot (1-p)^{n-r} \tag{25}$$

with d_{min} is the minimum Hamming distance, p is the probability of non-zero bits (p_{E1}, Equation 15) and (1-p) is the probability of correct bits ($p_{E0,}$ Equation 14) in the error mask. With k = 2, it is

$$p_{E0} = (1-p)^2 + p^2 \tag{26}$$

and

$$p_{E1} = 2 \cdot p \cdot (1-p). \tag{27}$$

It follows for the residual error probability of the final result:

$$P_u(p) = \sum_{r=d_{min}}^{n} W_r \cdot p_{E1}^r \cdot p_{E0}^{n-r} \tag{28}$$

The result of Equation 28 is depicted in Figure 9. The curve a) represents the residual error probability as a function of p for the linear code with generator polynomial $g(z) = z^3+z+1$ and two consecutive *XOR* instructions. For comparison, the residual error probability of the AN-code with the ordinary addition from [20] is also depicted in the figure (A = 3, curve b)). With the same code rate, the residual error probability of the arithmetic code in combination with the single addition is always higher than the comparable linear code processed by two *XOR* operations. This would mean that linear codes are better for coded data processing than arithmetic codes because the probability of non-detection is lower in spite of more instructions (= computational effort).

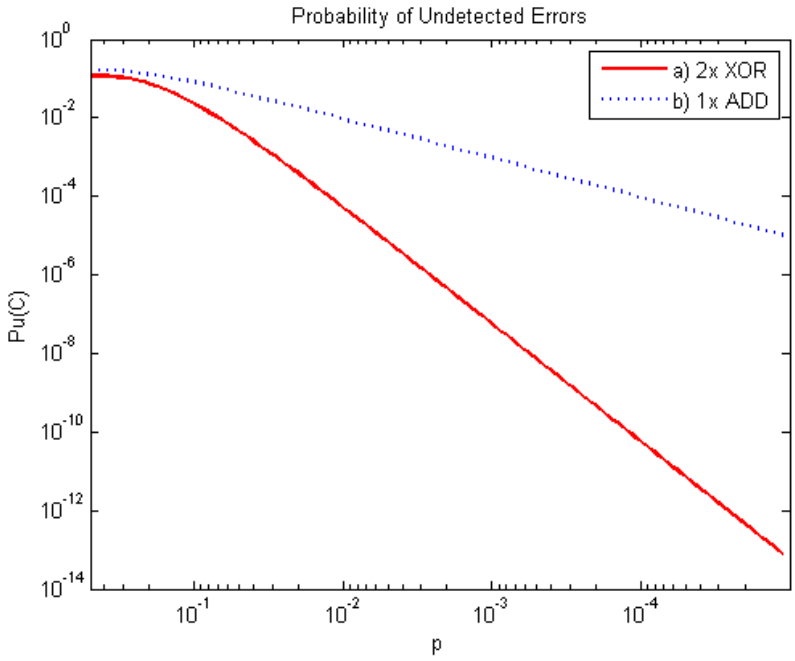

Figure 9: Residual error probability of linear codes and arithmetic codes.

But, can both results be compared? The presented approach assumes Equation 24 as a replacement of an ordinary addition. There is no consideration of the carry-bits, which are part of the ordinary addition. In real microcontrollers, there is no instruction available to calculate only the carries of an addition. There is the way to extract the carries like this

$$c(z) = \varphi^{-1}(X+Y) \oplus x(z) \oplus y(z). \tag{29}$$

Or additional synthesized hardware in an FPGA can generate the carries. But all solutions have an additional unreliability to be considered. And this means that the curve a) in Figure 8 will be shifted up and the advantage of the linear codes decreases or maybe disappears. As a conclusion of this investigation, linear codes are probable not applicable for coded data processing by arithmetic operations. However following [26], linear codes are optimal for data storage and transmission. And with two different error codes which are applicable in data processing systems, there is the need for code transformations as it is described in [32].

5 Discussion

The data flow of software consists of a set of instructions executed in a defined order. Assuming unreliable hardware which arbitrarily injects faults into any instructions, there is the effect of fault compensation which is already reported in [13], as well. In the previous section, we derived a mathematical model which describes this effect within a strict linear data flow. However, fault compensation is not restricted to this kind of data flow. In [28], the fault compensation between the operands and a single addition was shown as an example of a conjunctive data flow. Further, it was shown that the effect of fault compensation cannot be ignored for reliability evaluations with an increasing complexity of software and with an increasing number of instructions.

Indeed, the evaluation of a complete data flow is very complex and it is not possible without all required error models of each instruction. Thus, there is the simple data flow of Equation 24 in Section 4.3 for this discussion. The given data flow of two consecutively executed *XOR* instructions represent a linear flow without any branches and merges and furthermore it is an example of coded data processing using linear codes as another aspect of the evaluation. There are several cases to distinguish: First, both instructions are executed without any faults according Equation 6 with

$$R_0 = (1-p)^{2n}. \tag{30}$$

Second, both instructions represent independent events with respect to their fault vulnerability and fault compensation is possible. According Equation 19, the effective reliability is increased by the factor

$$r_c = \cosh(2p)^n. \tag{31}$$

With a basic fault probability p = 10e-6 and a data width n = 8, the factor of Equation 31 is r_c = 1+16e-12. But this covers the data flow including two instructions only. With increasing number of instructions (e.g. k = 100000), the factor increases to r_c = 1.04 which is obviously higher than before and it cannot be ignored any more. A similar behavior can be observed with increasing fault probability p.

The remaining faults should be detected by using coded data words, but there is still a small residual risk for undetection (Equation 28). Figure 10 qualitatively summarizes all outcomes of the data flow.

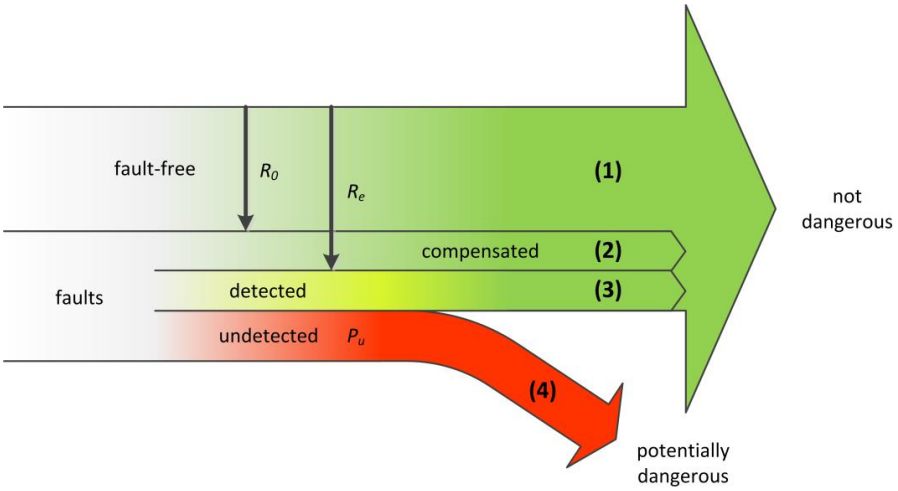

Figure 10: The entire probability space is partitioned into several groups: (1) fault-free execution, (2) fault compensation, (3) fault detection by error codes and (4) undetection of faults determines the residual error probability.

A further aspect of fault compensation is related to Publication [27]. Here, we introduced an extended Markov model which describes the error behavior of a given data flow in a more abstract layer (see Figure 6 in Section 3). It defines several system states which correspond to the number of active faults (columns of the Markov model) and the probability of the detection. But this model doesn´t cover any fault compensation as discussed before. The compensation of a fault resets the system back to a fault-free state or rather it reduces the number of active faults. Thus, there are additional transitions to the left column with an increasing rate depending on the progress of the data flow (Figure 11).

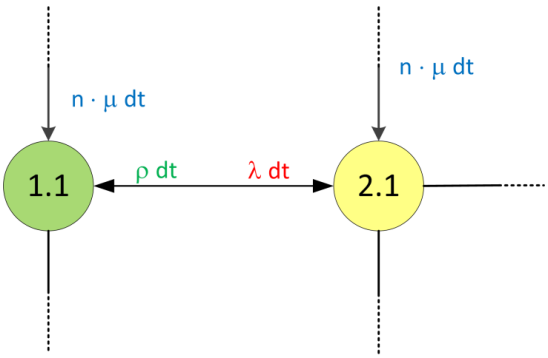

Figure 11: Detail of the extended Markov model with the effect of fault compensation by additional transitions.

6 Conclusion

The presented paper investigated the effect of fault compensation during the data processing on unreliable hardware. Because of changing manufacturing processes, the risk for arbitrary faults in logical circuits becomes a bigger impact on recent and especially in future systems. Thus, it becomes more probable for multiple faults in more complex software systems and therefore the chance for compensation rises. This fault compensation effectively increases the reliability and it must be considered in such reliability analysis to avoid a deviation in the calculations. It was shown that the influence of fault compensation becomes bigger with increasing number of instructions (Section 4.2).

This paper also shows a possible evaluation for a strict linear data flow wherein there are no dependencies between nearby bits. This is still a simplification because in a data flow there are also more complex instructions related to the error behavior and the dependencies to each other and between nearby bits. In future works, the set of error models will be extended to further instructions to derive a complete reliability model for a given data flow and also for a control flow. Such a reliability model can also be used by a compiler environment for data flow analysis. An enhanced compiler knows the data flow and it is possible to evaluate the reliability of a given data flow just after the build process of a software development automatically. So, necessary changes in the data flow to optimize the reliability by software can be done very early in the development process.
This paper also investigates the alternative usage of linear codes instead of arithmetic codes (AN-codes). The ordinary addition of two integer numbers can be replaced by two polynomial additions of the operands and the resulting carry-bits of both. However, there is no possibility to create the carry polynomial without further (unreliable) instructions and the presented evaluation is not complete. But it already shows the better error performance of linear codes in combination with BSC-based instructions (e.g. *XOR, MOV*). The conclusion of this study is the existence of an optimal code for a given instruction (= channel). With different channels in an arithmetic processor, the usage of different codes would increase the error detection probability by matching the characteristic of the underlying hardware. But, the code words must be transformed to each other depending on the currently executed instruction. Such a transformation requires additional execution of unreliable hardware. In [32], we already presented a possible transformation rule of linear code words to arithmetic code words and vice versa for this purpose.

Acknowledgement

This work is a result of the research project *Safe Oriented Programming of Software-Intensive Embedded Systems* (S^3OP). This project was established and carried out at the *Laboratory for Safe and Secure Systems* (LaS^3), which is located at the University of Applied Sciences Regensburg, in cooperation with the University of West Bohemia in Pilsen. The S^3OP project was supported by the Bavarian State Ministry for Science, Research and Arts (Code: D2-F1116.RE/3/4).

References

[1] P. Dodd and L. Massengill, “Basic mechanisms and modeling of single-event upset in digital microelectronics,” IEEE Transactions on Nuclear Science, vol. 50, no. 3, pp. 583 – 602, June 2003.
[2] Functional safety of electrical/electronic/programmable electronic safety-related systems - part 1: General requirements, April 2010.
[3] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante, Software-Implemented Hardware Fault Tolerance. Secaucus, NJ, USA: Springer-Verlag New York, Inc., 2006.
[4] B. Nicolescu and R. Velazco, “Detecting soft errors by a purely software approach: Method, tools and experimental results,” Design, Automation and Test in Europe Conference and Exhibition, vol. 2, p. 20057, 2003.
[5] F. Wang and V. Agrawal, “Single Event Upset: An Embedded Tutorial,” in Proc. 21st International Conference on VLSI Design, January 2008, pp. 429 – 434.
[6] N. Oh, S. Mitra, and E. McCluskey, “ED4I: Error Detection by Diverse Data and Duplicated Instructions,” IEEE Transactions on Computers, vol. 51, pp. 180–199, 2002.
[7] D. Brown, “Error detecting and error correcting binary codes for arithmetic operations,” in IRE Trans. Electron. Comput., 1960, pp. 333–337
[8] Thammavarapu R. N. Rao. Error coding for arithmetic processors. Electrical science series. Academic Press, New York and London, 1974.
[9] P. Forin, “Vital coded microprocessor principles and application for various transit systems,” in IFA-GCCT, 1989, pp. 79–84.
[10] David Mandelbaum. Error Correction in Residue Arithmetic. Computers, IEEE Transactions on, C-21(6):538 _545, June 1972.

[11] Hao Dong. Berger Codes for Detection of Unidirectional Errors. Computers, IEEE Transactions on, C-33(6):572 _575, June 1984.
[12] U. Schiffel, A. Schmitt, M. Süsskraut, and C. Fetzer, "ANB- and ANBDmem-encoding: Detecting hardware errors in software," in Computer Safety, Reliability, and Security, ser. Lecture Notes in Computer Science. Springer Berlin / Heidelberg, 2010, vol. 6351, pp. 169–182.
[13] P. Ozello, "The coded microprocessor certifcation." in International Conference on Computer Safety, Reliability and Security. Springer Munich, 1992, pp. 185–190.
[14] P. Raab, S. Kramer, J. Mottok, H. Meier and S. Racek. Safe Software Processing by Concurrent Execution in a Real-time Operating System. In Applied Electronics (AE), 2011 International Conference on, pages 315-319, Pilsen, September 2011.
[15] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri, "A C/C++ source-to-source compiler for dependable applications," in Dependable Systems and Networks, 2000. DSN 2000. Proceedings International Conference on, 2000, pp. 71–78.
[16] Mei-Chen Hsueh, T.K. Tsai, and R.K. Iyer. Fault injection techniques and tools. Computer, 30(4):75_82, 1997.
[17] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G.H. Leber. Comparison of physical and software-implemented fault injection techniques. Computers, IEEE Transactions on, 52(9):1115_1133, 2003.
[18] Jeffrey M Voas and Gary McGraw. Software fault injection: inoculating programs against errors. John Wiley & Sons, Inc., 1997.
[19] R. H. Morelos-Zaragoza, The Art of Error Correcting Coding. John Wiley & Sons, Ltd, 2006, pp 1-21.
[20] P. Raab, S. Krämer, and J. Mottok, "Error Model and the Reliability of Arithmetic Operations," In 2013 IEEE EUROCON - International Conference on Computer as a Tool, pages 630-637, July 2013.
[21] M. Bossert, Kanalcodierung, 2nd ed. Teubner, 1998.
[22] E. Härtter. Wahrscheinlichkeitsrechnung, Statistik und mathematische Grundlagen: Begriffe, Definitionen und Formeln. Vandenhoeck & Ruprecht, 1987.
[23] V. Vais and S. Racek. Experimental evaluation of regular events occurrence in continuous-time markov models. In Informatics, 2011, Nov. 2011.
[24] R. W. Hamming, "Error detection and error correction codes," The Bell System Technical Journal, vol. 26, no. 2, pp. 147– 160, April 1950.
[25] W. E. Peterson W.W., Error Correcting Codes, 2nd ed. MIT Press, 1972.
[26] P. Raab, S. Krämer, and J. Mottok, "Cyclic codes and error detection during data processing in embedded software systems," in Proceedings of the 4rd Embedded Software Engineering Congress, December 2011, pp. 577–590.
[27] P. Raab, S. Kramer, S. Racek and J. Mottok. Reliability of Task Execution during Safe Software Processing. In Proceedings of the 15th Euromicro Conference on Digital System Design, September 2012.
[28] P. Raab, S. Kramer, S. Racek and J. Mottok. Data Flow Analysis of Software Executed by Unreliable Hardware. Accepted for the 16th Euromicro Conference on Digital System Design, September 2013.
[29] IEC 61508-7. Functional safety of electrical/electronic/programmable electronic safety-related systems - Part 7: Overview of techniques and measures, April 2010.
[30] R. Billinton and R.N. Allan. Reliability evaluation of engineering systems: concepts and techniques. Plenum Press, 1992, pp. 171-173.
[31] L. Papula, Mathematische Formelsammlung für Ingenieure und Naturwissenschaftler: Mit zahlreichen Rechenbeispielen und einer ausführlichen Integraltafel, ser. Viewegs Fachbücher der Technik. Vieweg, 2006, p. 180.
[32] P. Raab, V. Vavricka, S. Krämer and J. Mottok. Isomorphism between Linear Codes and Arithmetic Codes. ***Accepted*** for Computing and Informatics (CAI).