University of West Bohemia
Faculty of Applied Sciences

Doctoral Thesis

December 2015

Stefan Krämer, M. Eng.

University of West Bohemia
Faculty of Applied Sciences

Development and Simulation of Fault-Tolerant Multicore Real-Time Scheduling Covering Transient Faults

Stefan Krämer

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 2015

Západočeská univerzita v Plzni
Fakulta aplikovaných věd

Návrh a simulace FT plánovacího algoritmu pro vícejádrový procesor a RT aplikace

Stefan Krämer

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 2015

Abstract

The request for more powerful processing units is not only rising in the field of PCs, but also in the embedded domain. Nowadays embedded systems with a high demand for performance can be found almost everywhere, starting from households, entertainment devices and in the automotive domain [1]. Furthermore the latter have an increasing demand for safety, reliability and dependability. In the future the trend towards multi-processors in the entertainment and consumer industry will be visible moreover in the embedded domain in automotive and the avionic industry. Multicore processors are not only advantageous regarding the performance. But they also offer opportunities to increase the reliability and dependability by introducing redundancy, which can be flexibly implemented in software. These embedded systems often have hard real-time requirements, which as well result in challenges regarding multicore real-time scheduling, especially if global dynamic real-time schedulers are used.

Combining these with the requirement of high reliability and the demand for dependability, results in the necessity to develop new software concepts. Especially scheduling strategies are required that can harden a system against such (transient) faults – with minimum additional hardware effort and therefore produce competitive and cost-effective embedded systems.
In this thesis, discrete event simulation is used to evaluate the benefits of the newly developed dynamic hard real-time multicore scheduling algorithm LB-*Pfair* under the presence of transient hardware faults.

Keywords

Multicore, Manycore, Multiprocessing, SMP, Multicore-Scheduling, Safety, Reliability, Monte-Carlo Simulation, Reliability Analysis, Safe Software Processing, real-time Operating System, Discrete Event Simulation

Abstrakt

Zvýšené požadavky na větší výpočetní výkon se nevyskytují pouze v oblasti PC, ale také v oblasti vestavěných počítačových systémů. Současné vestavěné systémy s požadavkem na zvýšený výpočetní výkon lze nalézt v mnoha aplikacích počínaje jejich využitím v domácnosti, zařízeních pro zábavu a v automobilech [1]. Navíc zejména v posledním jmenovaném využití přistupují požadavky na bezpečnost a spolehlivost zařízení. V budoucnosti lze předpokládat, že současný trend využití multiprocesorů ve spotřební elektronice se více projeví také v automobilovém a leteckém průmyslu. Přitom vícejádrové procesory nejsou výhodné pouze s ohledem na zvýšený výkon, ale nabízí také příležitost zvýšit spolehlivost zařízení zavedením redundantních prvků, které mohou být pružně implementovány i prostřednictvím SW. Uvažované vestavěné systémy často mají také zvýšené požadavky na dodržení časových limitů výpočtu, což je výzvou pro zdokonalování plánování úkolů ve vícejádrových procesorech, zejména pokud jde o globální dynamické real-time plánovače úkolů.

Uvedené okolnosti spolu se zvýšenými požadavky týkajícími se spolehlivosti systému jsou motivací pro vývoj nových přístupů k vývoji SW. Zejména jsou zapotřebí plánovací postupy, které způsobí zvýšení odolnosti systému proti přechodným poruchám s minimální potřebou dodatečného HW, takže výsledný systém vyjde jako schopnější konkurence a zároveň jako cenově dostupný.

K vyhodnocení výhod nově navrženého dynamického plánovacího algoritmu LB-Pfair pro vícejádrové procesory a časově kritické aplikace s uvažováním výskytu přechodných poruch HW je v předložené disertační práci využita metoda diskrétní simulace orientované na události.

Klíčová slova

Vícejádrové procesory, multiprocesorový výpočet, SMP, plánování, bezpečnost, spolehlivost, Monte-Carlo simulace, spolehlivostní analýza, bezpečné softwarové zpracování, operační systémy reálného času, simulace orientovaná na události.

Zusammenfassung

Die Nachfrage nach immer leistungsfähigeren Prozessoren steigt nicht nur in der Konsumerelektronik, sondern auch im Bereich der eingebetteten Systeme. Eingebettete Systeme mit hohen Performanzanforderungen sind heutzutage überall vertreten, angefangen bei Haushaltsgeräten, über die Unterhaltungselektronik bis hin in die Automobilindustrie [1]. Sicherheit und Zuverlässigkeit der Systeme rücken immer weiter in den Fokus. Mehrkernsysteme sind aus der Konsumerelektronik und der Unterhaltungselektronik nicht mehr wegzudenken. Auch in der Automobilindustrie ist der vermehrte Einsatz von Mehrkernprozessoren zu erkennen. Mehrkernprozessoren bringen nicht nur Vorteile hinsichtlich der Performanz mit sich, sie erlauben es auch in gleicher Weise die Zuverlässigkeit und Fehlertoleranz zu steigern, indem Redundanz-Mechanismen flexibel in Software umgesetzt werden. Diese eingebetteten Systeme haben häufig harte Echtzeitanforderungen, die wiederum Herausforderungen bezüglich des Multicore Real-time Schedulings darstellen – besonders dann, wenn globales, dynamisch Echtzeitscheduling Verwendung findet.

Die Verbindung dieser Echtzeitanforderungen mit den Anforderungen an Sicherheit und Zuverlässigkeit führen zu der Notwendigkeit neue Software-Lösungen zu entwickeln. Schedulingstrategien, die es ermögliche ein System gegen transienten Fehlern abzuhärten und den zusätzlichen kostenintensiven Hardwareaufwand zu reduzieren, müssen entwickelt werden, um ein konkurrenzfähiges, kosteneffizientes eingebettetes System zu entwickeln.
In dieser Arbeit wird der neue dynamische Hard Real-time Scheduling Algorithmus LB-*Pfair* entwickelt, welcher adaptiv auf die Einflüsse von transienten Fehlern reagiert. Mit Hilfe der diskreten Event-Simulation wird der Algorithmus durch simulierte Fehlerinjektion evaluiert.

Keywords

Multicore, Manycore, Multiprocessing, SMP, Multicore-Scheduling, Sicherheit, Zuverlässigkeit, Monte-Carlo Simulation, Zuverlässigkeitsanlyse, Sichere Softwareverarbeitung, Echtzeitsysteme, Diskrete Event Simulation

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 páce je původní a vypracoval jsem jí samostatně.

Použil jsem jen citované zdroje uvedené v přehledu literatury.

In Pilsen

V Plzni dne _____

Acknowledgements

This thesis was written during the past five years at the Laboratory for Safe and Secure Systems (LaS^3) at the OTH Regensburg and at the University of West Bohemia Pilsen. There are many people who supported me during that time with their guidance and by creating an inspiring working environment and therefore made this thesis possible.

At first 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 OTH Regensburg.

I thank my research colleagues at the LaS^3, especially Peter Raab, for many fruitful discussions and for having a great time.

Special thanks go to my family for their intellectual and emotional support and for placing great trust in me.

This research was funded by the research project Safe Oriented Programming of Software-Intensive Embedded Systems (S^{3}OP). S^{3}OP was financed by the Bavarian Ministry of Science, Research and Arts (Code: D2-F1116.RE/3/4). Further funding was provided by the research project “Multicore Processor Technologies for Safe Scheduling Simulation” (ZeloS^3). ZeloS^3 was funded by the German Federal Ministry of Economics and Technology (Code: KF2870902BZ3).

Contents

1. Introduction

The functionality and complexity of embedded systems in different domains have been increased to a greater extent in recent years. In addition to that the requirements for safety, reliability and availability are continuously rising. The contrary requirement is to reduce costs for these systems [2]. Hence a trend of shifting safety mechanisms like diverse hardware towards software approaches can be observed [3]. These demands influence the used hardware platforms. In the past years the clock rate was enhanced for obvious reasons. This leads to problems with EMC as well as to thermal issues, that is why multicore systems are getting accepted more and more. They only do not offer advantages regarding the performance but also a high potential of increasing the reliability of a software intensive embedded system by making use of multicore systems.

1.1 Motivation

These multicore systems provide the possibility of a diverse execution of software. Thus, additional to the improved performance, some of the required external hardware redundancy – which is necessary to achieve a certain reliability level – can be reduced. These mechanisms can then be implemented in multicore software.
In general, a trend towards multicore systems with smaller feature size can be observed, as this allows lower power consumption. These controllers are often used in cost-effective commodity hardware, but this has a drawback. The controllers are more prone to transient hardware faults [4] and therefore a higher probability of soft errors occur [5].

The higher probability of soft errors has to be intercepted by enhanced fault-tolerant mechanisms.
When speaking of real-time embedded systems, the attention has to be turned to the involved scheduling algorithms. Currently there are several approaches in the field of multicore real-time scheduling [6]–[9].
This dissertation will deal with robust reliable multicore real-time scheduling under the presence of transient faults.

Overview

In general, a task set scheduled on a multicore processor with a global scheduling algorithm can [10] react more flexibly to influences like transient faults than single-core scheduling algorithms. Some vendors (e.g. Freescale) provide dual or multicore controllers with integrated hardware lockstep mechanisms. These controllers use the cores redundantly on a hardware basis and compare their results after each instruction by an external compare unit. For the software the underlying hardware can be treated like a single-core processor. An embedded system has typically different tasks with different demands for reliability. That means that there are tasks which do not require execution in lockstep mode. For that reason a software approach can be more flexible. Various safety requirements are handled differently. Non-critical tasks are executed in a single instance. However, for some safety critical software components (tasks) safety mechanisms – like redundancy – can be applied. The performance for non-safety critical tasks as well as the reliability for safety critical tasks can be increased by this flexible software design approach.

The same principle can be applied to coded-processing approaches to provide higher safety levels on the one hand, and to make use of the higher performance potentials of multicore platforms on the other hand.

State-of-the-art normative regulations (IEC 61508 [11], ISO 26262 [12]) for functional safety suggest several possible solutions for error detection and handling in safety-critical embedded systems. The Laboratory for Safe and Secure Systems (LaS^3) has developed the Safely Embedded Software (SES) approach [13] based on AN-codes in high level programming languages. The concept of combining coded processing and the integration of these mechanisms into a real-time operating system was presented in [14].

These approaches have all in common that they have to be verified by testing techniques. For a single-core scheduling algorithm the proof of feasibility can be done analytically [15]. The task – of finding an analytic proof of feasibility – becomes harder or even impossible for global, dynamic multicore scheduling algorithms.

There are several methods like Markov Models or network modelling [16] to evaluate the reliability of a system or software-intensive system. These analytic methods have in common that they produce respectable results for simple systems. However, complex real-life embedded systems cannot be modelled adequately or have to be abstracted to a certain level of simplicity.

In a hard real-time system a system failure is not only a wrongly computed result – caused e.g. by a transient fault – but also the violation of certain timing con-

straints. These deadline violations can be triggered e.g. by the transient faults themselves, by affecting the scheduler or by the safety mechanisms initiated by the safety supervisor. Safety mechanisms include the re-execution of the task, backward or forward recovery [2], [17]. These strategies consume an extra amount of execution time, which – in addition – influences the feasibility of the schedule. Worst case assumptions were widely used to deal with these scenarios in the past [15].

The approach presented in this thesis will use a stochastic simulation environment to evaluate the reliability of the newly developed scheduling algorithms in a software-intensive embedded system. The simulation is used, not only with regard to the simulation of standard reliability indices, but also with regard to the simulation of real-time characteristics of an embedded system. On the one hand stochastic simulation in combination with discrete event simulation is used to model the variance of task execution durations and the general feasibility of a schedule in a multicore real-time embedded system [18]. On the other hand it is used to model the influence of transient errors affecting the system in a holistic approach.

Transient hardware faults only propagate to system failures if and only if the resulting errors are not detected by the error-detecting mechanism and the errors cannot be repaired within the real-time timing constraints, e.g. the deadline.

The goal of this dissertation is to develop robust, fault-tolerant multicore real-time scheduling algorithms to provide a safe multicore operating system. As these multicore scheduling algorithms can – as mentioned – hardly be analysed for feasibility by an analytic approach, Monte Carlo Simulation for software fault injection is applied to evaluate the results of the developed approaches.

1.2 Basic Concepts of Fault Handling Strategies in Real-time Multicore Operating Systems

During every time of the live cycle of an embedded system, errors of any kind can occur. This can happen during the design-phase of a product by incorrect or insufficient specification of software or hardware, by not considering the proper environment conditions or by mistakes during the implementation of the product. All these mistakes mentioned are systematic and present in the system from the beginning of its lifetime. These deterministic errors are not considered in this thesis. Although some approaches like N-version programming [19], where multiple versions of a functionality are independently implemented, based on a common specification, can also have a benefit in case of non-systematic faults because of the voting property. This work focuses mainly on non-systematic faults which can

occur randomly. These faults have to be detected and handled during run-time. The faults mentioned first can be avoided by the application of appropriate development and testing processes.

There are several concepts to increase the fault tolerance (see Chapter 2.3.2) of an embedded system and thus those increase the dependability of a system. In this introductorily chapter, the main notions regarding dependability and real-time scheduling will be introduced.

1.2.1 Dependable Embedded Systems

Laprie et al. [20] define the term dependability as follows:

> **Definition 1**: Computer system **dependability** is the quality of the delivered service such that reliance can justifiably be placed on this service.

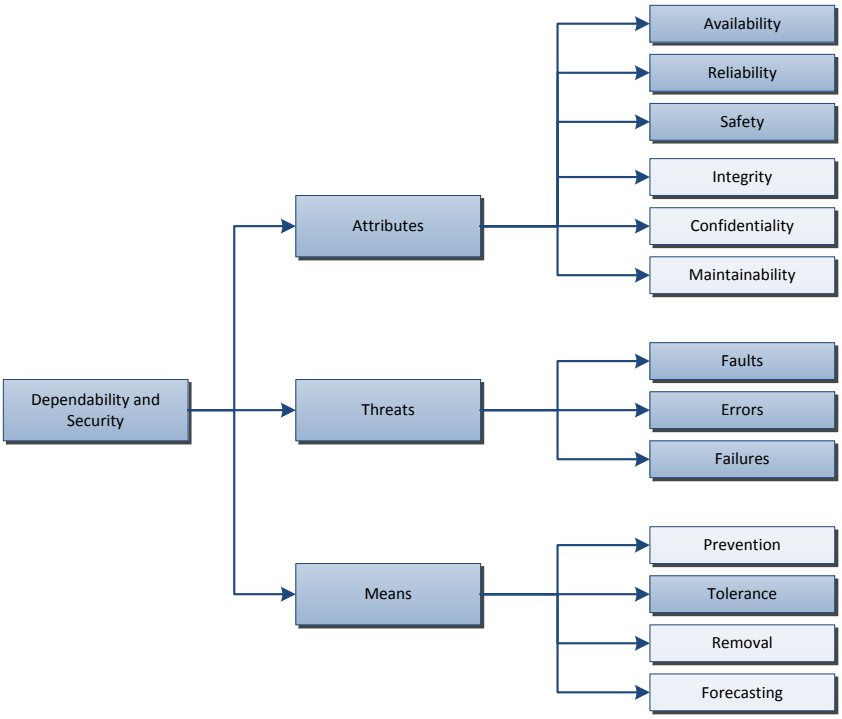

Figure 1: Dependability in accordance with Laprie et al. [20]

Generally speaking, the terms dependability and security[1] are used to classify an embedded real-time system. Dependability and security are defined by the corresponding attributes, threats and means. This relation can be depicted in Figure **1**.

[1] It is referenced to security in this case, because it might be a future aspect to introduce coding mechanisms that not only ensure the error correction but at the same time offer some kind of encryption ("holomorphic transformation").

1.2.1.1 Attributes

Attributes of a system are qualitative or quantitative measures to observe the dependability of a system. Avizienis et al. [21], [22] name the following main attributes:

- Availability
- Reliability
- Safety
- Maintainability
- Integrity
- Confidentiality

1.2.1.2 Threats

These dependability attributes define measures how a system reacts to the threats that influence this system and therefore reduce its overall dependability. According to [20], there are the following main threats:

- Failure:
  A failure exists, if the delivered service deviates from the specified service. The effects can be observed from the outside of the considered component. A failure is the result of an error.
- Error:
  An error is an internal system state which deviates from the normal operational state and might lead to a failure.
- Fault:
  A fault is the main cause of an error. It is applied externally to the considered component. It is the source of the whole error chain.

1.2.1.3 Fault-Error-Failure Chain

The fault propagation can be depicted in the so called fault-error-failure chain (see Figure 2) at component level [11].

The fault, the source of the chain, is the external effect applied to the system. This fault can be an erroneous bit – caused e.g. by an alpha particle – in the memory of a microcontroller. This flipped bit then causes an error in the data which is stored in the memory of the microcontroller. Thus there is a deviation from the normal operating state of the software, but this does not inevitably lead to a failure of the whole microcontroller, if

- the error is in a memory region that is not used (at that time) by the software (silent errors) or
- the error is detected and repaired.

If the internal error is not handled, it probably leads to a failure of the microcontroller.

As a consequence of the failure of the microcontroller the upper level component – the controller unit – has to deal with a fault.

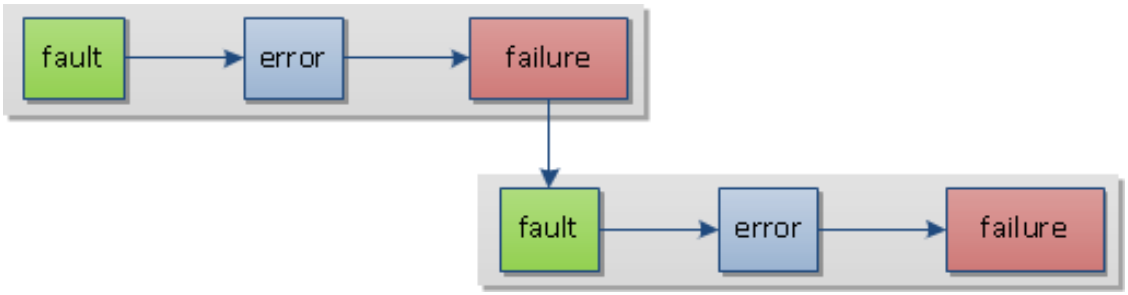

Figure 2: Fault-Error-Failure Chain – fault propagation on different system components.

1.2.1.4 Means

As described above an error does not inevitably cause a system failure. There are means which can break the error chain. The main means found in corresponding literature [20] are:

- Prevention
- Tolerance
- Removal
- Forecasting

Faults can be prevented – when speaking of systematic faults – by using strict development processes and testing processes. In case of non-systematic faults like the named arbitrary hardware faults, they can be prevented by hardening the system to a certain level against these faults. Transient hardware faults can be caused e.g. by:

- radiation
- electromagnetic interferences
- spikes in power supply
- etc.

Faults can be removed if there is the possibility of maintenance which is not an option as we are speaking of real-time systems.

And because arbitrary hardware faults randomly occur, they cannot be predicted.

Since this work will deal with arbitrary hardware faults, this section will focus on tolerance against faults.

1.2.2 Arbitrary Faults

A classification can be made to distinguish three categories of arbitrary faults by the duration of presence:

- Transient faults:
  Transient faults occur during life-time and have a short temporal extent. They can arise from multiple sources: external sources such as high-energy particles that cause voltage pulses in digital circuits, as well as internal sources that include coupling, leakage, power supply noise, and temporal circuit variations [23]. No permanent defect of the system remains after the environment conditions have been normalized.
- Intermittent faults:
  Intermittent faults have the same origins (e.g. aging) as permanent faults, but their duration of them is shorter and they occur in random intervals.
- Permanent faults:
  Permanent faults exist for an infinite time span and therefore, the component is destroyed. It has to be replaced or repaired during maintenance.

1.2.3 Fault Tolerance

This work will deal with arbitrary transient hardware faults lowering the dependability of a software intensive real-time embedded system and therefore the preferred solution for dealing with transient faults is the design of a fault-tolerant system.

> **Definition 2:** The ability of a system or component to continue normal operation despite of the presence of hardware or software faults [24] is defined as fault tolerance.

To achieve fault tolerance, the embedded system has to include mechanisms which detect the errors and to correct them in a way that they do not propagate to a dangerous system failure.
The requirements of a hard real-time system claim that the timing constraints are even kept in the presence of faults. That means a detected and corrected fault can propagate to a system failure if the deadlines are not met (see 1.2.5). Accordingly,

the used scheduling algorithm has to react robustly in case of timing disturbances caused by the fault.

Fault tolerance is mainly achieved by redundancy; it can be subdivided as follows:

- Information redundancy:
  This includes error detection and correction mechanisms, like checksums, cyclic redundancy checks (CRC), coded processing, SES, etc. (see Chapter 0).
- Time redundancy:
  Redundancy is added by use of extra time, this means a computation is performed several times. Transient faults can be detected, if their duration is shorter than the repetition interval [25].
- Hardware redundancy:
  There are several levels of hardware redundancy, like DMR (Dual Modular Redundancy) [26] or TMR (Triple Modular Redundancy) [25] (see Chapter 0). A job is performed on N different hardware components. A voting mechanism compares the results of the different components. A voter can perform a majority voting if enough (N > 2) parallel components produce the result and thus it can determine the components not affected.
    - Static redundancy:
      All parallel components are active and produce the results continuously.
    - Standby system:
      Only one component is active. If an error is detected, the system activates and switches to the redundant component. This has the advantage that an inactive component has a lower failure rate [16], but a failure sensing and changeover to the standby component has to be implemented.

1.2.4 System Behaviour

Depending on the level of redundancy, the system can tolerate a certain level of faulty components. It can be classified as follows [27]:

- Fail operational:
  The system stays operational in case of an error.
- Fail safe:
  After one or more errors have occurred, they can be detected. The system is no longer operational, but a transition to safe state is guaranteed.

- Fail silent:
  After one or more errors have occurred, the component stays silent and does not influence any other component.

1.2.5 Real-time Operating Systems

Real-time operating systems guarantee – additionally to the known services that are offered by a normal operating system – the execution of tasks within default and predefined constraints. They have a deterministic timing behaviour. Real-time operating systems are mainly classified in two categories [15]:

- hard real-time and
- soft real-time systems.

1.2.5.1 Soft real-time

In general, a system with soft real-time requirements can ensure that the timing constraints can be adhered. There is no guarantee in a soft real-time system that the timing constraints (e.g. deadlines) are kept. The effect of a violation of a timing constraint has a minor impact on the overall functionality of the system. The main functionality maintains. There is no harm in case of a deadline violation, but the helpfulness of the computed results declines after the missed deadline [15].

1.2.5.2 Hard real-time

In contrast to the soft real-time requirements, the violation of timing-constraint in a hard real-time system leads to a complete system failure. All computed results that are finished after the deadline are completely worthless. The deadline violation and the resulting system failure cause great harm by damaging the surrounding or even by compromising human lives [15].

1.2.5.3 Scheduling in Multicore Embedded Systems

In order to meet the real-time requirements of a system, real-time operating systems are used in embedded systems. These operating systems include a scheduler, which is responsible for the timing behaviour of the embedded system.

For standard single-core systems there are several well-known and investigated scheduling algorithms, like [15], [28]:

- priority based scheduling,
- rate monotonic scheduling (RMS),
- earliest deadline first scheduling (EDF).

Moreover there is a great variety of algorithms, which are well-studied. An analytical proof for feasibility and optimality exists for the most of them. In contrast to single-core scheduling in the domain of multicore scheduling, there is – besides the heavy research effort in the past years – a lot of development going on. There are multiple approaches to handle the multicore challenge [29], [30]. These will be discussed in detail in chapter (2.1 and 2.3.2). Therefore, only a brief summary is given:

Multicore scheduling approaches can generally be categorized in:

- partitioned and
- global

approaches. The partitioned approaches have the advantage that the original multicore problem is reduced to well-studied single-core problems by allocating the given tasks to a certain core. After allocating, standard scheduling algorithms (e.g. EDF) can be applied to each partitioned task set. The allocation of tasks to cores, which is a NP-hard problem, is accomplished during the design phase of the system and consequently does not influence the real-time behaviour.

These single-core scheduling algorithms are not optimal for multicores and do not provide proper results with regard to real-time constraints and utilization of cores [31].

Global scheduling means that one central scheduling algorithm is responsible for all scheduling decisions and controls all execution resources (cores). This offers the opportunity that tasks or task instances can migrate between different cores during execution time. Thus an optimal schedule can be achieved. Another advantage of global scheduling is that for every point in time it can be ensured that the task or tasks with the highest – dynamically calculated – priority are executed. The algorithm can react flexibly to the absence of an execution resource and therefore it offers some basic fault tolerance. To ensure optimal scheduling, the priorities of the executing tasks have to be calculated and adapted during runtime. This results in a higher overhead for executing the scheduling routines of the operating system.

1.3 Problem Analysis

Most publications either focus on multicore real-time scheduling or on fault tolerance mechanisms like DMR or TMR, while applying standard static partitioned scheduling approaches. Generating efficient and optimal scheduling algorithms – that guarantee a feasible schedule that ensures the absence of deadline violations – is the main concern of these publications.

On the other hand there is an increasing variety of approaches dealing with fault tolerance even in multicore systems, but mostly they use standard scheduling algorithms as RMS or EDF and partitioned scheduling approaches. They consider the additional time needed for error detection and correction, but in some cases the actual detection is neglected.

The global dynamic multicore algorithms which can react more flexible to timing disturbances [10] can handle certain disruptions in the timing. But they do not consider the – a priori known – time for error correcting mechanisms for computing the dynamic schedule.

Combining these global dynamic multicore algorithms with integrated fault handling techniques – considering the error reaction and planning the error reaction time directly in the dynamic schedule – lead to the creation of a reliable and still high performing scheduling algorithm. Such a scheduling algorithm (LB-*Pfair*) was developed in the course of this thesis. The LB-*Pfair* algorithm is explained in Chapter 4.3.1.

2. State of the Art – Related Work

In the previous chapter some basic ideas of dependable real-time embedded systems were introduced. This chapter will focus on the state of the art in the field of reliable multicore real-time scheduling. An overview of relevant safety design concepts – diversity, replication and combined approaches – is given in Chapter 2.3.2. The techniques named in Chapter 2.3.2 are general techniques that not inevitably relate to real-time systems. These techniques include methods to deal mainly with arbitrary hardware faults. The attention is turned on software based approaches to handle transient hardware faults.
This chapter – as mentioned – focuses on reliable multicore scheduling approaches. Single core scheduling mechanisms are described shortly, as they are still used widely in partitioned multicore scheduling approaches. They have a very low scheduling overhead and are well studied and proven in practice. There are several proposals in literature to use this kind of algorithms for reliable systems [32], [28], [33], [34]. The remaining part of this chapter will deal with multicore scheduling approaches. The toleration of transient faults leads to a temporary higher processor load, caused by the error detection and reaction. Thus these algorithms have to be robust regarding to temporary overload conditions. Therefore, dynamic global approaches play a very important role [7], [9]. Different algorithms are presented. The last category of scheduling algorithms are the so called hierarchical scheduling algorithms [35], [36]. These algorithms allocate (according to the requirement of reliability) the tasks to different local schedulers which are themselves controlled by a master scheduler. Thus a certain level of isolation is provided. An introduction to the basic timing parameters is given in Chapter 2.3.1.

2.1 Multicore Scheduling

There are different concepts for multicore scheduling. Davis et al. [30] give an overview of existing multicore scheduling algorithms. The concepts differ in the way tasks are allocated to cores, the execution resource of the scheduling algorithm itself and the possibility of task migration between different cores.

- Asymmetric multiprocessing describes a system, where different instances of real-time operating systems are located on different cores. Actually the

operating systems do not have to be of the same type. This approach reduces the actual scheduling problem to a single-core scheduling problem. The allocation of tasks to cores is done during design time. This kind of algorithms is described in greater detail in chapter 2.1.2.

- Master-Slave multiprocessing is a concept, where the execution of the scheduling algorithm is static located on one certain core, the master. All other cores do not have any scheduling logic, they just execute the tasks scheduled and allocated to them by the master. This concept is easy to implement, because there is no need for a synchronization between the cores. But the major drawback is that there is single point of failure – the core executing the scheduler – and therefore this concept is not considered in this work for scheduling safety critical task sets.
- A third category is the symmetric scheduling approach. In this concept only one instance of the scheduler is existing. The scheduler can – as well as the tasks – migrate between different cores during runtime. This means this approach can dynamically react to disturbances in the timing. As the tasks are not statically allocated to cores and the tasks can migrate between cores during runtime. This also offers the opportunity to stay fail operational in case a core has a permanent error. A comparison of this type of algorithms is given by Xin [29].

2.1.1 Static Scheduling

Eles et al. [37] propose a concept, based on a predefined "Fault tolerant Conditional Process Graph" where all assumed error conditions and the according scheduling decisions are precalculated and stored. This concept has the advantage that it is highly predictable because even in the case of an error the scheduling decisions are defined a priori. But regarding performance this system has major drawbacks as it has to allocate runtime for all possible scenarios.

2.1.2 Partitioned Scheduling

Partitioned scheduling is a step towards multicore scheduling. Nearly all single-core scheduling algorithms – priority based, RMS, EDF, round robin, etc. – can be applied for multicore usage in systems with asymmetric multiprocessing. The algorithms described in this section can not only be used for the partitioned approach, but also for the hierarchical scheduling approach which are described later (see Chapter 2.1.4). For hierarchical scheduling these algorithms can be em-

ployed for the local schedulers as well as for the master scheduler [38]. The round robin algorithm might be more suitable for the hierarchical scheduling approach as the master scheduling algorithm.

2.1.2.1 Priority based Scheduling

This is a very basic approach to the scheduling problem. A certain priority *Pi* is assigned to all tasks of a task set. The scheduler holds an ordered list of the task set:

$$\mathrm{T} = \{\tau_1, \tau_2, \ldots, \tau_n\} \text{ where } \mathrm{P}_1 \geq \mathrm{P}_2 \geq \mathrm{P}_n \tag{2-1}$$

In case of equal priorities some tie-breaking rules have to be applied or the scheduling has to be done according the round robin algorithm [15].

If this scheduling algorithm is used for symmetric multiprocessing M tasks (M = number of cores) are executed simultaneously.

2.1.2.2 Rate Monotonic Scheduling (RMS)

This scheduling algorithm is very similar to the priority based scheduling because the priorities assigned to tasks are chosen according to the period of the task. The task with the shortest period obtains the highest priority. This scheduling algorithm – according to Stallings [39] – can handle a maximal utilization S:

$$S \leq N(2^{1/N} - 1) \quad \text{(with N: the number of tasks)} \tag{2-2}$$

For an infinity number of tasks an overall utilization – which still ensures a feasible schedule – is given by:

$$S\big|_{N\to\infty} \leq \ln(2) = 0{,}69 \tag{2-3}$$

If this scheduling algorithm is used for symmetric multiprocessing, this limit can no longer be applied. There is no proof of feasibility in this case. This algorithm is used by Ghosh et al. [40] for fault-tolerant scheduling.

2.1.2.3 Earliest Deadline First Scheduling (EDF)

The basic idea for this algorithm is to determine the execution order according to the deadline. The pure EDF-algorithm is not suitable for multiprocessors: “For example, on multiprocessors EDF is not optimal. In fact, EDF might miss dead-

lines on multiprocessors even if processors are idle approximately half the time." [7]

Beitollahi et al. [28] use this algorithm for fault-tolerant scheduling in an uniprocessor system. As mentioned this algorithm is not optimal on multiprocessors, but there are extensions to this algorithm, global EDF until zero Laxity (see 2.1.3) which overcome this lack.

2.1.2.4 Round Robin

This algorithm uses no special priority assignment. All tasks – which have to be preemptable – of the task set are executed for a certain window. At the end of the window the task is preempted, and the next one is continued. Usually this algorithm is not used as a standalone algorithm, but in addition to other algorithms as a tie-breaking rule.

For the hierarchical scheduling approach this may be a suitable algorithm for the master scheduler because it allocates defined runtime budgets to the different slave schedulers. This can be an advantage compared to the priority based master scheduler in "Multicore Operating-System Support for Mixed Criticality" by Anderson [36].

2.1.3 Global Multicore Scheduling Concepts

A selection of promising algorithms – with regard to include fault tolerance mechanisms in the scheduler – should be described in this chapter. Carpenter et al. [8] give an overview of real-time multiprocessor scheduling algorithms. Moreover Davis et al. [30] have done a survey of real-time scheduling algorithms. The grouping is in accordance with [41].

2.1.3.1 Global fixed priority Scheduling

Global EDF until zero Laxity

Persya et al. [33] use global EDF for fault-tolerant scheduling. At global scheduling this algorithm is not optimal [7]. As Figure 3 shows, no feasible schedule is produced although the utilization is below 100%.

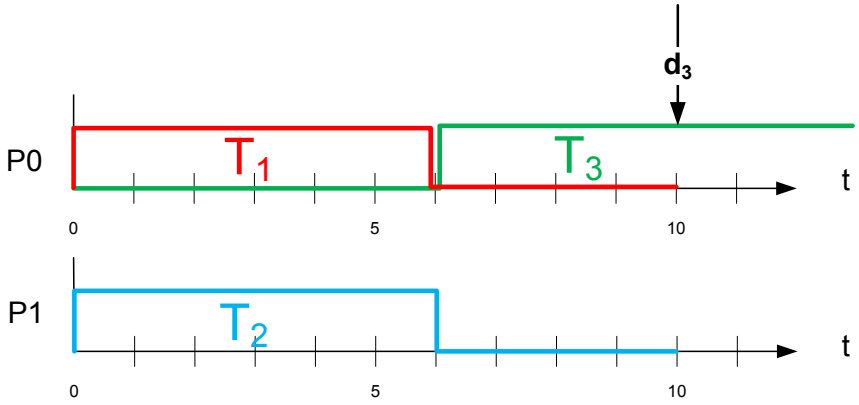

Figure 3: **Standard Global EDF: Deadline for all tasks is 10 ms. $e_1 = 6$ ms, $e_2 = 6$ ms, $e_3 = 7$ ms; total utilisation: 95%**

An extension to the original algorithm is zero laxity [42], [43]. The tasks are now preemptable. An additional scheduling event is introduced, the zero laxity event. It is triggered at time t_z if the laxity of a task has become zero. The resulting schedule is depicted in Figure 4.

$$t_z = d_i - e_i \tag{2-4}$$

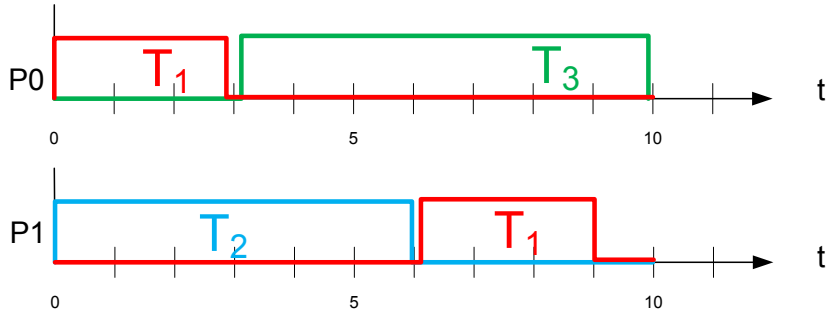

Figure 4: **Global EDF with "zero lexity" extension; Deadline for all tasks is 10 ms. $e_1 = 6$ ms, $e_2 = 6$ ms, $e_3 = 7$ ms; total utilisation: 95%**

This event based scheduling approach has the advantage of low scheduling overhead and because of the consideration of the laxity, the observation of a change of the execution time – caused by error reactions – can be considered in the calculation of the schedule.

2.1.3.2 Global dynamic Scheduling

Proportionate fairness Agorithms

The Proportionate fairness (Pfair) algorithms were first published by Baruah [44].This kind of algorithm is suitable for periodic task sets [41]. The basic idea is the fluid schedule, which is shown in Figure 5.

Fluid schedule

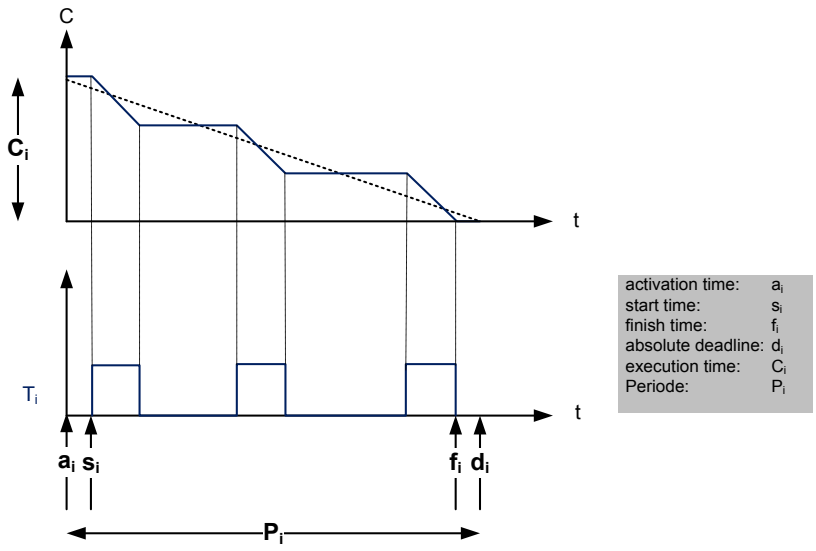

Figure 5: Fluid schedule, task execution proportionate to its utilization

The real execution of a task instance follows the imaginary fluid schedule. The slope of the fluid schedule is defined by the utilization (see equation 3-4), that is generated by the task. As the execution speed of a processor cannot be varied, the task is either executed or set to idle. The lower part of Figure 5 shows in a Gantt-Chart the actual execution of the task. "Each task makes progress proportionate to its utilisation (or weight in *Pfair* terminology)." [41] The advantage of this kind of schedule is the low jitter of the task set because all tasks finish just before their deadline.

Pfair Scheduling

Holman et al. [6], [45] implemented the algorithm and made adaptions for symmetric multiprocessing. It is shown [44] that *Pfair* is optimal for periodic task sets with implicit deadlines. The utilization bound is the number of cores M:

$$S_{Pfair} = M \tag{2-5}$$

Due to the quantum based approach – to follow the fluid schedule as close as possible – there is a very high scheduling overhead. Moreover the individual-cores have to be synchronized to the time quanta to make scheduling decisions. The queue where the task set is stored has to be accessed by all cores. And therefore this queue needs to be guarded by some kind of semaphore mechanism which costs additional time [6].

Pfair supports unlimited migrations of tasks and is non-work-conserving. That means the core can be idle, although tasks are ready for execution. There are several extensions like the ERFair which softens the tight binding to the fluid schedule and executes task earlier if there is free execution time. With this modification the algorithm is work-conserving. Other extensions are PD or PD². These extensions group tasks in heavy and light tasks and therefore improve the performance [41].

TL-plane based Scheduling Algorithms

The time and local execution time plane (TL-plane) was introduced by Cho et al. [9]. The concept of the TL-plane is based on the fluid scheduling model. The TL-plane represents a period within the schedule; it is the superposition of the fluid schedules of all tasks. The overall execution time of a task instance (e_i) is splitted proportionate to the task period and the length of the TL-plane. This splitted execution time is called local execution time.

LLREF

The basic scheduling principle is: largest remaining execution time first (LLREF) [9]. At the start of each plane M tasks with the largest remaining local execution time are selected for execution. The local execution time diminishes during the execution of the task. Scheduling events are triggered,

- Event B:
  if a task has consumed its local execution time and hits the bottom of the TL-plane.

- Event C:
  if the laxity of a task within the plane has become zero. This event is called ceiling hitting event.

This event driven scheduling reduces the number of scheduling points – compared to the quantum based Pfair algorithm – substantially. There are at most n + 1 scheduling events [9].

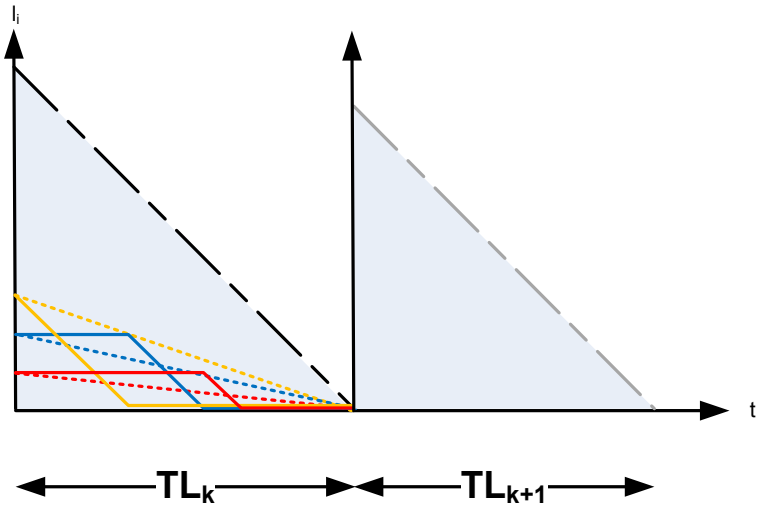

Figure 6: Task execution visualized in the TL-plane.

LLREF supports unlimited migrations of tasks and is non-work-conserving. It produces an optimal schedule for periodic task sets with a utilization bound of:

$$S_{LLREF} = M \quad \text{(where } M\text{: number of cores)} \tag{2-6}$$

Funaoka et al. [46] modifies the algorithm to be work-conserving by executing other task if one task finishes earlier than expected. Thus the number of preemption can be reduced [30].

This algorithm was implemented and modified by the author [47], [48] in a multicore port of a OSEK-based operating system, the R³TOS.

With this modification – the algorithm is now work-conserving – LLREF is suitable for the usage in a system where fault-tolerance is required; as no execution time gets lost at begin of a TL-plane.

LRE-TL

Funk et al. [7] introduce a modification of the LLREF algorithm that is designed for periodic and sporadic tasks. They managed to reduce the scheduling overhead from *O(n)* to *O(log n)*. It is called LRE-TL (local remaining execution-TL). They no longer schedule according to the largest remaining execution time first principle because as shown it is sufficient to select any of the local tasks, if M task are

executed at the beginning of the plane. If the M tasks execute continuously, it can be shown that the total local utilization never increases.

This algorithm is suitable for fault-tolerant scheduling because it covers general sporadic task sets. Former periodic task sets can become sporadic ones, if tasks are re-executed in case of faults.

TL-DVFS

Zhang et al. introduce in "TL-plane-based multicore energy-efficient real-time scheduling algorithm for sporadic tasks" [49] a dynamic voltage scaling extension to LRE-TL.

This scaling of the execution speed can be used to introduce fault-tolerance mechanism like the primary backup strategy [50] in the LLREF-scheduling. If no fault reaction has to be performed (normal operation), the system clock can be reduced. Only in case of fault reaction, the system clock can be increased for a faster error correction.

2.1.4 Mixed Criticality and Hierarchical Scheduling

In this subchapter hierarchical and mixed Criticality scheduling is discussed. Applications with different kind of criticality can be located on one target device. The different application parts have to be isolated from each other.

2.1.4.1 Temporal Isolation

In "Multicore Operating-System Support for Mixed Criticality", Anderson et al. [36] set the focus on temporal isolation across criticality levels. Lower levels should not interfere with higher levels. "Task execution costs may be determined using more-stringent methods at high criticality levels and less- stringent methods at low criticality levels." [36] They assume five levels of criticality – based on the effects on a commercial aircraft – according to the avionic standard DO-178B:

- catastrophic
- hazardous
- major
- minor
- no effect.

To ensure temporal isolation, the key idea is to encapsulate every of the five criticality classes in a container. For each container an "appropriate intra-container"

scheduler is chosen. The temporal isolation is achieved by assigning a higher priority to a higher criticality container. They suggest among others global EDF for soft-real-time tasks and partitioned EDF for hard-real-time tasks.

They use LITMUS – a real-time extension for Linux – for the evaluation of their proposal. Their focus is mainly on the timing aspects. They do not consider any fault detection or reaction. Moreover, their usage of a priority based scheduling algorithm for the master scheduler has a drawback for systems with long running high critical task and short running tasks with a shorter period. As a result of the priority assignment, the low critical task would not be considered for scheduling and thus eventually miss their deadline.

2.1.4.2 Hierarchical Scheduling with Hardware Support

In "A Flexible Scheme for Scheduling Fault-Tolerant Real-Time Tasks on Multiprocessors" [35] Cirini et al. propose a technique based on a combination of hardware mechanisms and real-time operating system mechanism. They use hierarchical scheduling techniques to classify three different kinds of tasks:

- fault-tolerant,
- fault-silent and
- non-fault-tolerant

tasks.

A multicore processor executes fault-tolerant and fault silent tasks in a hardware lockstep mode. The non-fault-tolerant tasks are executed in parallel mode for performance reasons. Therefore, they suggest a dynamically online reconfiguration of the processor.

There is currently no hardware platform which supports this dynamic reconfiguration to the best of our knowledge. A statement from the vendor Freescale says that this is hardly achievable in reality because of the dense connections of the cores in lockstep mode. It would be a great challenge to accomplish the synchronization of the cores, especially of the pipelines after switching modes.

2.2 Safety related, Fault-tolerant Scheduling

This chapter deals with scheduling solutions that offer fault-tolerant properties. They are based on the earlier named static scheduling algorithms.

2.2.1 Redundant Execution

Subramanyan et al. [51] use a variation of the dual modular redundancy (DMR) approach. They employ single chip multiprocessors (CMP) which executes a single application as two threads – a leading and a tailing thread. They call their solution multiplexed redundant execution (MRE). The tailing thread can be executed faster than the leading thread by "providing execution assistance" [51]. They promise an increase in the throughput of 16% compared to DMR.

In this approach, the faults are basically detected by periodically exchanging fingerprints of the involved cores. A second way to detect errors is, to observe branch mispredictions in the second thread.

The faster execution of the second core is a result of modified hardware. Two cores are strongly connected. One concept is "Branch outcome forwarding", which increases the execution of the trailing task, by using this value instead of the own branch prediction.

This is a solution for enhancing the DMR approach, but it requires hardware modifications. There is no statement made, how the comparing of the checkpoints is done.

2.2.2 Energy aware redundant execution

Wei et al. [32] suggest in "Fixed-Priority Allocation and Scheduling for Energy-Efficient Fault Tolerance in Hard Real-Time Multiprocessor Systems" the optimal balancing of the workload among the cores. The main assumption is that the fault free execution is the dominating one and therefore, energy can be saved in this case. "An optimistic fault-tolerant heuristic is then proposed to achieve optimum energy savings in the absence of faults and meet application timing requirements in the worst case faults at the cost of energy inefficiency." [32]

Their main scheduling strategy is rate monotonic scheduling (RMS). The overhead for checkpointing, rollback, etc. is neglected. Based on the different techniques – triple modular redundancy, primary backup and checkpointing – they propose schedules which are optimized either for energy reduction or for increasing reliability. There proposals are evaluated by a simulation approach.

The energy reduction is a result of applying energy saving mechanisms as dynamic voltage scaling or switching off of cores, in case no faults occur. Additionally, they make a statement about the optimal granularity of checkpoints. No special hardware is needed for their approach.

2.2.3 Deadline relating Fault-tolerant Scheduling Approaches

Beitollahi et al. [28] modify in "Fault-Tolerant Earliest-Deadline-First Scheduling Algorithm" the earliest deadline first algorithm in that way that "enough and efficient time redundancy" [28] is added to the EDF scheduling policy. The insertion of slack and recovery mechanism is called the FT scheme. This scheme is mainly designed for uniprocessor systems. If a fault occurs, the system switches to recovery mode where the faulty task is re-executed. They provide a feasible solution in recovery mode; additional slack was added to the system, according to their policy.

The basic idea is to add slack to the schedule and predict the amount of time needed for error reaction. A gain factor is defined to determine this additional slack. They do not make a statement on how to detect errors.

Gresser et al. [33] also use the EDF scheduling scheme as a base for fault-tolerant scheduling. In their case – the global variant for multiprocessors is used. They apply primary backup as fault tolerance strategy, where the backup has not to be executed if the primary instance successfully finishes. The novelty in their solutions is that they overload time slots of backup instances with backups of other tasks as long as they can be allocated to cores, different from the ones executing the primary instance.

Due to the overloading and the assumption that only one fault occurs during a period, the possible maximal utilization can be increased.

2.2.4 Hardware based approach

In "A Safety-Related PES for Task-Based Real-Time Execution" Skambraks [52] describes a solution for implementing the real-time scheduler in a digital logic circuit (FPGA). It uses the EDF scheduling theme. Erroneous hardware parts can be detected; the device can be reconfigured to use backup parts.

This solution is not in the main focus of this thesis, but it shows that hardware based solutions may offer an opportunity, too.

2.3 Reliable Scheduling in Embedded Multicore real-time Systems

This chapter will deal with the reliability aspects of real-time multicore scheduling. After a short introduction of the most important timing parameters, the focus is set to new approaches: the adaption of the fluid scheduling concept and the

usage of a hierarchic scheduling approach to ensure the separation of task sets with different safety levels. In the following, periodic task systems are considered. Sporadic task systems can also be considered in a later step.

2.3.1 Definitions

A task system of an embedded real-time application is a set of tasks:

$$T = \{\tau_1, \tau_2, \ldots, \tau_n\} \tag{2-7}$$

Each task τ_i is usually defined by a triple $(p_i,\ e_i,\ d_i)$.

The period p_i of a task is defined by the interval of two subsequent activations a_i of a task. The actually executed task is called job. It is denoted as $\tau_{i,j}$, where j is the index of the activated job. The job is often referred as task instance.

The execution time e_i is the assumed (error–free) worst-case execution time (WCET) of the running task instance.

The parameter d_i represents the relative deadline of the task instance. This is the latest time, when the task instance has to finish its execution.

The deadline is equal to the period of the task in a periodic task system. If no multiple task activations are allowed, the following equation must hold.

$$e_i < p_i \leq d_i \tag{2-8}$$

Additionally to the standard task parameters, we have to define parameters considering the safety characteristics of a task.

An important aspect for the introduced predicting scheduling algorithms is the knowledge about the assumed error reaction and error correction times.

The error reaction time r_i is the time, which is needed by the applied detection mechanism to detect the error occurred at time t_e. The error is detected at time t_d. The time assumed for executing the failure reaction is called the repair time c_i.

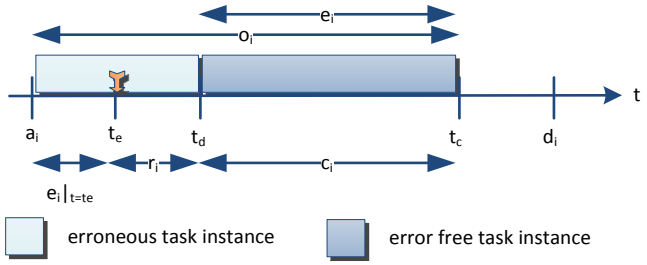

Figure 7 **The relevant timing parameters of a task. An error is detected at time t_d and repaired within the task deadline d_i at time t_c. Re-execution is considered, and therefore $e_i = c_i$.**

To meet the task deadline in case of an error, equation (3-2) has to be modified:

$$o_i < p_i \leq d_i \tag{2-9}$$
$$\text{where: } o_i = e_i|_{t=t_e} + r_i + c_i$$

The overall execution time o_i considers the fraction of the normal execution time e_i until the fault occurred and the time needed for detection r_i and correction c_i. In case of multiple errors this has to be considered separately.

Therefore, a task is now defined by the parameters: e_i, d_i, p_i, r_i and c_i.

To determine the feasibility of a task set, the utilization u_i produced by the task set is an important factor.

$$u_i = \frac{o_i}{p_i} = \frac{o_i}{d_i} \leq 1 \tag{2-10}$$

The total utilization (S) – produced by all tasks of the task set – is defined by

$$S = \sum_{i=1}^{N} u_i \leq M \tag{2-11}$$

where the maximal utilization can be the number of cores M.

In the naming of Pfair-scheduling the utilization is also called task weight.

The laxity x_i describes the time spa, for which a task can be at maximum delayed and still meets its deadline. It is defined by:

$$x_i = d_i - a_i - o_i \tag{2-12}$$

2.3.2 Error Handling in Embedded Systems

Possible concepts for achieving a higher dependability in an embedded system are illustrated in this chapter. There is a great variety of concepts. Here, only a short overview of the basic ideas of these concepts is given. Full implementation details will be omitted as well as the different variants of these concepts because the main concern in this thesis is the impact of these concepts on the runtime behav-

iour of the real-time task set. Error detection mechanisms – with regard to real-time operating systems – are described in the first part of the chapter. After that, possible error reactions are shown. This chapter concludes with first simulation results, comparing information redundancy and homogenous redundancy, further research topics are listed at the end.

2.3.2.1 Error Detection Concepts

For launching error reactions in an embedded system, concepts for error detection are vital. The presented concepts are described by design patterns that can be applied to error detection and error handling in safe real-time operating systems. All patterns have in common that the adherence to timing constraints and error detection is handled by a central unit, the safety supervisor [19] (see Chapter 4.1). The safety supervisor is implemented within the safe multicore scheduler (SMS) (see Chapter 2.3.2 and 4.3). The concept of integrating this safety supervisor into the architecture of a real-time operating system is described in Chapter 4.1.

Besides N-version programming [53] – which mainly aims to reduce systematic errors – literature ([54], [19]) describes several safety-related design patterns. But the focus is set to transient hardware faults, thus the design patterns – information redundancy, homogenous redundancy and the combination of both – are described in the following, because they are adequate to handle these kinds of faults.
Transient hardware faults can cause different kinds of errors in an embedded real-time system. It depends on the region – memory, bus, CPU, etc. – of the controller, that is affected by the fault. Generally, two types of errors result from hardware faults:

- data errors and
- program flow errors.

The listed design pattern can handle both types of errors. As mentioned in the introduction, all described means for increasing the dependability of a system are based on different kinds of redundancy.

Information Redundancy Pattern – Coded Processing

In this pattern, the data and the computations within the task are transferred to a coded domain, where redundant information is added by coding. Different coding mechanisms as Parity Codes, Checksums, Cyclic Codes, AN-Codes etc. can be applied and several methods of using coded executions can be found in publica-

tions [13], [55], [56], [53], [57]–[59]. The coding can be applied during different phases: during compile time or during runtime. There are approaches of safety compilers [60], safety interpreters or the direct integration of coded processing within the actual code by using special software libraries [14]. Error detection can be performed during the processing of the data, during the program execution. That is an advantage of coded processing in comparison to other concepts like homogenous redundancy (see Chapter 0). Thus, the error detection time (t_d) (see Figure 7) is located within the execution time (e) of the task instance. Error-correcting mechanisms triggered by the SMS (safe multicore scheduler, see chapter 4.3.1 of implementation details of the developed LB-*Pfair* algorithm) can be initiated immediately. Therefore, the error reaction time (r_i) is shorter compared to a homogenous redundancy design pattern (see Chapter 0). Moreover, the error-detecting probability is higher [61], especially for permanent errors compared to the homogenous redundancy pattern. The disadvantage of coded processing is the huge overhead of execution time caused by the coded operation, if this coded operation is not executed in hardware.

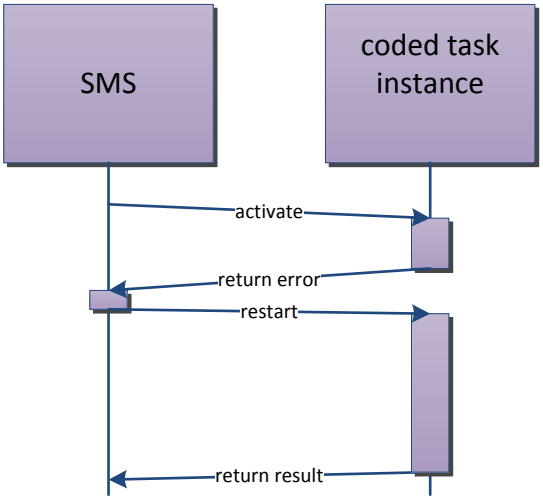

Figure 8: Information Redundancy pattern applied to the task execution in a real-time operating system. Errors can be detected during the execution of a task. Thus the error reaction time can be shorter than in the homogeneous redundancy approach.

Figure 8 depicts the execution of the information redundncy pattern by coded processing. For further considerations of the information redundancy pattern of the Safely Embedded Software (SES) approach (see 0) will be used. An advantage of the herein used AN-Codes is that all computation can be executed in the coded domain – by corresponding arithmetic operations – without transfering data back to the original domain for the actal computation. Further possibilities of applying coded processing were presented in the Chapter 2 “State of the art”.

Safely Embedded Software (SES)

One part of the Safely Embedded Software (SES) [56] approach is the coding of operations using AN-Codes, based on Forin's, the vital coded approach [62]. Thus SES can be used as a coding technique for the information redundancy design pattern (see Chapter 0). Data diversity is achieved by transferring all operations to the coded domain.

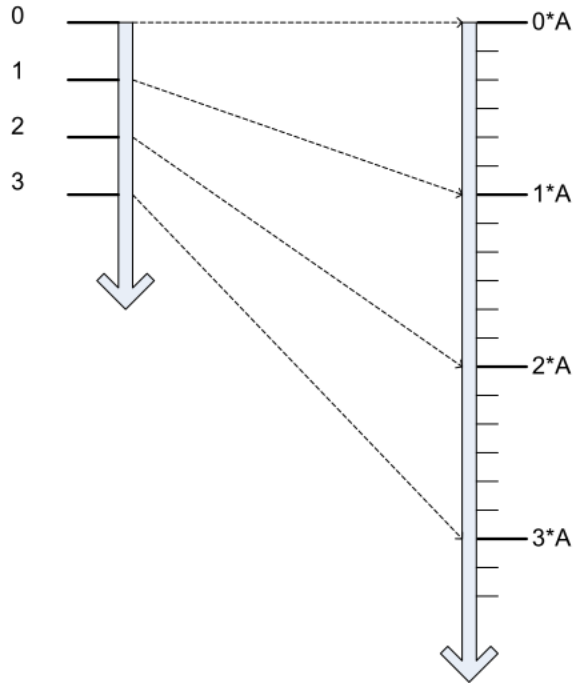

Figure 9: Depiction of AN-Codes used by SES; the data is transferred to the coded domain [55]**.**

During the past research, the described SES approach was implemented using C++ with template classes [14]. Hence, a programmer can transparently use coded variables, when including the library and using the safe data types. 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:

$$x_c = A * x_f + B_x + D_j \tag{2-13}$$

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 on 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 these 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 task instance (j) of a task [56].
The correctness of the computation can be tested with a simple modulo-check during execution.

Homogenous Redundancy Pattern – Redundant Execution

In this pattern, the same task (same program code) is duplicated into n task instances which are executed one after another in a single-core system (time redundancy) or on different cores in a multicore system (space redundancy). This approach is called n-modular redundancy (NMR) or dual modular redundancy (DMR) for n = 2. The computation results of the n instances are compared and evaluated after complete execution by the voter which can be integrated in the safety supervisor of the safe multicore scheduler (SMS) (see Chapter 4.3.1). In consequence, the time of error detection (t_d) is at least greater than the execution time (e) of a task instance. If the number of redundant instances n is greater than 2, the voter can determine the correct result by majority voting.

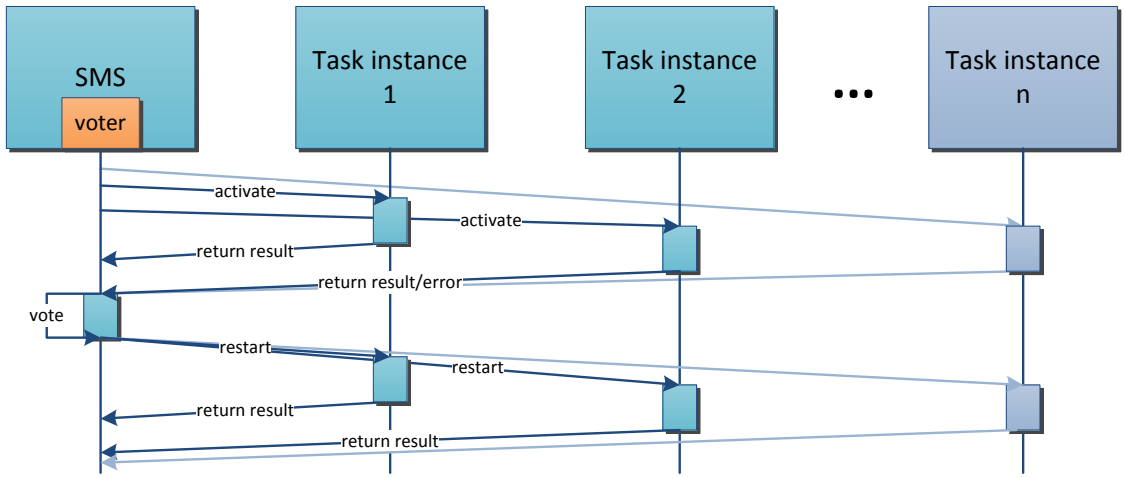

Figure 10: Homogenous Redundancy Design Pattern applied to the duplicated task execution in a real-time operating system. An error during one of the executions of the two instances is detected by the voter – which is integrated in the SMS – at the end of the execution.

A single point of failure in this approach is the voting mechanism. Hence, there have to be other mechanisms to ensure its reliability. A possible solution would be to use an intelligent hardware unit that performs the compare operation. Another solution is to include a voter that is implemented by using coded processing. (see Chapter 0).

In Figure 10 the execution of such a safety task with redundant execution is shown how it could be executed in the safety multicore scheduling architecture (see Chapter 4.1). The results are compared at defined synchronization points

(see also Figure 11). The scheduler can add a delay for the execution of the second instance. That way, both time and space redundancy can be realized.

This design pattern can be combined with the information redundancy design pattern to achieve a higher error detection probability and a lower failure reaction time; this is illustrated in the next paragraph.

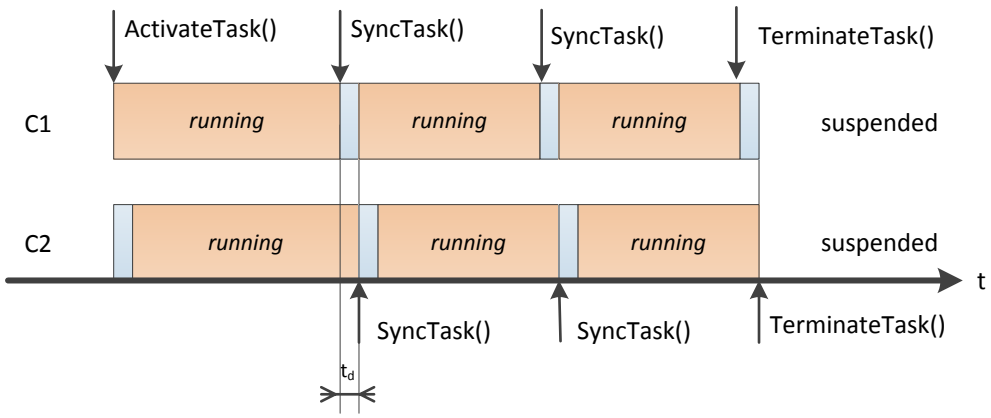

Figure 11: Gantt-chart of a time and space redundant execution of a safety critical task on a multicore system.

Combined Error Detection Approaches

The presented patterns – information redundancy and homogenous redundancy – have advantages and disadvantages. Therefore, it seems appropriate that combined approaches are evaluated. As mentioned earlier, when using the redundant method (dual or triple modular), the single point of failure is the voting mechanism. When speaking of a triple modular redundancy, it can be determined by the voter (assumption: single fault) which channel is the erroneous one. Hence, the correct result is produced by the two remaining channels; no further recovery action is required.

But the voter itself has to have very high reliability. It is possible to implement the voter in a coded processing technique. Thus, the actual decision is safeguarded by coded processing. The time-consuming coding is reduced to the voting sections – the synchronization points of the homogenous redundant executed task. These sections are marked as SyncTask() in Figure 11. Hence, the drawback of the coding – the longer execution times – can be minimized. The main computations can be performed redundantly in the original domain. Figure 12 depicts this combined approach. Ulbrich et al. [58] describe a similar concept, but they do not really cover real-time scheduling demands.

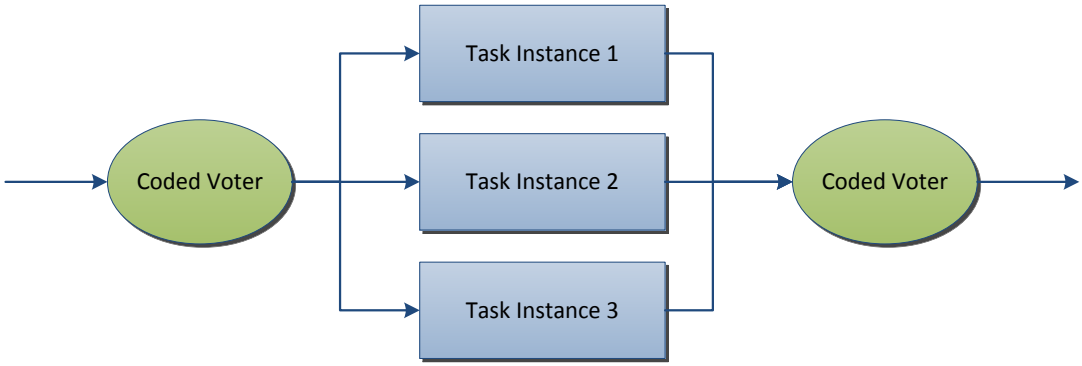

Figure 12: Combination of coded processing and homogenous redundant execution, the single point of failure – the voter – is equipped with coded processing.

2.3.2.2 Error Reactions

There are several known error reaction concepts. After detecting an error an appropriate error reaction has to be triggered. Important concepts for this work are:

- Recovery with check-pointing
  In case of duplicated execution (e.g. lockstep) Subramanyan [51] describes the procedure as follows: Two cores exchange fingerprints to detect errors. If the fingerprints are found to match in both cores, the cores store all architecture registers in the checkpoint store. If the fingerprints do not match in at least one core, then recovery is performed in three steps. Firstly, the register states of the two processors are restored from the checkpoint store [51]. Finally, the leading processor restarts execution from the next instruction after the last checkpoint.
- Active and passive Replication
  In the case of active replication, all replicas of processes are executed independently of fault occurrences. This concept is described in more detail in Chapter 0 by the homogenous redundancy pattern. In the case of passive replication, also known as primary-backup, replicas are executed only if faults occur [2].
- Re-execution
  In the case of error detection, the influenced task instance is aborted and the complete task instance is started again with the same input parameters. According to Beitollahi [28], this is a relatively inexpensive method for an error reaction.

The method of re-execution is the preferred and primarily investigated solution because of the existing presence of the possibility of re-starting task instances in contemporary real-time operating systems.

3. Objectives of the Thesis

In recent years, the functionality and complexity of embedded systems in different domains have been increased to a greater extent. In addition, the requirements for safety, reliability and availability are continuously growing [2]. As a result of decreasing feature sizes, the probability of arbitrary hardware faults is rising, thus the necessity of fault-tolerant, real-time software approaches is growing. Multicore systems played an important role in the field of commodity hardware in the past years. In the automotive industry, first embedded projects are realized nowadays by multi-core embedded systems to satisfy the requirement of higher performance. In the future, they will be employed in safety relevant applications to combine performance requirements and fault-tolerance requirements. Multicore embedded systems offer promising premises to implement a fault-tolerant real-time system that can tolerate transient faults and at the same time meet its timing constraints.

In the first part of this dissertation, the basic state of the art concepts of software fault-tolerance (Chapter 2.3.2) are presented. These concepts are based on redundancy. Redundancy can be implemented as symmetric software redundancy, where data or program code is duplicated. It can further be classified by time and space redundancy. Diversity is a second kind of redundancy, there, an alternative channel is implemented or coded in a different way. These can be pure software solutions or hardware solutions as well.
The second part of this thesis relates to different methods of multicore scheduling (Chapter 2.3). This includes static, partitioned multicore approaches and global approaches with dynamic scheduling. The fluid schedule concept, which is the base of many global, dynamic scheduling algorithms, is enhanced by the concept of the discontinuous fluid schedule (Chapter 4.3.1.1). This allows dynamical reaction to load changes, caused by detected faults.

Discrete event simulation with fault injection is introduced to evaluate the fault-tolerant multicore scheduling algorithm LB-*Pfair* (see Chapter 4.3.1 and 4.4). The simulation driven approach offers the possibility to model the application of an embedded system in an abstract way, during an early phase of development. The system's reliability indices can thus be determined by the simulation. Moreover,

possible modifications of the system can be done in a very early stage, thus costs for iterating tests – including hardware tests – can be saved.

Most publications either focus on efficient multicore real-time scheduling or on the improvement of standard fault-tolerance mechanisms like DMR or TMR in combination with static, partitioned scheduling algorithms.

The main goal of this Ph.D. thesis is the development of a reliable, fault-tolerant, multicore, real-time scheduling algorithm for mixed criticality embedded applications that can tolerate transient hardware faults and at the same time keep the timing constraints of the safety-critical tasks.

This main goal can be subdivided to the following objectives which are covered in the following sections:

Objective 1: Specification of suitable software and operating system architecture to integrate fault-tolerance mechanisms in a real-time operating system

A basic concept of encapsulating safety relevant parts of the software in safety-supervisor and the SES-library is going to be developed (see Chapter 4.1). The usage of different approaches of fault-tolerance – such as DMR, TMR, diversity by coding and combination of the mechanism (see Chapter 2.3.2) – will be evaluated in detail. A flexible architecture that can apply the different fault-tolerance mechanisms according to the required fault-tolerant level of the application will be specified. An architecture has to be developed that ensures the best trade-off between increased fault-tolerance and performance. Thus, an interface has to be defined to safeguard interaction of the scheduler and the so called Safety-Supervisor.

Objective 2: Specification of Error models for reliability evaluation of task execution

For developing fault-tolerant real-time scheduling algorithms, the foundation of error-detection has to be built. Hence, the reliability of the execution of each single task instance has to be analysed in detail to determine the impact of the caused disturbances. Therefore, it is essential for this thesis to formulate a suitable error model that contains the so called operation errors that is based on a probabilistic error model of the used hardware platform. To cover multiple faults, the concept of fault compensation shall be analysed according to its influence on the overall system timing behaviour.

Objective 3: Development of a fault-tolerant robust multicore scheduling algorithm

Moreover, the most crucial part in such a real-time system is the scheduler and its scheduling algorithm which has to incorporate both the timing disturbances of the fault-tolerance mechanisms and has to ensure the real-time constraints at the same time.

There are several approaches for robust multicore scheduling algorithms based on the fluid schedule which show promising real-time properties; but they do not consider the additional error reaction times. Thus, the main goal is:
combining global, dynamic, multicore algorithms with integrated fault handling techniques – considering the error reaction and planning the error reaction time directly in the dynamic schedule – that leads to the creation of a fault-tolerant and still high performing scheduling algorithm.

Objective 4: Validation: Development and application of a discrete-event based simulation methodology for evaluating the developed scheduling algorithm – comparison with analytic extended Markov Model

The developed scheduling algorithms and their interaction with the fault-tolerance mechanisms – integrated in the architecture of a real-time operating system – have to be evaluated according to the gained reliability, the achieved fault-tolerance and the adherence to real-time constraints. Hence, the discrete event simulation with fault injection shall be used to analyze a task set from the automotive domain by simulated fault injection during the simulation run.

The discrete event simulation itself has to be verified by an analytic approach.

4. Fault-Tolerant Multicore Real-Time Scheduling

Due to the fact that all the objectives declared in the previous chapter were covered by published or accepted publications, we decided to organize the core of this thesis as a set of commented publications. This chapter summarizes the publications and depicts the connection and the logical ordering of the individual publications. The publications are referenced by the stated contributions of this thesis.

The following figure gives an overview and visualizes the publications and their relation to each other.

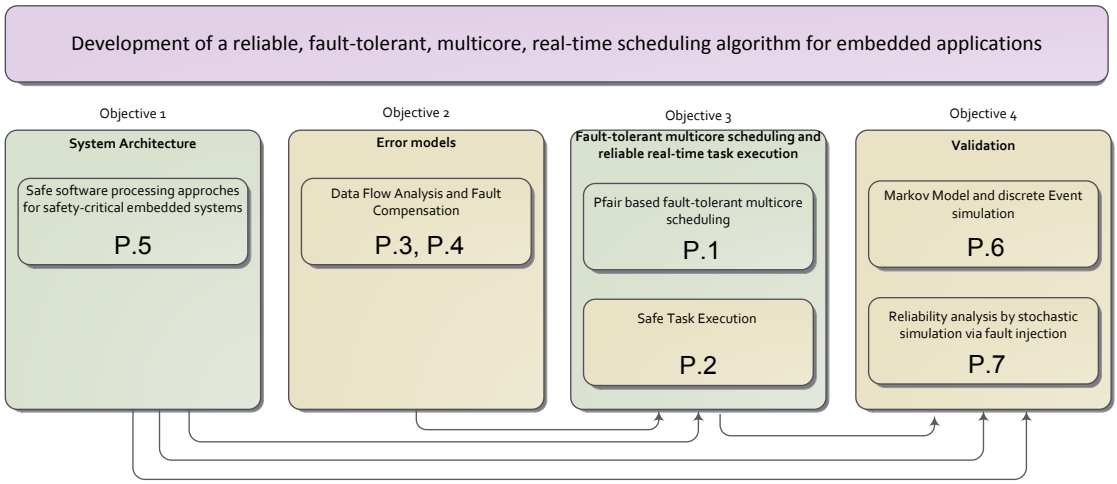

Figure 13: Overview and connection of the publications of the cumulative part of the thesis

The proposed system architecture of a reliable fault-tolerant real-time system is described in publication P.5. This is the basis for the developed fault-tolerant scheduling algorithm (LB-*Pfair*). Publication P.1 is the crucial part of this thesis, as it presents the developed LB-*Pfair* algorithm. The different underlying error models and error detecting mechanisms are presented in publication P.3, P.4 and partly in P.2. They are essential for the developed multicore scheduling algorithm, as they are the foundation for the necessary error detection.
Publication P.4 furthermore shows – as an essential part – that the discrete event simulation can further be incorporated for in depth-analysis of the underlying fault effects. The fourth part of the objectives – the validation – uses a discrete event simulation approach (P.7) and Markov Models (P.6) to verify the correct-

ness and the usability of the previously presented reliability approaches and multicore scheduling algorithms.
In detail, the cumulative part of this thesis is based on the following publications:

P.1 Proportionate Fair based Multicore Scheduling for Fault Tolerant Multicore Real-Time Systems
In Proceedings of the IEEE International Conference on Electrical and Information Technologies, pages 88 – 93, March 2015, Marrakech
Referenced in chapter: 4.3.1

P.2 Reliability of Task Execution during Safe Software Processing
In Proceedings of the 15th Euromicro Conference on Digital System Design, pages 84 - 89, September 2012, Cesme
Referenced in chapter: 4.3.2

P.3 Data Flow Analysis of Software Executed by Unreliable Hardware
In Proceedings of the 16th Euromicro Conference on Digital System Design, pages 243 – 249, September 2013, Santander
Referenced in chapter: 4.2

P.4 Reliability of Data Processing and Fault Compensation in Unreliable Arithmetic Processors
Accepted for Microprocessors and Microsystems: Embedded Hardware Design (MICPRO), http://dx.doi.org/10.1016/j.micpro.2015.07.014, 2015
Referenced in chapter: 4.2

P.5 Safe Software Processing by Concurrent Execution in a Real-time Operating System
In Proceedings of 16th IEEE International Conference on Applied Electronics, pages 315 – 319, September 2011, Pilsen
Referenced in chapter: 4.1

P.6 Comparison of Enhanced Markov Models and Discrete Event Simulation - for evaluation of probabilistic Faults in safety-critical real-time task sets
In Proceedings of the 17th Euromicro Conference on Digital System Design, pages 591 – 598, August 2014, Verona
Referenced in chapter: 4.4

P.7 Reliability Analysis of Real-time Scheduling by Means of Stochastic Simulation
In Proceedings of 17th the IEEE International Conference on Applied Electronics, pages 151 – 156, September 2012, Pilsen
Referenced in chapter: 4.4.2

The subsequent subchapters summarize and explain in detail how the stated publications relate to the declared objectives and how these objectives are achieved

by these publications. Every subchapter contains a summary and a detailed discussion of the results related to the set objectives.

4.1 System Architecture for Fault-tolerant Real-time Multicore Scheduling

This section is mainly based on publication P.5. Several concepts of safe system architectures are also depicted in the publications P.6 and P.2.

> P. Raab, S. Kraemer, J. Mottok, H. Meier, and S. Racek. Safe Software Processing by Concurrent Execution in a Real-time Operating System. *In Proceedings of 16th IEEE International Conference on Applied Electronics*, pages 315 - 319, September 2011.

4.1.1 Summary

The main goal of this publication is to show a simple but effective approach for a system design that includes error detecting mechanism in a real-time operating system in a scalable manner. This publication and the herein stated system architecture provide the basis for the developed enhancements of the *Pfair* algorithm presented in publication P1.

To achieve an increased overall reliability and higher fault tolerance of a system, some kind of redundancy is required. On a single core system, redundancy can be achieved by time redundancy. This means, two instances of safety critical tasks are executed one after another.

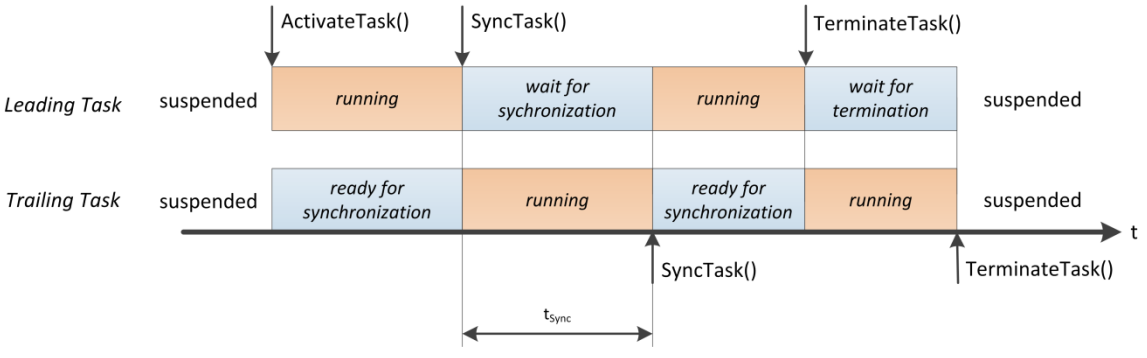

Figure 14: Synchronized redundant task execution on a single core system (see [P.5])

Therefore, arbitrary transient faults can be detected, if the transient fault only affects a single execution instance of the task. If the duration of the transient faults lasts longer than the consecutive execution, an error may not be detected if the effect of the result is the same for all execution instances. The redundant task instances can be allocated on different cores by using a multicore system, space

redundancy can be achieved in this way. Transient faults which only affect a certain core, can be detected by comparing the results.

By using duplication in a DMR (dual modular redundancy) approach, backward recovery is possible if a discrepancy between the two results is recognized. Figure 14 depicts such a DMR-task execution with a leading and a trailing task instance.

The results are compared at synchronization points and an error handling action is triggered by the so called Safety-Supervisor (see Figure 15). The Safety-Supervisor is an extension of a real-time operating system. It incorporates the existing application interface of the operating system to realize the synchronization of the task instances. For example, this can be realized by event-mechanisms that are available in most out of the shelf real-time operating systems. The error reaction – such as task re-execution – is triggered by the Safety-Supervisor. Figure 15 depicts the scalable system architecture.

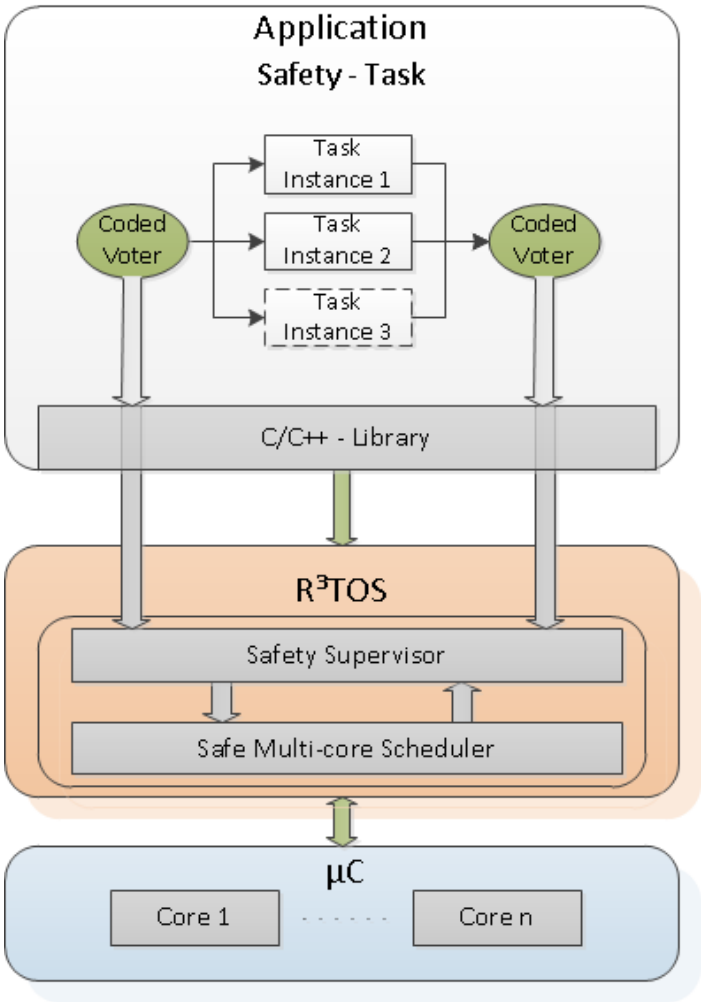

Figure 15: Operating system architecture, with integrated Safety-Supervisor, see [P.5]

Data flow monitoring is demonstrated to be implemented by means of operating systems functionality which can as well be checked by the Safety-Supervisor. To check the correct execution order, the different task instances have to pass the synchronization points at the same execution cycle in a certain time-span. If they reach different synchronization points – that are determinable by the program counter – or if the temporal delay is too big, this can be detected by the Safety-Supervisor.

Instead of adding simple duplication of task instances, arithmetic codes can be used to add data diversity and hereby add coded processing to the system. AN-Codes are often used for coded processing. The Safely Embedded Software (SES) research developed at our laboratory bases on AN+B+D codes. Publication P.5 showes a C++ implantation of a coded processing library that uses C++ template classes and operator overloading to create an easy to use programmer interface for encapsulating coded processing. Therefore, the programmer does not get in touch with the underlying coded processing. The library is tightly connected to the Safety-Supervisor and therefore to the operating system. Coded processing offers the possibility – if two coded instances have been used – to perform a forward recovering by setting the faulty task instance to a valid state by using the error-free task instance. The different influences on the real-time behaviour of the different strategies are observed in publication P.7.

4.1.2 Discussion

The presented concept allows a scalable approach for different levels of required reliability of the embedded system. This reliability or residual error probability is derived from the Safety Integrity Level (SIL) [12], which is determined by a risk assessment of the system. Multichannel hardware can be reduced in a certain level by a coded processing approach. The usage of multicore systems offers the possibility to additionally add space redundancy. The software fault tolerance of this approach leads to an increased processing time requirement, which has a huge impact on the real-time requirements of such an embedded system. The effect on the overall system by this approach is analysed in detail in publication P.7. It has to be considered as well that the higher execution time requirement for coded processing or redundant execution lead to a higher error probability due to the longer execution time. The temporal impact is also dependant on the number of synchronisation points.

This architecture is used as a fundamental experimental platform; all subsequent approaches are based on this architecture. The proposed LB-*Pfair* (publication P.1) uses the Safety-Supervisor to get feedback of the application and the occurrence of errors, although it is used to define the additional amount of execution in case of error detection for backward or forward recovery.
As already mentioned, publication P.7 deals with the implications of the error

detection and correction regarding several important real-time metrics by a simulation approach.

The redundant task execution with synchronisation points and re-execution is in detail analysed by an enhanced Markov Model in publication P.2 (see Chapter 4.3.2, Figure 22). In this figure, the first two columns in the Markov Model represent the redundant execution of task instances with synchronisation points (states 2.x). The transition back from the erroneous state 2.(n-1) to the initial error-free state (1.0) represents the restart of the task instance. There exists a probability that both instances – if affected by a fault – produce the same result and therefore, the error is not detected. This is represented by the third column which shows the split of the execution flow between the undetected error state (3.n) and the transition back to the state 1.0.

The outcome of both validation methods – Markov Model and simulation approach – are compared in publication P.6.
The experimental target platform is the so called R³TOS which was developed at the Laboratory for Safe and Secure System [63] and enhanced to support homogenous multicore scheduling by the usage of the fluid schedule based LLREF-Scheduling algorithm [47], [48].

4.2 Underlying Error Models and Fault Compensation for Fault-tolerant Real-time Scheduling

The impact of arbitrary faults on the timing behaviour of the real-time system is affected by the influence of faults to the data flow. Therefore, error models have to be developed. A special case considered in this section, is fault compensation which may have a positive effect to the overall system timing.

The content of this chapter is based on publications P.4 and P.3.

> P. Raab, S. Krämer and J. Mottok. Reliability of Data Processing and Fault Compensation in Unreliable Arithmetic Processors. *Accepted for Microprocessors and Microsystems: Embedded Hardware Design (MICPRO)*, http://dx.doi.org/10.1016/j.micpro.2015.07.014, 2015.
>
> 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*, pages 243-249, September 2013.

4.2.1 Summary

This section is based on publication P.3, wherein the principle of fault compensation was introduced. This concept is further developed in publication P.4. Moreover linear codes are considered and analyzed for coded data processing. This paper (P.4) connects the analytic consideration of fault compensation and the analysis by discrete event simulation framework used for the evaluation of the multicore scheduling algorithm (publication P.1 and P.6).

The error model for a ripple-carry adder is presented in publication P.3. This model only considers faults affecting the operation itself. However, the result of an operation is dependent on its inputs as well. Considering the whole program flow, it becomes apparent that faults in the input of a subsequent operation have impact on the correctness and reliability of the final result. For an add-operation the two inputs have to be loaded before executing the add-operation by two mov-operations to copy their content to the working registers. The reliability of the final result is the serial concatenation of the reliabilities of each single instruction, therefore:

$$R = R_n * R_{mov} * R_{mov} * R_{add} * R_{n+x} \tag{4-14}$$

The overall reliability of the chain of operations decreases with an increasing number of instructions. The more instructions, the longer the execution time, the higher the probability that different independent faults influence the execution in a way that faults can compensate each other. This effect will be visible if the faults influence the registers in a contrary way. This means that the overall reliability of the program is higher than expected by simply applying the serial reliability model. Thus the overall reliability is higher than the analytically calculated one. The effective reliability R_e can be expressed by:

$$R_e = r_c * \prod R_n \tag{4-15}$$

The faults can occur in different locations of the computation. They can affect the operation itself and/or the operands. In P.3 a Markov Chain describing the behaviour of a faulty addition including the memory effect resulting from the carry bit of a ripple-carry adder was introduced. This model considered a single fault that leads to a transition to an error state. It is a kind of abstraction and has to be adapted to the concrete implementation of the hardware.

A reliability evaluation of multiple transient faults by a Markov Model has to consider all three elements (both inputs and the operation itself) that can lead to an erroneous outcome. Fault compensation will only be possible if two or more faults occur. A fault in an operand can be overridden by the successive add operation. In publication P.3 multiple consecutive instructions with no memory (no carry bit) are considered. It is shown that a longer execution results in a higher

effective reliability. This compensating effect is shown analytically and verified by fault injection in a discrete event simulation framework (P.4).

4.2.2 Discussion

These publications show that fault compensation cannot be neglected, especially not, when considering a larger amount of instructions as with an increased number of instructions, the probability that faults compensate each other, rises. The resulting effective reliability is higher than the reliability defined by the serial concatenation of the individual reliabilities of the operations. Both publications P.3 and P.4 cover simple data flow examples. Real applications consist of multiple instructions with non-linear data flow, where the outcome of an operation is dependent on the outcomes of the previous operations. In these more complex data flows, the probability of fault compensation is assumed to be higher. This is based on the assumption that the probability of a fault in each operation rises with its complexity and execution history. This aspect has to be considered in future research.

The analytic model was evaluated using discrete event simulation and fault injection. A consecutive, linear instruction flow was modelled and faults were injected independently in different bits of the data word while simulating the execution of an XOR-instruction. P.4 shows that with some assumptions a fault compensation factor r_c can be derived:

$$r_c = \cosh(pk)^n \tag{4-16}$$

Where p is the probability of a faulty bit, k is the number of consecutive instructions and n is the considered bit width.

The discrete event simulation results – to determine the compensation factor – have shown that the analytical model produces nearly equivalent results. The deviations are in the range of numerical inaccuracy.

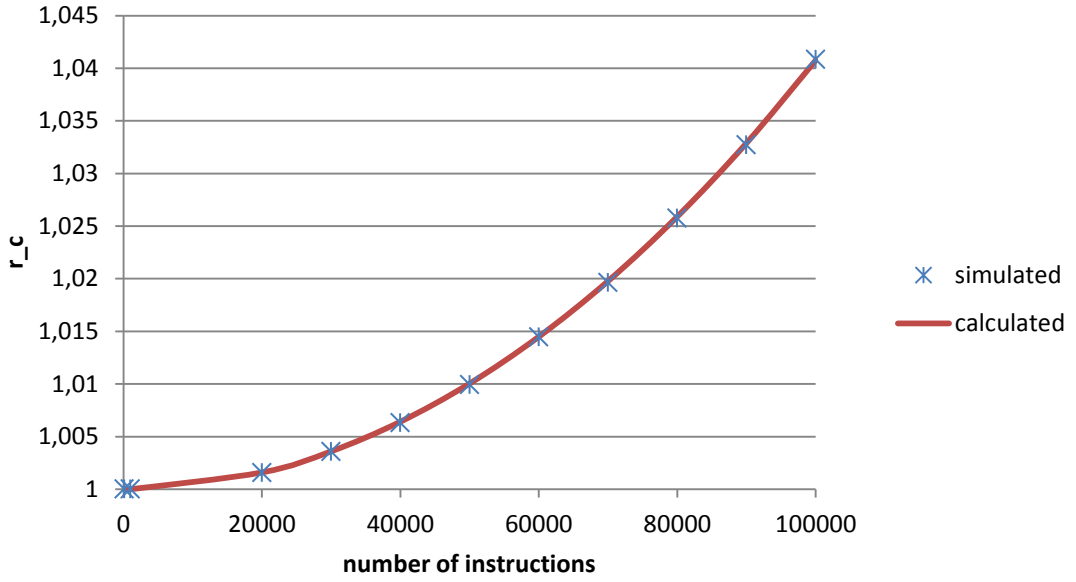

Figure 16: Comparison of the compensation factor r_c; the red line shows the analytically calculated r_c factor, the blue crosses selected simulation results; for individual values, see Table 1 in publication P.4.

Figure 16 depicts the evaluation results. This evaluation illustrates the possibility to investigate fault compensation in complex systems with discrete event simulation, because it was shown that the results of the analysis by the Markov Model produce equivalent results for the investigated system.

Outlook: The effects of the fault compensation can be considered in the timing simulation of an entire embedded real-time system.

4.3 Fault-Tolerant Multicore Scheduling

For the purpose of fault-tolerant multicore scheduling, different global scheduling algorithms were evaluated. The fluid-schedule based algorithms offer the possibility to dynamically react on different load conditions that occur, when the effects of transient faults have to be handled during safe software execution. An overview of these algorithms was given in chapter 2.3.

4.3.1 Enhancement of PFair based Multicore Scheduling for Fault-Tolerant Real-Time Systems

The content of this chapter is derived from publication P1.

> S. Krämer, J. Mottok and S. Racek. Proportionate Fair based Multicore Scheduling for Fault Tolerant Multicore Real-Time Systems. *In Proceedings of IEEE International Conference on Electrical and Information Technologies 2015*, pages 88 – 93, March 2015

4.3.1.1 Summary

In chapter 2.1.3.2 global dynamic scheduling concepts were introduced including the *PJair* algorithm. They are based on the fluid scheduling approach, which describes a continuous task execution. The utilization or the task weight defines the slope of the fluid schedule. According to literature [9], [64], the utilization is independent of the point in time of the task period. For achieving the fault-tolerance, the same methods – TMR with coded voter or coded DMR – have to be applied to the task set (see Chapter 0).
In this publication the developed concept of the discontinuous fluid schedule (see Figure 17) was applied to the group of Proportionate Fair (*PJair*) global dynamic scheduling algorithms. *Pfair* [6] based multicore real-time scheduling for periodic tasks is different in one aspect, compared to other real-time scheduling approaches. The task instances are executed at a steady rate. Meaning they execute at constant speed in a way that they finish right within their deadline, derived from their period. This execution along this steady rate is called fluid schedule. As the execution rate of a core cannot be varied, the execution of a fluid schedule is approximated by dividing the task execution time in so called quants, that are either executed or not. The division into quants – by bypassing the bin-packing-problem – is one of the reasons, why a *Pfair* Schedule can be calculated online in polynomial time for periodic task sets [6].

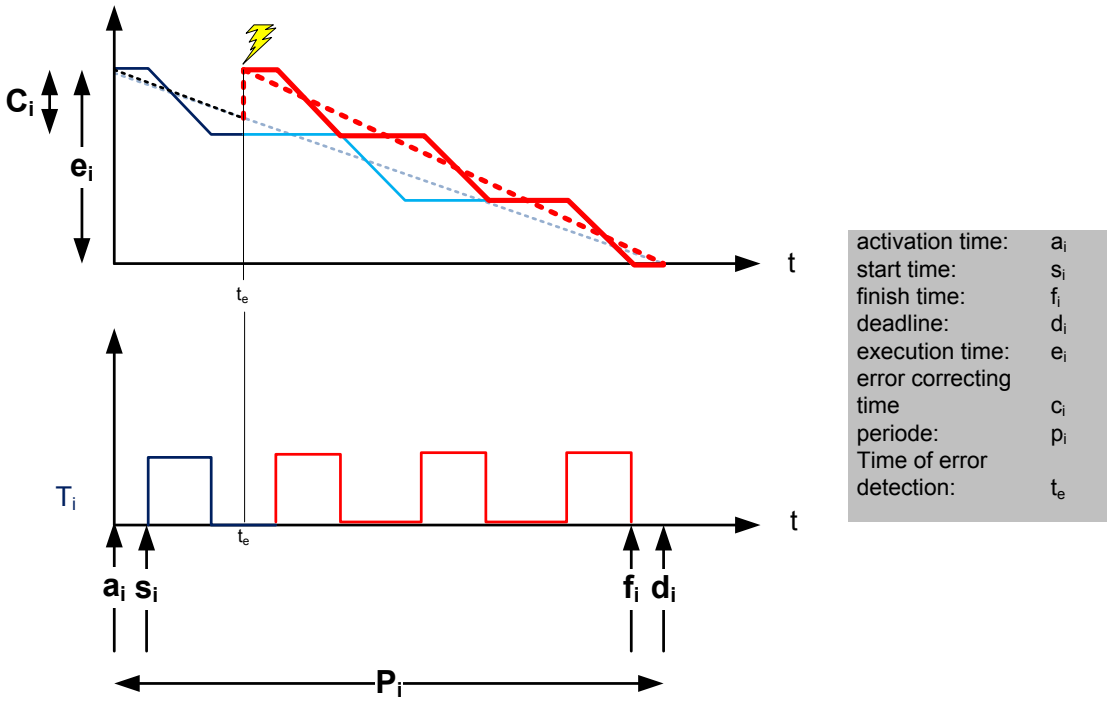

Figure 17: Depiction of the enhanced fluid schedule – the discontinuous fluid schedule – considering the additional execution time in case of fault detection and error reaction.

The developed enhancement of the fluid schedule model leads to a no longer constant utilization of the cores. Therefore, the slope of the fluid schedule is no longer constant within the task period. Now the utilization could increase during the execution of task, if a fault reaction is reported to the scheduler by the Safety-Supervisor. The utilization rises at time t_e by c_i – the time necessary for the repair action. This additional time c_i is known a priori (by the Safety-Supervisor) and hence predictable for the scheduler. This is depicted in Figure 17 by the jump discontinuity of the fluid schedule at time t_e.

This new discontinuous fluid schedule is the foundation for modifying the *Pfair* algorithm introduced in Chapter 2.1.3.2. The herein presented extension is applicable for many fluid schedule based algorithms such as LLREF (see Chapter 2.1.3.2).

Coupling of Scheduling Algorithm and Error Detection Sub-System

The input to determine the dynamic slope of the discontinuous fluid schedule is the feedback loop of the error detecting sub-system. The error detecting subsystem consists of the Safety-Supervisor and the coded processing (SES) library (see P.5 for the description of the Safely Embedded Software library (SES)). They

form an additional layer between the operating system and the application. Figure 18 illustrates the architecture of the operating system with the LB-*Pfair* scheduling algorithm. The Safety-Supervisor is responsible for reporting the additional execution time (c_i) – in case an error was detected – and for the synchronization of the different instances of the safety critical task instances.

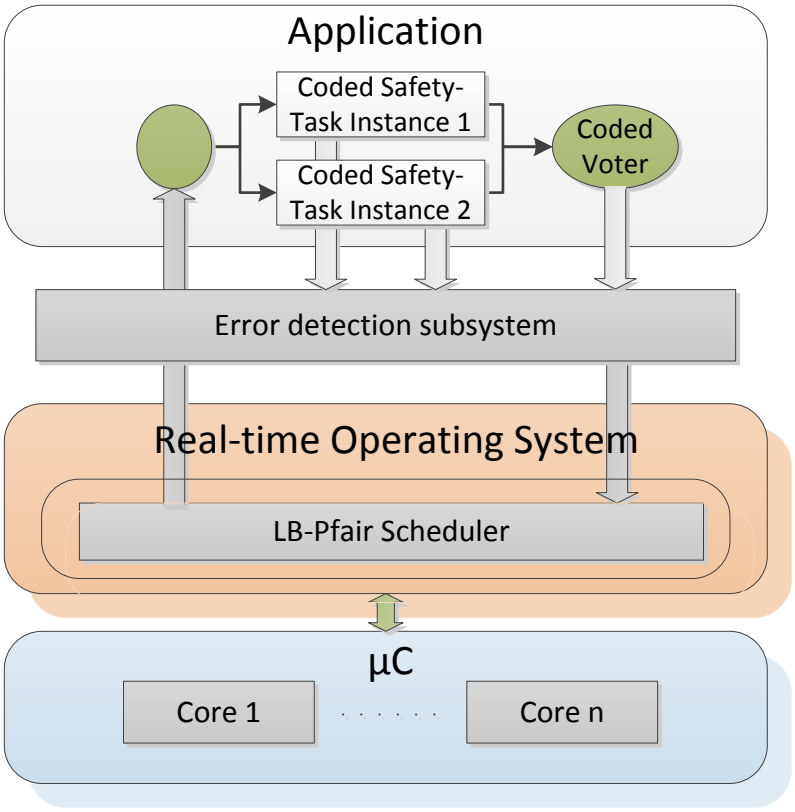

Figure 18: Operating system architecture for the LB-Pfair scheduling algorithm, based on the general architecture of Figure 15.

Basic Pfair Scheduling

Pfair scheduling is designed for scheduling periodic task instances. A periodic task instance T_i is defined by its execution time e_i and period (p_i). To satisfy the requirement for approximate a uniform execution rate, each subtask (quant) has to be executed in a certain window. Different subtasks may execute on different cores, but they may not execute in parallel on different cores.

There are several extensions and modifications of the original *Pfair* algorithm, such as PF, PD, PD^2 or ER [41]. The latter (ER = early release) is suitable for fault tolerant scheduling as it does not misspend execution time. It will immediately schedule tasks, if a free execution resource R is available [P.1].

The main scheduling criteria, besides some tie breaking rules, is referred as the task weight wt_i that is the criteria for calculating the pseudo deadline $d(T_{i,j})$ for each individual subtask (j) of T_i. It is defined by:

$$wt_i = \frac{e_i}{p_i} \tag{4-17}$$

Informally that is the rate at which the task should be executed. The pseudo deadline for a subtask follows:

$$d(T_{i,j}) = \left\lceil j \frac{1}{wt_i} \right\rceil. \tag{4-18}$$

Pfair is proven to create feasible schedules for k tasks on M execution resources, if [45]:

$$\sum_{i=0}^{i<k} \frac{e_i}{P_i} \leq M. \tag{4-19}$$

Fault-Tolerant Enhancement of Pfair Scheduling – LB-Pfair

The discontinuous fluid schedule mainly has an effect to the task weight wt_i, it is now no longer constant over the execution period of a task instance. It is incremented according to the reported additional execution-time by the Safety Supervisor.

Moreover, the actual extension of the scheduling algorithm LB-*Pfair* includes execution allocation constrains and subtask-synchronization. Same subtask instances of coupled safety critical tasks that are executed in space redundant manner are not allowed to execute on the same execution resource R. For the couple T_i, T_{i+1} follows:

$$R(T_{i,j}) \neq R(T_{i+1,j}) \tag{4-20}$$

To avoid delays for comparing voting results, the coupled task instances have to be synchronized. This criterion can be realized indirectly by considering the same execution rate and initial task weights. Additionally, the usage of event mechanisms offered by operating systems can be applied to achieve the necessary synchronisation (see Figure 1 in publication P.1).

As already mentioned, with this extension to the fluid schedule, the task weight wt_i is no longer constant over system lifetime (In original *Pfair* algorithm it is considered as constant over time, but nevertheless the real execution time on a target may also fluctuate.). The task weight is now additionally dependent on the occurrence and detection of an error. For correcting a detected error, the task instance is reset. In other words, it is re-executed. This extra execution time c_i results in a temporally increased task weight wt_i*.

Different system behaviours can be applied according to the system configuration and the required safety level. The algorithm supports backward-recovery – as mentioned before – and forward recovery. Forward recovery can by incorporate if

both instances use coded processing (see Chapter 4.1). Then it is possible to reset the faulty task instance with the results of the error free task instance because by coded processing, it is possible to detect which task instance is the faulty one. LB-Pfair can be applied to systems with simple DMR or TMR architectures. In this case it cannot be determined which instance is the erroneous one, thus both instances have to be re-executed.

The error reaction and the higher dynamic weight in turn, leads to a temporarily higher dynamic priority of the faulty task instance. The ER-extension for Pfair-algorithms is also applied for LB-Pfair scheduling. Due to its non-work-conserving properties, it is suitable for a safety-critical task to ensure free resource capacity at the end of a period in case of error occurrence. If the Safety Supervisor layer detects an error the scheduler will be informed about the time of error occurrence t_e by the control loop. The error correcting time c_i is defined by:

$$c_i = t_i - s_i \tag{4-21}$$

The dynamic task weight wt_i* follows:

$$wt_i^* = \frac{e_i + c_i}{p_i} \tag{4-22}$$

wt_i^* is derived from the system error rate λ. Because of the complexity of the task systems and the time of error occurrence as well as the considered execution time, wt_i^* can hardly be determined in an analytical way for an arbitrary system. Therefore evaluation is done by a discrete event simulation approach (See section 4.4.1.2).

The local deadline can therefore be defined by:

$$d^*(T_{i,j}) = \left\lceil j \frac{1}{wt_i^*} \right\rceil \tag{4-23}$$

A feasible schedule can be calculated if the following equation is satisfied for every point in time.

$$\sum_{i=0}^{i<k} wt_i^* \leq M\big|_{\forall t}. \tag{4-24}$$

Evaluation by Simulation

For the simulation based evaluation of the LB-*Pfair* scheduling algorithm, a multi seed discrete event simulation is used. The discrete event simulation is used to model the variance of task execution durations and the general feasibility of a schedule in a multicore real-time embedded system. The reliability indices are determined by probabilistic fault injection of transient faults. To cover a wide range of the design space, each simulation run is executed multiple times. In order

to establish a reasonable confidence in the result and to gain a high precision, the simulated time span of one single run was set to 5000 seconds that were executed 1000 times each [P.1].

The modelling of the task execution is very close to real task execution on a target (see Figure 1 and Figure 2 in publication P.1). In this example, at the beginning, input data is read at first, then the safeguarded processing ("Actual Function") is executed and the result is checked. If no error occurred, the task terminates normally and writes to the output. Otherwise, if an error was detected, the scheduler is called and informed about the additional execution time in case of LB-Pfair scheduling.

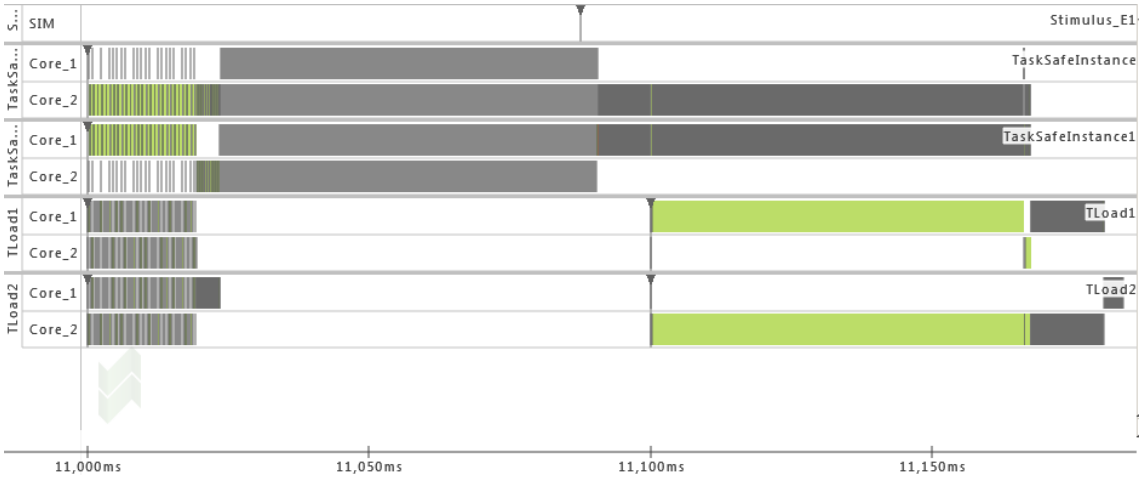

Figure 19: Simulated execution of the LB-Pfair scheduling algorithm with DMR (TaskSafeInstance1 and TaskSafeInstance2) and two non-safety-critical tasks (TLoad1 and TLoad2). Stimulus_E1 represents the injected fault. (visualized with ©Timing Architects Tool Suite [65])

Figure 19 illustrates the execution of two safety-critical task instances (TaskSafeInstance1 and TaskSafeInstance2) and two non-safety-critical instances (TLoad1 and TLoad2). A fault is injected with Stimulus_E1. It can be seen that in the time span where no error has occurred, the safety-critical and the not-safety-critical task instances are executed alternatingly according to their weight in the fluid schedule. After the fault was injected by Stimulus_E1 the dynamic weight of the safety-critical instances is increased. The non-critical tasks are periodically activated but are not scheduled by the algorithm because of the higher dynamic weight of the safety-critical tasks. Their execution is delayed until successfully finishing the critical tasks.

Simulation Model

The real system behaviour can be depicted quite accurately, starting at the level of instructions up to the modelling of the application with tasks and operating

systems by usage of Monte-Carlo based discrete event simulation framework. The used model consists of a hardware part defining the processor which itself consists of multiple cores. These cores execute the instructions defined in the application part of the model. The execution time of these instructions is defined by the clock of the core and the amount of instructions the core can execute per tick.

A next part of the model describes the operating system architecture, including the LB-Pfair algorithm.

In the application part of the model the different task and interrupt service routines are defined. A task itself contains several function calls and possesses the capability to model conditional branching of the execution flow. Functions are assembled by instruction blocks and define the amount of instructions (constant number, or defined by a distribution) that have to be executed by the core.

Furthermore the stimulation of the simulated system has to be described. This stimulation includes the activation of tasks and ISRs or the triggering of the fault injection. These trigger mechanisms can either be configured periodically or can be varied by a distribution [P.6].

The investigated system is represented by its random behaviour. To obtain valuable results, several repetitions of the simulation with several randomly generated input properties have to be performed to reduce the variability of the evaluated results. The basic input of each property is produced by a pseudo-random number generator.

For each simulation run – which is a set of individual simulations with randomly generated runtime behaviour – of the Monte-Carlo Simulation, the input parameters such as the applied fault rate, is varied.

Input parameters for the discrete event simulation are (among others): the studied task-set (runtime, executed functions), the multicore scheduling algorithm, the fault detection and correction mechanisms and the transient fault rate applied to the simulated system.

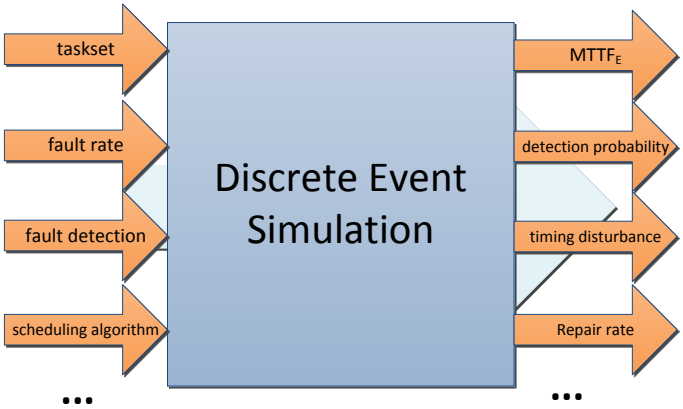

Figure 20: Overview of the discrete event simulation for evaluating the results of the applied strategies.

Simulated Software Fault Injection

During the simulated task executions faults are injected via system stimulation. The fault injection rate λ is varied over simulation time in a certain range. Different kinds of timing behaviour of faults were applied to the system. Used fault patterns are single faults, multiple faults and bursts of faults.

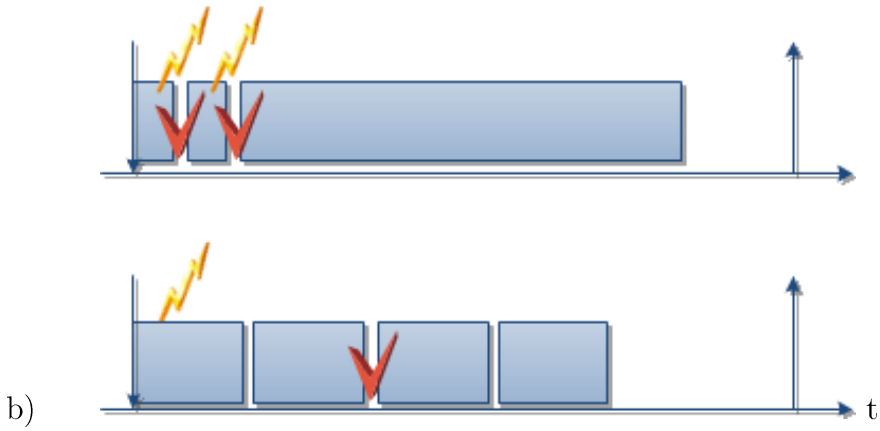

Figure 21: Simulated fault injection and simulated fault reaction: re-execution.
a) Multiple faults: Coded task
b) Single fault: Redundant task on a single-core

To do a worst-case analysis, the simulation is configured to simulate bursts of faults and therefore covers heavily faulty situations, e.g. caused by fluctuations of the power supply. Thus multiple faults are quite likely. The injected fault rate (λ) is varied in a range from 1ms to 200ms, which means (in this setup) there will be at least one injected fault during the execution period of the safety-critical task. Faults are injected on each core individually. Thus faults would affect safety-critical and QM-tasks. However, error detection and correction is only implemented for safety-tasks. Another aspect of using burst fault injection is that usual fault

rates (ASIL C, $10^{-8} < \lambda < 10^{-7}$, [12]) would lead to long simulation times with no additional benefits. Thus the simulation focuses only on the error case.

Observed Error Detection Architecture

According to Figure 1 in publication P.1 error detection for the two safety-critical tasks is considered by dual modular redundancy and coded processing. That means that data is processed in the coded domain by two different instances (different coding) in space redundancy on the dualcore. In case an error is detected by the comparator the two task instances are activated again and therefore re-executed. See also publication P.2 for an analytical approach (see Chapter 4.3.2).

Metrics and Benchmarks for Evaluation of the Simulation Results

By tracing the different events of the simulation a great variety of timing and reliability metrics can be evaluated. The simulation is used to simulate not only the standard reliability indices, but also to simulate the real-time characteristics of an embedded system. To evaluate the simulation results of the investigated task sets with the applied LB-Pfair, standard metrics such as the number of deadline misses, the maximum normed lateness, the laxity, the jitter and the response time were evaluated. These metrics mainly relate to the time domain and are common metrics to make a statement about the temporal properties of a task set.

In this publication we focused on the response time as an evaluation criterion.

4.3.1.2 Discussion

To evaluate the benefits of the extensions to the *Pfair* algorithm, analytic approaches can hardly be used because of the complex interaction of the global scheduling algorithm (bin packing-problem for allocating the task instance to cores) and the direct influence of the fault injection. A simplified model of safe task execution has been analyzed by a Markov-Model. The results of this analysis where crosschecked with the result of the simulated fault injection that was presented in publication P.6 to ascertain the validity of the discrete event simulation.

Evaluation Results – analysis by Response time metric

The exact simulation results and the comparison to standard priority based scheduling and standard *Pfair* scheduling is noted in Table 2 and Table 3 of publication P.1. They summarize the results of the discrete event simulation runs. As mentioned, the individual simulations were executed 1000 times, each with a sim-

ulation time of 5000 seconds and different seeds for the Monte Carlo-based simulation.
In general it can be seen that there is a difference between the response times for the QM-tasks with LB-*Pfair* scheduling and OSEK scheduling. The average response times (see Table 3 in P.1) are better for QM tasks in *Pfair* and LB-*Pfair* scheduling, because of dynamic task allocation during runtime which allows a more evenly utilization of the cores. For the important safety-critical tasks, the average response time for LB-*Pfair* is better compared to OSEK and *Pfair*. The comparable poor result for standard *Pfair* is an effect of the not considered additional execution time in case of error handling and the originally relative low weight (because of the long period) of the safety critical task and therefore a low priority for this task is the consequence.

It was shown that the developed LB-*Pfair* scheduler reduces the response times of the safety-critical tasks and therefore reduces the number of deadline misses. The execution time of the operating system and the time for context switching was set to the same amount of instructions, as for the OSEK-system.

In future work the exact time for executing the LB-*Pfair* algorithm has to be estimated to improve the simulation model. An implementation of the LB-*Pfair* algorithm is planned to be integrated in the R^{3}TOS operating system to get tracing results from a real hardware device.

4.3.2 Safe Task Execution, Analysis by Markov Models

The content of this section is derived from publication P.2.

> 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.

4.3.2.1 Summary

In this paper the reliability of a single task in an embedded system is analyzed by means of an extended Markov Model. This publication prepares the fundamentals for further extending the analysis of the reliability of a complete real-time embedded system, including the consideration of the temporal behaviour of this system. The coherence to hard timing-requirement is an essential part of the reliability evaluation of a hard real-time system. In a hard real-time system a missed deadline can lead to a system error as well and therefore has to be considered for a complete system reliability analysis. This is further investigated in chapter 4.4.1 and publication P.6.

Markov Models are normally used to analyze the behaviour of a system. In this section we focus on the analysis of the behaviour of a task system. The emphasis is set on the observation of the reliability of this task set. As mentioned, the timing aspects are handled in chapter 4.4.1.

A task consists of a number of consecutive instructions that form the program and the data flow. Transient faults that influence the system can lead to corrupted data or corrupted program code. The outcome of the task is affected in this way. The probability of an error occurrence during the task execution rises with the number of instructions of the task since this results in a longer execution time of the task. The basic principle is to divide the system into several states in the Markov Model, corresponding to the instructions. The states model, the error free and erroneous task executions and therefore the system can be modelled as a Markov process with nodes, representing the states, and the transition between the nodes. The transitions are dependent on the program flow and the applied fault rate. A constant fault rate can be considered in an electronic device. This implies that the time in which a fault occurs is exponentially distributed. This correlates with the underlying exponential distribution of a Markov process.

Regular events like repairs, which are initiated by periodical checks or the runtime of certain software tasks follow another distribution (e.g. normal distribution). This detection or repair rate is not Markovian and the Markov Process cannot be used for this in its original form. Accordingly this rate is based mainly

on a constant runtime of the task [66]. It can contain some variance in execution time, due to timing effects of interrupts or the memory subsystem. Therefore, a Gaussian distribution or a normal distribution is more appropriate to model the task runtime. 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 [67]–[69] is a superposition of several exponential distributed transitions. The enhancement of a Markov Process by adding further states realizes the Erlang distribution. Other distributions like the normal distribution [70] can be approximated by the Erlang distribution. Figure 22 represents the redundant task execution by the extended Markov Model [P.2]. The model was extended by the runtime information. The runtime itself is represented by transitions ($n * \mu$) between the vertical states that denote the instructions of the tasks. With the number of vertical stages (n) the resolution of time of the model can be set. State *3.n* represents the termination of the task with an undetected error and therefore can result in a system failure. State *1.n* represents the error free termination of the task. The horizontal states represent the number of occurred faults. The error rate is denoted by $\boldsymbol{\lambda}$. There are no outgoing transitions from these states (*3.n* and *1.n*), thus these states are absorbing states and the probability for entering these states is unity. The doted transitions depict the periodic task repetition. The effort for computing the time-based state probabilities rises with the number of states. In many cases the probability of all stages is not as important as the probability of the absorbing states. If we consider a periodic non-stop execution (doted transitions) the Markov Model becomes ergodic and reliability metrics such as availability and MTBF can be calculated by frequency analysis.

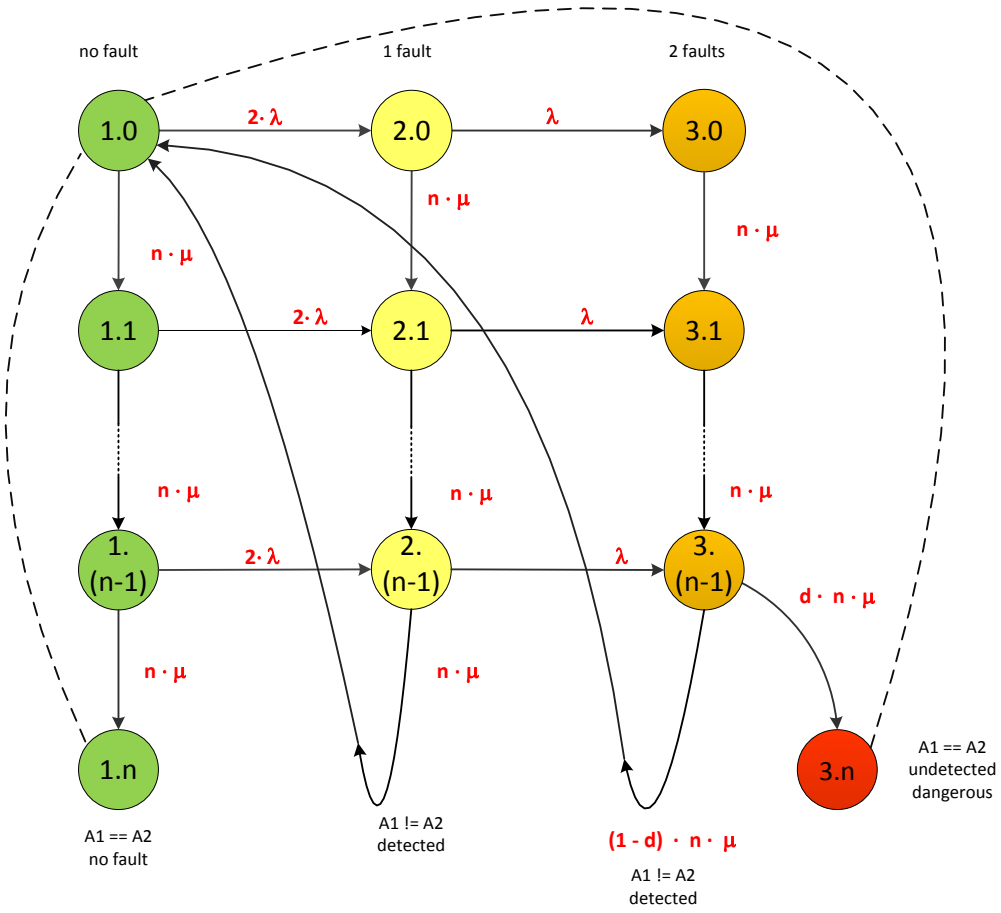

Figure 22: Parallel redundant task execution represented by an Extended Markov Model. In this model re-execution (transition back to state 1.0) is used to achieve error correction.

4.3.2.2 Discussion

A high-level reliability model of task execution by an extended continuous-time Markov Model was presented. This approach can be further used to determine the reliability of a single, uncoded task, a duplicated task or a coded task for single error detection. The different approaches were compared by metrics such as failure probability or MTBF. Coded task processing uses arithmetic error codes for error detection that replaces the duplicated task execution. 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. One task is executed twice as instance A and B, either on different cores (space redundancy) or sequentially on the same core (time redundancy). 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.

For all evaluated approaches, the extended Markov Model allows the determination of task termination and further reliability relevant metrics such as repetitions. The representation by a stage-matrix shows the execution time (column) and the applied redundancy level (row). In this publication the needed additional runtime for coded processing was not considered. All three compared architectures used the same timing parameters. For a more detailed consideration of the timing of real-time task execution the Monte Carlo Simulation was used in publications P.6 and P.7. There the model contains implementation details like branches, variation of task execution times and impacts of the real-time operating system. This paper presents a generic approach which allows different fault rates at each stage, and is therefore capable to model different fault-effects (code, data, and branches) on every single stage (instruction).

The developed Markov Model is the basis for more detailed reliability evaluations. The effect of faults that can compensate each other can be analyzed with this method for example (see Chapter 4.2). By including the task execution time in the Markov Model, it is now possible not only to determine reliability metrics, but also timing relevant metrics of a system. This is extended continuous-time Markov Model is the basis to evaluate the accuracy of the Monte Carlo-based discrete event simulation in publication P.6. Therefore this publication is the foundation for a combined reliability and schedulability analysis of a real-time system.

4.4 Validation of Scheduling Concepts and Error Models

The presented new approaches for multicore scheduling with the LB-*Pfair* algorithm and the safe multicore system architecture have to be validated. Therefore discrete event simulation and an extended Markov Model were used.

4.4.1 Comparison of Markov Model and Discrete Event Simulation

The content of this section is based on publication P.6.

> S. Krämer, P. Raab, J. Mottok, and S. Racek. Comparison of Enhanced Markov Models and Discrete Event Simulation - for evaluation of probabilistic Faults in safety-critical real-time task sets. In *Proceedings of the 17th Euromicro Conference on Digital System Design*, pages 591 - 598, August 2014.

The extended continuous-time Markov Model that was presented in publication P.2 (see Chapter 4.3.2) is used to evaluate a discrete event simulation, which is capable not only to determine the timing characteristics of a multicore real-time system, but at the same time evaluates reliability metrics.

4.4.1.1 Summary

In this publication extended an continuous-time Markov Model and a discrete event simulation are compared to each other by analyzing two architectures for increasing the reliability of an embedded system. After achieving comparable results, the discrete event simulation model is extended to represent a complete real-time system consisting of a simplified automotive task set and a detailed model of the real-time operating system. The two compared system architectures are coded processing and dual modular redundancy (DMR).

As mentioned in Chapter 4.3.2, continuous-time Markov Models are an alternative method for evaluating faults and their impact on task sets. But originally, Markov Models describe the probability of faulty outcomes based on the given fault rate. The necessary runtime information of a task for following fault recovery is not covered. Therefore, the basic Markov Model must be enhanced by the runtime information using the *Erlang* distribution (see Chapter 4.3.2).

Figure 22 and Figure 23 show the enhanced Markov Model for the previously mentioned scenarios – coded processing and symmetric redundancy. The aim of this publication was not to compare the different architecture, but to compare the

results of simulation and analytic Markov Model. The metrics that were used for comparison are expected value for the response time – that is the time from the activation of a task until it successfully completes its execution – and the steady state probability of task execution times. This is the probability of terminating in a certain discrete time period. This discrete execution time steps are a result of the task re-execution in case of a detected error. Therefore, the step width is represented by the error-free execution time of the task instance.

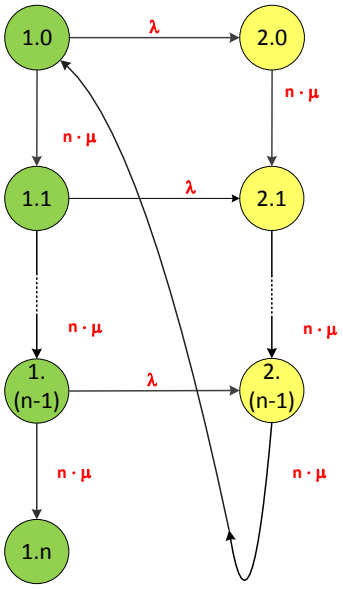

Figure 23: Enhanced Markov Model that simulates the runtime of a single coded task with several stages (n).

The model in Figure 23 describes the faulty execution of a coded task. It models the injection of only one fault (= one additional column) during the task execution which is detected with certainty (= transition to the state *1.0*).

Figure 22 (Chapter 4.3.2) depicts the Markov Model of symmetric redundant task execution. At the end of the redundant (space or temporal) execution both task outputs are compared for fault detection. In case of only one faulty task, the fault is detected with certainty and the re-execution of both tasks is triggered. In case of two faulty tasks, the outputs of both tasks are probably different and the same fault recovery (= task repetition) can be done. However, there is also the probability of non-detection in case both tasks result in the same faulty output. The probability of not detecting a fault is now set to $d = 0.125$ (compare with simulation approach) and it results in the absorbing state *3.n* of Figure 22.

Basically, a continuous-time Markov Model is described by a set of differential equations with one equation for each state which get quite complex for larger and more details systems. Hence, a more detailed analysis of the real-time system by a

discrete event simulation approach based on Monte Carlo Method is used. To determine the timing characteristics for a multicore system with multiple execution resources and dependent tasks, there is no analytic method available to calculate the timing metrics – such as response times, deadline valuations or laxity.

The simulation model is quite accurate. The complete microcontroller system including the cores, the quartzes and the timers are modelled. The cores execute instructions which are the smallest unit. The number of instructions of a function can be constant or can be modelled by different distributions such as Gaussian, Weibull, Normal or Exponential. Tasks are a set of function calls. The tasks themselves are executed by a simulated operating system with exchangeable scheduling algorithms (see [P.1], Chapter 4.3.1). Dependant or probability based program flow branches can be modelled as well as operating system functionality for synchronisation of task instances. Moreover there is a stimulation system that defines a set of stimuli. These stimuli are used to not only activate tasks, but also to incorporate the simulated fault injection. Like for instructions the same distributions can be applied for the stimulation. For task activation a constant period stimulus was used. The stimulus for the fault injection was modelled by an exponential distribution, since electronic devices have a constant error rate which results in an exponential distribution for the fault injection. Figure 24 depicts the simplified model of the symmetric redundant architecture.

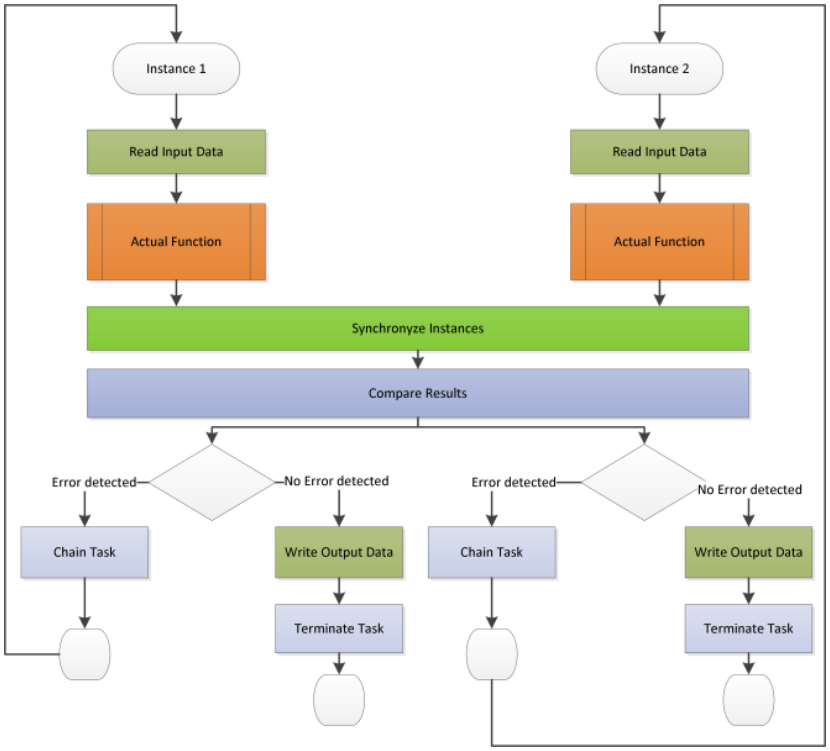

Figure 24: Modelling of symmetric redundant task execution in discrete event simulation approach.

By tracing the occurrence of defined events, numerous timing and reliability metrics can be generated. Figure 25 shows the graphical representation of these events in a Gantt chart. It can be seen that the injected faults cause a task re-execution of the tasks SafeTaskA and SafeTaskB. Thereby the re-execution causes a delay in the execution of the non-critical tasks T1 and T2.

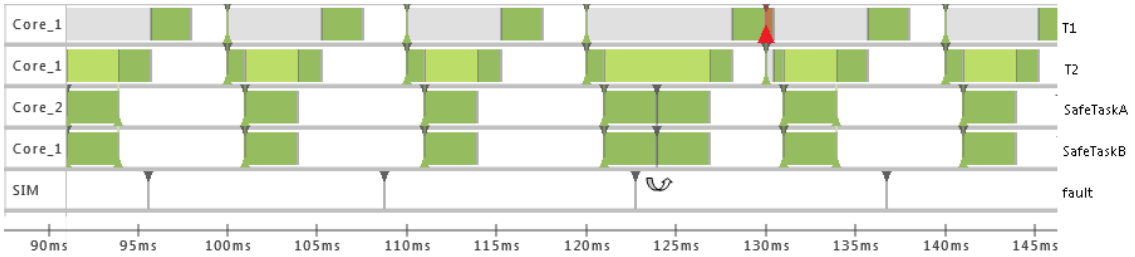

Figure 25: Depiction of Gantt chart of symmetric redundant execution (SafeTaskA and SafeTaskB) with simulated fault injection and task re-execution at error detection (visualized with ©TA Tool Suite [65]).

4.4.1.2 Discussion

In this publication it was shown that both approaches – the enhanced Markov Model as well as the Monte Carlo based discrete event simulation with fault injection – deliver comparable results according task response times and frequency of task execution times.

The result of the analytical Markov approach for the expected response time was within the range of the simulated results for the expected response time of the multi seed simulation. There were some small deviations between the simulated results and the analytical results. The consideration of operating system calls – such as synchronization – and the overhead for scheduling in the more detailed simulation was the reason for this deviation [P.6].
The main outcome of this publication is that with the simulation more complex real-life systems can be analyzed. The timing impacts of the error handling of the redundant task on other tasks were demonstrated with a case study of an automotive-like task set.

It is shown that simulation is capable for analyzing the system in greater detail and that it is usable for examining the influence of the scheduling algorithm on a given task set. The applied scheduling algorithm has to be considered in the whole system analysis which hardly can be achieved by Markov Modelling and is even impossible for multicore systems [15], [53], [P.5].

To achieve a major goal of this thesis – to develop a reliable multicore real-time scheduling algorithm – this paper provides the basis for a tool to evaluate such a scheduling algorithm in a "reality-like" environment. This simulation framework

and the proposed reliable system architecture were used to evaluate the benefits of the LB-*Pfair* scheduling algorithm presented in publication P.1 (see Chapter 4.3.1).

4.4.2 Reliability Analysis by Stochastic Simulation

The content of this section is mainly based on publication P.7.

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

4.4.2.1 Summary

This publication focuses on the reliability analyses of embedded real-time systems by stochastic simulation. Timing and reliability metrics are considered in a holistic approach. This paper is the basis for the subsequent publications and deals with the impact of different error detecting design patterns. A reliability network model was proposed to include timing metrics as an additional serial component, the software triggered error detection and correction as an additional parallel repair element. The violation of timing constraints is considered a soft error. Thus the network model has to be extended by a further serial component, the scheduler integrated in the real-time operating system. On the contrary the depicted error repairing mechanisms diminish the error rate of the whole system. Hence, there is an additional parallel element in the network model. To lower the error rate, the safe multicore scheduling algorithm has to encounter the error and the error repairing action has to return the system to an error free-state without violating any timing constraints.

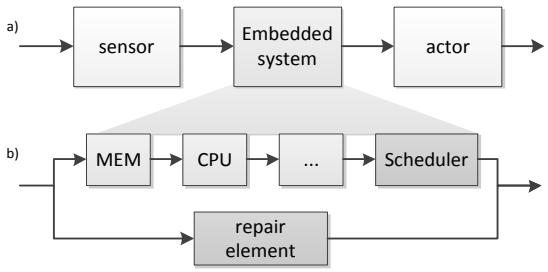

Figure 26 **a) Reliability network model of a software-intensive real-time embedded system, including sensors and actors.**
b) Detailed reliability network of the embedded system including real-time requirements.

The embedded system can be considered as a serial architecture of elements, consisting of memory, CPU, buses, etc. The scheduler represents the timing errors caused by the error-handling mechanisms of the scheduling subsystem. The error-handling / correction software mechanism increases the system reliability and it is therefore depicted as a parallel element (see Figure 26).

Important real-time parameters and classifications are summarized and reconsidered under the aspect of reliable scheduling. Figure 27 depicts the main timing parameters. Main criteria are timing characteristics, like $MTTF_E$, deadline violations and repaired errors. The influence of the type of scheduling algorithm used for a single or multicore system was neglected in this publication (see Chapter 4.3.1).

The two design patterns – coded processing (AN-codes) and dual modular redundancy – were analyzed in detail. The presented patterns refer to error handling in a safe real-time operating system. All patterns have in common that the adherence to timing constraints and error detection is handled by a central unit, the safety supervisor (see Chapter 4.1). It can be configured in different ways. The safety supervisor (see Chapter 4.1) is implemented within the safe multicore scheduler (SMS). In chapter 4.3.1 the safe multicore scheduling algorithm LB-*Pfair* is introduced.

Compared to the analytic analysis (see Chapter 4.3.2 and [P.2]), the execution time for coded processing is considered. The additional execution time of coded processing has a significant influence on the gained reliability.

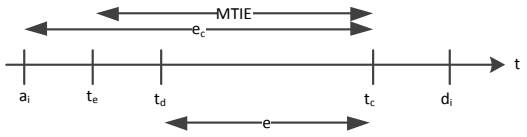

Figure 27: The relevant timing parameters of a task instance. An error is detected at time t_d and repaired within the task deadline d_i at time t_c

A new metric, the effective mean time to failure ($MTTF_E$) for classifying the effectiveness of applied error detection and correction mechanisms, was presented.

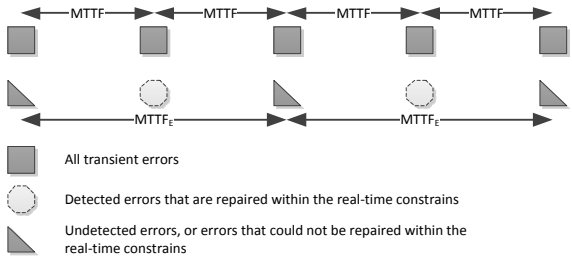

Figure 28: Illustration of $MTTF_E$. Detected and repaired errors lead to an increased $MTTF_E$

A transient error only propagates to a system failure if it is not possible to repair the effects of the error within the real-time timing constraints, namely the task deadline, thus the *Effective Mean Time to Failure* $MTTF_E$ is the MTTF after applying error detection and correction. Figure 28 illustrates this.

4.4.2.2 Discussion

In this publication coded processing and heterogeneous redundancy (dual modular redundancy) were compared by stochastic simulation and fault injection. It was shown that, at very high error rates, neither approach can handle the error correcting within the given timing constraints. Moreover, the two approaches produced very similar results. The high transient error rate applied in the simulation leads to a relatively high number of re-executions. The percentage of remaining free execution time (slack) during the task period is a benefit for the homogeneous redundancy, as the overall slack is higher in this scenario. The probability of deadline violation is lower for coded processing at moderate fault rates due to the shorter error reaction time.

The simulation results have indicated that the slowdown factor of the coded processing has a significant impact on the effectiveness of the coded processing. The elevated execution time leads to higher error probabilities during the execution of the coded task instance compared to the redundant one. For this reason implementation details of the coded processing are very important for its effectiveness.

The developed simulation framework with the integrated fault injection engine is the basis for having an evaluation tool for multicore scheduling algorithms in a safety critical environment (see Chapter 4.3.1). This framework was used to develop the LB-*Pfair* scheduling algorithm ([P.1]). The correctness of this stochastic simulation approach was evaluated in publication P.6 (see Chapter 4.4.1.)

5. Conclusion and Future Work

Multicore systems are nowadays widely used in consequence of the increasing demand for performance and as well as for reliability demands. This trend can be observed in the embedded domain, too. Especially the automotive domain – where costs play an important role – requests concepts for using software approaches instead of pure hardware approaches to achieve the reliability requirements. Multicore systems are perfectly suitable for providing performance and are offering the possibility of multichannel architectures for ensuring the reliability requirements.

To achieve flexibility and cost efficiency, fault-tolerance has to be implemented in software as far as possible. Therefore detailed error models are vital to develop safe embedded system architectures with a flexible and dynamic global multicore scheduler. A scheduling algorithm that fulfills this requirement – of high performance during normal operation and increased fault-tolerance – was implemented in this work (see publication P.1). Basic error models, underlying the scheduling layer, were depicted in publications P.3 and P.4.

Global multicore scheduling in combination with the influence of sporadic faults forms complex systems which have to be considered in a combined approach of handling the reliability requirements and at the same time ensuring the timing requirements. Because of this complexity, discrete event simulation was used to evaluate the effectiveness of the scheduling algorithm (publication P.7) and was verified by an analytic approach in publication P.6.

Thus the connection of these two aspects leads to increased performance in normal operation and gains fault-tolerance in case of the presence of transient faults.

5.1 Results and Objectives

The main contribution of this work is a multicore real-time scheduling algorithm (LB-*Pfair*) that can guarantee an optimal schedule during fault free operation and maximize the adherence of timing constraints during the occurrence of transient faults which was presented in publication P.1. This main contribution can further be subdivided according to objectives declared in Chapter 0:

Objective 1: Specification of suitable software and operating system architecture to integrate fault-tolerance mechanisms in a real-time operating system

Based on the Safely Embedded System concept (SES), system architecture for embedded systems was developed. The focus was set to create a generic architecture that can be integrated in existing real-time operating systems. This architecture uses coded processing for error detection. The coded processing can be used seamlessly due to the developed SES-Software library. The second part to fulfill this objective was the introduction of the concept of the Safety-Supervisor that integrates the SES-library and to provide an interface to the operating system. It is a main pillar of the developed LB-*Pfair* multicore scheduling algorithm, as the Safety-Supervisor is the connection of the scheduling subsystem and the error detecting subsystem. With that tight connection of these subsystems, balance between performance and reliability can be obtained. This concept was described in publication P.5.

Objective 2: Error models for reliability evaluation of task execution

The knowledge of the behaviour, the propagation and the detectability of faults in embedded systems is crucial, since it has impact on the overall reliability and influences the timing behaviour. An error model describing basic instructions – such as MOV or ADD (ripple-carry-adder) instructions – based on a discrete Markov Model was introduced in publication P.3. Hence a basic principle for creating models of different instruction types was introduced. In publication P.5 a further aspect was analysed. With the example of an XOR operation it was shown that faults can compensate each other during program execution, thus that the effective resulting error rate is lower than the expected one. This was presented in publication P.4. This publication is a link between the analytic Markov approach and the discrete event simulation approach that was used to evaluate the timing characteristics of the developed LB-*Pfair* algorithm.

Objective 3: Development of a fault-tolerant robust multicore scheduling algorithm

The main objective was to create a multicore scheduling algorithm that can produce an optimal schedule and handle disturbances of the timing by transient faults dynamically. Therefore the concept of fluid-schedule based algorithms was further developed. The discontinuous fluid-schedule principle was introduced (see publication P.1). It allows the integration of timing information of the error detecting system into the fluid-schedule based algorithms. The *Pfair* algorithm was

extended by this discontinuous fluid-schedule principle. In that way, the LB-*Pfair* scheduling algorithm ensures an optimal schedule during normal operation and dynamically increases the local priority of the safety-critical tasks in case additional error correcting time is reported by the error detecting subsystem.

Objective 4: Validation: Development and application of a discrete-event based simulation methodology for evaluating the developed scheduling algorithm – comparison with analytic extended Markov Model

For the practical usage of the presented LB-*Pfair* algorithm, it is important to determine its correctness. Thus a vital part of this dissertation is the development of an evaluation framework that can analyze timing metrics and at the same time evaluate reliability metrics. Fault injection was used in combination with Monte Carlo-based discrete event simulation. This framework was shown in publication P.7. Publication P.6 is the link between the simulation model and an extended Markov Model. In this paper the correctness of the results of the simulation was compared with the results of a Markov Model of the system. The discrete event simulation is capable to simulate detailed error models which was shown in publication P.4. Fault compensation in a program flow is analyzed herein by Markov Model and discrete event simulation.

As a conclusion this thesis can be seen as the starting point of the development of holistic approaches handling reliability and real-time requirements of future embedded multicore systems. The evaluation of the developed LB-*Pfair* multicore scheduling algorithm – realized by discrete event simulation with integrated fault injection – showed good results compared to state of the art scheduling algorithms. The simulation results were crosschecked with an analytic approach of Markov Models.

5.2 Future Work

Although the goals of this thesis – regarding the set objectives – were fulfilled, there are several issues and further items that have to be studied in future research. The main open issues are summarized in this concluding section.

More detailed Analyzing of the Scheduling Overhead

In the simulations for evaluating the LB-*Pfair* algorithm and comparing it to other scheduling algorithms, scheduling- and operating-system-overhead was consid-

ered. The execution times for context switching and the execution of the algorithm were assumed to be constant over time and equal for all observed algorithms. However, the execution time is dependent e.g. on the number of currently active tasks, the location of the task in the memory or on potential locking times while accessing common resources. This more detailed system model is a vital part of future research.

Effects of the Hardware Architecture of the Controller

In multicore systems, especially with local and global memories, the allocation of tasks in different memory regions has an impact on the memory access times and therefore on the execution times of the tasks. This has to be considered for task migrations. Another aspect is that, in a multicore system, race conditions while accessing shared resources can occur which further influence the timing. Therefore a more detailed hardware model has to be used. The simulation framework is capable to consider a detailed hardware model which can be applied in a case study.

Case Study

The system architecture and the scheduling algorithm were evaluated using task sets that were derived from real task sets. In general, real-live task sets are more complex. Thus they contain e.g. memory accesses, shared resources, interrupt loads, etc. To benchmark the real performance of the architecture and the algorithm such a task set has to be analyzed and evaluated against state of the art concepts.

Integration and Implementation in a real-time Operating System

To verify that the simulation covers all aspects of the multicore hardware and to determine the exact scheduling and error detecting overhead, the concept of the Safety-Supervisor and the LB-*Pfair* algorithm has to be implemented on a target platform. There, fault injection can be applied by using in system debugging and tracing devices. The integration of LB-*Pfair* in the operating system R^{3}TOS – which was developed in our laboratory LaS^3 – and hardware based fault injection is part of ongoing research in the project ZeloS^3.

Bibliography

[1] C. Berg, "Kohärentes Single-Chip-Multiprocessing," *DESIGN&ELEKTRONIK*, vol. 08, no. WEKA FACHMEDIEN GmbH, 2008.

[2] V. Izosimov, "Scheduling and Optimization of Fault-Tolerant Embedded Systems," Linköpings universitet, 2009.

[3] A. Burns, R. Davis, and S. Punnekkat, "Feasibility Analysis of Fault-Tolerant Real-Time Task Sets," in *Eighth Euromicro Workshop on Real-Time Systems*, 1996, pp. 29–33.

[4] P. Dodd and L. Massengill, "Basic mechanisms and modeling of single-event upset in digital microelectronics," *IEEE Trans. Nucl. Sci.*, vol. 50, no. 3, pp. 583–602, 2003.

[5] N. Ignat, B. Nicolescu, Y. Savaria, and G. Nicolescu, "Soft-error classification and impact analysis on real-time operating systems," in *Proceedings of the Design Automation Test in Europe Conference*, 2006, vol. 1, pp. 1–6.

[6] P. Holman, "On the implementation of Pfair-scheduled multiprocessor systems," University of North Carolina at Chapel Hill, 2004.

[7] S. Funk and V. Nanadur, "LRE-TL: An Optimal Multiprocessor Scheduling Algorithm for Sporadic Task Sets," in *In Proceedings of the 17th International Conference on Real-Time and Network Systems RTNS'2009, Paris, ECC*, 2009, pp. 26–27.

[8] J. Carpenter, S. Funk, P. Holman, A. Srinivasan, J. Anderson, and S. Baruah, "A categorization of real-time multiprocessor scheduling problems and algorithms," Citeseer, 2004.

[9] H. Cho, B. Ravindran, and E. Jensen, "An Optimal Real-Time Scheduling Algorithm for Multiprocessors," *2006 27th IEEE Int. Real-Time Syst. Symp.*, pp. 101–110, 2006.

[10] M. Deubzer and B. Andreas, "Dependability-Betrachtung von Multicore-Scheduling," *Hansa Automot.*, 2010.

[11] "IEC 61508 Functional safety of electrical/electronic/programmable electronic safety-related systems." 2010.

[12] "ISO 26262: road vehicles - functional safety." 2009.

[13] T. Völkl, "A Concept for a Safe Realization of a State Machine in Embedded Automotive Applications," University of Applied Sciences Regensburg, 2007.

[14] P. Raab, S. Kraemer, J. Mottok, H. Meier, and S. Racek, "Safe Software Processing by Concurrent Execution in a Real-Time Operating System," in *Proceedings of 16th International Conference on Applied Electronics*, 2011, pp. 315–319.

[15] G. Buttazzo, *Hard real-time computing systems: predictable scheduling algorithms and applications*. New York: Springer, 2004.

[16] R. Billinton and R. Allan, *Reliability evaluation of engineering systems-Concepts and techniques(Book)*. New York: Plenum Press, 1992.

[17] A. Burns, S. Punnekkat, L. Strigini, and D. R. Wright, "Probabilistic scheduling guarantees for fault-tolerant real-time systems," in *Dependable Computing for Critical Applications 7*, 1999, pp. 361–378.

[18] S. Kraemer, 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*, 2012, pp. 151–156.

[19] N. Leveson, *Safeware: System Safety and Computers*. Amsterdam: Addison-Wesley, 1995.

[20] J. Laprie, "Dependable Computing and Fault Tolerance: Concepts and terminology," in *TwentyFifth International Symposium on FaultTolerant Computing 1995 Highlights from TwentyFive Years (1995)*, 1995, pp. 2–11.

[21] A. Avizienis, J. Laprie, and B. Randell, "Fundamental concepts of dependability," *Tech. Rep. Ser. NEWCASTLE UPON TYNE Comput. Sci.*, vol. 1145, no. 010028, pp. 7–12, 2001.

[22] A. Avizienis and J. Laprie, "Basic concepts and taxonomy of dependable and secure computing," *Ieee Trans. Dependable Secur. Comput.*, vol. 1, no. 1, pp. 11–33, 2004.

[23] N. J. Wang, J. Quek, T. M. Rafacz, and S. J. Patel, "Characterizing the effects of transient faults on a high-performance processor pipeline," *Int. Conf. Dependable Syst. Networks, 2004*, pp. 61–70, 2004.

[24] "IEEE Standard Glossary of Software Engineering Terminology," vol. 121990. IEEE Computer Society, New York, New York, USA, pp. 1 – 84, 1990.

[25] W. Torres-Pomales, "Software fault tolerance: A tutorial," in *NASA Technical Report*, 2000, no. October.

[26] A. Golander, S. Weiss, R. Ronen, and I. Israel, "DDMR: Dynamic and scalable dual modular redundancy with short validation intervals," *Comput. Archit. Lett.*, vol. 7, no. 2, pp. 65–68, 2008.

[27] R. Isermann, *Mechatronische Systeme - Grundlagen*, 2nd ed. Heidelberg: Springer-Verlag, 2007.

[28] H. Beitollahi, S. G. Miremadi, and G. Deconinck, "Fault-Tolerant Earliest-Deadline-First Scheduling Algorithm," *2007 IEEE Int. Parallel Distrib. Process. Symp.*, pp. 1–6, 2007.

[29] X. Jin, "Survey of Scheduling Schemes for Multiprocessor Real Time System," in *Department of Electrical and Computer Engineering, Michigan Technological University*, 2008.

[30] R. I. Davis and A. Burns, "A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems," *Univ. York, Dep. Comput. Sci.*, 2009.

[31] M. Deubzer, U. Margull, M. Niemetz, J. Mottok, and G. Wirrer, "Efficient Scheduling of Reliable Automotive Multi-core Systems with PD2 byWeakening ERfair Task System Requirements," 2010.

[32] T. Wei, P. Mishra, K. Wu, and H. Liang, "Fixed-Priority Allocation and Scheduling for Energy-Efficient Fault Tolerance in Hard," vol. 19, no. 11, pp. 1511–1526, 2008.

[33] A. Persya and T. Nair, "Fault Tolerant Real Time Systems," in *International Conference on Next Generation Software Application*, 2008, pp. 177–180.

[34] K. G. Shin, "A fault-tolerant scheduling algorithm for real-time periodic tasks with possible software faults," *IEEE Trans. Comput.*, vol. 52, no. 3, pp. 362–372, Mar. 2003.

[35] M. Cirinei, E. Bini, G. Lipari, A. Ferrari, S. Superiore, and S. Anna, "A Flexible Scheme for Scheduling Fault-Tolerant Real-Time Tasks on Multiprocessors," *2007 IEEE Int. Parallel Distrib. Process. Symp.*, pp. 2–9, Mar. 2007.

[36] J. Anderson, S. Baruah, and B. Brandenburg, "Multicore operating-system support for mixed criticality," in *Workshop on Mixed Criticality: Roadmap to Evolving UAV Certification*, 2009.

[37] P. Eles, V. Izosimov, P. Pop, and Z. Peng, "Synthesis of Fault-Tolerant Embedded Systems," *2008 Des. Autom. Test Eur.*, pp. 1117–1122, Mar. 2008.

[38] A. Chandra and P. Shenoy, *Hierarchical Scheduling for Symmetric Multiprocessors*, vol. 19, no. 3. 2008, pp. 418–431.

[39] W. Stallings, *Operating Systems – Internals and Design Principles*. Pearson Education, 2009.

[40] S. Ghosh, R. Melhem, D. Mossé, and J. J. S. E. N. Sarma, "Fault-tolerant rate-monotonic scheduling," *Real-Time Syst.*, vol. 181, pp. 149–181, 1998.

[41] R. Davis and A. Burns, "A survey of hard real-time scheduling for multiprocessor systems," *ACM Comput. Surv.*, vol. 1, no. 216682, 2011.

[42] J. Lee, A. Easwaran, I. Shin, and I. Lee, "Zero-laxity based real-time multiprocessor scheduling," *J. Syst. Softw.*, vol. 84, no. 12, pp. 2324–2333, Dec. 2011.

[43] S. K. Lee, "On-line multiprocessor scheduling algorithms for real-time tasks," in *TENCON "94. IEEE Region 10"s Ninth Annual International Conference. Theme: Frontiers of Computer Technology. Proceedings of 1994*, 1994, pp. 607–611 vol.2.

[44] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. a. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, Jun. 1996.

[45] P. Holman and J. Anderson, "Adapting Pfair scheduling for symmetric multiprocessors," *J. Embed. Comput.*, vol. 1, pp. 543–564, 2005.

[46] K. Funaoka, S. Kato, and N. Yamasaki, "Work-Conserving Optimal Real-Time Scheduling on Multiprocessors," in *Proceedings of the 2008 Euromicro Conference on Real-Time Systems*, 2008, pp. 13–22.

[47] S. Kraemer, J. Mottok, and H. Meier, "Osek-basierende Implementierung des LLREF-Scheduling-Algorithmus für eine Dual-Core-Architektur," in *2. Landshuter Symposium Mikrosystemtechnik*, 2010, pp. 343–349.

[48] S. Kraemer, J. Mottok, and H. Meier, "Modifikation des Taskzustandsmodells des LLREF-Schedulers auf einem Dual-Core-Prozessor," in *2nd Embedded Software Engineering Conference*, 2009, pp. 628–636.

[49] D. Zhang, D. Guo, F. Chen, F. Wu, T. Wu, T. Cao, and S. Jin, "TL-plane-based multi-core energy-efficient real-time scheduling algorithm for sporadic tasks," *ACM Trans. Archit. Code Optim.*, vol. 8, no. 4, pp. 1–20, Jan. 2012.

[50] J. Wei-peng, L. Ya-qiu, and W. Qu, *Fault-tolerant task scheduling in multiprocessor systems based on primary-backup scheme*. Ieee, 2010, pp. 670–675.

[51] P. Subramanyan and V. Singh, "Multiplexed redundant execution: A technique for efficient fault tolerance in chip multiprocessors," *Des. Autom. Test Eur. Conf. Exhib.*, pp. 1572–1577, 2010.

[52] M. Skambraks, "A Safety-Related PES for Task-Based Real-Time Execution," Faculty of Electrical and Computer Engineering, Fern Universität Hagen, 2004.

[53] L. Yang, Z. Cui, and X. Li, "A Case Study for Fault Tolerance Oriented Programming in Multi-core Architecture," *2009 11th IEEE Int. Conf. High Perform. Comput. Commun.*, pp. 630–635, 2009.

[54] B. Douglass, *Doing hard time. Developing real-time system with UML, objects, frameworks, and patterns*. Addison-Wesley, 2007.

[55] M. Steindl, J. Mottok, H. Meier, F. Schiller, and M. Fruechtl, "Diskussion des Einsatzes von Safely Embedded Software in FPGA-Architekturen," in *2nd Embedded Software Engineering Congress*, 2009, pp. 655–661.

[56] J. Mottok, T. Zeitler, F. Schiller, and T. Völkl, "A Concept for a Safe Realization of a State Machine in Embedded Automotive Applications," pp. 283–288.

[57] G. a. Reis, J. Chang, and N. Vachharajani, "SWIFT: Software implemented fault tolerance," in *international symposium on Code generation and optimization*, 2005, pp. 243–254.

[58] P. Ulbrich, M. Hoffmann, R. Kapitza, D. Lohmann, W. Schroder-Preikschat, and R. Schmid, "Eliminating Single Points of Failure in Software-Based Redundancy," *2012 Ninth Eur. Dependable Comput. Conf.*, pp. 49–60, May 2012.

[59] A. Shye, S. Member, J. Blomstedt, T. Moseley, V. J. Reddi, and D. a. Connors, "PLR: A software approach to transient fault tolerance for multicore architectures," *IEEE Trans. Dependable Secur. Comput.*, vol. 6, no. 2, pp. 135–148, 2009.

[60] C. Fetzer, U. Schiffel, and M. Süßkraut, "AN-encoding compiler: Building safety-critical systems with commodity hardware," *Comput. Safety, Reliab. Secur.*, vol. 5775, pp. 283–296, 2009.

[61] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer, "Software-Implemented Hardware Error Detection: Costs and Gains," *2010 Third Int. Conf. Dependability*, pp. 51–57, Jul. 2010.

[62] P. Forin, "Vital Coded Microprocessor principles and application for various transit systems," in *IFAC Symposia Series*, 1989, p. 7984.

[63] P. Raab, J. Mottok, and H. Meier, "OSEK-RTOS für Jedermann (Teil 1)," *Embed. Softw. Eng. Rep.*, pp. 15–15, 2009.

[64] S. Baruah and N. Fisher, "Real-Time Scheduling of Sporadic Task Systems When the Number of Distinct Task Types Is Small," *11th IEEE Int. Conf. Embed. Real-Time Comput. Syst. Appl.*, pp. 232–237, 2005.

[65] Timing Architects, "TA Tool Suite." www.timing-architects.com.

[66] P. Raab, "Model-based Reliability Evaluation of Data Processing in HW-fault-tolerant Processor Systems," University of West Bohemia in Pilsen, 2013.

[67] Magdi S. and Moustafa, "Availability of k-out-of-n:g systems with m failure modes," *Microelectron. Reliab.*, pp. 385–386, 1996.

[68] K. D. Thies, *Elementare Einführung in die Wahrscheinlichkeitsrechnung.* Shaker Verlag, 2010.

[69] E. Härtter, *Wahrscheinlichkeitsrechnung, Statistik und mathematische Grundlagen: Begriffe, Definitionen und Formeln.* Vandenhoeck & Ruprecht, 1997.

[70] V. VAIS and S. RACEK, "Experimental Evaluation of Regular Events Occurrence in Continouous-time Markov Models," in *Proceedings of the Eleventh International Conference on Informatics*, 2011, pp. 143–146.

Publications of the author

Conferences

[A1] S. Krämer, J. Mottok and S. Racek. Proportionate Fair based Multicore Scheduling for Fault Tolerant Multicore Real-Time Systems. *In Proceedings of IEEE International Conference on Electrical and Information Technologies 2015*, pages 88 – 93, Marrakech, March 2015

[A2] S. Krämer. Adaption of Proportionate Fair Multicore Scheduling for Safety-Critical Applications, Integrating Fault-Awareness in the Pfair-Scheduling-Algorithm. In *Proceedings of 2015 PESW - The 3rd Prague Embedded Systems Workshop*, Prague, July 2015.

[A3] M. Deubzer, A. Stingl, E. Simsek, S. Krämer and J. Mottok. Performante und zuverlässige Embedded-Multi-Core-System, *In Proceedings of the 7th Embedded Software Engineering Congress*, pages 183 – 194, Sindelfingen, December 2014.

[A4] S. Kraemer, P. Raab, J. Mottok, and S. Racek. Comparison of Enhanced Markov Models and Discrete Event Simulation - for evaluation of probabilistic Faults in safety-critical real-time task sets. In *Proceedings of the 17th Euromicro Conference on Digital System Design*, pages 591 – 598, Verona, August 2014.

[A5] S. Kraemer. Effects of Arbitrary Hardware Faults on Multicore Scheduling in Safety-critical Applications - Evaluation by enhanced Markov Models and discrete event simulation. In *Proceedings of 2014 PESW - The 2nd Prague Embedded Systems Workshop*, Prague, June 2014.

[A6] S. Kraemer, Simsek E., Deubzer M., Stingl A., Hobelsberger M., and J. Mottok. Mut zu Fehlern um die Qualität zu steigern - Fault-Injection zur Steigerung der Zuverlässigkeit. In *Proceedings of the 6th Embedded Software Engineering Congress*, pages 577 – 585, Sindelfingen, December 2013.

[A7] P. Raab, S. Racek, S. Kraemer, and J. Mottok. Data Flow Analysis of Software Executed by Unreliable Hardware. In *Proceedings of the 16th Euromicro Conference on Digital System Design*, pages 243 – 249, Santander, September 2013.

[A8] P. Raab, S. Kraemer, and J. Mottok. Error Model and the Reliability of Arithmetic Operations. In *Proceedings of 2013 IEEE EUROCON - International Conference on Computer as a Tool*, pages 630 – 637, Zagreb, July 2013.

[A9] S. Kraemer, P. Raab, and J. Mottok. Simulationsbasierte Reliability-Analyse - Einflüsse von zufälligen Fehlern auf das Echtzeit-Scheduling. In *Proceedings of the 5th Embedded Software Engineering Congress*, pages 613-622, Sindelfingen, December 2012.

[A10] S. Kraemer, P. Raab, J. Mottok, and S. Racek. Reliability analysis of real-time scheduling by means of stochastic simulation. In *Proceedings of 17th IEEE International Conference on Applied Electronics*, pages 151 – 156, Pilsen, September 2012.

[A11] P. Raab, S. Racek, S. Kraemer, 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, Cesme, September 2012.

[A12] P. Raab, S. Kraemer, 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, Sindelfingen, December 2011.

[A13] P. Raab, S. Kraemer, J. Mottok, H. Meier, and S. Racek. Safe software processing by concurrent execution in a real-time operating system. In *Proceedings of 16th IEEE International Conference on Applied Electronics*, pages 315 – 319, Pilsen, September 2011.

[A14] S. Kraemer, J. Mottok, and H. Meier. Osek-basierende Implementierung des LLREF-Scheduling-Algorithmus für eine Dual-Core-Architektur. In *2. Landshuter Symposium Mikrosystemtechnik*, ISBN 978-3-00-030325-8, pages 343 – 349, Landshut, February 2010.

[A15] 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, Sindelfingen, December 2009.

Journals

[A16] P. Raab, S. Krämer and J. Mottok. Reliability of Data Processing and Fault Compensation in Unreliable Arithmetic Processors. *Accepted for Microprocessors and Microsystems: Embedded Hardware Design (MICPRO)*, http://dx.doi.org/10.1016/j.micpro.2015.07.014, 2015.

[A17] S. Krämer, R. Grave. Systematische Verteilung und Optimierung von Softwarekomponenten in einer Autosar-Multicore-Umgebung, *Elektronik Automotive*, Würzburg, pages 33 – 37 November 2014.

[A18] P. Raab, V. Vavricka, S. Krämer and J. Mottok. Isomorphism between Linear Codes and Arithmetic Codes. In *Computing and Informatics Vol. 33*, pages 721 – 734, October 2014.

[A19] S. Kraemer, J. Mottok, and H. Meier. Multi-Core Scheduling in Embedded Systemen, Teil 1. *Hanser Automotive, München*, (1-2), pages 18 – 22, 2010.

[A20] S. Kraemer, J. Mottok, and H. Meier. Multi-Core Scheduling in Embedded Systemen, Teil 2. *Hanser Automotive, München*, (3-4), pages 23 – 25, 2010.

[A21] S. Kraemer, J. Mottok, and H. Meier. Multi-Core Scheduling in Embedded Systemen, Teil 3. *Hanser Automotive, München*, (5-6), pages 14 – 16, 2010.

Appendix - List of cumulated Articles

P.1. Proportionate Fair based Multicore Scheduling for Fault Tolerant Multicore Real-Time Systems

Authors:	S. Krämer, J. Mottok and S. Racek
Published in:	In *Proceedings of the IEEE International Conference on Electrical and Information Technologies*, Marrakech
Year:	2015

Proportionate Fair based Multicore Scheduling for Fault Tolerant Multicore Real-Time Systems

by Tight Coupling of Error Detection and Scheduling

Stefan Krämer[1,2], Jürgen Mottok[1], Stanislav Racek[2]

[1]OTH Regensburg
Faculty of Electronics and Information Technology
Seybothstr. 2, D-93053 Regensburg, Germany
{stefan.kraemer, juergen.mottok}@oth-regensburg.de

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

Abstract – **In this paper *we* present a scheduling approach for safety critical, fault tolerant, multicore real-time embedded systems. For this kind of systems, not only the correctness of a computed result but also the strict adherence to timing requirements of computation is essential to avoid any kind of damage. To react to unpredictable, arbitrary hardware faults suitable error detection mechanisms have to be applied. The caused error itself and the detection and correction have great impact on the system's timing behavior. To still keep the real-time requirements, the used scheduling algorithm has to ensure maximum flexibility to disturbances of the timing. The group of Proportionate Fair (Pfairness) multicore scheduling algorithms has been proven to create an optimal schedule in polynomial time. The contribution of this paper is a Pfair-based algorithm that uses tight coupling between the error detection mechanisms and the scheduler of the real-time operating system to establish a loop-back connection.**

Fault tolerant systems; safe software processing; real-time operating system; multicore scheduling; discrete event simulation; fault injection; P-Fair scheduling; Stochastic simulation;

I. INTRODUCTION

The usage of multicore processors in embedded real-time systems is gathering pace. Even if for example in the automotive domain the migration towards multicore systems is done quite conservatively. Different legacy systems are combined on one new multicore ECU to reduce the number of ECU and to reduce costs. Because of this trend mixed criticality, embedded systems will become more common due to the fact that multiple functional components are combined into one ECU [15]. Nowadays static partitioning of tasks to cores and manly priority based scheduling approaches are used. Different timing domains and different levels of criticality for each application part are quite common. This means that in these mixed criticality systems QM-tasks (non-safety critical application part) and safety-critical real-time tasks – where not only a wrong computed result, but also a missed deadline for a calculated value can cause damage – have to be scheduled. The safety-critical tasks need additional steps to ensure the systems reliability requirements. QM-tasks do not require additional steps of protection. Multicore systems offer not only an advantage regarding performance, but also have a high potential of increasing the reliability of software intensive embedded system [12]. That is why a software approach can be more flexible and handles the different safety requirements differently. It executes non-critical tasks in a single instance and safety-relevant tasks in e.g. redundant execution on a multicore system. Thus the overall performance and reliability can be increased by using software reliability mechanisms. The requirement to reduce costs of those systems leads to a trend of shifting safety mechanisms like diverse hardware towards software approaches can be observed [10].

A task set scheduled on a multicore with a global scheduling algorithm can in general react more flexible in case of influences by transient faults than singlecore scheduling algorithms [6], because a single scheduling instance is responsible for all computation resources (cores). In case of a detected transient error – e.g. by usage of coded processing – the scheduling unit can distribute the load over all execution resources more easily for the benefit of the safety critical tasks. The impact of timing disturbances of the overall system can be reduced if the scheduling algorithm has a tight connection to the error detection and correction subsystem. With this feedback – without assuming always worst case execution times but using specific error reaction execution times – more efficient schedules can be created and the disturbance of the timing of all tasks (including QM-tasks) can be reduced to a minimum.

Therefore we present an extension for fluid based schedules like Pfair that uses this connection to the error handling subsystem to optimize the system timing behavior in case of the occurrence of arbitrary hardware faults.

This paper is structured as follows. The first section gives a short overview of the general considered system architecture, including the impact of this architecture on the reliability of the system.

In the second section, the Loop-back (LB) - Pfair

scheduling algorithm is introduced. The evaluation of the LB-Pfair is done by discrete event simulation that offers the possibility of a holistic approach for reliability analysis combined with schedulability analysis of complex safety-critical multicore real-time systems. A case-study with a representative automotive task set is given in section V. Finally Section VI gives a conclusion and outlook for further research.

II. Considered System Architecture

This section will describe the two safety mechanisms considered in this paper, coded processing and symmetric redundant execution [2], [9] and their combined application. The principal modelling of a safety task is shown. In case an error was detected the corresponding task instances are re-executed. The operating system architecture with loop-back connection from the error detecting subsystem to the operating system's scheduler is depicted.

A. Considered fault handling startegies

1) Coded processing

Coded processing is the protection of calculations and their results during operations in an arithmetic unit by means of error detection codes. Important of error detecting codes are the so-called arithmetic codes (AN-codes) that are based on ordinary algebra like addition and multiplication [1]. Forin made the first use of coded processing in a real application [13], [11], [17].

2) Symetric redundant execution

The duplication of components is a common method to increase the reliability of systems. The same task is duplicated into *n* task instances, which are executed one after another on a singlecore system (time redundancy) or on different cores on a multicore system (space redundancy). The computation results of the *n* instances are compared and evaluated after complete execution by the voter and possible single faults can be detected [12], [16].

3) Combined approach

According to the needed Safety Integrity Level (SIL) both approaches can be combined. By coding an error can be detected in each individual instance. For resolving the error re-execution is only necessary if both redundant and coded instances have detected an error at the same time.

B. Safety-Task Design

1) Coded Task Processing

The scheduler and within that the LB-Pfair algorithm is directly called by a system call, if the underlying coded processing library detects an error. The check for error occurrence at coding processing is directly integrated in the task execution. The time stamp and the additional execution time are reported to the LB-Pfair algorithm.

2) Symmetric redundant task execution

For employing symmetric redundant execution in a multicore system some effort has to be made to achieve the necessary synchronization between the two task instances using standard scheduling algorithms. The synchronisation in case of LB-Pfair scheduling is included in the actual scheduling algorithm. Paired safety-task instances are controlled by a task allocation constraint. The first task waits for the second instance to reach the synchronisation point. The comparison of the computation results is then be done in a global voter.
This could be integrated in a safety supervisor [7]. After comparison the LB-Fair scheduler gets the information if additional execution time for re executing the instances is necessary and can consider this for computation of the next scheduling decision. In the following a combined approach with coded processing and redundant execution is assumed.

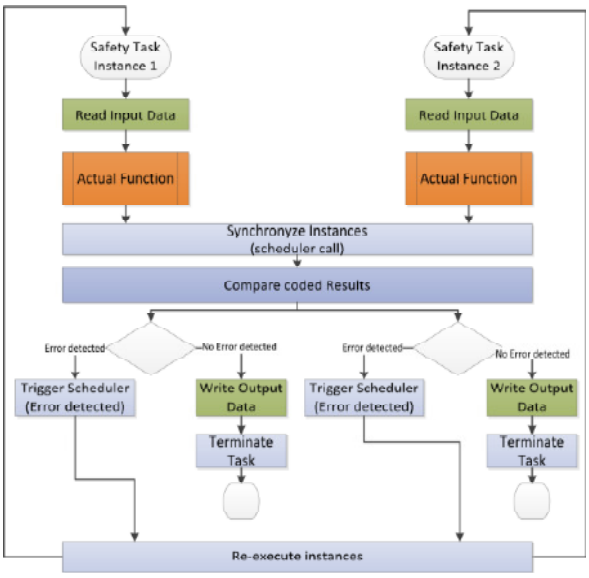

Figure 1: Modeling of symmetric redundant task execution of two coded safety-critical task execution with loop-back connection (trigger Scheduler) to the LB-Pfair scheduling algorithm.

C. Operating System architecture

For considering overhead caused by error detection and correction, a tight coupling between the error handling subsystem and the LB-Pfair scheduling algorithm executing within the operating system's scheduler has to be established.

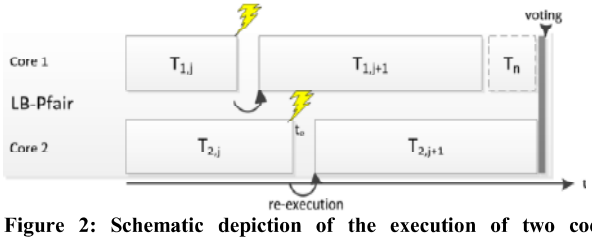

Figure 2: Schematic depiction of the execution of two coded instances of a safety task, scheduled by the LB-Pfair algorithm. For impact on the scheduling see Figure 4 at time t_e.

Figure 3 shows the principle concept. The error detection – e.g. by coded processing – is encapsulated in a library function, such that the user can incorporate this functionality transparently [7]. The so called safety supervisor is the connection between the application, the error handling subsystem and the scheduler. The safety supervisor provides information of time and position of error detection. With that information the LB-Pfair algorithm can predict – at the moment of error detection (t_e) – how much additional execution time (c_i) is needed for this error correction and can change the schedule dynamically.

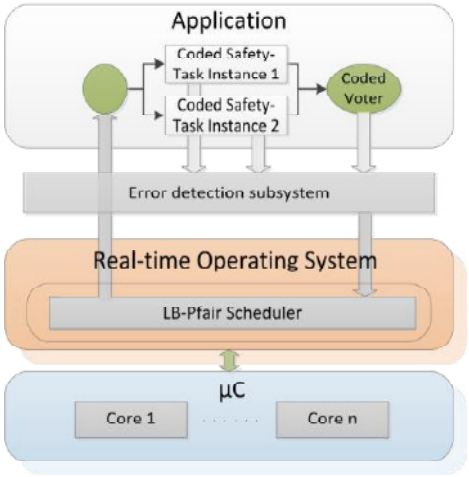

Figure 3: Basic architecture of an operating system using LB-Pfair scheduling. The error detection subsystem is connected via a feedback-loop to the LB-Pfair scheduler.

III. LB-PFAIR SCHEDULING

The basic principles of the Pfair scheduling algorithm are briefly summarized. In the second part of our contribution of the discontinuous fluid schedule as main property of the LB-Pfair algorithm is introduced.

A. *Pfair scheduling*

Pfair [14] based multicore real-time scheduling for periodic tasks is different in one aspect compared to other real-time scheduling approaches. The task instances are executed at a steady rate. Meaning they execute at constant speed in a way that they finish right within their deadline derived from their period. This execution along this steady rate is called fluid schedule. As the execution rate of a core cannot be varied, the execution of a fluid schedule is approximated by dividing the task execution time in so called quants. That are either executed or not. The division into quants – by bypassing the bin-packing-problem – is one of the reasons, why a Pfair schedule can be calculated online in polynomial time for periodic task sets [14].

A periodic task instance T_i is defined by its execution time e_i and period (p_i). To satisfy the requirement for approximate an uniform execution rate each subtask (quant) has to be executed in a certain window. Different subtasks may execute on different cores but they may not execute in parallel on different cores.

There are several extensions and modifications of the original Pfair algorithm, such as PF, PD, PD² or ER (early release) [19]. The latter is suitable for fault tolerant scheduling because it does not misspend execution time by executing tasks, if a free execution resource R is available.

The main scheduling criteria, besides some tie breaking rules, is referred as the task weight wt_i that is the criteria for calculating the pseudo deadline $d(T_{i,j})$ for each individual subtask (j) of T_i. It is defined by:

$$wt_i = \frac{e_i}{p_i} \qquad (1)$$

Informally the weight is the rate at which the task should be executed. The pseudo deadline follows:

$$d(T_{i,j}) = \left\lceil j \frac{1}{wt_i} \right\rceil. \qquad (2)$$

Pfair is proven to create feasible schedules for k tasks on M execution resources, if:

$$\sum_{i=0}^{i<k} \frac{e_i}{P_i} \leq M. \qquad (3)$$

B. *Extension of Pfair Scheduling – LB-Pfair*

The herein introduced extension – the discontinuous fluid schedule (see Figure 4) – is applicable for many fluid schedule based algorithms.

Besides the actual extension of the scheduling algorithm LB-Pfair includes execution allocation constrains and subtask-synchronization. Same subtask instances of coupled safety critical tasks – that are executed in space redundant manner – are not allowed to execute on the same execution resource R. For the couple T_i, T_{i+1} follows:

$$R(T_{i,j}) \neq R(T_{i+1,j}) \qquad (4)$$

To avoid delays for comparing voting results the coupled task instances have to be synchronized. This criterion is realized indirectly by considering the same execution rate and initial task weight. Therefore on an equally balanced system this criterion is fulfilled.

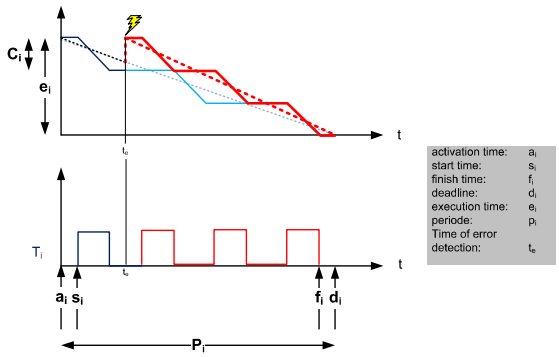

Figure 4: Discontinuous fluid schedule, representing erronous task execution, with error detection at time t_e.

With the proposed discontinuous fluid schedule the task weight wt_i is no longer constant over system lifetime (In original Pfair algorithm it is considered as constant over time, but nevertheless the real execution time on a target may fluctuate.). It is now additionally dependent on occurrence and detection of an error by the error handling subsystem and the loop-back connection to the algorithm. For correcting the error the task instance is re-executed. This additional execution time c_i results in a temporally increased task weight wt_i*.

This in turn leads to a temporarily higher dynamic priority of the faulty task instance. The ER-extension for Pfair-algorithms is applied for LB-Pfair scheduling. Because of its non-work-conserving properties it is suitable for safety-critical task to ensure free resource capacity at the end of a period in case of error occurrence. If the safety supervisor layer detects an error the scheduler is informed about the time of error occurrence t_e by the loop-back connection. The error correcting time c_i is defined by:

$$c_i = t_e - s_i \quad (5)$$

The dynamic task weight wt_i* follows:

$$wt_i^* = \frac{e_i + c_i}{p_i} \quad (6)$$

wt_i^* could be derived from the error rate λ affecting the system.

The schedule criteria – the local deadline – can now be defined by:

$$d^*(T_{i,j}) = \left\lfloor j \frac{1}{wt_i^*} \right\rfloor \quad (7)$$

According to equations (3) and (6) a feasible schedule can be calculated, if the following equation is satisfied for every point in time.

$$\sum_{i=0}^{i<k} wt_i^* \leq M\big|_{\forall t}. \quad (8)$$

Because of the complexity of task systems and time of error occurrence as well as the considered execution time, wt_i^* can hardly be determined in an analytical way for an arbitrary system. Therefore evaluation is done by a discrete event simulation approach (See section IV).

IV. Evaluation by Simulation

For the simulation based evaluation of the LB-Pfair scheduling algorithm a multi seed discrete event simulation is used. The discrete event simulation is used to model the variance of task execution durations and the general feasibility of a schedule in a multicore real-time embedded system. The reliability indices are determined by probabilistic fault injection of transient faults. To cover a wide range of the design space each simulation run is executed multiple times. In order to establish a reasonable confidence in the result and to gain a high precision, the simulated time span of one single run was set to 5000 seconds that were executed 1000 times each.

The modeling of the task execution is very close to real task execution on a target (see Figure 1 and Figure 3). In this example, at the beginning input data is read, then the safeguarded processing ("Actual Function") is executed and the result is checked. If no error occurred the task terminates normally and writes to the output. Otherwise, if an error was detected, the scheduler is called and informed about the additional execution time in case of LB-Pfair scheduling.

A. Simulation Model

The real system behavior can be depicted quite accurately starting at the level of instructions up to the modeling of the application with tasks and operating systems by usage of Monte-Carlo based discrete event simulation framework. The used model consists of a hardware part defining the processor, which itself consists of multiple cores. These cores execute the instructions defined in the application part of the model. The execution time of these instructions is defined by the clock of the core and the amount of instructions the core can execute per tick.

A next part of the model describes the operating system architecture, including the LB-Pfair algorithm.

In the application part of the model the different task and interrupt service routines are defined. A task itself contains several function calls and has the capability to model conditional branching of the execution flow. Functions are assembled by instruction blocks, they define the amount of instructions (constant number, or defined by a distribution) that have to be executed by the core.

Furthermore the stimulation of the simulated system has to be described. This stimulation includes the activation of tasks and ISRs or the triggering of the fault injection. These trigger mechanisms can either be configured periodically or also varied by a distribution [18].

B. Simulation input parameters

During the simulated task executions faults are injected via system stimulation. The fault injection rate is varied over simulation time in a certain range (see section V). The runtimes of the tasks are defined by uniform distribution to model the actual variation of task execution on a real target.

C. Simualtion output parameters

By tracing the different events of the simulation a great variety of timing and reliability metrics can be evaluated. The simulation is used to simulate not only the standard reliability indices but also to simulate the real-time characteristics of an embedded system. In this paper we focus on the response time as an evaluation criterion.

V. Case Study Automotive Task Set

A. Task set for evalutation

The LB-Pfair algorithm is evaluated using a simplified automotive-like task set. The task set consists of one safety-critical application executed with coded processing and redundant execution. Four further non-safety-critical applications are executing on the ECU and are causing additional load. The tasks are located in different timing domains. This represents the different applications types, running on the target. A symmetric multicore processor is used as target device.

Standard OSEK-algorithm with partitioned priority based scheduling, Pfair and LB-Pfair are compared in this study. The quant-size of the latter is set to 0.1ms. Safety-critical tasks cannot be preempted by QM-tasks in the OSEK-scheduler. The safety-critical task has the highest priority in the task-set, despite a short running task (QM-task 4). For OSEK-scheduling the tasks are statically allocated to cores. The priorities are given in Table 1. For Pfair algorithms the priorities are dynamically calculated by the weight and in case of LB-Pfair additionally depending on the presence of a detected error.

Table 1: Configuration of the simplified task set

	Execution time [ms]	Period/ Deadline [ms]	OSEK Core	OSEK Priority
QM-task 1	10 – 20	100	1	30
QM-task 2	10 – 20	100	2	30
QM-task 3	90 - 110	500	2	20

QM-task 4	0.1 – 0.2	1	1	50
Safty-task Instnace 1	75	1000	1	40
Safty-task Instnace 2	75	1000	2	40

1) Error detection

According to Figure 1 error detection for the two safety-critical tasks is considered by dual modular redundancy and coded processing. That means that data is processed in the coded domain by two different instances (different coding) in space redundancy on the dualcore. In case an error is detected by the comparator the two task instances are activated again and therefore re-executed.

2) Fault injection

The simulation is configured to simulate bursts of faults and consequently covers heavily faulty situations, e.g. caused by fluctuations of the power supply. Thus multiple faults are quite likely. The injected fault rate (λ) is varied in a range from 1ms to 200ms, which means that there will be at least on injected fault during the execution period of the safety-critical task. Faults are injected on each core individually. Thus faults would affect safety and QM-tasks. However error detection and correction is only implemented for safety-task.

B. Evaluation of simulation results

1) Response time metrics

Table 2 and Table 3 summarize the results of the discrete event simulation runs. As mentioned, the individual simulations were executed 1000 times each with a simulation time of 5000 seconds and different seeds for the Monte Carlo-based simulation.

In general it can be seen that there is a difference between the response times for the QM-tasks with LB-Pfair scheduling and OSEK scheduling. The average response times (see Table 3) are better for QM tasks in Pfair and LB-Pfair scheduling, because of dynamic task allocation during runtime, which allows a more evenly utilization of the cores. For the important safety-critical tasks the average response time for LB-Pfair is better compared to OSEK and Pfair. The bad result for standard Pfair is an effect of the not considered additional execution time in case of error handling and the originally relative low weight (because of the long period) of the safety critical task. Hence a low priority for this task is the consequence. It can be seen (see Table 2) that for safety-critical tasks the maximum response times are almost half as long as for LB-Pfair scheduling compared to OSEK scheduling. LB-Pfair scheduling recognizes the higher execution time requirements of safety-critical tasks in case of an error and therefore increases their local scheduling priority. A side effect is the higher maximum response time at LB-Pfair scheduling for QM-tasks compared to the other two algorithms.

In average the response times of the safety-critical task are lower compared to the standard algorithms. The maximum response time of the relevant safety-critical tasks ($t_{response,\ max} = 3421\text{ms}$) can be lowered significantly compared to OSEK ($t_{response,\ max} = 6024\text{ms}$) and original Pfair scheduling.

The response times of the safety-critical tasks at LB-Pfair scheduling have a standard deviation of 480ms over all multi-seed simulation runs. At OSEK scheduling the standard deviation is 949ms. This shows that LB-Pfair scheduling produces results with less outlier.

Table 2: Maximum response time for OSEK, Pfair and LB-Pfair scheduling for task set of Table 1.

	OSEK	Pfair	LB-Pfair
QM 1[ms]	6042.94	48.47	3135.63
QM 2[ms]	184.78	47.99	3133.83
QM 3[ms]	2381.29	176.55	3040.67
QM 4[ms]	0.19	0.69	303.13
safety-task instance 1 [ms]	6023.76	53091.50	3420.50
safety-task instance 2 [ms]	6023.60	53088.55	3421.00

Table 3: Average response time) for OSEK, Pfair and LB-Pfair scheduling for task set of Table 1.

	OSEK	Pfair	LB-Pfair
QM 1[ms]	142.53	21.087	71.39
QM 2[ms]	71.66	21.11	71.51
QM 3[ms]	551.73	136.58	255.00
QM 4[ms]	0.14	0.14	0.75
safety-task instance 1 [ms]	803.82	7069.14	667.70
safety-task instance 2[ms]	803.65	7069.85	667.95

2) Deadlines and response time distribution

Table 4 shows the percentage of task instances of the safety-critical tasks that violate their deadline requirement. Task instances, which could not be started as the previous instance is still executing – due to longer execution time resulting from error correction – are counted as deadline violation.

Table 4: Quota of deadline violations for OSEK, Pfair and LB-Pfair scheduling for task set of Table 1.

	OSEK	Pfair	LB-Pfair
deadline violations safety [%]	53.55	99.05	29.54

Figure 5 and Figure 6 depict the distribution of response times for OSEK and LB-Pfair scheduling. The step in the aggregated frequency graph represents task instances that are not executed. Low response times do not occur at LB-Pfair scheduling. This results from the fair scheduling policy with early release. Thus QM-tasks are executed in a higher quota compared to the OSEK scheduling, where the safety-critical task can be executed right at the beginning of the period because of the higher task priority. But the deviation of response times and the number of longer response times for OSEK scheduling are higher compared to LB-Fair scheduling (see also Table 2 and Table 3). The extensions by LB-Pfair algorithm leads to a better performance regarding response times and deadline violations.

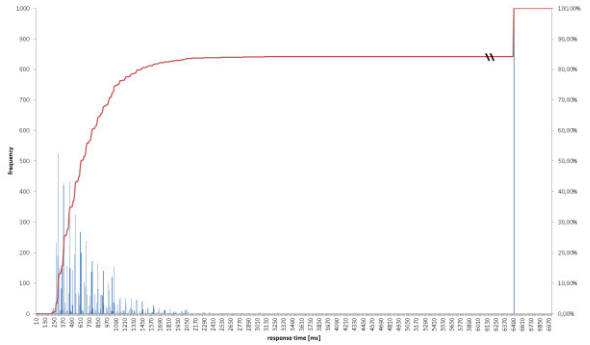

Figure 5: Response time histogram (frequency, blue and aggregated values, red) for the safety-critical tasks in LB-Pfair scheduling. Not activated task instances – because of overload condition – are depicted by the step of the aggregated values at the right hand side.

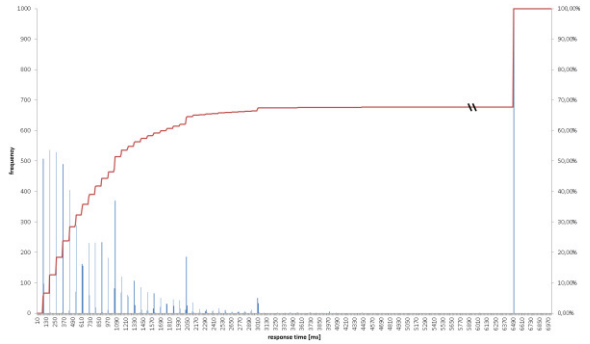

Figure 6: Response time histogram (frequency, blue and aggregated values, red) for the safety-critical tasks in OSEK scheduling. Not activated task instances –- because of overload condition – are depicted by the step of the aggregated values at the right hand side.

VI. CONCLUSION AND FURTHER STEPS

It has been shown that with the extension of the Pfair scheduling by the discontinuous fluid schedule, and the execution time loop-back from the error handling subsystem, the important response time metric of the safety-critical tasks can be lowered compared to other scheduling approaches. The LB-Pfair algorithm achieves this by dynamic priority reassignment for the benefit of safety-critical tasks. At the same it maintains the advantage of global dynamic algorithms for scheduling complex task sets dynamically on a multicore. This has been shown with a case study of a simplified task set applicable for the automotive domain.

For the LB-Pfair scheduling the early release scheduling policy was chosen. When applying this policy to all tasks the result is a low percentage of short response times compared to the OSEK algorithm. A further extension here is selective ER, which will be topic of future research.

Other points of research are the evaluation of more complex systems and the evaluation of the influence of different applied fault rates to determine a maximum fault rate the system can tolerate. Furthermore the determination of the optimal quant-size and its relation to the synchronization granularity will be part of future research. The basic idea of the discontinuous fluid schedule will be ported to other fluid schedule based algorithms.

REFERENCES

[1] P. Raab, S. Kraemer, and J. Mottok. 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.

[2] B. Douglass. Doing hard time. Developing real-time system with UML, objects, frameworks, and patterns. Addison-Wesley, 2007.

[3] Mottok, F. Schiller, Th. Völkl, and Th. Zeitler. A concept for a safe realization of a state machine in embedded automotive applications. In *Proceedings of the 26th Safecomp Conference*, ISBN 978-3-540-75100-7, pages 283-288, 2007.

[4] L. Yang, Z. Cui and X. Li. A case study for fault tolerance oriented programming in multi-core architecture. IEEE computer society, pages 630 – 635, 2009.

[5] E. Beckschulze, F. Salewski, T. Siegbert and S. Kowalewski. Fault handling approaches on dual-core microcontrollers in safety-critical automotive applications. CCIS 17, pages 82 – 92, 2008.

[6] M. Deubzer, J. Mottok, and A. Baerwald. Dependability-Betrachtung von Multicore-Scheduling. *HANSER Automotive*, November 2010.

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

[8] G. Buttazzo. Hard real-time computing systems – Predictable Scheduling Algorithms and Applications. Springer, 2004

[9] N. Leveson. Safeware: System Safety and Computers. Addison-Wesley, 1995.

[10] A. Burns, R. Davis, and S. Punnekkat. Feasibility Analysis of Fault-Tolerant Real-time Task Sets. In Proceedings of EURWRTS, 1996

[11] P. Raab, S. Racek, S. Kraemer, 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

[12] S. Kraemer, 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

[13] P. Forin, "Vital Coded Microprocessor principles and application for various transit systems," in IFAC Symposia Series, 1989, pages 79-84.

[14] S. K. Baruah, N. K. Cohen, C. G. Plaxton, and D. a. Varvel, "Proportionate progress: A notion of fairness in resource allocation," *Algorithmica*, vol. 15, no. 6, pp. 600–625, Jun. 1996.

[15] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson, "RTOS Support for Multicore Mixed-Criticality Systems," *2012 IEEE 18th Real Time Embed. Technol. Appl. Symp.*, pp. 197–208, Apr. 2012.

[16] M. Hoffmann and P. Ulbrich, "A Practitioner's Guide to Software-Based Soft-Error Mitigation Using AN-Codes," *(HASE), 2014 IEEE*, no. 0704, 2014.

[17] M. Hoffmann, C. Borchert, and C. Dietrich, "Effectiveness of Fault Detection Mechanisms in Static and Dynamic Operating System Designs," *cs.fau.de*.

[18] S. Krämer, P. Raab, J. Mottok, and S. Racek, "Comparison of Enhanced Markov Models and Discrete Event Simulation," in *2014 Euromicro Conference on Digital System Design (DSD)*, 2014.

[19] R. I. Davis and A. Burns, "A survey of hard real-time scheduling algorithms and schedulability analysis techniques for multiprocessor systems," *Univ. York, Dep. Comput. Sci.*, 2009.

P.2. Reliability of Task Execution during Safe Software Processing

Authors:	P. Raab, S. Racek, S. Krämer, and J. Mottok
Published in:	In *Proceedings of the 15th Euromicro Conference on Digital System Design*
Year:	2012

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

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

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:

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

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).

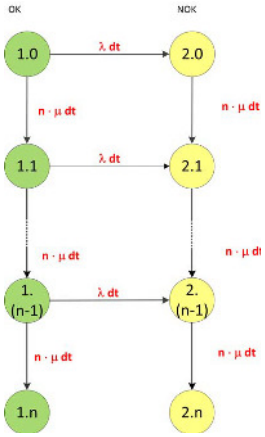

Figure 1. Enhanced Markov model that simulates the runtime of a single task with several stages. The total execution time with $t_{Task} = \frac{1}{\mu}$ is divided into n segments. The transition rate of every stage is therefore $n \cdot \mu$. The task terminates after t_{Task} either without any faults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

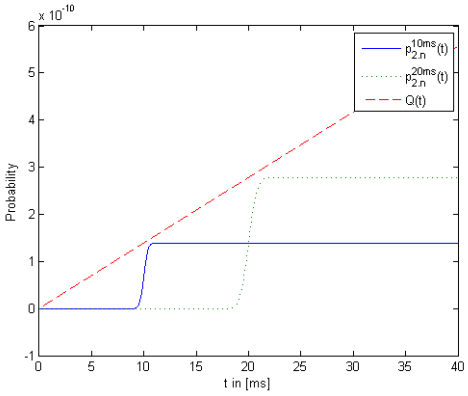

Figure 2. State 2.n represents the failure state which the system enters after termination of the task. The figure shows the probability of the failure state with two different execution times $t_1 = 10ms$ and $t_2 = 20ms$. The steady-state probability of both curves equals the probability of the failure 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.

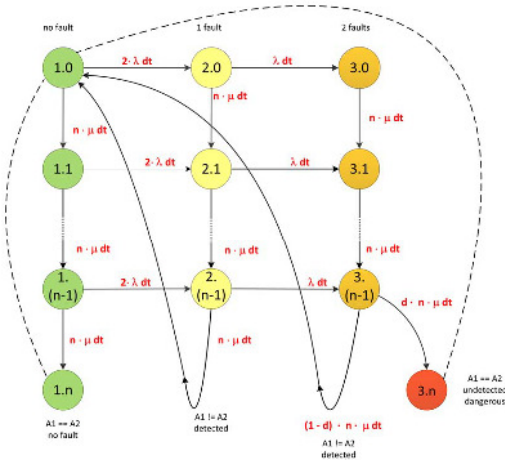

Figure 3. The basic Markov model is that of a two-component, non-repairable system with three states. It is enhanced by n stages to model the execution time of the task. The state 1.n represents the condition when the task terminates without any faults. If a faulty output is detected, then the task is 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.

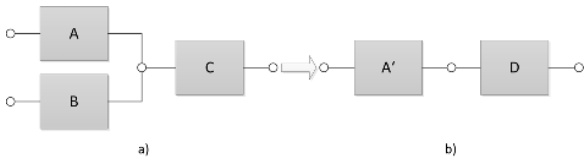

Figure 4. Change of the reliability model: The parallel structure of a duplicated task a) is replaced by the single structure of a coded task b).

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.

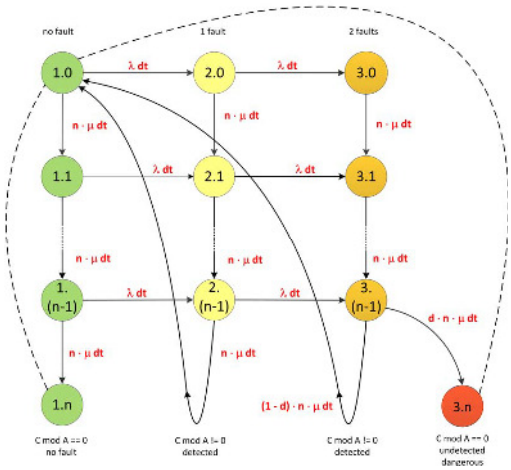

Figure 5. With the given code, all single faults are detectable. This is equivalent to a two-component, non-repairable system which detects all single faults. Because of the fact that there is only one instance of a task, the probability of a single fault is half compared to the duplicated task, whose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

P.3. Data Flow Analysis of Software Executed by Unreliable Hardware

Authors:	P. Raab, S. Racek, S. Krämer, and J. Mottok
Published in:	In *Proceedings of the 16th Euromicro Conference on Digital System Design*
Year:	2013

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.

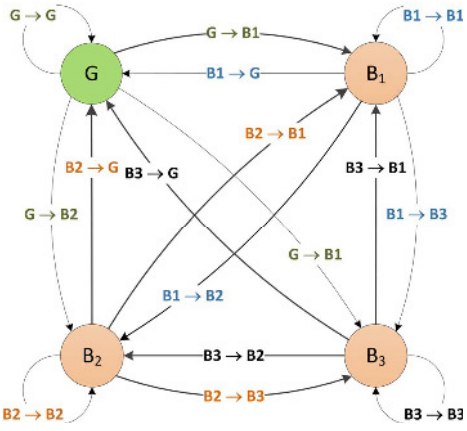

Figure 1. Discrete Markov process modelling the memory of an adder [23].

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.

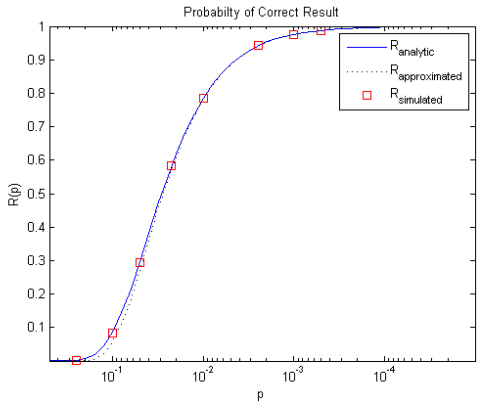

Figure 2. Effect of fault compensation in a given data flow model.

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.

P.4. Reliability of Data Processing and Fault Compensation in Unreliable Arithmetic Processors

Authors:	Raab, P. and Krämer, S. and Mottok, J.
Published in:	Accepted to Microprocessors and Microsystems: Embedded Hardware Design (MICPRO)
Year:	August, 2015

Microprocessors and Microsystems xxx (2015) xxx–xxx

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Reliability of data processing and fault compensation in unreliable arithmetic processors

Peter Raab [a,*], Stefan Krämer [b], Jürgen Mottok [b]

[a] Faculty of Mechanical Engineering and Automotive Technology, Hochschule Coburg, Friedrich-Streib-Straße 2, 96450 Coburg, Germany
[b] Faculty of Electronics and Information Technology, OTH-Regensburg, Seybothstr. 2, D-93953 Regensburg, Germany

ARTICLE INFO

Article history:
Available online xxxx

Keywords:
Data flow
Error probability
Fault compensation
Software-implemented hardware fault tolerance (SIHFT)
Stochastic simulation
Simulated fault injection

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 softwarés 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.

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 soft-errors (*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. However, diverse data processing does not detect all faults. There is still a residual probability for non-detection. This residual error probability is a crucial metric for the evaluation of error codes. But in contrast to error detecting codes used in transmission and storage systems [21,24,25], where sufficient error models are available for determining this probability, there are no comparable models for arithmetic operations in a processor system [13]. But by means of such error models, the analytical evaluation of faulty outcomes in arithmetic instructions is then possible opposed to state-of-the-art experimental methods like fault injection. Indeed, the higher complexity of software as a set of interdependent arithmetic operations results in further effects of fault compensation. This effect of fault compensation is evaluated by fault injection in a stochastic simulation framework based on the Monte Carlo method. There a detailed microcontroller model is the basis for the analysis of fault compensation.

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

* Corresponding author.
E-mail addresses: peter-j.raab@hs-coburg.de (P. Raab), stefan.kraemer@oth-regensburg.de (S. Krämer), juergen.mottok@oth-regensburg.de (J. Mottok).

http://dx.doi.org/10.1016/j.micpro.2015.07.014
0141-9331/

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 detailed analysis of fault compensation by a simulation approach is depicted in Section 5. The paper proceeds with a discussion of the results in Section 6 and ends with a conclusion for further works in Section 7.

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 (Fig. 1). With this view, coded processing is related to channel coding as a part of the coding theory.

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 [7–12]. 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 (Eq. (1)).

$$\mathbb{C}_{AN} := \{A \cdot X | A, X \in \mathbb{Z}\} \tag{1}$$

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 10,000 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 variablés 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

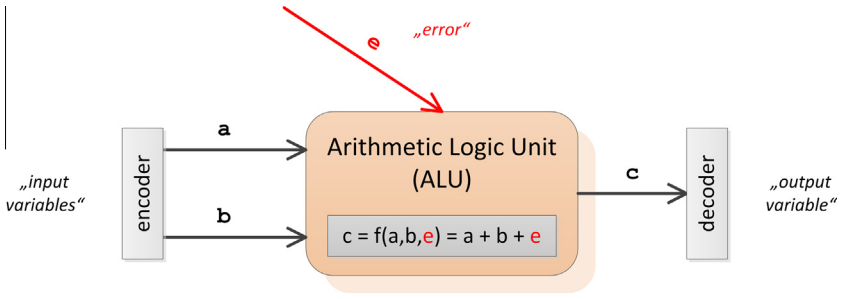

Fig. 1. Simplified processor model: The arithmetic unit in a processor represents a channel with respect to arbitrary (transient or permanent) faults which change the value of a result during the execution of an operation.

Please cite this article in press as: P. Raab et al., Reliability of data processing and fault compensation in unreliable arithmetic processors, Microprocess. Microsyst. (2015), http://dx.doi.org/10.1016/j.micpro.2015.07.014

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 [16–18] 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 Fig. 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.

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 (Fig. 3).

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 (Fig. 4).

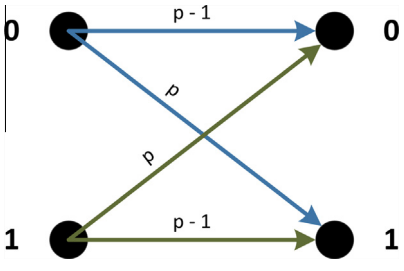

Fig. 2. The binary symmetric channel model describes the probability p that a single bit changes its value or remains unchanged $(1 - p)$.

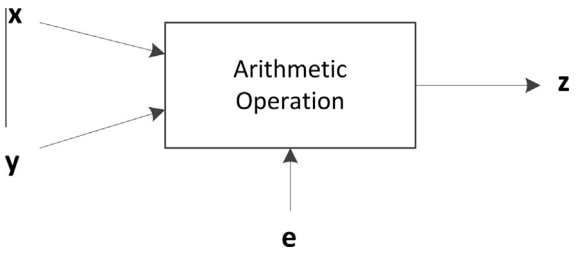

Fig. 3. An arithmetic operation usually processes at least one input and generates an output as a function of the inputs and the arithmetic error ($z = f(x, y, e)$).

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 Fig. 5 to model the faulty behavior caused by the dependencies between the carry-bits and the following digits.

Fig. 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. The data flow of a processor task is determined by e.g. its runtime among others. Somewhen during the task execution, the hardware arbitrary injects a fault. The concrete implementation of the fault detection mechanism induces a latency between the fault occurrence and the fault detection and the following fault recovery (see Fig. 6).

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). In Fig. 6, there are two different events that must be combined for the desired model of faulty task execution. The fault occurrence which follows the exponential distribution is delayed until the task is terminating. We know a similar behavior described by the *queueing theory* which combines two events with different distributions. In this context, 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 tasks runtime better. The fact that the Erlang distribution is a composition of exponential distributions the Markov model, which describes transitions with constant fault rate, can be extended to multi-stages representing the Erlang

Please cite this article in press as: P. Raab et al., Reliability of data processing and fault compensation in unreliable arithmetic processors, Microprocess. Microsyst. (2015), http://dx.doi.org/10.1016/j.micpro.2015.07.014

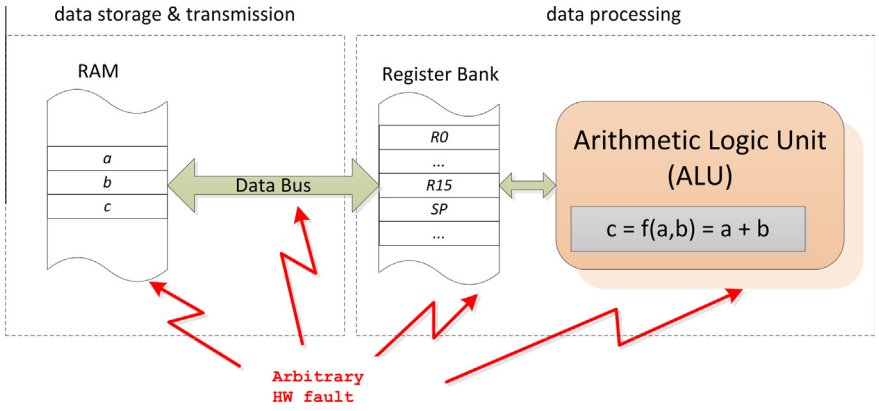

Fig. 4. Simplified fault model of a processor [26].

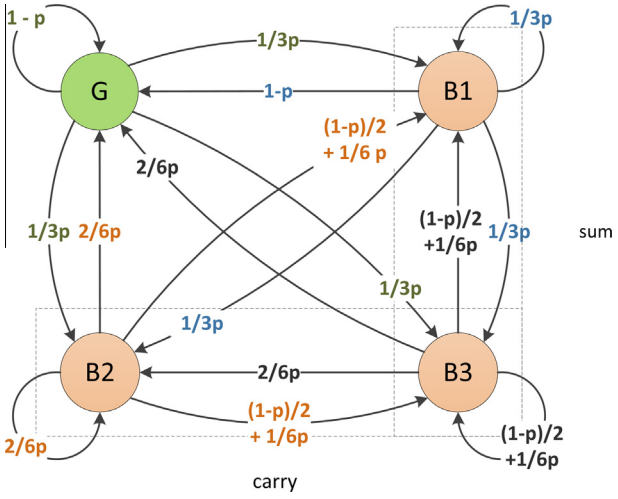

Fig. 5. Discrete Markov chain which models the iterative process of a ripple-carry-adder [20].

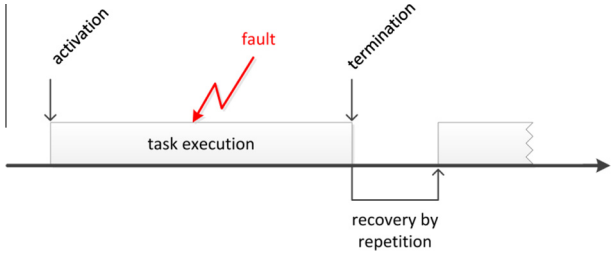

Fig. 6. Latency between fault occurrence and fault detection.

distribution (Fig. 7). 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 Fig. 7, this repetition is marked by the transitions back to the initial state 1.0.

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. The finite number of vertical stages makes a quantization of time which leads to some uncertainty of task termination. In general, the task terminates after a constant number of instructions whose time is well defined. With the presented model, this time is approximated by the Erlang distribution which allows deviations in the tasks runtime depending on this number of stages. The challenge is then to configure the Markov model in a way that it approximates the real distribution of the runtime.

Furthermore, the vertical stages can also be considered in a descriptive way as a time interval which corresponds to a single or a set of instructions. So, it is possible to map all instructions into the Markov model with an individual fault probability (horizontal transitions) and runtime (vertical transitions). This is important in case of fault detection within a task and not only at the end. In Fig. 7, there would be additional transitions from this state to the state 1.0.

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.

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

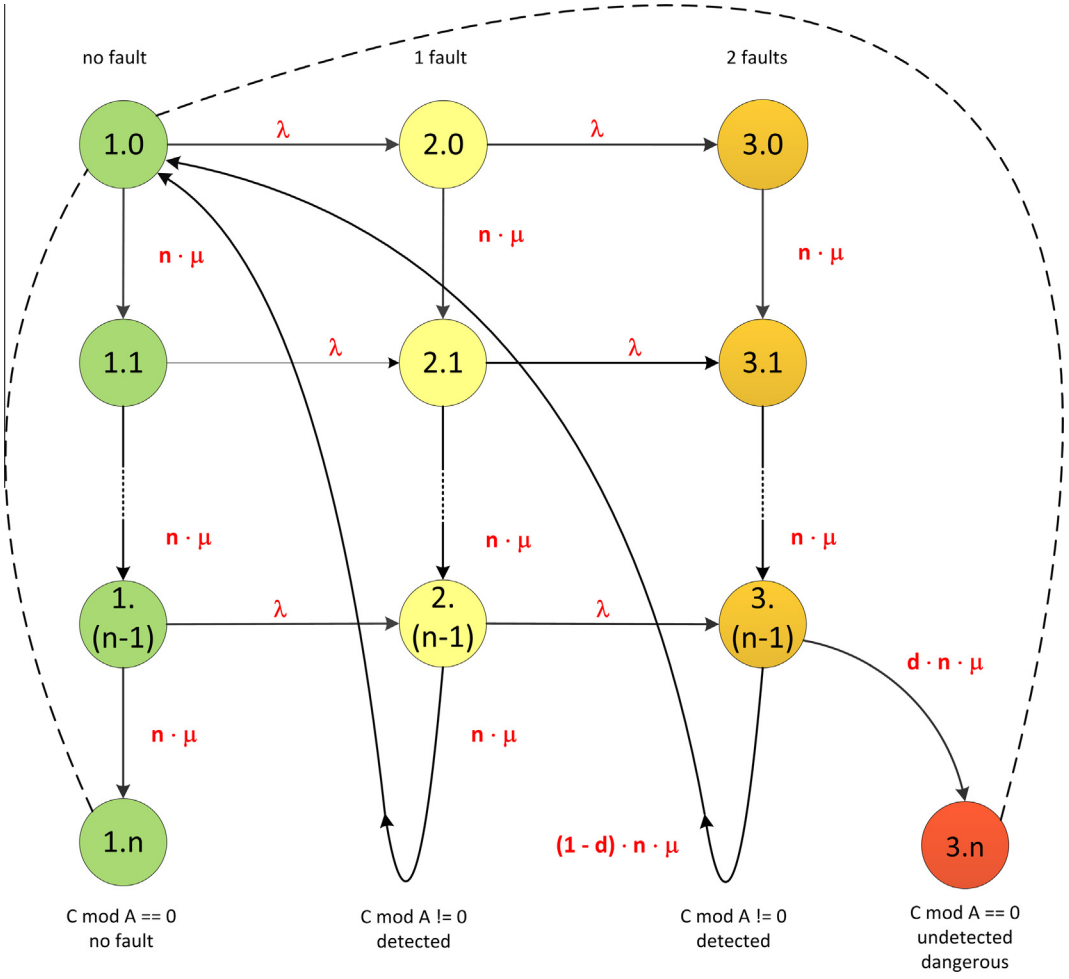

Fig. 7. Extended Markov model for a coded data flow [27].

```
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.

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 Eq. (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 Eq. (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 Eq. (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

$$R_e = r_c \cdot \prod_k R_k. \tag{10}$$

Please cite this article in press as: P. Raab et al., Reliability of data processing and fault compensation in unreliable arithmetic processors, Microprocess. Microsyst. (2015), http://dx.doi.org/10.1016/j.micpro.2015.07.014

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 a sequence of faulty instructions, the error mask after the k-th instruction is the superposition of all previous error masks by

$$E = E_1 \oplus E_2 \cdots \oplus E_k. \tag{12}$$

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 Eq. (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, Eq. (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 $x = pk$. The Eq. (14) derives to

$$p_{E0} = \cosh(pk) \cdot e^{-pk} \tag{17}$$

as the probability for a single correct bit. The effective reliability R_e is the probability that all n bits in the resulted processor word are correct:

$$R_e = p_{E0}^n = \cosh(pk)^n \cdot e^{-pkn} \tag{18}$$

The term e^{-pkn} in Eq. (18) is identical to the binomial expression $(1-p)^{kn}$ for small pk and corresponds to the basic reliability R_0 of Eq. (9) (see also Chapter 6.6.3 in [30]). It follows:

$$R_e = \cosh(pk)^n \cdot R_0 \tag{19}$$

The hyperbolic cosine in Eq. (19) 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 (Fig. 8).

Assuming an infinite number of instructions k in Eq. (17), we can evaluate the steady-state probability of a single correct 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 k is

$$\lim_{k\to\infty} p_{E0} \approx \frac{1}{2} \cdot e^{pk} \cdot e^{-pk} = \frac{1}{2}. \tag{21}$$

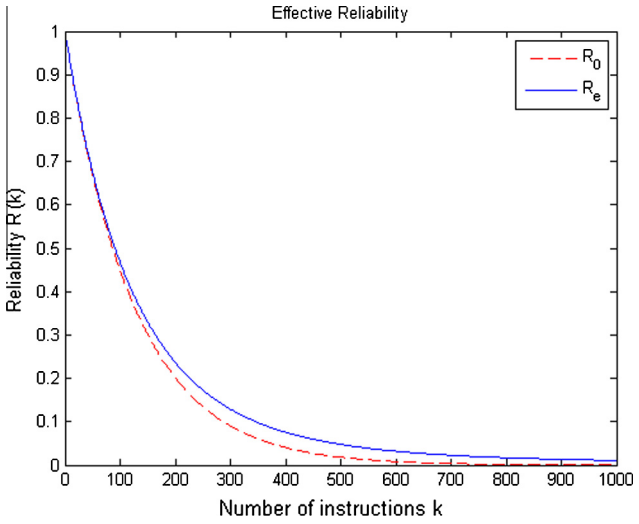

Fig. 8. The effective reliability R_e is higher than the basic reliability R_0 without consideration of fault compensation.

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 which is equivalent to the availability. This means that the described fault compensation can be considered as some intrinsic repair events in the data flow.

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 (Eq. (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 (Eq. (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 Eq. (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 $k = 2$, Eq. (19) determines the reliability of the result after two consecutively executed *XOR* operations. This reliability is shown as a function of p in Fig. 9 (continuous line). Furthermore, the figure

Please cite this article in press as: P. Raab et al., Reliability of data processing and fault compensation in unreliable arithmetic processors, Microprocess. Microsyst. (2015), http://dx.doi.org/10.1016/j.micpro.2015.07.014

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.

Using linear codes, the result after the two *XOR* operations in Eq. (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}, Eq. (15)) and $(1-p)$ is the probability of correct bits (p_{E0}, Eq. (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 Eq. (28) is depicted in Fig. 10. 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).

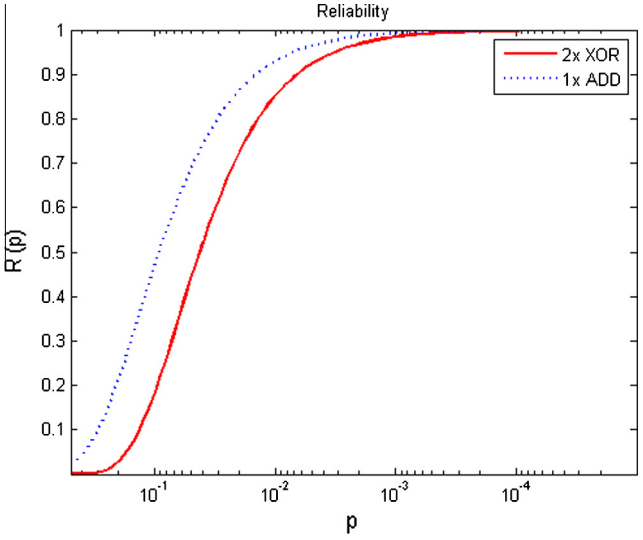

Fig. 9. Comparison of the reliability: (1) two consecutive XOR instructions and (2) one single ADD instruction.

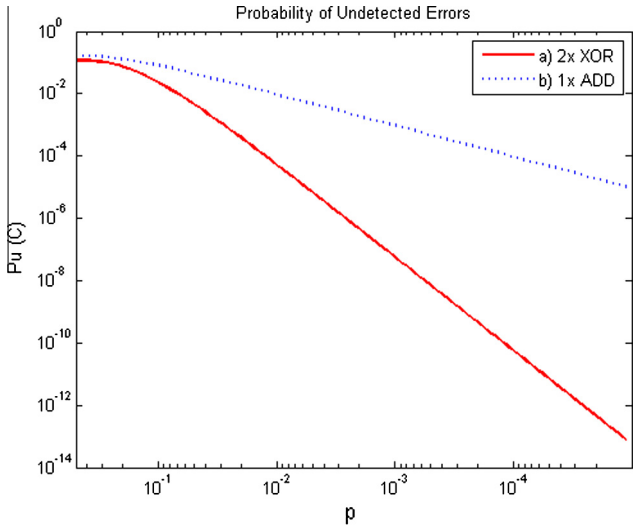

Fig. 10. Residual error probability of linear codes and arithmetic codes.

But, can both results be compared? The presented approach assumes Eq. (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 Fig. 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 [33].

5. Evaluation of fault compensation by stochastic simulation

To verify the analytical result (Chapter 4.2), the effect of fault compensation was analyzed by a discrete event simulation approach that is based on the Monte Carlo method. Within this simulation framework the technique of simulated fault injection was used.

The simulation and the used simulation model is a detailed description of an embedded microcontroller device. It models hardware properties, such as memory accesses, by defining the memory modules, the crossbars, the arbitration and bus timing effects. The model furthermore includes a description of the number of cores in a multicore system and the computing capabilities (clock frequency, number of instructions per tick, memory connection, etc.) of the cores. The second level of the simulation model describes the operating system incorporated in the simulation setup. This includes the used scheduling algorithm as well as the partionating of execution resources (cores) to scheduling units and the timing overhead of scheduling. Standard operating system capabilities like synchronization or time triggering are also supported by the model. Below the hardware and operating system layer, the application layer is placed. The application layer includes the description of Interrupt Service Routines (ISR) and Tasks. Tasks and ISR are defined by functions. Each function contains a certain

Please cite this article in press as: P. Raab et al., Reliability of data processing and fault compensation in unreliable arithmetic processors, Microprocess. Microsyst. (2015), http://dx.doi.org/10.1016/j.micpro.2015.07.014

set of instruction. Complex program flow and branching is supported. The instructions themselves are modeled by different configurable distributions, such as Normal-, Exponential-, Gaussian- or Weibull-distribution that define the individual number of instructions of each instance. The last part of the simulation model contains the stimulation of the system. That means that task activations or external events – such as injected faults or external communication on buses – can be defined. These events are again modeled by various types of distributions, like for the instructions.

For the purpose of analyzing the fault compensation, a single task that continuously executes the XOR instruction was chosen. Faults are injected in each single bit of an 8-bit variable independently. At the end of the execution of the task instances (T_i) the effect of the injected faults to the variable are evaluated. That means the number of compensated and not compensated faults – that leads to an error – and the number of injected faults are counted. This experiment was executed $N = 2e6$ times to gain a reasonable confidence in the results.

The execution of consecutive XOR instructions is a linear data flow with independent operations, such that fault can compensate each other. With Eq. (19) the compensation factor r_c can be determined by:

$$r_c = \frac{R_e}{R_0} = \cosh(pk)^n \tag{30}$$

The following parameters were used to evaluate the analytic result against the simulation approach:

The basic fault probability per instruction was set to $p = 1e{-}6$. The bit size of the variable was $n = 8$ bit. The number of instructions was varied in the range of 100–10e6 instructions.

For the simulation a single core system with a clock rate $f = 200$ MHz was chosen. With the clock rate f and the fault probability p the expected value E of the exponential distribution that is used to model the temporal behavior of the fault injection can be calculated by:

$$E = \frac{1}{\lambda} = \frac{t_{instruction}}{p} = \frac{1}{p * f} = 5\ \text{ms} \tag{31}$$

Fig. 11 visualizes the simulation result based on the named parameters. On top the exponentially distributed fault injection is depicted (FI_Bit_1 ··· FI_Bit_7). The bottom line in the Gantt chart shows the task that is executing the XOR instructions. In Fig. 11 the period of task activation was set to 5 ms and the task contains 1e6 instructions. As mentioned before, the task is executed multiple times (N), in multiple task instances T_i. Each execution of a task instance T_i represents a single experiment.

At the assumed XOR operation, each single bit is independent from the other bits, thus the compensation of a faulty bit is determinated by the number $N_f(i,j)$ of faults occurred for a single bit (j) during the execution of an instance T_i. If the number of faults affecting a single bit $N_f(i,j)$ is even at the termination of a task instance T_i, the faults at this bit where compensated and do not propagate to an error in this bit. For evaluation of the task instance all n bits have to be considered. Thus all injected faults during the execution of the task instance T_i were compensated, if:

$$\sum_{j=0}^{n} (N_f(i,j)\ mod\ 2) = 0 \quad \text{holds.} \tag{32}$$

For evaluating the simulation run, the number of faulty task instances N_f, that means task instances where at least one fault was injected:

$$N_f = \sum_{i=1}^{N} \begin{cases} 1, & \text{if at least one fault was injected on any bit in } T_i \\ 0, & \text{if no fault was injected in } T_i \end{cases} \tag{33}$$

has to be determined.

For evaluating the effective reliability R_e (and with that Q_e), the number of task instances N_{nc}, where no fault in no bit was compensated has to considered. N_{nc} is defined by:

$$N_{nc} = \sum_{i=1}^{N} \begin{cases} 0, & \text{if } \sum_{j=0}^{n} (N_f(i,j) mod\ 2) = 0 \\ 1, & \text{else} \end{cases} \tag{34}$$

Therefore the basic reliability R_0 can be expressed by the number of experiments N and the number of task instances where at least one fault was injected (N_f):

$$R_0 = 1 - Q_0 = 1 - \frac{N_f}{N} \tag{35}$$

And the effective unreliability Q_e and with that the reliability R_e is defined by:

$$R_e = 1 - Q_e = 1 - \frac{N_{nc}}{N} \tag{36}$$

With that the Eqs. (30), (35) and (36) the compensation factor r_c can be calculated for the analytical and the simulative approach. The following Table 1 and Fig. 12 summarize the results.

The effects of fault compensation where analyzed by two methods. Both approaches provide quite similar results. The deviation is in the range of the numeric inaccuracy of the calculation.

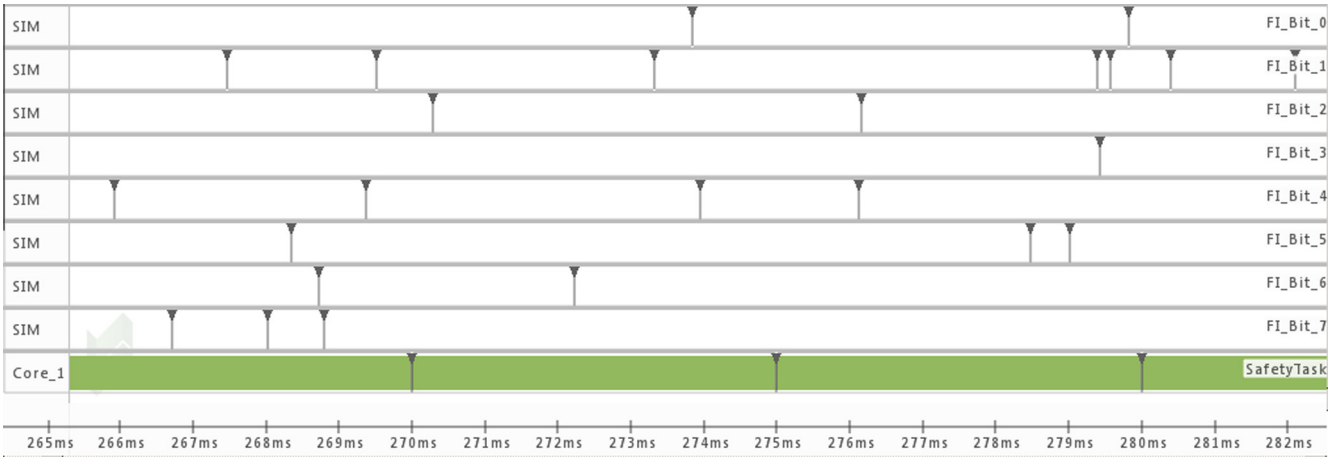

Fig. 11. Simulated task execution with fault-injection (FI_Bit_0 ··· FI_Bit_7); Task SafetyTask has a execution time equivalent to $k = 1e6$ instructions and is activated with a period of 5 ms. Screenshot of TA Tool Suite [34].

Please cite this article in press as: P. Raab et al., Reliability of data processing and fault compensation in unreliable arithmetic processors, Microprocess. Microsyst. (2015), http://dx.doi.org/10.1016/j.micpro.2015.07.014

The validity of the used error models is essential for practical application. And so, the validation by Monte Carlo based discrete event simulation is vital part to prove the correctness of the analytic consideration and to further extend the model in future.

6. 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 Eq. (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 Eq. (6) with

$$R_0 = (1-p)^{2n} \tag{37}$$

Second, both instructions represent independent events with respect to their fault vulnerability and fault compensation is possible. According Eq. (19), the effective reliability is increased by the factor

$$r_c = \cosh(2p)^n \tag{38}$$

With a basic fault probability $p = 10e{-}6$ and a data width $n = 8$, the factor of Eq. (31) is $r_c = 1 + 16e{-}12$. But this covers the data flow including two instructions only. With increasing number of instructions (e.g. $k = 100{,}000$), 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 (Eq. (28)). Fig. 13 qualitatively summarizes all outcomes of the data flow.

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 Fig. 7 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 doesńt 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

Table 1
Comparison of the analytically calculated and the simulative compensation factor r_c for selected number of instructions k.

Number of instrucions k	1e3	5e6	1e6	5e6
r_c analytic	1.00000454	1.00994644	1.04082739	2.60903486
r_c simulated	1.00000400	1.01004596	1.04074157	2.61409003

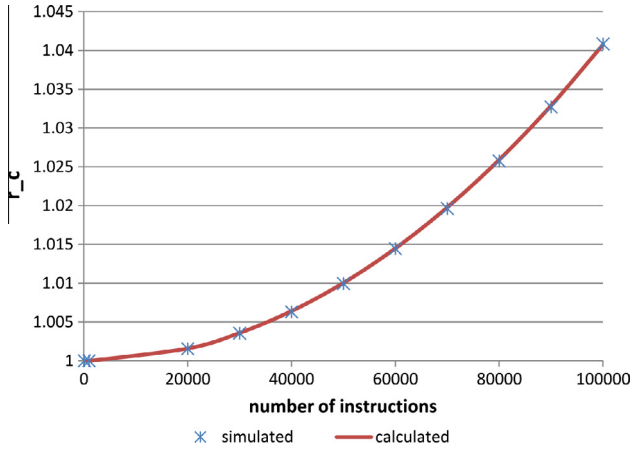

Fig. 12. Comparison of the compensation factor r_c; the red line shows the analytically calculated r_c factor according to Eq. (30), the blue crosses show selected simulated r_c values, which where generated by discrete event simulation. (For interpretation of the references to color in this figure legend, the reader is referred to the web version of this article.)

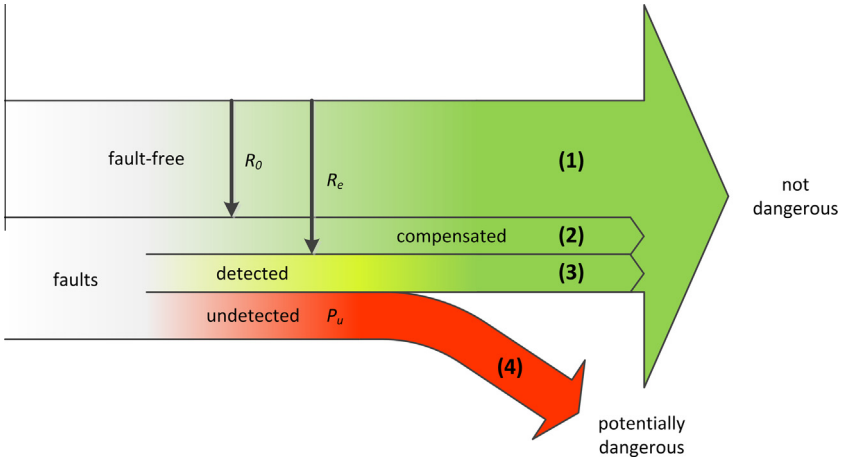

Fig. 13. The entire probability space is partitioned into several groups: (1) fault-free execution, (2) fault compensation, (3) fault detection by error codes and (4) undetection of faults determines the residual error probability.

Please cite this article in press as: P. Raab et al., Reliability of data processing and fault compensation in unreliable arithmetic processors, Microprocess. Microsyst. (2015), http://dx.doi.org/10.1016/j.micpro.2015.07.014

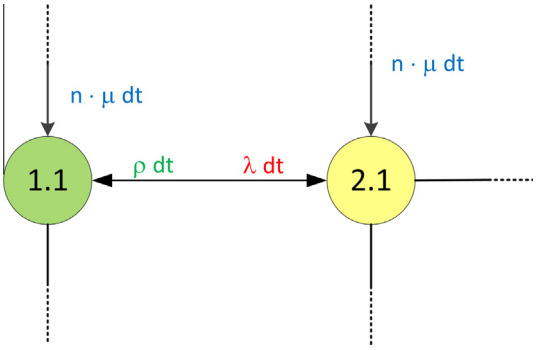

Fig. 14. Detail of the extended Markov model with the effect of fault compensation by additional transitions.

transitions to the left column with an increasing rate depending on the progress of the data flow (Fig. 14).

7. 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.

The validity of the presented models is crucial for its practicability and should be necessarily proofed in future works. For validation, there are two possible approaches. FPGA-based fault injection environment allows the insertion of faults in defined parts of a soft-core [32]. The advantage of this approach is the test of software systems with more realistic conditions. Secondly, the simulation approach as shown in Chapter 5 refers to the same assumptions and validates the correct realization of the single error model. But it can be further improved to reproduce the underlying hardware in greater detail and include a wider range of instructions types.

Furthermore, 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 [33], we already presented a possible transformation rule of linear code words to arithmetic code words and vice versa for this purpose.

Acknowledgements

This work is a result of the research projects Safe Oriented Programming of Software-Intensive Embedded Systems (S^3OP) and Safe Scheduling for Multicore Processors (ZeloS^3). These projects were 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). The project ZeloS^3 is supported by the German Federal Ministry for Economic Affairs and Energy (Code: KF2870902BZ3).

References

[1] P. Dodd, L. Massengill, Basic mechanisms and modeling of single-event upset in digital microelectronics, IEEE Trans. Nucl. Sci. 50 (3) (June 2003) 583–602.
[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, M. Violante, Software-Implemented Hardware Fault Tolerance, Springer-Verlag New York Inc., Secaucus, NJ, USA, 2006.
[4] B. Nicolescu, R. Velazco, Detecting soft errors by a purely software approach: Method, tools and experimental results, in: Design, Automation and Test in Europe Conference and Exhibition, vol. 2, 2003, p. 20057.
[5] F. Wang, 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, E. McCluskey, ED4I: error detection by diverse data and duplicated instructions, IEEE Trans. Comput. 51 (2002) 180–199.
[7] D. Brown, Error detecting and error correcting binary codes for arithmetic operations, IRE Trans. Electron. Comput. (1960) 333–337.
[8] R. Thammavarapu, 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, IEEE Trans. Comput. C-21 (6) (1972) 538–545.
[11] Hao Dong, Berger codes for detection of unidirectional errors, IEEE Trans. Comput. C-33 (6) (1984) 572–575.
[12] U. Schiffel, A. Schmitt, M. Süsskraut, C. Fetzer, ANB- and ANBDmem-encoding: detecting hardware errors in software, in: Computer Safety, Reliability, and Security, Lecture Notes in Computer Science, vol. 6351, Springer, Berlin/Heidelberg, 2010, pp. 169–182.
[13] P. Ozello, The coded microprocessor certification, in: International Conference on Computer Safety, Reliability and Security, Springer, Munich, 1992, pp. 185–190.
[14] P. Raab, S. Kramer, J. Mottok, H. Meier, S. Racek, Safe software processing by concurrent execution in a real-time operating system, in: 2011 International Conference on Applied Electronics (AE), Pilsen, September 2011, pp. 315–319.
[15] A. Benso, S. Chiusano, P. Prinetto, L. Tagliaferri, A C/C++ source-to-source compiler for dependable applications, in: Proceedings International Conference on Dependable Systems and Networks, 2000, DSN 2000, 2000, pp. 71–78.
[16] Mei-Chen Hsueh, T.K. Tsai, R.K. Iyer, Fault injection techniques and tools, Computer 30 (4) (1997) 75–82.
[17] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, G.H. Leber, Comparison of physical and software-implemented fault injection techniques, IEEE Trans. Comput. 52 (9) (2003) 1115–1133.
[18] Jeffrey M. Voas, 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, J. Mottok, Error model and the reliability of arithmetic operations, in: 2013 IEEE EUROCON – International Conference on Computer as a Tool, July 2013, pp. 630–637.
[21] M. Bossert, Kanalcodierung, second ed., Teubner, 1998.

[22] E. Härtter, Wahrscheinlichkeitsrechnung, Statistik und mathematische Grundlagen: Begriffe, Definitionen und Formeln, Vandenhoeck & Ruprecht, 1987.
[23] V. Vais, S. Racek, Experimental evaluation of regular events occurrence in continuous-time markov models, in: Informatics, 2011, November 2011.
[24] R.W. Hamming, Error detection and error correction codes, Bell Syst. Techn. J. 26 (2) (1950) 147–160.
[25] E.J. Weldon, W.W. Peterson, Error Correcting Codes, second ed., MIT Press, 1972.
[26] P. Raab, S. Krämer, 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, 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, J. Mottok, Data flow analysis of software executed by unreliable hardware, in: 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, 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] S. Felis, J. Mottok, Die Fault Bench for Integrated Incircuit Injection zur Verifikation von Safely Embedded Software, in: Embedded Software Engineering Conference, December 2012.
[33] P. Raab, V. Vavricka, S. Krämer, J. Mottok, Isomorphism between linear codes and arithmetic codes, in: Computing and Informatics (CAI), vol. 33(4), 2014.
[34] Timing Architects Embedded System GmbH, TA Tool Suite. <www.timing-architects.com>.

Prof. Dr. Peter Raab is professor for embedded systems and vehicle communication systems at the University of Applied Sciences and Arts Coburg (www.hs-coburg.de/raabp).
Formerly, he was research assistant at the Laboratory for Safe and Secure Systems (www.las3.de) working in the research project S^{3}OP for model – based reliability evaluations.

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 ZeloS^3. 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).

Please cite this article in press as: P. Raab et al., Reliability of data processing and fault compensation in unreliable arithmetic processors, Microprocess. Microsyst. (2015), http://dx.doi.org/10.1016/j.micpro.2015.07.014

P.5. Safe software processing by concurrent execution in a real-time operating system

Authors:	P. Raab, S. Krämer, J. Mottok, H. Meier, and S. Racek
Published in:	In *Proceedings of 16th IEEE International Conference on Applied Electronics*
Year:	2011

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:

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

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^{3}TOS, 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

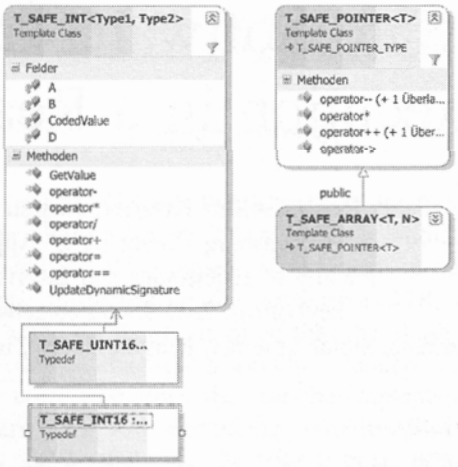

Figure 1. Class-diagram of the SES library: The class consists of overloaded operators, which perform the coded computation. The class encapsulates data fields like the coded representation of the value itself, the applied signature (B and D) and the diversity factor (A).

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^{3}TOS 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^{3}TOS 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^{3}TOS (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^{3}TOS.

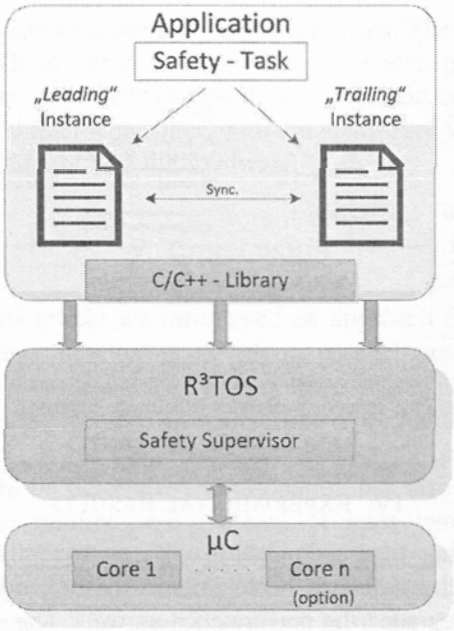

Figure 2. Simplified software architecture of R³TOS. The Safety-Supervisor as an extension of R³TOS monitors the execution of a safety-task and improves the fault-tolerance. The safety task is executed in two separate instances - the leading and the trailing task instance.

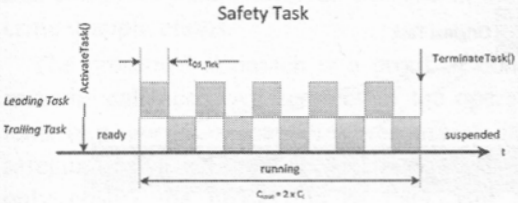

Figure 3. Time redundancy achieved by duplicated task execution

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).

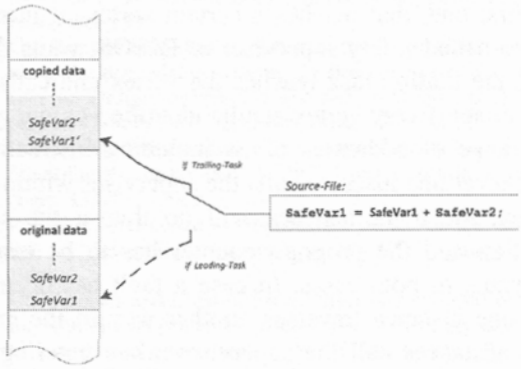

Figure 4. Access of duplicated global variables by a safety task: Original data is generated by the linker during software build process. Copied data are generated by the operating system during startup.

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].

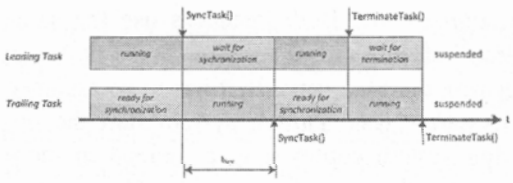

Figure 5. Synchronization of two task instances

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.

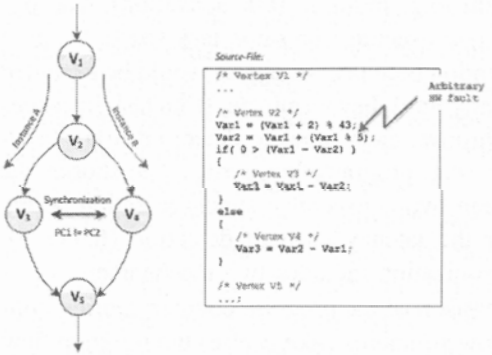

Figure 6. Example for a program graph of a task

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

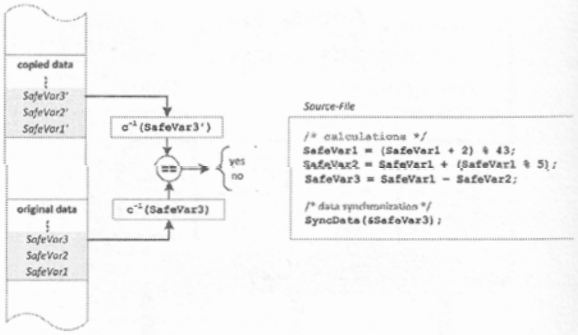

Figure 7. Synchronization of two data channels: In case of different coded data, the supervisor decodes it before comparison.

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.

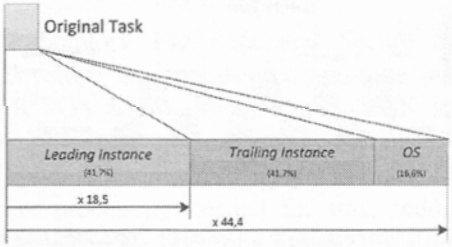

Figure 8. Slowdown of application runtime compared to its original version

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.

P.6. Comparison of Enhanced Markov Models and Discrete Event Simulation - for evaluation of probabilistic Faults in safety-critical real-time task sets

Authors:	S. Krämer, P. Raab, J. Mottok, and S. Racek
Published in:	In *Proceedings of the 17th Euromicro Conference on Digital System Design*
Year:	2014

2014 17th Euromicro Conference on Digital System Design

Comparison of Enhanced Markov Models and Discrete Event Simulation

for evaluation of probabilistic Faults in safety-critical real-time task sets

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

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

Abstract **– In this paper we present simulation and model based approaches for evaluating and validating the temporal and safety relevant properties of software intensive safety-critical real-time embedded systems. A high level reliability model of a safe task execution is described by a continuous-time Markov process, enhanced by the modeling of execution times. It is shown that the behavior – regarding real-time and safety metrics – of this theoretical model can be transferred into an abstract system timing model, which then can be analyzed by a discrete event simulation approach. The verification of the discrete event simulation by Markov models offers the possibility of a holistic approach for reliability analysis combined with schedulability analysis of complex safety-critical multicore real-time systems by the discrete event simulation.**

Markov Model; Stochastic simulation; reliability analysis; safe software processing; real-time operating system; multicore scheduling; discrete event simulation; fault injection

I. INTRODUCTION

The functionality and complexity of embedded systems in different domains was increasing to a greater extent in the recent years, accompanied by the continuously rising requirements for safety, reliability and availability. But the contrary requirement is to reduce costs for those systems. Therefore a trend of shifting safety mechanisms like diverse hardware towards software approaches can be observed [15]. Multicore systems are more and more used as they offer not only an advantage regarding performance, but also have a high potential of increasing the reliability of software intensive embedded system [18].
Moreover a task set scheduled on a multicore with a global scheduling algorithm can in general react more flexible in case of influences by transient faults than singlecore scheduling algorithms do[11].

A. Background

Mixed criticality embedded systems will become more common due to the fact that multiple functional components are combined into one ECU [21]. That means that there are tasks which need additional steps to ensure the systems reliability requirements, e.g. by using hardware lockstep mode. Non-safety-critical QM-tasks do not require additional steps. For that reason a software approach can be more flexible and handle the different safety requirements differently and execute non critical tasks in a single instance and safety-relevant tasks in e.g. redundant execution. Thus the overall performance and reliability can be increased.

In a hard real-time system a system error is not only the wrong computed result caused e.g. by a transient fault but also the violation of certain timing constrains. These deadline violations can be caused e.g. by the transient faults itself, by affecting the scheduler or by the safety mechanisms triggered by the safety supervisor. Safety mechanisms to harden systems against soft errors can e.g. be implemented by coded processing or diverse execution. These strategies consume an extra amount of computation time which in addition influences the feasibility of the schedule. In the past, worst case assumptions were widely used to handle these scenarios [18].

B. Evaluation approaches

The usage of such multicore systems for safety-critical tasks in a mixed criticality system makes it harder to analyze the systems fault and timing behavior analytically. Such analytical approaches like Markov Models or network modeling [1] for evaluating the reliability of a system or software intensive systems are common and widely used. These methods have in common that they produce respectable results for simple systems. However complex real-life embedded systems – including the real-time scheduling behavior of multicore scheduling algorithms – cannot be modeled adequately or have to be abstracted to a certain level of simplicity [1], [11].
Regarding schedulability analysis, there is no analytical approach for proofing feasibility of a global dynamic multicore scheduling algorithm until now. For singlecore scheduling algorithm the proof of feasibility can be done analytically [13].

Our approach is to use an enhanced Markov model which includes the runtime behavior of task execution

978-1-4799-5793-4/14 $31.00 © 2014 IEEE
DOI 10.1109/DSD.2014.42

to determine not only reliability but also timing metrics – like the task deadline – of a task. Therefore it is possible to consider the basic reliability and timing properties of inspected simplified tasks.
The same system description is used to setup a model based on stochastic simulation environment. With the Monte Carlo based discrete event simulation reliability and timing metrics of the system and the applied safety mechanisms are analyzed by simulated fault injection.
These two evaluation concepts – Markov Model and discrete event simulation – are then verified against each other by analyzing basic metrics such as the probability of a deadline violation and the resulting execution time when different safety mechanisms are applied to handle the occurrence of sporadic faults.

The simulation model is then extended to simulate not only the standard reliability indices but also to simulate the real-time characteristics of an embedded system. The discrete event simulation is used to model the variance of task execution durations and the general feasibility of a schedule in a multicore real-time embedded system. The reliability indices are determined by probabilistic fault injection of transient faults. The simulation approach can easily be enriched by a complete mixed criticality tasks set of an automotive embedded system.

Thus a complex embedded real-time system can be evaluated with regard to reliability metrics and at the same time with regard to real-time requirements.

This paper is organized as follows. Section II gives an overview of the example task set and the applied fault handling mechanisms. The modeling of this system by a Markov Model is explained in Section III. In Section IV the discrete Event Simulation and the transition from the original Markov model is presented and the results are discussed. The importance of the simulated approach is highlighted by considering a representative automotive task set in section V. Finally Section VI gives a conclusion and outlook for further research.

II. BASIC CONCEPTS FOR CVALUATING RELIABILITY INDECIES OF A REAL-TIME SYSTEM

This section will describe the two safety mechanisms considered in this paper, coded processing and symmetric redundant execution [4], [14]. After that relevant system parameters and metrics are focused and the properties of the evaluated task are introduced.

A. Considered fault handling startegies

1) Coded processing

Coded processing is the protection of calculations and their results during operations in an arithmetic unit by means of error detection codes. An important group of error detecting codes are the so-called arithmetic codes (AN-code) that are based on ordinary algebra like addition and multiplication [2]. Forin made the first use of coded processing in a real application [20], [17].

2) Symetric redundant execution

The duplication of components is a common method to increase the reliability of systems. The same task is duplicated into *n* task instances, which are executed one after another on a singlecore system (time redundancy) or on different cores on a multicore system (space redundancy). The computation results of the *n* instances are compared and evaluated after complete execution by the voter and possible single faults can be detected [18].

B. Relevant system parameters

The following parameters are used for the Markov Model and the abstract timing model of the discrete event simulation.

1) Fault rate, Intensity I:

This is the fault rate λ of the occurrence of transient faults. It is defined as the reciprocal value of the mean time to failure (MTTF) of transient faults [1].

$$I = \lambda = \frac{1}{MTTF} \tag{1}$$

A hazard rate of:

$$10^{-8}\,{}^{1}/_{h} < \lambda < 10^{-7}\,{}^{1}/_{h} = \frac{1}{MTTF} \tag{2}$$

for the overall system is stated in the ISO 26262 for ASIL C. According to formula (2) this would result in an error probability of

$$p = \frac{10^{-7}\,{}^{1}/_{h}}{3600\,{}^{s}/_{h}} * 10 * 10^{-3}s = 2.8 * 10^{-14} \tag{3}$$

for execution duration of a task instance of 10ms. In other words $2.8*10^{14}$ task executions have to be performed during the simulation run to catch only one error. Moreover the effects of multiple occurrences of faults within the execution of a task instance could not be shown so clearly.
Due to that a rate of

$\lambda = \frac{1}{10\ ms}$ is assumed.

2) Response time

The response time of a task instance is defined by delta of its activation time and its termination time. Included are the execution time (*e*) itself and any possible preemptions of the task instance by other task instances or by a delayed start of task instance execution [13].

$$\mathrm{t_{response}} = t_{terminate} - t_{activate} \tag{4}$$

3) Deadline

To fulfill hard real-time timing requirements a task instance (*i*) has to be finished successfully and terminated within its deadline *D* [13]. Thus:

$$D < \mathrm{t_{response}}(i) \tag{5}$$

C. Paramaters of the task sets

For comparison and verification of the both approaches – Markov Model and discrete event

simulation – two task sets are applied.
First coded task processing is used. The error detection probability for coded processing is set to 100% because for this simplified consideration a sufficient hamming distance is assumed to detect every occurred fault. Fault detection leads to a re-execution of the task instance, until no further fault occurs.
Secondly, symmetric redundant task execution is applied to the safety task. Thus the instance is duplicated and executed twice, either in space redundancy on a multicore or in time redundancy on a singlecore processor. The results are compared by a voter at the end of task execution. Based on the usage of 8bit variables, the probability of not detecting different results in both instances is assumed by $d = 0.125 = 1/8$.
In both scenarios the task execution time is 10ms and the task period is 100ms. The applied fault rate λ is set to $\frac{1}{10ms}$. Table 1 summarizes this.

TABLE I. MAIN TASK PARMAETERS

	Execution time [ms]	**Period/Deadline [ms]**	**Fault rate λ [1/ms]**
A: Coded	10	100	0.10
B: Redundant	10	100	0.10

III. ENHANCED MARKOV MODEL

Continuous-time Markov models is an alternative method for evaluating faults in task sets. But originally, Markov models describe the probability of faulty outcomes based on the given fault rate. The necessary runtime information of a task for following fault recovery is not covered. Therefore, the basic Markov model must be enhanced by the runtime information using the *Erlang* distribution [19]. The *Erlang* distribution is well-known from the queuing theory and combines the distribution of two different events, e.g. the randomly fault injection and the more regular task termination. Because the *Erlang* distribution is the composition of several exponential distributions, the Markov Model in form of a matrix structure can be used.

Figure 1 and Figure 3 show the enhanced Markov Models for the previously mentioned scenarios – coded processing and symmetric redundancy. Every vertical stage represents a certain time interval within the task processing following the Erlang distribution. The more stages there are the higher the resolution is in time. Whereas, the horizontal stages represent the original Markov model describing the fault injection events following the exponential distribution.

Basically, a continuous-time Markov Model is described by a set of differential equations with one equation for each state. For the Markov Model in Figure 1, these are:

$$\begin{aligned} P'_{1.0} &= -\lambda \cdot P_{1.0} - n\mu \cdot P_{1.0} \\ P'_{2.0} &= +\lambda \cdot P_{2.0} - n\mu \cdot P_{2.0} \\ &\dots \\ P'_{1.k} &= -\lambda \cdot P_{1.k} - n\mu \cdot P_{1.k} + n\mu \cdot P_{1.(k-1)} \\ P'_{2.k} &= +\lambda \cdot P_{1.k} + n\mu \cdot P_{2.(k-1)} - n\mu \cdot P_{2.k} \\ &\dots \\ P'_{1.n} &= +n\mu \cdot P_{1.(n-1)} \end{aligned} \quad (6)$$

The solution of these equations leads to the state probability as a function of time for all states. However, the last states *1.n/2.n* represent the termination of the task and they are more of interest. As there are no outgoing transitions, these states are absorbing states and the probability of entering one of these states is unity in total at the end. But during the task execution (vertical stages *1 ... n-1*), the probability of entering such an absorbing state is constant and zero in the first cycle. It depends on the injected fault if the task will be repeated and the absorbing state is entered again.

The density function *f(t)* describes the normalized probability of a random variable. In this case, the interesting random variable is the total execution time (response time) of the task whose time-based sum of probabilities is the sum of the probabilities of all absorbing states. The density function is the first derivative of *p(t)* with respect to the time

$$f(t) = \frac{dp_x(t)}{dt}. \quad (7)$$

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

$$E = \int_{-\infty}^{\infty} t \cdot f(t)\ dt\ . \quad (8)$$

This equals the expected value of task execution time (response time) [17] and it represents the average time the task will terminate after all repetitions.

A. Markov Model of coded task processing

The model in Figure 1 describes the faulty execution of a coded task. It models the injection of only one fault (= one additional column) during the task execution which is detected with certainty (= transition to the state 1.0). The differentiation of the probability $p_{1.n}(t)$ of the terminating state 1.n results according to formula (6) in the density function of the absorbing state *1.n*:

$$f(t) = \frac{d(p_{1.n}(t))}{dt} \quad (9)$$

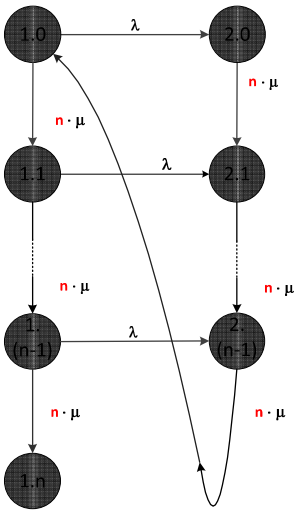

Figure 1: Enhanced Markov model that simulates the runtime of a single task with several stages. The total execution time with t_{Task} =1/μ is divided into n segments. The transition rate of every stage is therefore n*μ. The task terminates after t_{Task} in state 1.n assuming the detection of the fault and the following repetition of the task.

With given parameters (see also simulation approach in Chapter IV.E) the expected value of the task execution time for this task is

$$E = 26.81ms.$$

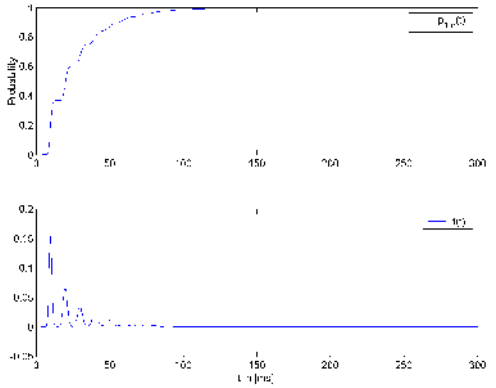

Figure 2: Cumulative probability function *p(t)* and density function *f(t)* of task execution. The steady-state probability is approached step by step with each task repetition.

In Figure 2 (as well in Figure 4) 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 the probabilities of absorbing states converges to unity. For better visualization, the fault rate is dramatically increased to an unrealistic value.

B. Markov Model of symmetric Redundant Execution

Figure 3 depicts the Markov Model of symmetric redundant task execution. This means the redundant execution of the same task. At the end, both task outputs (A1, A2) are compared for fault detection. In case of only one faulty task, the fault is surely detected and the execution of both tasks is repeated. There is a transition from the end of second column to state *1.0* in Figure 3. In case of two faulty tasks, the outputs of both task are probably different and the same fault recovery (= task repetition) can be done. But there is also the probability of non-detection in case both tasks results in the same faulty output. The probability of not detecting a fault is now set to $d = 0.125$ (see II.C) and it results in the absorbing state *3.n* of Figure 3.

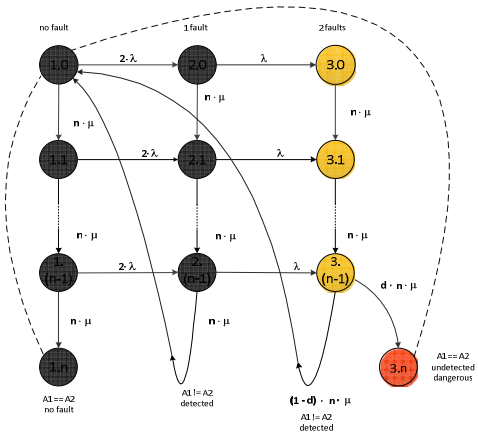

Figure 3: Model of redundant execution with absorbing states. The task execution consists of n stages. There are two absorbing states. 1.n represents the fault-free termination whereas the state 3.n represents the faulty case which is not detected.

The last states represent the termination of the task either without any fault (state *1.n*) or with an undetected fault (state *3.n*). The differentiation of the probability functions $p_{1,n}(t)$ and $p_{3,n}(t)$ of the absorbing states results in the density function *f(t)* [17].

$$f(t) = \frac{d(p_{1,n}(t)+p_{3,n}(t))}{dt}. \qquad (10)$$

With equation (7) and with the given parameters the expected value for the response time of this task is

$$\mathrm{E} = 52.5ms.$$

This means that the task terminates after 52.5ms in average, which represent about 5 task repetitions.

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.

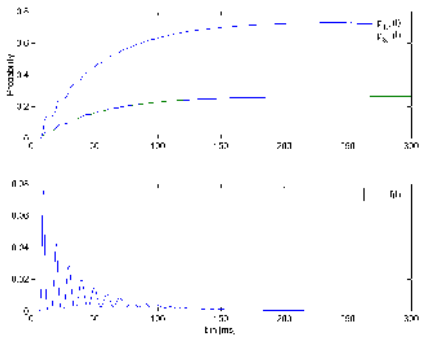

Figure 4: Cumulative probability p(t) and the densit function f(t) of symmetric redundant task execution. The steady-state probability is approached step by step with each task repetition.

IV. SIMULATION MODEL

For the simulation based evaluation of the two mentioned error handling strategies a multi seed discrete event simulation is used. To cover a wide range of the design space each simulation run is executed multiple times. In order to establish a reasonable confidence in the result and to gain a high precision, the simulated time span of one single run was set to 10,000 seconds that were executed 10,000 times each.

As depicted in Figure 5 and Figure 7 the modeling of the task execution is very close to real task execution on a target. At the beginning input data is read, then the safeguarded processing ("Actual Function") is executed and the result is checked. If no error occurred the task terminates normally and writes to the output. Otherwise, if an error was detected the task is restarted (by calling e.g. API-Function ChainTask() of the a real-time operating system, such as e.g. OSEK, see [22]).

A. Simulation Model

The Monte-Carlo based discrete event simulation framework - used for the analysis in this paper - uses an abstract system description model.

The model consists of a hardware part defining the processor, which itself consists of multiple cores. These cores execute the instructions defined in the application part of the model. The execution time of these instructions is defined by the clock of the core and the amount of instructions the core can execute per tick.

The second part of the model describes the operating system architecture, including the used scheduling algorithms, the type of multiprocessing (symmetric, asymmetric, or combined) and the mapping of scheduling algorithms to cores.

In the application part of the model the different task and interrupt service routines are defined. A task itself contains several function calls and has the capability to model conditional branching of the execution flow. Functions are assembled by instruction blocks, they define the amount of instructions (constant number, or defined by a distribution) that have to be executed by the core.

The last part of the model describes the stimulation of the simulated system. This stimulation contains the activation of tasks and ISRs, the changing of values of variables or the triggering of the fault injection. These trigger mechanisms can either be configured periodically or also varied by a distribution.

Thus the real system behavior can be depicted quite accurately starting at the level of instructions up to the modeling of the application with tasks and operating systems.

B. Simulation input parameters

During the simulated task executions faults are injected via system stimulation. The fault injection time is defined by an exponential distribution with the given λ (see (1) and Table 1). For comparability with the Markov Model constant execution times according Table 1 are assumed. A priority based scheduling algorithm is chosen.

C. Simualtion output parameters

By tracing the different events of the simulation a great variety of timing and reliability metrics can be evaluated. In this paper we focus on the average response time.
It is defined by:

$$\overline{t_{response}} = {}^{1}/_{N} \sum_{i}^{N} t_{response}(i) \qquad (11)$$

and is equivalent to the expected value of the execution time (7) in the Markov Model.

The probability *p(t)* of finishing the execution within a certain upper limit of the response time (deadline) can be achieved by determine the relative frequency of the single response times $t_{response}(i)$. Therefore the fulfillment of a deadline requirement can be checked as well.

D. Modelling of Coded Task Processing

The check for error occurrence at coding processing can be directly integrated in the task execution. This is shown in Figure 5. After detecting an error the task is re-executed like it is described in the introduction of this chapter.

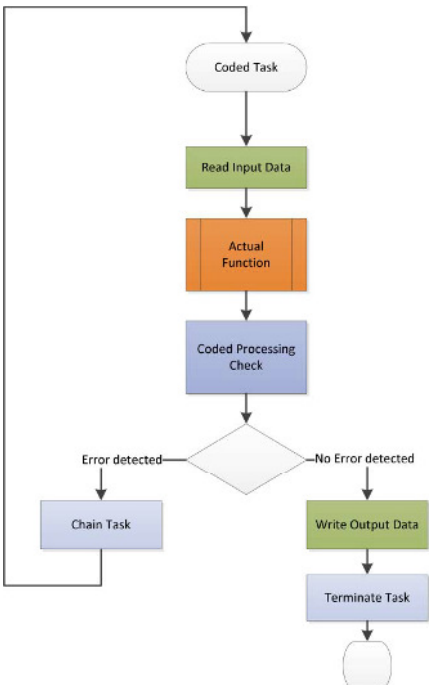

Figure 5: Modeling of the program flow for coded task processing in the discrete event simulation.

The evaluation of the simulation runs produces the following results. Figure 6 displays the cumulative probability that the response time of the task is below a certain time span.
The probability that a task can fulfill its deadline requirement is in this case $p(t_{response} < deadline) = 0.99$.

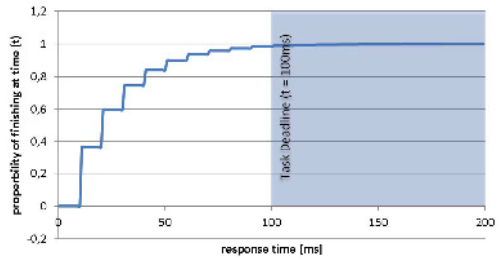

Figure 6: Cumulative probability of coded task successfully finishing after response time (t), the task deadline is given at t = 100ms.

The average response time was evaluated to:

$$\overline{t_{response}} = 27.28ms \qquad (12)$$

The comparison of the result of the Markov model and the simulation result shows that only little deviations, which are within the expected precision, have occurred. It has to be mentioned that the simulation considers the behavior of the complete system, including scheduler overhead, context switching delay, etc.; therefore the higher value of the response time is reasonable.

E. Modelling of symmetric redundant task execution

For employing symmetric redundant execution on a multicore system some effort has to be made to achieve the necessary synchronization between the two task instances. In the simulation model this is handled equally to a real embedded target. The tasks are synchronized by usage of the operating system's event functionality. If a task waits for the second instance to reach the synchronisation point, it is put into state ready, which means, that other task allocated to that core can be executed. The comparison of the computation results should be done in a global voter. The error detecting probability $(1 - d)$ (see II.C) is considered within the simulated voter. This could be integrated in a safety supervisor as suggested in [12]. After comparison, the further implementation is similar to that of codded processing.

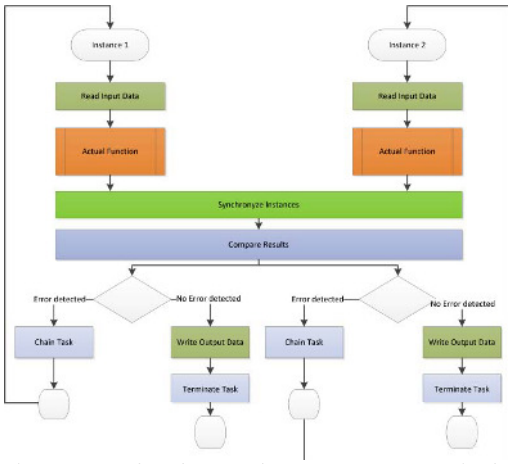

Figure 7: Modeling of symmetric redundant task execution in discrete event simulation approach.

The evaluation of the simulation produces the following results. Figure 8 displays the cumulative probability that the response time of the task is below a certain time span.
The probability that a task can fulfill its deadline requirement $p(t_{response} < deadline)$ is in this setup 0.87.

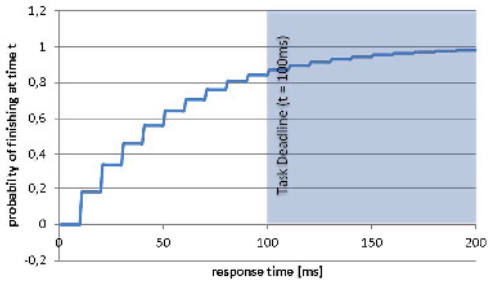

Figure 8: Cumulative probability of redundant task successfully finishing after response time (t), the task deadline is given at t = 100ms.

After evaluating M = 10,000 simulation runs an average response time of:

$$\overline{t_{response\ Multi\ Seed}} = \frac{1}{M}\sum_{j=0}^{M} \overline{t_{response}}\ (j) \quad (13)$$

$$\overline{t_{response\ Multi\ Seed}} = 53.95ms \quad (14)$$

can be observed. Table II contains the statistical analysis of the multi seed simulation run.

TABLE II. RESPONS TIME AVERAGE FOR MULTI SEED SIMULATION

	min [ms]	average [ms]	max [ms]
Respons time	51.67	53.95	55,82

It can be seen that the analytical result of the response time E = 52.5ms is within the range of the simulation results. The higher average value compared to the Markov Model is an effect of the additional modeling of the synchronization mechanisms and the general scheduler overhead.

V. TRANSITION TO REAL A AUTOMOTIVE TASK SET

In the next step the basic task set (TABLE I. B) used for validation of the simulation approach by Markov-model is extended to cover a more realistic task set, such as it is used e.g. in the automotive industry, to reveal that the simulation approach is capable to analyze more complex systems, as well.

In the extended model additionally to the existing safety-critical tasks using the symmetric redundant approach (see section II), normal, non-safety-critical tasks (QM-Tasks) are added. These two tasks are equally mapped to the cores of the used dualcore processor. These tasks have an uniform distributed execution time between 2ms and 3ms. They are triggered every 10ms and have the same priority such as the safety tasks.

TABLE III. ADDITIONAL QM-TASKS

	Execution time [ms]	Period/Deadline [ms]
QM-tasks	2 - 3	10

The result of this additional load is an increased average response time and increased number of deadline violations of the safety task instance (see Table IV). The re-execution is influenced by the non-critical tasks as they get in state running, while the safety tasks are waiting for each other at synchronization points. Furthermore there is a big impact on the QM-tasks, as well because the relatively long safety-task execution, in case of an error causes jitter in the start time and as well leads to higher probability of deadline violations (1.3%). Additionally 0.02% of the activations of the QM-tasks where skipped, because of the system overload conditions when executing safety-critical tasks.

TABLE IV. INFLUENCES OF ADDITIONAL QM-TASKS

	Without QM-tasks	With QM-tasks
Saftey-task: response time [ms]	53,95	57,53
Saftey-task: deadline violations [%]	12.8	16.3
QM-task: start to start jitter [ms]	-	0 … 462

VI. CONCLUSION AND FURTHER STEPS

The enhanced Markov Model as well as the Monte Carlo based discrete event simulation with fault injection produces comparable results. The result of the analytical Markov approach for the expected response time is within the range of the simulated results for the expected response time of the multi seed simulation. The low deviation of the simulated results compared to the analytical results is caused by consideration of operating system calls (synchronization, scheduling) in the more detailed simulation.

Moreover it has been shown, that with the simulation more complex real-life systems can be analyzed (V). The timing effects of the error handling on safety-critical as well as on QM-tasks have been demonstrated.

The applied scheduling algorithm has to be considered in the whole system analysis. That can hardly be achieved by Markov modelling and is even impossible for multicore systems [9], [13].

The verification of the simulation approach by Markov Models has shown that discrete event simulation is capable to evaluate safety-critical systems in a holistic timing and reliability view. This means the simulation approach can reasonable be used for evaluating such safety-critical multicore systems.

Furthermore for future research, the results of the simulation point out that the scheduling algorithm [15] has to be aware of the fault reaction and has to consider this in its schedule decisions. This will be part of future research.

REFERENCES

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

[2] P. Raab, S. Kraemer, and J. Mottok. 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.

[3] H. Beitollahi, S. Miremadi, G. Doconick. Fault-tolerant earliest deadline first scheduling algorithm. IEEE, 2007.

[4] B. Douglass. Doing hard time. Developing real-time system with UML, objects, frameworks, and patterns. Addison-Wesley, 2007.

[5] Mottok, F. Schiller, Th. Völkl, and Th. Zeitler. A concept for a safe realization of a state machine in embedded automotive applications. In *Proceedings of the 26th Safecomp Conference*, ISBN 978-3-540-75100-7, pages 283-288, 2007.

[6] S. Ross. Introduction to probability models. Academic Press, 2003.

[7] C. Han, K. Shin, J. Wu, A fault-tolerant scheduling algorithm for real-time perodic tasks with possible faults, IEEE TRANSACTIONS ON COMPUTERS, VOL. 52, NO. 3, 2003.

[8] S. Ghosh, R. Melhem, D. Mossé and J. Sarma. Fault-tolerant rate-monolitic scheduling. Kluwer Academic Publishers, pages 149 – 181, 1998.

[9] L. Yang, Z. Cui and X. Li. A case study for fault tolerance oriented programming in multi-core architecture. IEEE computer society, pages 630 – 635, 2009.

[10] E. Beckschulze, F. Salewski, T. Siegbert and S. Kowalewski. Fault handling approaches on dual-core microcontrollers in safety-critical automotive applications. CCIS 17, pages 82 – 92, 2008.

[11] M. Deubzer, J. Mottok, and A. Baerwald. Dependability-Betrachtung von Multicore-Scheduling. *HANSER Automotive*, November 2010.

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

[13] G. Buttazzo. Hard real-time computing systems – Predictable Scheduling Algorithms and Applications. Springer, 2004

[14] N. Leveson. Safeware: System Safety and Computers. Addison-Wesley, 1995.

[15] A. Burns, R. Davis, and S. Punnekkat. Feasibility Analysis of Fault-Tolerant Real-time Task Sets. In Proceedings of EURWRTS, 1996

[16] U. Schiffel, M. Süßkraut, and C. Fetzer. AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware. in The 28th International Conference on Computer Safety, Reliability and Security (SafeComp 2009), 2009.

[17] P. Raab, S. Racek, S. Kraemer, 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

[18] S. Kraemer, 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

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

[20] P. Forin, "Vital Coded Microprocessor principles and application for various transit systems," in IFAC Symposia Series, 1989, pages 79-84.

[21] J. L. Herman, C. J. Kenna, M. S. Mollison, J. H. Anderson, and D. M. Johnson, "RTOS Support for Multicore Mixed-Criticality Systems," *2012 IEEE 18th Real Time Embed. Technol. Appl. Symp.*, pp. 197–208, Apr. 2012.

[22] OSEK/VDX Operating System Version 2.2.3, http://portal.osek-vdx.org/files/pdf/specs/os223.pdf

P.7. Reliability Analysis of Real-time Scheduling by Means of Stochastic Simulation

Authors:	S. Krämer, P. Raab, J. Mottok, and S. Racek
Published in:	In *Proceedings of 17th IEEE International Conference on Applied Electronics*
Year:	2012

Reliability Analysis of Real-time Scheduling

by Means of Stochastic Simulation

Stefan Krämer, Peter Raab, Jürgen Mottok
Regensburg University of Applied Sciences
Faculty of Electronics and Information Technology
Seybothstr. 2, D-93053 Regensburg, Germany
{stefan.kraemer, peter.raab, juergen.mottok}@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 – We* present a holistic approach of reliability analysis combined with schedulability analysis of software intensive embedded real-time systems by stochastic simulation. In such a system not only the software execution has to be hardened against soft errors e.g. by means of coded processing or diverse execution. Moreover the real-time requirements have still to be met in presence of such error to guarantee a safe operation of the system. For that reason the influence regarding the real-time characteristics of a given sporadic error with a certain error rate is analyzed by means of Monte Carlo simulation. Different safety design pattern are introduced and compared. Furthermore the impact on the schedulability of an embedded system is discussed.**

Stochastic simulation; reliability analysis; safe software processing; real-time operating system; multicore; scheduling; discrete event simulation

I. INTRODUCTION

In recent years the functionality and complexity of embedded systems in different domains was increasing to a greater extent. But also the requirements for safety, reliability and availability are continuously rising. The contrary requirement is to reduce costs for those systems. Thus a trend of shifting safety mechanisms like diverse hardware towards software approaches can be observed [19]. These demands influence the used hardware platforms. In the past years the clock rate was enhanced for obvious reasons. But this results in problems with EMC as well as in thermal problems. That is why multicore systems are more and more used. They offer not only advantages regarding performance. But furthermore there is a high potential of increasing the reliability of software intensive embedded system by making use of multicore systems. They provide the possibility of diverse execution of software. Summarizing a trend towards multicore systems with smaller feature size and thus higher probability of soft errors can be observed.

A task set scheduled on a multicore with a global scheduling algorithm can in general [13] react more flexible to influences like transient faults than single core scheduling algorithms. Some vendors provide dual or multicore controllers with integrated hardware lockstep mechanisms. For the software the underlying hardware can be treated like a single core. Typically an embedded system has different tasks with different demands of reliability. That means that there are tasks which do not require the execution in lockstep mode. For that reason a software approach can be more flexible and handle the different safety requirements differently and execute non critical tasks in a single instance. The performance can be increased.

The same principle can be applied to coded processing approaches to provide higher safety levels on the one hand and to take usage of the higher performance potentials of multicore platforms on the other hand.
State of the art normative regulations (IEC61508, ISO 26262) for functional safety name several possible solutions for error detection and handling in safety-critical embedded systems. The laboratory for safe and secure system (LaS^3) has developed the Safely Embedded Software (SES) [7] approach based on AN-codes in high level programming languages. The concept of combining coded processing and the integration of these mechanisms into a real-time operating system was presented in [14].

All these approaches have in common that they have to be verified by testing techniques. For single core scheduling algorithm the proof of feasibility can be done analytically [15]. This task – of finding an analytic proof of feasibility – becomes harder or even impossible for global, dynamic multicore scheduling algorithms.

There are several methods like, Morkov models or network modeling [1] to evaluate the reliability of a system or software intensive system. These analytic methods have in common that they produce respectable results for simple systems. However complex real life embedded systems cannot be modeled adequately or have to be abstracted to a certain level of simplicity.

In a hard real-time system a system error is not only the wrong computed result caused e.g. by a transient fault but also the violation of certain timing constrains. These deadline violations can be caused e.g. by the transient faults itself, by affecting the scheduler or by the safety mechanisms triggered by the safety supervisor. Safety mechanisms can be the re-execution of the task, backward or forward recovery. These strategies consume an extra amount of computation time which in addition influences the

feasibility of the schedule. In the past worst case assumptions were widely used to handle these scenarios.

Our approach is to use a stochastic simulation environment to prove the reliability of the scheduling subsystem in a software intensive embedded system. Not only with the regard to simulate the standard reliability indices but also to simulate the real-time characteristics of an embedded system. On the one hand stochastic simulation in combination with discrete event simulation is used to model the variance of task execution durations and the general feasibility of a schedule in a multicore real-time embedded system. On the other hand it is as well used to model the influence of transient errors affecting the system in a holistic approach. Transient hardware faults only propagate to system errors if and only if the faults are not detected by the fault detecting mechanism and the errors cannot be repaired within the real-time timing constraints, e.g. the deadline.

This paper is organized as follows. Section II gives an overview of error mechanisms to be considered. The design patterns which are applied to the safe operating system are explained in Section III. In Section IV the Monte carlo simulation is presented by introducing a sample task set. The results are summarized in Section V. Finally Section VI gives a conclusion and outlook for further research.

II. SAFETY ISSUES OF SCHEDULING

In this section we will describe the influence of transient hardware faults on the feasibility of the scheduling system. Transient errors are brought into focus because these are short disruption of the timing of the schedule. These disruptions can be handled by a robust scheduling algorithm [3]. It should be mentioned that in case of permanent errors there cannot be a scheduling strategy in a single core system to handle such errors. Whereas in multicore systems a permanent error on certain affected core can be detected by the scheduler. This can be achieved by evaluating the results of coded data processing. The affected core can be deactivated and the system has to adapt the schedule to the remaining cores.

There are several criteria to evaluate the schedulability of a certain task set T with a certain scheduling algorithm. According to [13] the schedulability of dynamic scheduling system can be defined as follows:

Definition A: A system has a verified task set T^V if and only if a variation of all properties of the task set in a defined range do not cause a violation of the real-time timing constrains of the system.

If the variation of the task duration differs from the defined range of variation due to the influence of some transient faults the resulting task set is named a disrupted task set.

Definition B: A system has a disrupted task set T^D if the disturbance variable D results in a variation of the properties of the task set which differs from the defined variations of the verified task set.

The disruption of a verified task set can be caused by different kinds of disturbances. This will be discussed in the following section.

A. *Definition of different error types*

Transient hardware faults can raise different kinds of errors in an embedded system. For reason of completeness two important consequences of these transient faults should be mentioned:

- Data errors:
  caused e.g. by bit-flips in registers, memory or ALU and result in corrupted data
- Program flow errors:
  Errors that led to an abnormal program flow

For observing the consequences arising out of faults affecting the scheduling subsystem the actual reason of the fault will be neglected in the following. It is just important that the system has a temporal disruption. At first the error type is not important for the impact on the timing of the task set.

B. *Time dependent classification of transient errors*

Several parts of the scheduling system can be influenced by soft errors. [13] points out three characteristics – kind of timing disruption, duration and intensity of disruption – which are relevant for the scheduling system:

- *Kind of timing disruption*:
  These are the individual properties of the task set influenced by the error. These properties are the duration, the task activation time or the availability of a resource.
- *Duration D_i:*
  The duration mainly denotes the temporal occurrence of errors. Permanent errors (which are not discussed in detail in this paper) and transient errors can be distinguished. Transient faults have a short temporal extend. Consequently they only affect the current running task instance. They are located to one core (in case of a multicore system). Transient errors are defined by the frequency of occurrence (Intensity I).
- *Intensity I:*
  This is the error rate λ_t of the occurrence of transient errors. It is defined as the reciprocal value of the mean time to failure ($MTTF_t$) of transient errors. This value consists of several input parameters which will be discussed later [III.C].

$$I = \lambda_t = \frac{1}{MTTF_t} \tag{1}$$

Additional definitions:

- *Error probability p*:
  With the execution time of the task instance e_i the probability of error occurrence during the execution of a task instance is defined as:

$$p = e_i * \lambda_t \tag{2}$$

- *Time of error occurrence t_e:*
  This is the time the error influences the system during the execution of a task instance. With time of activation a_i and relative task deadline d_i, t_e is defined as:

$$a_i \leq t_e \leq d_i \quad (3)$$

- *Time of error detection t_d:*
  The point in time the scheduler or the safety subsystem detects the error is denoted t_d. The error can be detected during the execution if codded processing is applied to the task instance. In case of diverse execution and a voting system the error can only be detected at the end of the execution of this task.

- *Time of error correction t_c:*
  is defined as the time when the system is back in a failure free state. It has detected the error and the repairing of the affected task instance was successful.

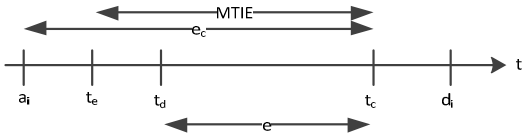

Figure 1. The relevant timing parameters. A error is detected at time t_d and repaired within the task deadline d_i.

- *Mean time in error MTIE:*
  is defined as the time span between error occurrence and successfully applying error repairing actions. It is the sum of the error reaction time and the mean time to repair (MTTR).

$$MTIE = t_c - t_e \quad (4)$$

- Effective Mean Time to Failure $MTTF_E$:
  As stated above a transient fault is only a system error if it is not possible to repair the effects of the faults within the real-time timing constrains, namely the task deadline. Figure 2 illustrates this.

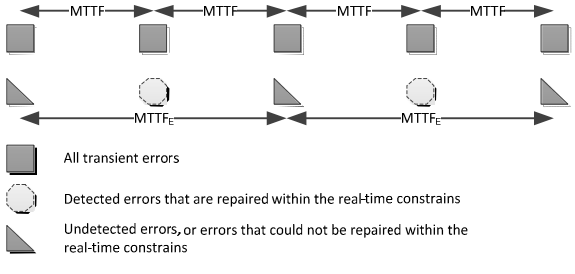

Figure 2. Illustration of $MTTF_E$.

A high diagnostic coverage – which e.g. can be determined by fault injection – is assumed hence the contribution of the undetected dangerous and undetected undangerous errors can be neglected.

C. Resulting timing errors

The named errors and the triggered error repairing actions can cause a disruption of the verified task set T^V due to the needed additional computation time of the error correction. If the disruption variable D exceeds the defined limits, timing errors are the consequence. Possible error handling mechanisms are forward, backward recovery or re-execution of the affected task instance, etc.

The most important errors are deadline errors and jitter in the activation period or finishing period of a task instance.

- *Deadline Errors:*
  A deadline error appears at time t if the overall computation time of the task (e_c) is not completely and correctly executed at time t. The overall task computation time is defined by the task computation time for the failure free execution (e) in addition to the time needed for error correction mechanisms (MTIE).

- *Jitter-Errors:*
  Jitter-Errors like start to start or end to end jitter appear if the activation period or finishing time deviate as a result of transient faults.

Subsequently in the simulation only deadline errors are considered.

III. Safe Real-time-operating system

The possible solution of gaining more reliability in an embedded system is depicted by the use of safety design patterns.

Patterns describe a general way to solve recurring problems. The presented patterns refer to error handling in a safe real-time operating system. All patterns have in common that the adherence of timing constrains and error detection is handled by a central unit, the safety supervisor [16]. It can be configured in different ways as described in the following sections. The safety supervisor is implemented within the safe multicore scheduler (SMS).

Besides N-version programming [11], [16] literature denotes several safety related design pattern [6]. The following depicted design patterns apply to the operating system and the scheduling subsystem.

A. Homogeneous Redundancy Pattern

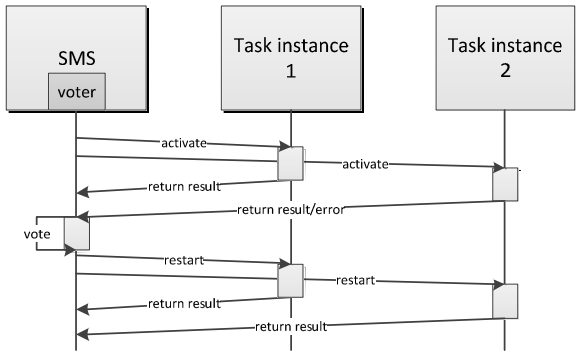

Figure 3. Homogenous Design Pattern applied to the dupplicated task execution in a real-time operting system. An error during the first execution of the two instances is detected by the voter which is integrated in the SMS at the end of the execution period. Then a re-execution of the task instance is triggerd by the SMS.

The same task is duplicated into n task instances which are executed one after another on a single core system (time redundancy) or on different cores on a multicore system (space redundancy). The computation results of the n instances are compared and evaluated after complete execution by the voter integrated in the safety supervisor. In consequence the time of error detection t_d is at least greater than the computation time (e) of a task instance.

B. *Coded Processing Pattern*

In this pattern the computations within the task are transferred to a coded domain. Different coding mechanisms can be applied and several methods of using coded executions can be found in publications. There are approaches of safety compilers [20], safety interpreters or the direct integration of coded processing within the actual code by providing special software libraries [14]. Error detection can be applied during execution. That is a advantage of coded processing. Thus the error detection time t_d is within the execution time (e) of the task instance. Error correcting mechanisms triggered by the SMS can be initiated immediately. Therefore the failure reaction time is shorter compared to homogenous design pattern. Moreover the error detecting probability is higher [21], especially for permanent errors. The disadvantage is the huge overhead of execution time caused by the coded operation.

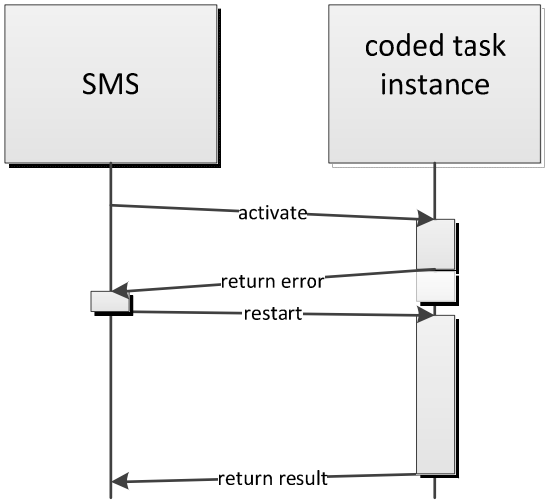

Figure 4. Coded design pattern applied to the task execution in a real-time operating system, errors can be detected during execution, thus the error reaction time can be shorter than in the homogenious redundancy approch.

C. *Influence on the system hazard rate*

In ISO 26262 the Safety-Integrity Level ASIL (A, B, C, D) for the automotive domain is defined. For ASIL C a hazard rate of

$$10^{-8} < \lambda < 10^{-7} = \frac{1}{MTTF} \qquad (5)$$

for the complete embedded system is defined. This rate is composed of the different error rates of the sensors λ_s, actors λ_a and of the micro-controller unit λ_c as well. The controllers' error rate can be further subdivided into the single rate of each component. This can be depicted in a reliability network model.

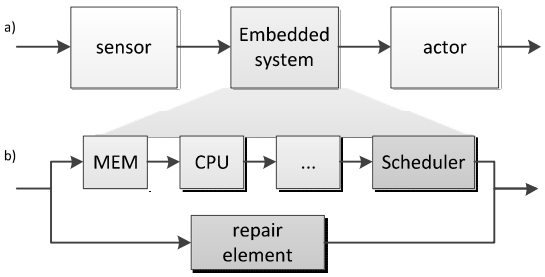

Figure 5. a) Reliability network model of a software intensive real-time embedded system, including sensors and actors.
b) The embedded system can be considered as a serial architecture of elements, consisting of memory, CPU, busses etc. The scheduler represents the timing errors caused by the error handling mechanisms of the schduling subsystem. The error handling / correction software mechanism increase the system reliability and are therfore depicted as a parallel element.

The violation of timing constrains is considered as a soft error. Thus the network model has to be extended by a further serial component, the scheduler integrated in the real-time operating system. The named error repairing mechanisms on the contrary lower the error rate of the whole system. Therefore there is an additional parallel element in the network model. To lower the error rate the SMS has to encounter the error and the error repairing action has to return the system to an error free state without violating any timing constrain.

IV. STOCHASTIC SIMULATION

The reliability indices of a system can be evaluated by using one of two basic approaches. These are direct analytical techniques on the one hand and stochastic simulation on the other [1]. The analytic approach of evaluating a system like a multicore system is usually done by applying a great amount of simplification. The stochastic simulation can represent the real system more accurately. The actual process is simulated by its random behavior. To obtain valuable results several repetitions of the simulation with several randomly generated input properties have to be performed, to reduce the variability of the evaluated results. The basic input of each property is produced by a pseudo random number generator. It is based on a modified version of Donald E. Knuth's subtractive random number generator algorithm [18]. The output of the pseudo random generator is then converted to the appropriate distribution e.g. by the inverse transformation method.

A. *Simulation goals*

In this paper we compare the two approaches of increasing the fault tolerance of a real-time embedded system, homogenous redundancy and coded processing. Criteria are timing characteristics, like $MTTF_E$, deadline violations and repaired errors. The influence of the used type of scheduling algorithm for a single or multicore system is neglected and observed in a second step. The simulation is focused on the impact of the applied faults to the timing characteristics of the task instance.

B. Simulation constraints

While executing a task instance only a single error can occur. The applied error recovery strategy is re-executing the affected task instance as soon as the error is detected. To prevent the system from stucking in an endless loop the maximum amount of re-executions is limited to meet the hard real-time timing constrains, like the task deadline. In case of soft real-time demands, the number of re-executions has to be limited, too. A certain amount of slack is available in the task set.

As mentioned above a hazard rate of

$$10^{-8} < \lambda < 10^{-7} = \frac{1}{MTTF} \qquad (6)$$

for the overall system is stated in the ISO 26262 for ASIL C. According to formula (2) this would result in an error probability of

$$p = \frac{10^{-7}\,{}^{1}/_{h}}{3600\,{}^{s}/_{h}} * 10 * 10^{-3}s = 2.8 * 10^{-14} \qquad (7)$$

for execution duration of a task instance of 10ms. In other words $2.8*10^{14}$ simulation cycles have to be performed to catch one error.
Due to the high computation power needed for such a simulation higher error rates are assumed for the introduced comparison results.

C. Simulation input parameters

With each iteration cycle of performed simulations the error intensity (λ_t) is varied.
The following properties of a task instance are simulated using differently distributed input parameters:

- Error free computation time (e):
  The duration of execution of a single task instance is modeled by a normal distribution.
- Time of error occurrence (t_e):
  In case of a simulated transient error occurs the point of time during the task execution is modeled by an uniform distribution.
- Probability of error occurrence (p):
  This value is modeled by the error rate which is a parameter of each simulation set. It is exponentially distributed.

V. Simulation Results

A task which timing parameters are in the typical range of automotive task sets was used as basis for this simulation. To reduce the variance in the results each set of simulations was executed 100.000 times. The transient mean time to failure $MTTF_t$ was varied with each set of simulations.

A. Simulation setup

The following task instances (see TABLE I) were evaluated during the Monte carlo simulation. For the coded task instance AN-codes were used as coding technique. Because of the assumed high diagnostic coverage each injected transient fault is detected by the SMS. Further transient errors can occur during the re-execution of a task instance. The voter's execution time was neglected.

B. Discussion of the simulation results

With the named assumptions it can be stated that as expected if very high error rates appear both approaches cannot handle the error correcting within the given timing constrains. Moreover the two approaches produced very similar results. The low transient error rate applied in the simulation leads to a relatively high amount of re-executions.

The repair rate – that is the percentage rate of occurred errors that could be repaired within the timing constrains and do not lead to a system error – is higher for the coded processing approach (Figure 6).

The percentage of remaining free execution time (slack) during the task period is better for the homogeneous redundancy as the overall slack is higher in this scenario (Figure 7).

The probability of deadline violations is lower in the coded processing approach at lower error rates because of the higher repair rate. The detecting mechanism has a shorter failure reaction time. But at high error rates this advantage is no use due to the higher computation time of the coded approach. This affects a higher probability of faults during the re-execution (Figure 8).

The better effective mean time in failure is a result of the higher error repair rate and the resulting lower probability of deadline violations (Figure 9). The $MTTF_E$ can be increased to a factor of 50 by applying coded processing.

The simulation results indicate: The slowdown factor of the coded processing has a significant impact on the effectiveness of the codded processing. The elevated execution time leads to higher error probabilities during the execution of the coded task instance compared to the redundant one. With an increased slowdown factor we showed an advantage for the redundant approach. But the coded approach achieves better results in the detection of common cause errors or permanent errors.

VI. Conclusion

In this paper we demonstrated: Provided that only transient faults which cannot be repaired in time cause a system error, the effective mean time in failure can be reduced by applying safety software pattern to single task instances. The stochastic simulation approach offers the possibilities to evaluate the reliability indices of an embedded system as well as the scheduling characteristics and the interaction of both domains.

In future work we will enhance the simulation model to include the error detecting mechanisms and the error detection probability in detail. With regard to the development of new dynamic, robust, multicore scheduling algorithms the holistic stochastic simulation approach offers good prospects to evaluate these concepts.

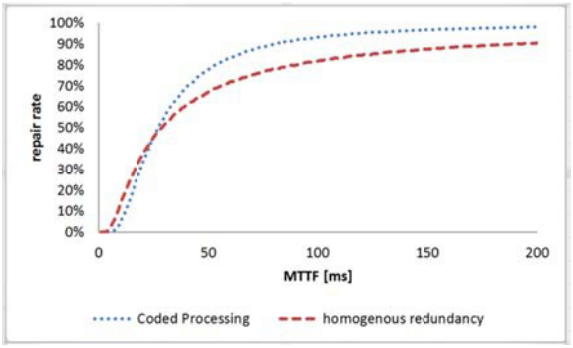

Figure 6. Comparison of rate of successfully repaired transient errors of the task instances, with coded processing and homogenous redundancy.

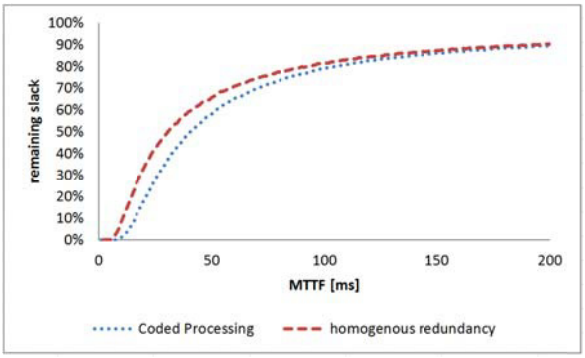

Figure 7. Comparison of the remaining slack – with applied error correcting mechanisms – within the task set related to the initial slack.

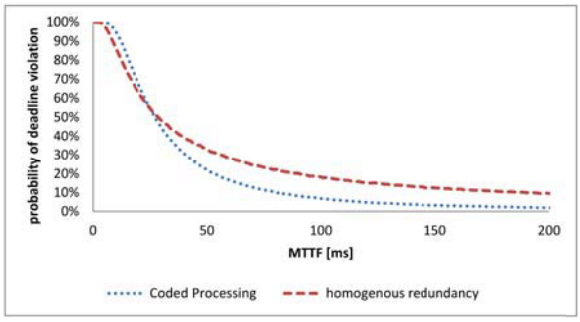

Figure 8. Comparison of the probability of violating the given timing constrains.

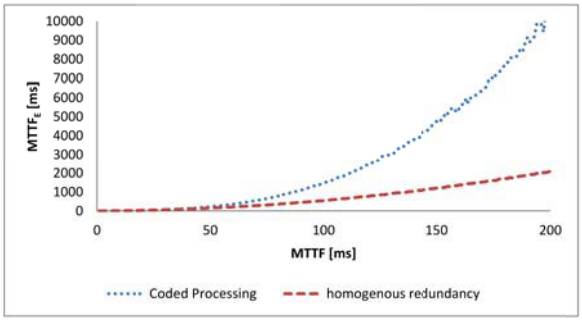

Figure 9. Effective mean time to failure.

TABLE I. MAIN TASK POPERTIES

	Computation time [ms]	**Period [ms]**	**Slack [ms]**
Coded	40	60	20
Redundant	10	60	40

REFERENCES

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

[2] P. Raab, S. Kraemer, and J. Mottok. 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.

[3] M. Cirinei, E. Bini, G. Lipari and A. Ferrari. A flexible scheme for scheduling fault-tolerant real-time tasks on multiprocessors. IEEE, 2007.

[4] H. Beitollahi, S. Miremadi, G. Doconick. Fault-tolerant earliest deadline first scheduling algorithm. IEEE, 2007.

[5] S. Künzli, L. Thiele. Generating event traces based on arrival curves. Measuring, Modelling and Evaluation of Computer and Communication Systems (MMB), 2006.

[6] B. Douglass. Doing hard time. Developing real-time system with UML, objects, frameworks, and patterns. Addison-Wesley, 2007.

[7] Mottok, F. Schiller, Th. Völkl, and Th. Zeitler. A concept for a safe realization of a state machine in embedded automotive applications. In *Proceedings of the 26th Safecomp Conference*, ISBN 978-3-540-75100-7, pages 283-288, 2007.

[8] S. Ross. Introduction to probability models. Academic Press, 2003.

[9] C. Han, K. Shin, J. Wu, A fault-tolerant scheduling algorithm for real-time perodic tasks with possible faults, IEEE TRANSACTIONS ON COMPUTERS, VOL. 52, NO. 3, 2003.

[10] S. Ghosh, R. Melhem, D. Mossé and J. Sarma. Fault-tolerant rate-monolitic scheduling. Kluwer Academic Publishers, pages 149 – 181, 1998.

[11] L. Yang, Z. Cui and X. Li. A case study for fault tolerance oriented programming in multi-core architecture. IEEE computer society, pages 630 – 635, 2009.

[12] E. Beckschulze, F. Salewski, T. Siegbert and S. Kowalewski. Fault handling approaches on dual-core microcontrollers in safety-critical automotive applications. CCIS 17, pages 82 – 92, 2008.

[13] M. Deubzer, J. Mottok, and A. Baerwald. Dependability-Betrachtung von Multicore-Scheduling. *HANSER Automotive*, November 2010.

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

[15] G. Buttazzo. Hard real-time computing systems – Predictable Scheduling Algorithms and Applications. Springer, 2004

[16] N. Leveson. Safeware: System Safety and Computers. Addison-Wesley, 1995.

[17] F. Schiller, J. Mottok, M. Blum, F. Duckstein, R. Egen, M. Hummel, and T. Mattes. Generische Safety-Architektur für KFZ-Software. *Hanser automotive*, pages 52-54, November 2006.

[18] D. E. Knuth. The Art of Computer Programming, volume 2: Seminumerical Algorithms. Addison-Wesley, Reading, MA, second edition, 1981.

[19] A. Burns, R. Davis, and S. Punnekkat. Feasibility Analysis of Fault-Tolerant Real-time Task Sets. In Proceedings of EURWRTS, 1996

[20] U. Schiffel, M. Süßkraut, and C. Fetzer. AN-Encoding Compiler: Building Safety-Critical Systems with Commodity Hardware. in The 28th International Conference on Computer Safety, Reliability and Security (SafeComp 2009), 2009.

[21] U. Schiffel, A. Schmitt, M. Süßkraut, and C. Fetzer. Software-Implemented Hardware Error Detection: Costs and Gains. Third International Conference on Dependability, 2010