

---

# Flexible Error Handling for Embedded Real-Time Systems

– Operating System and Run-Time Aspects –

---

## Dissertation

zur Erlangung des Grades eines

DOKTORS DER INGENIEURWISSENSCHAFTEN

der Technischen Universität Dortmund

an der Fakultät für Informatik

von

Andreas Heinig

Dortmund

2015

Tag der mündlichen Prüfung: 30. April 2015  
**Dekan / Dekanin:** Prof. Dr. Gernot A. Fink  
Gutachter / Gutachterinnen: Prof. Dr. Peter Marwedel  
Prof. Dr. Hermann Härtig (TU Dresden)

# Acknowledgments

I would like to thank my advisor Prof. Peter Marwedel for the initial ideas that lead to this dissertation. Without his advises, writing this thesis in this form would not be feasible. Very special thanks go to Dr. Michael Engel for all the discussions, the advices he gave over the years, and his intensive proof-reading support of this thesis and my publications.

I would also like to thank my collaborating colleagues at the Department of Computer Science 12 with special thanks to Florian Schmoll. He provided the compiler support which is required by the flexible error handling approach presented in this thesis. Furthermore, he was available for high valuable technical discussions. Thanks go also to Björn Bönninghoff for providing the timing model for the H.264 decoder and to Ingo Korb for providing various bug fixes for the H.264 library. For proof-reading, I would like to thank Helana Kotthaus, Olaf Neugebauer, Florian Schmoll, and Björn Bönninghoff.

My research visit at NTU Singapore was enabled by Prof. Dr. Vincent Mooney and Prof. Dr. Krishna Palem. I like to thank both for the advices that led to a publication which received a best paper award.

Additional thanks go to the German Research Foundation (DFG) for supporting the FEHLER-Project (Grant No. MA-943/10) as part of the Priority Programme SPP1500 which leads to this thesis. The initial parts of this dissertation were supported by the European Community as part of the Artist Design Network of Excellence EU FP7 ICT Grant No. 39316.

Finally, I would like to thank my mother. Without her continual support and encouragement, I would never have made it this far.



# Abstract

Due to advancements of semiconductor fabrication that lead to shrinking geometries and lowered supply voltages of semiconductor devices, transient fault rates will increase significantly for future semiconductor generations [Int13]. To cope with transient faults, error detection and correction is mandatory. However, additional resources are required for their implementation. This is a serious problem in embedded systems development since embedded systems possess only a limited number of resources, like processing time, memory, and energy. To cope with this problem, a software-based flexible error handling approach is proposed in this dissertation. The goal of flexible error handling is to decide **if**, **how**, and **when** errors have to be corrected. By applying this approach, deadline misses will be reduced by up to 97% for the considered video decoding benchmark. Furthermore, it will be shown that the approach is able to cope with very high error rates of nearly 50 errors per second.



# Publications

Parts of this thesis have been published in proceedings of the following conferences and workshops (in chronological order):

1. A. Heinig, M. Engel, F. Schmoll, and P. Marwedel, “Improving transient memory fault resilience of an H.264 decoder,” in Proc. of Embedded Systems for Real-Time Multimedia (ESTIMedia), pp. 121–130, 2010.
2. A. Heinig, M. Engel, F. Schmoll, and P. Marwedel, “Using application knowledge to improve embedded systems dependability,” in Proc. of HotDep, Oct. 2010.
3. A. Heinig, “R2G: Supporting POSIX like semantics in a distributed RTEMS system,” Technical Reports in Computer Science, no. 836, TU Dortmund, Faculty of Computer Science, Dec. 2010.
4. M. Engel, F. Schmoll, A. Heinig, and P. Marwedel, “Unreliable yet useful – reliability annotations for data in cyber-physical systems,” in Proc. of Workshop on Software Language Engineering for Cyber-physical Systems (WS4C), 2011.
5. D. Cordes, A. Heinig, P. Marwedel, and A. Mallik, “Automatic Extraction of Pipeline Parallelism for Embedded Software Using Linear Programming,” in Proc. of Parallel and Distributed Systems (ICPADS), 2011, pp. 699–706.
6. M. Engel, A. Heinig, F. Schmoll, and P. Marwedel, “Temporal properties of error handling for multimedia applications,” in Proc. of Conference on Electronic Media Technology (CEMT), 2011, pp. 1–6.
7. A. Heinig, V. J. Mooney, F. Schmoll, P. Marwedel, K. V. Palem, and M. Engel, “Classification-Based improvement of application robustness and quality of service in probabilistic computer systems,” in Proc. of international conference on Architecture of Computing Systems (ARCS), 2012. – **Best Paper Award**
8. A. Heinig, I. Korb, F. Schmoll, P. Marwedel, and M. Engel, “Fast and Low-Cost Instruction-Aware Fault Injection,” in Proc. of Software-Based Methods for Robust Embedded Systems (SOBRES), 2013.
9. F. Schmoll, A. Heinig, P. Marwedel, and M. Engel, “Improving the fault resilience of an H.264 decoder using static analysis methods,” Transactions on Embedded Computing Systems (TECS), vol. 13, no. 1, pp. 1–27, Nov. 2013.

10. F. Schmoll, A. Heinig, P. Marwedel, and M. Engel, "Passing error handling information from a compiler to runtime components," Technical Reports in Computer Science, no. 844, TU Dortmund, Faculty of Computer Science, 2014.
11. A. Heinig, F. Schmoll, P. Marwedel, and M. Engel, "Who's using that memory? A subscriber model for mapping errors to tasks," in Proc of Workshop on Silicon Errors in Logic - System Effects (SELSE), Stanford, 2014.
12. A. Heinig, F. Schmoll, B. Bönninghoff, P. Marwedel, and M. Engel, "FAME: Flexible Real-Time Aware Error Correction by Combining Application Knowledge and Run-Time Information," in Proc of Workshop on Silicon Errors in Logic - System Effects (SELSE), Austin, 2015.



# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Motivation . . . . .	3
1.2	Faults, Errors, and Failures . . . . .	3
1.2.1	Masking . . . . .	4
1.2.2	Duration . . . . .	5
1.3	Embedded Real-Time Systems . . . . .	5
1.3.1	Real-Time . . . . .	5
1.3.2	Efficiency . . . . .	7
1.3.3	Dependability . . . . .	7
1.4	Contribution of this Work . . . . .	8
1.5	Outline . . . . .	10
1.6	Author's Contribution to this Dissertation . . . . .	10
<b>2</b>	<b>Related Work</b>	<b>13</b>
2.1	Fault Tolerance . . . . .	13
2.2	Reliability in Real-Time Operating Systems . . . . .	16
2.3	Checkpoint and Recovery . . . . .	19
2.4	Fault Injection . . . . .	21
2.5	Scheduling of Real-Time Tasks . . . . .	25
2.6	Timing Estimations . . . . .	27
<b>3</b>	<b>Models and Infrastructure</b>	<b>31</b>
3.1	Reliable Computing Base . . . . .	31
3.2	RAP Model . . . . .	32
3.3	Transient Memory Fault Model . . . . .	34
3.3.1	Thread-Based Fault Injection . . . . .	35
3.3.2	Fault Injection in CoMET . . . . .	35
3.3.3	Injection into Demonstrator Platform . . . . .	37
3.4	Probabilistic Error Model . . . . .	42
3.4.1	Supply Voltage Schemes . . . . .	43
3.4.2	Three-Stage Model . . . . .	43
3.4.3	Probabilistic ARM Platform . . . . .	47
3.5	Task Model . . . . .	48
3.6	Quality of Service Metrics . . . . .	48
3.6.1	$\Delta E$ . . . . .	49
3.6.2	Peak Signal-To-Noise Ratio . . . . .	49

<b>4</b>	<b>Flexible Error Handling</b>	<b>51</b>
4.1	Flexible Error Handling Strategy . . . . .	51
4.2	Error Classification . . . . .	53
4.2.1	H.264 Application Analysis . . . . .	54
4.2.2	LEGO Mindstorms Application . . . . .	56
4.2.3	Further Examples . . . . .	59
4.2.4	Classification Annotation . . . . .	59
4.2.5	Implications . . . . .	61
4.3	Real-Time Aspect . . . . .	62
4.3.1	A kind of Priority Inversion Problem . . . . .	63
4.3.2	Subscriber Model . . . . .	63
4.3.3	Using the Subscriber Model to Schedule Error Correction . . . . .	67
4.4	Summary . . . . .	69
<b>5</b>	<b>FAME: Fault Aware Microvisor Ecosystem</b>	<b>71</b>
5.1	REPAIR Compiler . . . . .	72
5.1.1	Classification Parsing . . . . .	73
5.1.2	Subscriber Model Support . . . . .	75
5.1.3	Code Generation . . . . .	76
5.1.4	Summary . . . . .	77
5.2	Runtime Overview . . . . .	77
5.3	Microvisor . . . . .	78
5.3.1	RCB Components . . . . .	79
5.3.2	Virtualization Concepts . . . . .	79
5.3.3	Memory Management . . . . .	85
5.3.4	Subscriber Model Support and Data Object Management . . . . .	88
5.3.5	Checkpoint-and-Recovery . . . . .	89
5.3.6	Summary . . . . .	92
5.4	FAMERE . . . . .	93
5.4.1	Para-Virtualization Support . . . . .	93
5.4.2	Checkpoint-and-Recovery . . . . .	94
5.4.3	Scheduling and Subscriber Model Support . . . . .	94
5.4.4	Memory Allocation . . . . .	95
5.4.5	Error Handling . . . . .	95
5.4.6	Summary . . . . .	95
5.5	Para-Virtualizing RTEMS for FAME . . . . .	96
5.5.1	RTEMS Overview . . . . .	96
5.5.2	Board Support Package and Infrastructure . . . . .	97
5.5.3	Context Switching . . . . .	97
5.5.4	Memory Management . . . . .	98
5.5.5	Summary . . . . .	100
5.6	Flexible Error Handling - Realization . . . . .	101
5.6.1	Stage 1: Low-Level Error Handling . . . . .	102
5.6.2	Stage 2: Mapping Errors to Tasks . . . . .	102

5.6.3	Stage 3: Classification and Handling Method Selection . . . .	103
5.6.4	Stage 4: Error Correction and Acknowledgment . . . . .	103
5.7	Summary . . . . .	104
<b>6</b>	<b>Demonstrator: H.264 Video Decoder Application</b>	<b>105</b>
6.1	H.264 Decoder Library – <i>libh264</i> . . . . .	105
6.2	Simple Decoder Application . . . . .	106
6.3	Real-Time Decoder Application . . . . .	106
6.3.1	Timing . . . . .	107
6.3.2	Scheduling . . . . .	110
6.3.3	RTEMS Modifications . . . . .	112
6.4	Summary . . . . .	113
<b>7</b>	<b>Evaluation</b>	<b>115</b>
7.1	Evaluation Platforms . . . . .	116
7.1.1	CoMET-based ARM926 Platform . . . . .	116
7.1.2	MPARM-based ARM7 Platform . . . . .	117
7.2	QoS Impact of Classification . . . . .	117
7.2.1	Transient Errors in Memory . . . . .	118
7.2.2	Probabilistic Error Model . . . . .	120
7.2.3	Summary . . . . .	123
7.3	Overhead . . . . .	124
7.3.1	Interrupt Latency . . . . .	124
7.3.2	Hypercalls . . . . .	125
7.3.3	Timing . . . . .	126
7.3.4	Summary . . . . .	127
7.4	Checkpointing . . . . .	128
7.4.1	Encoder Type and Implementation . . . . .	128
7.4.2	Block Size . . . . .	130
7.4.3	Reducing the Number of Data Objects . . . . .	130
7.4.4	Summary . . . . .	132
7.5	Flexible Error Handling and Transient Faults . . . . .	132
7.5.1	Naive Error Handling . . . . .	133
7.5.2	Flexible Error Handling . . . . .	134
7.5.3	Flexible Error Handling with Application Specific Error Cor- rection Methods . . . . .	135
7.5.4	Quality of Service Impact of Flexible Error Handling . . . . .	135
7.5.5	Summary . . . . .	137
7.6	Summary of Evaluation . . . . .	137
<b>8</b>	<b>Conclusions and Future Work</b>	<b>139</b>
<b>A</b>	<b>Hypercall Table</b>	<b>143</b>
	<b>Bibliography</b>	<b>145</b>

---

List of Figures	155
List of Tables	157
List of Code Listings	158
List of Abbreviations	159
Index	161

# Introduction

---

## Contents

<b>1.1</b>	<b>Motivation</b>	<b>3</b>
<b>1.2</b>	<b>Faults, Errors, and Failures</b>	<b>3</b>
1.2.1	Masking	4
1.2.2	Duration	5
<b>1.3</b>	<b>Embedded Real-Time Systems</b>	<b>5</b>
1.3.1	Real-Time	5
1.3.2	Efficiency	7
1.3.3	Dependability	7
<b>1.4</b>	<b>Contribution of this Work</b>	<b>8</b>
<b>1.5</b>	<b>Outline</b>	<b>10</b>
<b>1.6</b>	<b>Author's Contribution to this Dissertation</b>	<b>10</b>

---

In previous decades Information Technology (IT) was used in the context of business, enterprise, or government to store, retrieve, transmit, and manipulate data by applying computers and telecommunication equipment. In such contexts, computers typically are operated as large mainframes. With the invention of the Personal Computer (PC) in the nineties, IT becomes available to a vast portion of private individuals. Nowadays, IT is pervasive in our daily life. Information is available anytime and anywhere (ubiquitous computing). The computer systems enabling ubiquitous computing accompany the user permanently, either as part of systems encountered every day or as small portable devices carried around. Due to ongoing miniaturization, small portable computer devices and also wearable computers are available. Very prominent examples are smart phones and Google Glass [Gla], respectively. While PCs and smart phones can be easily noticed as computing systems, integrated products are less obvious to discover. If information processing systems are integrated into enclosing products, they are called embedded systems [Mar11]. Huge application domains for embedded systems exist in the field of telecommunication, consumer electronics, automotive electronics, railways, avionics, robotics, logistics, security, health sector, fabrication equipment, smart buildings, and military equipment. Besides the functional requirements, embedded systems often have to fulfill non-functional requirements too. An example of a very important non-functional requirement is dependability, especially in areas where humans can be injured.

Since the beginning of IT, the demand for computing power is steadily increasing. For example, in the entertainment and multimedia area, video codecs become more and more complex and video resolutions are getting higher and higher. Television broadcasting is currently preparing 8K UHD TV (rec. ITR-T BR.2020 [Int12]). This will result in sixteen times as many pixels as in current 1080p HDTV. Furthermore, UHD TV allows for frame rates of up to 120 fps (frames per second). To cope with the resulting high data rates, faster interconnects and/or more efficient video codecs are required. Fortunately, advances in semiconductor fabrications allows for production of integrated circuits (ICs) with more and more computation power. According to *Moore's Law*, the number of transistors in dense ICs will roughly double every 18 months [Moo65]. Moore's Law is not a real law of nature. It is rather a prediction on the basis of observations. Until now, however, the prediction is accurate. The doubling of the amount of transistors can only happen due to advancements in semiconductor fabrication. If current integrated circuits had been fabricated with the same technology as 10 years ago, the required chip area would have been  $2^{10}$  times larger. This means a chip with an edge length of 10 mm would have grown to a chip with an edge length of 320 mm which will be infeasible. On the one hand, there are performance penalties. Signals have to travel very long distances leading to increased signal delay, and hence lead to reduced clock speed. Furthermore, power consumption will rise significantly which also increases the emitted heat of systems. On the other hand, the huge dimension of a single integrated circuit (IC) would lead to fabrication problems. Only a few chips would fit onto one wafer. Consequently, the production will be very expensive.

However, due to the ongoing trend of shrinking semiconductor devices, the structure sizes are getting smaller and smaller. In 2004 the 90 nm process technology started. Now, ten years later, 14 nm is introduced. Shrinking of the structure sizes is not only advantageous in terms of required chip area. It also allows for lowered supply voltages. Consequently, energy consumption is reduced as well. However, due to smaller structure sizes and lowered supply voltages, less charge is stored in the capacities of the circuit and the distance between the voltage levels representing a '0' and a '1' is getting smaller. Hence, devices become more and more susceptible to external disturbances. According to forecasts of the International Technology Roadmap for Semiconductors (ITRS) [Int13], future semiconductors will be susceptible to transient faults to a much higher degree than current devices. Transient faults are single-shot phenomena that can lead to unintended status changes in components of a system. They are induced, for example, by natural radioactive decay, high energy cosmic particles, disturbance in supply voltages, electromagnetic interferences, or overheating. Due to the stochastic characteristics of their sources, faults affect components of a system in an unpredictable moment in time. Fortunately, transient faults can be corrected. They persist only until the next status change of the affected component. Increasing the resilience against transient faults is the target of this dissertation.

## 1.1 Motivation

Resilience against transient faults was traditionally only a concern for computer systems running in harsh environments or in fields where failures can lead to severe damages. Due to advancements of semiconductor fabrication that lead to shrinking geometries and lowered supply voltages of semiconductor devices, the rates of transient faults will increase significantly for future semiconductor generations. Hence, resilience against transient faults will become an issue also for everyday computing. To cope with transient faults, error detection and correction (EDAC) will be mandatory. However, EDAC does not come for free. Typically, additional resources are required for the implementation. This is a serious problem, especially in embedded systems development. On the one hand, transient faults have to be handled, and on the other hand, embedded real-time systems possess only a limited number of resources, like processing power, memory, and energy. If not enough sparse resources for EDAC are available, deadline misses will occur. The goal of this thesis is to reduce the amount of resources required for error correction while keeping the number of deadline misses low. To achieve this goal, a software-based flexible error handling strategy is introduced. This strategy is responsible to correct already detected errors. Hence, error detection has to be provided by other components. Error detection, however, is not within of the scope of this thesis.

The basic idea of flexible error handling is to decide **if**, **when**, and **how** errors have to be handled. This provides the necessary flexibility not available in typical EDAC systems where every error is handled alike. When timing constraints of a system do not allow correcting every occurring error, errors which are classified to lead to no or negligible impact can be *safely* ignored. In contrast, errors with *fatal* impact have to be corrected. Application knowledge is required to create such a classification and to decide whether or not an error has to be handled. This knowledge can be gathered from source code annotations as well as static source code analysis. Flexible error handling is hard to implemented in hardware, since hardware has no knowledge of the impact of an error on the application. Consequently, a software-based approach is required. With the *Fault Aware Microvisor Ecosystem (FAME)* a tool flow will be presented which automatically extracts the application knowledge to generate a classification of possible error impacts. *FAME* also provides a runtime system which is able to apply the classification.

The flexible error handling approach as well as its implementation (*FAME*) will be discussed in detail in Chapter 4 and Chapter 5, respectively. In the next section the terms fault, error, and failures are defined. Afterwards, some important aspects of embedded systems are detailed which are often stressed in this thesis.

## 1.2 Faults, Errors, and Failures

Sorin describes the terms faults, errors, and failures as follows:

*“We consider a **fault** to be a physical flaw, such as a broken wire or a*

*transistor with a gate oxide that has broken down. A fault can manifest itself as an **error**, such as a bit that is a zero instead of a one, or the effect of the fault can be masked and not manifest itself as any error. Similarly, an error can be masked or it can result in a user-visible incorrect behavior called a **failure**. Failures include incorrect computations and system hangs.” (Sorin in [Sor09])*

In these definitions the term “user-visible” means the software running on top of the hardware. The view point of Sorin is hardware-centric. Hence, the previous definitions are well suited if only hardware will be considered. In this thesis, however, the focus lies on software based fault handling. The software stack typically consists of several layers, like, e.g., device drivers, file system, scheduling, system libraries, and applications. Hence, the definitions have to be adopted to consider the currently investigated layer.

**Definition 1.1 (Fault)** *A fault is a phenomenon affecting the layer which is below the current layer. If the current layer is already the lowest layer, then the fault will be the physical effect.*

**Definition 1.2 (Error)** *If a fault is not masked at the lower layer, it will manifest itself as an error in the current layer.*

**Definition 1.3 (Failure)** *If the error renders the system useless at the current layer, the error will be called failure. The error will become also a failure if it propagates to the next layer.*

With Definitions 1.1, 1.2, and 1.3 a fault can ripple through the hardware layers and software stack as an error. If no layer is able to mask – or to correct – the error, the user will notice the error as failure in form of an incorrect computation or as system crash.

### 1.2.1 Masking

Masking of faults and errors can happen in various layers. *Logical masking* happens in combinational logic. If, for example, one of the two inputs of an OR gate is affected by a fault while the other input is logical '1', the fault will be masked. Faults in the *register transfer level* can be masked, if the fault event occurs outside a clock edge and a D flip-flop is used to store the result, for example. *Arithmetical masking* can be the result of an operation considering not all bits within a register, like a right shift operation. If an instruction does not use every unit of a processor, *architectural masking* can happen. For example, faults in the instruction decoding for the source and destination register of a NOP operation have no impact on the correct execution. Last but not least, masking can also occur at the *application level*. An example is a fault affecting a memory region, which is not used by the application.



### 1.2.2 Duration

Faults can be categorized according to their duration. A *permanent fault* occurs at some point in time, and persists until the faulty component is replaced. To cope with permanent faults, the ability to avoid using the affected component is required. Usually, this means to use a fault free spare component as replacement. In contrast to permanent faults, *transient faults* can be repaired by (re-)setting the affected component to a valid state. Transient faults are also called soft errors or single event upsets (SEUs).

## 1.3 Embedded Real-Time Systems

**Definition 1.4 (Embedded System)** “*Embedded systems are information processing systems embedded into enclosing products.*” (Marwedel in [Mar11])

Examples for embedded systems include devices used in telecommunication, planes, trains, cars, or fabrication equipment. In such areas of application, the link to physics and physical systems is very important. To stress the link even more, the term cyber-physical system (CPS) was introduced:

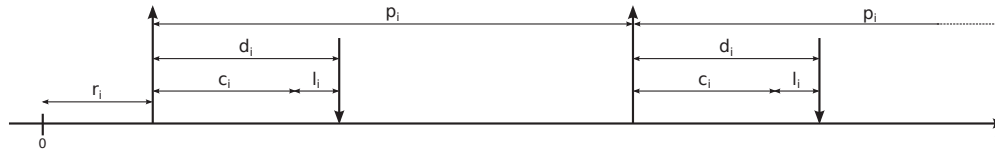
**Definition 1.5 (CPS)** “*Cyber-physical systems are integrations of computation and physical processes.*” (Lee in [Lee07])

Cyber-physical systems describe a subset of embedded systems. Entertainment systems, such as smart TVs, are embedded systems but not cyber-physical systems. In contrast, the controller of an anti-lock braking system (ABS) is an embedded system as well as a cyber-physical system. In the remainder of this thesis the term CPS is used whenever the link to physics should be stressed.

Embedded systems have important non-functional characteristics. The three most important characteristics for this thesis – real-time, efficiency, and dependability – are depicted in the following subsections.

### 1.3.1 Real-Time

In non-real-time systems, only the correct result of a computation is important. In embedded real-time systems, however, the correctness of a result depends not only on the logical result, but on the time at which the result is produced as well. The latest possible point in time for a valid result is called *deadline*. In hard real-time systems, every result delivered after the deadline passed is considered wrong. Contrary, soft real-time systems can cope with results delivered too late. For example, in a video application, a frame displayed too late would only lead to jitter. Embedded systems and especially cyber-physical systems inherently require timing correctness. If a robot is controlled, a too late calculated stop command can have fatal consequences. For example, the robot might collide with another object.



**Figure 1.1:** Definition of laxity, period, and activation time of a task

A typical embedded system comprises periodic real-time tasks which are scheduled according to the timing parameters. Figure 1.1 depicts a comprehensive overview of important times associated with each real-time task  $T_i$ . Arrows pointing upwards denote the point in time at which tasks become available. Deadlines are marked by downward pointing arrows. The *deadline interval*  $d_i$  is the time between the task's release time and its deadline. The time between start of the system and the first activation of  $T_i$  is called *release time*  $r_i$ . In embedded systems, tasks are typically executed periodically, to perform the same job, like, for example, updating the steering command of an actuator, in previously defined intervals. The duration between two consecutive activations is called *period*  $p_i$ . The time in which a task is using the processor is called *execution time*  $c_i$ . The *laxity* or *slack*  $l_i$  is the difference of the deadline and the execution time:  $l_i = d_i - c_i$ . If  $l_i = 0$ , the task  $T_i$  has to be started as soon as it becomes ready for execution. Otherwise, deadline misses will occur.

The knowledge of a task's execution time or more precisely the *worst case execution time* (*WCET*) is often required to schedule tasks in embedded real-time systems. Some definitions related to WCET are visualized in Figure 1.2. The WCET is extremely hard to compute since it is the largest execution time of a program for any initial execution state and any input. Due to the undecidability whether or not a program terminates, the WCET is in general undecidable. However, for certain programs which, for example, are without recursion and contain only loops with statically known iteration counts, calculation of the WCET is possible. Unfortunately, even under such constraints the calculation will be challenging since the behavior of modern processors is hard to predict at design time. Such systems contain caches, deep pipelines, interrupts, and virtual memory with Memory Management Unit (MMU) and Translation Lookaside Buffers (TLBs) which are all hard to predict. Consequently, the WCET has to be estimated ( $WCET_{EST}$ ) by computing a good *upper bound*. Two properties of such a bound are very important. On the one hand, the upper bound has to be safe ( $WCET_{EST} \geq WCET$ ). On the other hand, the upper bound has to be tight ( $WCET_{EST} - WCET \ll WCET$ ).

The *best case execution time* (*BCET*) and  $BCET_{EST}$  are defined analogously to WCET and  $WCET_{EST}$ , respectively. Timing analyses to calculate WCETs or BCETs are out of the scope of this dissertation. However, timing estimations are required later to schedule the used real-time applications.

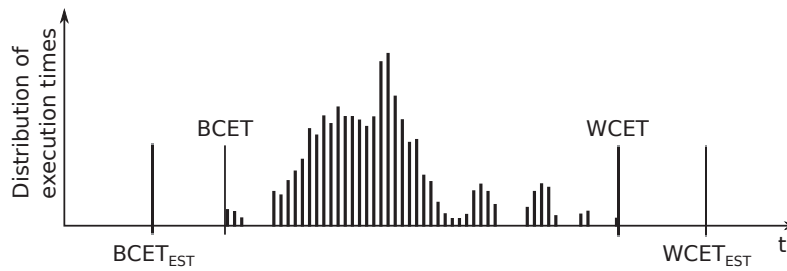


Figure 1.2: WCET-related terms (based on [Mar11])

### 1.3.2 Efficiency

Embedded systems have to be efficient in many ways:

- **Energy:** Energy consumption is an important factor in embedded systems. A huge amount of embedded systems are running on battery or gain their power via energy harvesting. Hence, the available energy is limited. For example, nobody would spend money for a smart phone which has to be recharged every hour due to high energy consumption.
- **Weight:** Reduced weight is very beneficial for cyber-physical systems. Especially, if the CPS is integrated into devices, like cars, air planes, or satellites. Every saved gram also conserves energy or fuel required for locomotion.
- **Code size:** A typical hardware platform for an embedded system is a system on a chip (SoC). In a SoC, the processing circuits as well as instruction memories (typically ROM) are integrated into one single chip. The capabilities of such a chip are limited and dynamic reloading of additional binary code is very unusual in embedded systems. Hence, the footprint of the embedded software has to be small.
- **Run-time efficiency:** Run-time efficiency basically means to exploit the underlying hardware resources as good as possible.
- **Cost:** Cost efficiency is mainly a driving factor for embedded systems produced in high volumes, like consumer electronics. In order to reduce costs, clock frequencies as well as supply voltages should be as low as possible, since this would, for example, reduce energy consumption which allows integrating a cheaper power supply. Furthermore, only required hardware components should be integrated in the circuits to reduce packaging costs.

### 1.3.3 Dependability

Dependability comprises many aspects, such as *safety*, *security*, *availability*, *maintainability*, and *reliability*. Safety expresses that a system must not cause harm.

Security guarantees authentic communication and that confidential data stay confidential. Availability and maintainability describe the probability that a system is available and that a failing system can be repaired within a given time-frame, respectively. Last but not least, reliability expresses the probability of mission survival. This includes, for example, the resistance against failures.

Unfortunately, efficiency and dependability are contradicting goals. For example, to establish a secure communication, extra program code responsible for encryption is required. To increase reliability redundant resources are required, since fault tolerance methods are inherently based on some kind of redundancy (temporal or spatial). Typically, a larger amount of redundant resources allows correcting a higher number of errors. If maintaining real-time properties of an embedded system has the highest priority, sufficient redundancy has to be available to handle every possible error without missing a deadline.

The remainder of this thesis shows how to reduce the amount of redundant resources while keeping the number of correctable errors high.

## 1.4 Contribution of this Work

Conventional error correction methods are not capable to consider application knowledge to apply selective correction. Instead, every error is handled alike. To overcome this problem, this dissertation proposes a flexible error handling approach. The following contributions are made:

- **Applicability for probabilistic as well as transient memory fault models:** The proposed flexible error handling approach is theoretically applicable to nearly every possible transient fault model. Only working error detection schemes as well as some guaranteed fault free hardware components are required. To demonstrate a wide range of applicability, a probabilistic as well as a transient memory fault model are considered.
- **Efficient fault injection techniques:** In this thesis different approaches to inject faults will be introduced to evaluate the developed methods. The injection techniques range from injection via a thread running in parallel over injection on system simulator basis to injection of faults in real systems. The first techniques have nearly zero overhead. For the latter technique significant overhead reductions are demonstrated.
- **Error classification according to impact on application:** Flexible error handling is based on the observation that errors can have different impacts on the application. The variety of impacts is shown using a multimedia and a robotic application. To specify the error impact, data have to be annotated. The easiest way to annotate data is to add reliability qualifiers to variable declarations in the application source code. Therefore, the qualifiers *reliable* and *unreliable* are used which specify that any possible error affecting

the annotated data can have severe impact or negligible impact, respectively. Negligible in this context means that an error cannot lead to system crashes. The impact on the output quality is not a major concern. The primary goal of applying these type qualifiers is to avoid crashes.

- **Delayed error handling:** Delayed error handling is advantageous in situations where the available time used for error correction is scarce. Sometimes the correction can be delayed to a later point in time when more time is available for error correction. Hence, higher prioritized tasks will not be interrupted.
- **Introduction of a new programming model for fine-grained resource mapping:** To determine which tasks are affected by an error a fine-grained resource mapping is required. In a typical operating system, however, resources are managed on a per-process basis. This is too coarse-grained since a process can contain many tasks. Instead, a subscriber-based model to map resources to tasks is introduced in this thesis. The subscriber model allows for mapping resources on the basis of tasks. Moreover, it also helps the operating system to determine whether or not a resource is currently used by any task.
- **Significant reduction of redundant resources:** By considering the subscriber model also for checkpoint creation, the checkpoint size as well as the checkpoint creation time is reduced. If the reliability annotation of data is also taken into account, additional reductions can be observed.
- **Para-virtualization to enable fault handling in software:** To avoid propagation of errors to important software components, virtualization is a key concept for isolation. The error detection mechanisms are also shielded by virtualization. Hence, the guest OS or the application are unable to (accidentally) deactivate error detection. The virtualization overhead is reduced by tailoring the applied virtualization techniques to the demands of reliability and embedded systems.
- **Improvement of real-time behavior under high fault rates:** Due to the low virtualization overhead, the impact on the real-time properties of the overall system is minimized. Furthermore, the applied techniques can handle very high effective fault rates of nearly 50 faults per second. Evaluation shows that such high fault rates can only be dealt with when all developed methods of this dissertation are enabled.
- **FAME:** All the previously mentioned contributions are integrated in one single ecosystem called *Fault Aware Microvisor Ecosystem (FAME)*.

Embedded real-time systems are in the focus of this dissertation. However, most of the developed methods are not constrained to embedded systems only and, hence, can be applied to other computer systems as well.

## 1.5 Outline

This thesis is structured as follows. In Chapter 2 related work is shown. Afterwards, different faults models and their implementation are depicted in Chapter 3. In Chapter 4 the flexible error handling approach is discussed. FAME is presented in Chapter 5. It is the ecosystem implementing the flexible error handling approach presented in Chapter 4. The main demonstrator application, an H.264 video decoder, is detailed in Chapter 6. Thereafter, this application is used for evaluation in Chapter 7. In Chapter 8 concluding remarks and hints for future work are given.

## 1.6 Author's Contribution to this Dissertation

According to §10(2) of the “Promotionsordnung der Fakultät für Informatik der Technischen Universität Dortmund vom 29. August 2011”, a Ph.D. dissertation requires a listing of research results achieved in cooperation.

**Chapter 3:** The probabilistic fault model used in Chapter 3 is based on research of Prof. Dr. Krishna Palem and Prof. Dr. Vincent Mooney. The integration into the MPARM[BBB+05] simulator, the operating system support, as well as porting the application that lead to publication [HMS+12] is the work of the author. The publication was entirely designed by the author of this dissertation. The co-authors assisted the author in writing the paper. All compiler integrations used in the paper are developed by Florian Schmoll.

In [HKS+13] a low-cost fault injection approach is presented. Ingo Korb developed the low-level JTAG implementation used for the communication between target and injection board. The fault injection algorithm as well as the paper is written by the author of this thesis. Again, the other co-authors assisted in writing the paper.

Ingo Korb also assisted integrating the transient error model for bit flips into the CoMET/METeor simulator.

**Chapter 4:** The general idea to flexibly handle errors in embedded real-time system was contributed by Prof. Dr. Peter Marwedel and Dr. Michael Engel.

Different error impact classes of an H.264 video decoder are shown in [HES+10a]. The co-authors, especially Dr. Michael Engel, assisted very much in writing of the paper. However, the application analysis as well as the evaluation was made solely by the author of this thesis. The error impact analyses of a robotic application was created by Dennis Nahberger in his diploma thesis [Nah12] supervised by the author of this thesis. The feasibility of delayed error handling is shown in [HES+10b]. Again, the other co-authors assisted in writing the paper. To track data usage the subscriber model was envisioned by the author of this dissertation and published in [HSM+14]. The co-authors assisted in writing the paper.

**Chapter 5:** The *FAME* ecosystem and the corresponding publication [HSB+15] is

designed by the author of this thesis. The coauthors of the paper assisted in writing the paper.

The conception and implementation of the *REPAIR compiler* is solely the work of Florian Schmoll. The corresponding publication highlighting compiler aspects is [SHM+13]. The majority of this journal article was written by Florian Schmoll. The author of this thesis contributed some sections, especially, the evaluation section excluding 4.6 (Static Analysis). In [ESH+11] source code annotations are shown. Dr. Michael Engel and Florian Schmoll contributed the majority of that paper. The author of this dissertation performed the experiments. The transfer of information from compiler to OS is described in [SHM+14]. This technical report was entirely written by Florian Schmoll. The author of this thesis and the co-authors only gave hints and assisted in writing.

All virtualization concepts presented in Chapter 5 are entirely envisioned by the author of this thesis. Papers, such as [HKS+13] or [HSM+14], are inherently based on the virtualization techniques.

**Chapter 6:** The H.264 codec used in this thesis was originally written by Martin Fiedler [FB04]. Ingo Korb contributed many bug fixes to the source code. However, the whole benchmark application using the H.264 codec was entirely designed by the author of this thesis. A timing analysis of the H.264 decoder was presented in [EHS+11]. Dr. Michael Engel provided most of the text for the paper. The author of this thesis provided all evaluation tools as well as the evaluation section.

The split version of *libh264* together with the corresponding timing model was provided by Björn Bönninghoff.

**Chapter 7:** Chapter 7 highlights evaluation results of the previously mentioned publications. All evaluations are performed by the author of this dissertation. There is only one exception. The values presented in Table 7.1 were measured by Florian Schmoll.

**Chapter 8:** The first step towards multi-core support is made with R<sup>2</sup>G [Hei10]. R<sup>2</sup>G was entirely envisioned by the author of this dissertation. The execution of automatically parallelized applications on top of R<sup>2</sup>G is shown in [CHM+11]. Daniel Cordes is the main contributor for the latter mentioned publication. The author of this thesis contributed the operating system, the simulation platform, and R<sup>2</sup>G.





# Related Work

---

## Contents

<b>2.1</b>	<b>Fault Tolerance</b> . . . . .	<b>13</b>
<b>2.2</b>	<b>Reliability in Real-Time Operating Systems</b> . . . . .	<b>16</b>
<b>2.3</b>	<b>Checkpoint and Recovery</b> . . . . .	<b>19</b>
<b>2.4</b>	<b>Fault Injection</b> . . . . .	<b>21</b>
<b>2.5</b>	<b>Scheduling of Real-Time Tasks</b> . . . . .	<b>25</b>
<b>2.6</b>	<b>Timing Estimations</b> . . . . .	<b>27</b>

---

This thesis covers many research areas. Therefore, related work is grouped into different categories. In Section 2.1 fault tolerance methods and fault detection are depicted. Related work concerning reliability improvements of real-time operating systems are discussed in Section 2.2. In Section 2.3 checkpoint and recovery methods are presented. Injecting faults is an important method to test and verify fault tolerance techniques. Hence, in Section 2.4 examples for fault injection are given.

Other none-reliability aspects are discussed in the remaining sections. Section 2.5 depicts relevant scheduling algorithms. Related work concerning timing estimations is depicted in Section 2.6.

## 2.1 Fault Tolerance

Fault tolerance methods are typically based on some kind of redundancy. Hereby, a larger amount of redundant resources allows correcting a higher number of errors. It is possible to distinguish between two types of redundancy: *temporal redundancy* and *spatial redundancy*. Temporal redundancy means to compute the same data value at two different moments in time, usually on the same hardware. If spatial redundancy is used, the same data value will be computed on different pieces of hardware, usually at the same time. Both redundancy forms can also be mixed.

Fault tolerance methods can be classified as *proactive* or *reactive*. A reactive method will only take action on demand, for example after an error is detected. In contrast, proactive methods tackle problems that may occur. If a fault occurs, proactive methods already have a solution available. Proactive methods permanently require resources since, for example, redundant data sets have to be stored. In contrast, reactive methods will only require resources if an error occurs.

**TMR:** Triple Modular Redundancy (TMR) is a typical proactive method to cope with errors. The computation is executed three times and the results are compared. If the computation results differ, a majority vote will determine the final result. The three instances can be executed in parallel (spatial redundancy) and/or sequentially (temporal redundancy). In the latter case TMR can be combined with a reactive strategy. It will be sufficient to execute only the first and the second calculation if the results are equal. Implementations of redundant computations can be found, for example, in Boeing airplanes [Yeh98] or IBM S/390 systems [SAC+99] (now System z). The likeliness that TMR is able to locate and to correct an error is very high. However, the drawback of TMR is the triplication of the required resources plus extra logic or program code needed for the voter. Another problem can be the voter as a single point of failure, if the voter is not replicated / protected as well.

**DMR:** If errors only have to be detected, Double Modular Redundancy (DMR) will be sufficient. DMR “only” doubles the amount of used resources. If the underlying hardware can be modified, Austin [Aus99] propose the dynamic implementation verification architecture (DIVA). In this architecture, the commit phase of the processor’s pipeline is augmented with a functional checker unit, called DIVA checker. The checker unit verifies the correctness of the main processor’s computation. Only correct results are committed. DIVA assumes that all registers and memories are protected from faults. Hence, any value the DIVA checker reads or writes will be without errors. The DIVA checker is a simple in-order core. The main processor can be a high-performance out-of-order core. The DIVA checker slows down the main processor by only 3%, on average.

**Parity Bit and ECC:** Using a parity bit to protect a word in memory is a low-cost solution to detect an uneven number of bit-flips. The parity bit is calculated by an exclusive OR (XOR) operation on all data bits. To check the stored data for an error, an XOR including the parity bit as well as the data bits has to be performed. If the result is ‘1’ (even parity), an error has occurred. To detect more than one bit flip, Error Correction Codes (ECC) can be used. Very prominent examples for such codes are Hamming code [Ham50] and Reed-Solomon code [RS60]. Both are also able to correct a certain amount of errors. However, the main drawback of ECC is the overhead in terms of additional resources required to store the extra bits and the additional time required by the memory controller to compute the checksum and to correct the error.

**Cache Scrubbing:** Caches are one of the largest structures in today’s processors. Hence, they are most vulnerable to transient faults. Therefore, caches are often protected by ECC which typically offer single-bit error correction and double-bit error detection. In [MEF+04] Mukherjee et al. analyzed the susceptibility to double bit faults in ECC-protected caches. They show that the Mean Time To Failure (MTTF) for such double-bit errors is very large for small caches (less than tens of megabytes in size). However, large caches need to be scrubbed to reduce the

temporal double-bit error rate. Scrubbing is a technology which periodically reads the content of the cache. Due to the read operation, the ECC gets evaluated and recomputed on each access. Hence, existing single-bit errors can be corrected. If the scrubbing interval is short enough, the probability of double-bit errors will be very low. Examples for CPUs with integrated cache scrubbing are AMD Athlon64 and AMD Opteron64 processors [AMD06].

**RAMPage:** RAMpage is a tool to test memory modules while the system is online. In the corresponding publication [SKS13], Schirmeier et al. show the integration of RAMpage into a x86-64 Linux-based system. RAMpage consists of two parts. The first part is a kernel module which is responsible for claiming physical pages. This operation includes moving data from used pages to unused pages, since testing a page for memory errors will overwrite the stored data. The second part of RAMpage runs in user-space and is responsible for the actual testing. RAMpage will remove pages from memory allocation if a permanent memory error is detected.

**EDDI:** In [OSM02] Sjorvani et al. introduce a pure software based technique to detect errors. This technique, called Error Detection by Duplicated Instructions (EDDI), is based on a customized compiler which duplicates instructions. The duplicated instructions use different registers and variables compared to the original instructions. In the evaluation, the authors show high fault coverage of over 98% without any extra hardware for error detection.

**SWIFT:** Reis et al. present Software Implemented Fault Tolerance (SWIFT) in [RCV+05] which is an evolution of the previously described EDDI. SWIFT adds control-flow checking with software signatures to EDDI. In the paper, a SEU fault model is used with exactly one bit flip throughout the entire program. Furthermore, it is assumed that the memory is somehow protected, for example, with ECC. Under these assumptions, it is not required to have two distinct memory locations to store data. However, loads still have to be duplicated since all values stored in registers require a redundant copy. In EDDI it will be possible that the control flow is incorrect if a fault happens during branch execution, for example. To eliminate this problem SWIFT introduces enhanced control flow checking. Therefore, a designated general purpose register stores a signature of the currently executed block. This register is called General Signature Register (GSR). The value stored in GSR is calculated by xor'ing GSR with a statically generated signature when entering a new basic block of the source code. SWIFT also defines a second register called RTS, which is storing the run-time adjusting signature. In the case of a branch, the RTS is calculated by xor'ing the statically generated signature of the current block with the signature of the target block. After branching the RTS is xor'ed with GSR. If the result matches the compile-time generated signature of the target block, no error has occurred. In this approach, the RTS together with the GSR serve as a redundant duplicate for the program counter. EDDI as well as SWIFT are used to detect faults. Upon detection, arbitrary user-defined error correction methods can

be used, like, for example, *FAME* which is the main contribution of this dissertation.

**ACFC:** Assertions for Control Flow Checking (ACFC) are proposed by Venkatasubramanian et al. in [VHM03]. Each basic block is assigned a pre-calculated execution parity. Control flow errors can be detected by comparing the calculated parity bit with the pre-computed parity. In contrast to SWIFT which operates at assembly level, ACFC operates on C code.

**Containment Regions:** Fault containment regions [HSL78] are used to limit the propagation of faults. A system supporting containment regions is designed in such a way that a fault which affects a specific containment region cannot spread to another region. The Reliable Computing Base concept [ED12] which is used in this dissertation can be seen as a special type of containment region.

Typically, error detection only approaches require less redundant resources than approaches including error correction. As mentioned earlier, this dissertation is focused on error correction. For the rest of this thesis it is assumed that at least one of the previously described error detection methods is available. When the applied method detects an error which cannot be handled directly, it is furthermore assumed that the error will be reported to the operating system. This will be the case if, for example, ECC memory is used which raises a Machine Check Exception when a multi-bit error cannot be corrected.

## 2.2 Reliability in Real-Time Operating Systems

Reliability in operating systems is a very important issue. Often, reliability is closely related to security concerns. In this section, selected approaches are shown increasing the reliability of the overall system based on the Operating System (OS).

**EROS:** One important development is the EROS system [SSF99] and its follow-up project, Coyotos. EROS is a capability-based microkernel. A capability is a pair of an object identifier and a set of authorized operations on that object. In a capability system, security is assured by the properties, that (1) capabilities are unforgeable, (2) processes are only able to obtain capabilities by using authorized interfaces, and (3) they are given only to processes with the authorization to hold them. EROS provides support to efficiently restructure critical applications into small communicating components. These components can then be efficiently isolated from each other and the rest of the system. Access to objects is controlled by capabilities. In this thesis, a subscription-based model of data object ownership is introduced which can be seen as orthogonal to capabilities. While capabilities have been mainly used for protection against intentional errors, like intrusions, they are useful for protection against hardware errors as well. Due to the strict isolation of the single components, capabilities can limit the propagation of faults.

**Microvisor:** The term microvisor was first used by Heiser and Leslie in [HL10]. It describes an approach where features of a *microkernel* are combined with features of a *hypervisor* into one operating system. The basic concept of microkernels is to reduce the kernel code to fundamental mechanisms. Liedtke formulated the following Minimality Principle [Lie95]:

**Definition 2.1 (Minimality Principle)** “A concept is tolerated inside the microkernel only if moving it outside the kernel [...] would prevent implementation of the system’s required functionality.” (Liedtke in [Lie95])

One application of this Minimality Principle is the minimization of the trusted computing base (TCB) [SPH+06]. The TCB is a set of components which are critical for security. Minimality has a further benefit. The number of lines of kernel code gets reduced which possibly enables formal verification of the kernel. In [KEH+09], Klein et al. formally verified the seL4 microkernel.

The other important component of the microvisor is virtual machine (VM) support. The classic definition of a (system-level) VM is given in [PG74]:

**Definition 2.2 (Virtual Machine)** “A VM is an efficient, isolated duplicate of a real machine concept.” (Popek in [PG74])

In a VM, virtual resources are either emulated or mapped to physical resources by a virtual machine monitor (VMM) or hypervisor. In the classic virtualization approach, virtual resources are essentially indistinguishable from physical resources. Since the virtual resources are indistinguishable, guest operating systems can be executed without modifications. However, when hardware limitations and efficiency are of major concern, a *para-virtualization* approach can be a better choice. In contrast to *pure* virtualization, para-virtualization provides a modified hardware ABI to which the guest OS has to be ported. This ABI is more suited for virtualization, and hence imposes less virtualization overhead. A hypervisor which supports para-virtualization provides an extra *hypercall* API. These hypercalls are generally more high-level than the hardware API. Typically, a hypercall can be seen as a system call to the hypervisor.

**Romain:** In the L4 microkernel family, several projects provide isolation mechanisms in order to prevent errors in one application from affecting other tasks. TU Dresden’s Fiasco [LW09] L4 variant is used by Döbel et. al in the Romain framework to build a system that supports the redundant execution of threads for typical embedded applications [DHE12]. Romain allows detecting and recovering from SEUs by using n-way modular redundancy. The redundancy is achieved by using software-implemented transparent multithreading. This means that a thread gets automatically replicated and distributed appropriate over the available CPU cores. The necessary synchronization and the state comparisons of the replicas take place before externalizing the thread state, e.g. when system calls are invoked. The

proposed method requires a maximum runtime overhead of 30% for TMR. However, the majority of the executed benchmarks of the MiBench suite remain below 5% overhead. A huge advantage of Romain is the support for applications only available in binary form since Romain is a transparent OS service.

**seL4:** A formal verification of seL4 is provided by Klein et al. in [KEH+09]. SeL4 is an L4-based microkernel [Lie96]. In [KEH+09] the authors show the correctness of the C implementation based on the high-level seL4 specification. Hereby, the correctness of the compiler, assembly code, boot code, and the hardware is assumed. To support formal verification, Haskell is used to implement a prototype of the kernel which is binary compatible to the later C implementation. Binary compatible means that applications can be executed with the Haskell or C implementation without recompiling. The used Haskell subset can be automatically translated into the language of the theorem proving tool the authors use. Hence, the Haskell code can be translated to the executable specification which can be verified via a refinement proof against the abstract specification of the seL4 microkernel. However, it is not possible to generate efficient C code out of Haskell. Therefore, the authors created the C code manually. The C code is verified against the executable specification via a refinement proof as well. In the end, the authors created a high-performance microkernel which is formally verified. Especially the verification is important for applications with high security demands, since security of a computer system can only be as good as that of the underlying OS. However, the focus of seL4 is rather on security than dependability.

**RMK:** The reliability microkernel framework (RMK) is presented by Wang et al. in [WKG+06]. RMK is a loadable module for the Linux kernel. It provides application-aware reliability and dynamically configurable reliability mechanisms. RMK supports detection of application and OS failures with a high coverage and low false positive rate and transparent application checkpointing. The modular design of RMK allows for high configurability. RMK consists of a core, a lower level (RMK pins), and an upper level (RMK modules). RMK pins encapsulate the underlying hardware and low-level system services. These can be, for example, hardware monitor registers available on modern processors. The RMK modules implement specific error detection or recovery mechanisms. Due to this design, RMK can be easily ported to support different platforms and operating systems, since only the RMK pins have to be ported. The main strength of RMK will be the low overhead if the CPU performance counters and exported kernel symbols are used. An application hang, for example, can be detected by comparing the instruction counter with the maximum number of instructions needed for the current code block. The RMK approach focuses on the detection of error effects and not on the detection of the error itself. The checkpointing is based on information about dirty memory pages, similar to the approach used in libckpt which will be depicted in the next section.

**Exokernel:** Engler et al. describe an exokernel in [EKO95]. The main idea of exokernels is to provide only a minimal layer of abstraction. In this architecture, all hardware resources are exported through a low-level interface to untrusted library operating systems. This enables the software running on top of it to have a greater level of control. Therefore, the functionality of exokernels is limited to protection and multiplexing of resources. In this regard, they are much less restrictive than typical microkernels, which usually add at least a message passing-based communication mechanism.

The operating system of *FAME*, which is presented in this dissertation, is based on microvisor design principles with features seen typically in exokernels. Hence, it can be seen as a kind of specialized exokernel which provides only those abstractions required for building fault-tolerant applications on top of it, while leaving other crucial resource allocation decisions, such as scheduling and memory management, to the library OS running on top of it.

## 2.3 Checkpoint and Recovery

*Checkpoint-and-recovery* is a commonly used strategy to deal with erroneous system states. The basic idea of this approach is to restore a valid state in such a case. Checkpoint and recovery consists of two phases:

1. In phase one, called *checkpoint creation*, the current system state is stored. Different strategies exist to determine the point where a checkpoint shall be created. One is to create checkpoints in previously defined time intervals. Another method is to add explicit calls to the checkpoint creation routine into the application's source code. While the former method ensures regular checkpoints, the latter method allows for placing checkpoint creation in strategically useful points. Such a location, for example, is a point in the application where the current working set is small. Hence, a checkpoint method optimized for storing only the working set can create smaller checkpoints.
2. The second phase is *checkpoint recovery*. This phase is reactive and will be only executed if an error affects the application or operating system. Therefore, a (previously) stored system state is used to override the current state. After restoring, execution is continued with the stored system state. Hence, the recovery returns to a previous point in time of program execution and all calculations from that point on have to be redone.

The following references depict different strategies to minimize the overhead of the checkpoint creation process.

**Flashback:** Srinivasan et al. [SKA+04] developed Flashback. Flashback is a tool for light-weight checkpointing of Linux processes. To create a checkpoint, a

snapshot of a process, called *shadow process*, is stored. A shadow process contains the state information of the original process including memory, registers, file descriptors, and signal handlers. Copy-on-write semantics are used to keep the creation overhead as low as possible. The basic idea of copy-on-write is to map pages read-only, hence a write access generates a page fault. When such a page fault occurs, the operating system triggers the creation of a copy of the old data. The drawback of copy-on-write, however, is a possibly high page fault rate, especially for applications which write to a variety of different memory locations. Each page fault will slow down the system due to the necessary processor mode switches.

**libckpt:** To reduce the amount of data which has to be stored in a checkpoint, libckpt [PBK+94] implements two optimizations. The first optimization is *incremental checkpointing*. Incremental checkpointing only stores data changed since the previous checkpoint. Data not stored in the current checkpoint can be found in previous checkpoints. The second optimization used by libckpt is the usage of copy-on-write semantics to determine whether or not data was changed. Like in Flashback, the drawback can be high page fault rates. However, the authors of the corresponding paper show huge savings in the checkpoint creation overhead which amortize the mode switching overhead. Mode switches occur, for example, at every page fault.

To further reduce the overhead, “dirty” and “accessed” bits of the page table can be used to track modified and accessed memory. In this solution, the MMU automatically keeps track of pages which are changed (dirty) and pages which are used (accessed). The operating system simply has to traverse the page table to find modified pages. However, this approach requires special hardware features which are not available on all architectures. For example, ARM processors do not implement the necessary functionality in the page table entries.

**Sequoia:** Bernstein describes the Sequoia multiprocessor in [Ber88]. To increase the resilience against faults in a processing element, accessed memory ranges are determined by tracking cache accesses. All dirty memory blocks are flushed and a copy of the processor registers is stored at each cache flush. Hence, there always a memory image available that is executable on another processor element. If the fault happens while the processor is not flushing its cache, the operating system needs only to identify the process assigned to the faulty processor and return it to the ready queue. Faults while flushing the cache are problematic. Therefore, the Sequoia System flushes the cache twice. The first flush is stored into a backup copy. When this flush is completed, the second flush will target the primary storage location. If a fault occurs during the backup phase, the primary storage location will be still intact. In the other case, if a fault occurs during flushing to the primary storage location, the backup will contain a consistent state. In both cases, the inconsistent copy can be restored by reading the consistent copy. The main drawback of this is that the underling hardware has to be modified.



Most highly optimized checkpoint approaches either require special hardware features or are limited to create only application checkpoints. In this dissertation, however, full-system checkpoints are mandatory to recover also from faults affecting the operating system. The checkpoint mechanism integrated in *FAME* is able to significantly reduce the checkpoint time without requiring special hardware features.

## 2.4 Fault Injection

In order to create effective fault-tolerance techniques, system designers require methods to assess the effectiveness of the applied EDAC techniques already in the design phase. This evaluation is commonly performed by exploring the effects of faults on the combined hardware/software system. To cover most of the possible faults, numerous fault injection experiments have to be performed. Many approaches to fault injection have been described in the literature so far. However, fault injection is limited. The system under test can only be stressed with the type of faults the injection system supports. In this section, an overview of typical injection approaches and their limits is given.

Hsueh et al. developed a classification of different fault injection techniques in [HTI97]. A global distinction is made between hardware and software fault injection techniques.

### Hardware-Based Fault Injection

Hsueh et al. define hardware fault injection as a technique that uses additional hardware to inject faults into the target system. The injection can be with direct physical contact, e.g., using pin-level probes and sockets, or without contact by exposing the circuit to a particle beam using lasers or electromagnetic inferences. Especially the contactless methods have the ability to inject faults in a realistic way. An alpha particle emitted by the particle beam, for example, will hit the circuit in a random location, which reflects the stochastic nature of faults quite accurately. However, this makes it hard to reproduce a specific fault sequence.

**MESSALINE:** An example for a tool injecting faults with direct physical contact is MESSALINE developed by Arlat et al. [ACL89]. The fault injection component of MESSALINE supports two modes. In the *forcing* mode, a fault is directly applied by means of multipin probes on the pins of the IC. For the second mode, called *insertion*, the IC under test is removed from the target system and inserted into a special test system where transistor switches isolate the IC from the rest of the system. Stuck-at faults can be evaluated with both methods. Stuck-at faults are pins which are permanently '0' or '1'. Physical bridging and intermediate voltage level simulation is only possible within the forcing mode. Intermediate voltages are voltages in between the ranges defining a logical '0' and '1'. For example, in 5V CMOS, 0V - 1.5V and 3.5V - 5V defines '0' and '1', respectively. Hence, all volt-

ages in between 1.5 V and 3.5 V can result in undefined behavior. In the insertion mode, logical bridging, inversion and open wires can be evaluated.

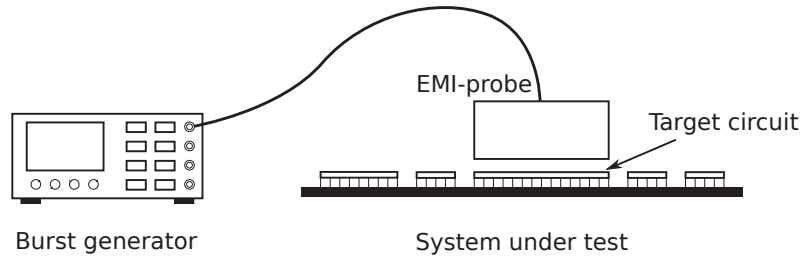
**Velazco:** In [VKC+92], Velazco et al. propose to validate and to verify the efficiency of detection mechanisms for faults induced by heavy ions. Therefore, the authors used a particle accelerator firing heavy ions on a Motorola 68020 microprocessor and its 68882 co-processor. The main goals of the authors are to study the influence of caches on the microprocessor sensitivity, the influence of the executed program on the error rate, the impact of the co-processor, and the interception and classification of faults. In the case of the 68020 processor, the cache has a huge influence on the heavy ion sensitivity. With caching enabled, the error rates are significantly higher. The executed program also has influence on the observed error rate. In the paper, it was shown that programs heavily using registers are more susceptible to faults. Faults within the co-processor are often detected by the main processor due to the communication protocol which will sometimes be perturbed by faults. Other errors are detected by the internal control mechanism of the main processor related to the co-processor management. The main reasons of errors are classified as wrong addresses, corrupted messages, and sequencing perturbation.

**Ecoffet:** Another ion based injection method is presented by Ecoffet et al. in [ELD+92]. The focus of the paper lies on sensitivity tests of different memory types from various technologies. Like in [VKC+92], a particle accelerator is used to emit ions. The used linear energy transfer (LET) ranges from 0.3 to 57  $MeV \cdot cm^2/mg$ . The authors concluded that the pattern stored in the memory have influence on the sensitivity. In particular, they observed that a higher number of stored '1' bits leads to a higher sensitivity to faults.

**Pouget:** Pouget et al. [PLF03] use a pulsed laser to inject transient fault into an analog-to-digital converter (ADC). In contrast to the previously mentioned radiation based fault injection, laser beams have the advantage that transient localized perturbation of the device under test is possible. Hence, lasers enable deterministic transient fault injection experiments. In addition to injection, the authors also use lasers to scan the internals of the used ADC. With the latter method, it is possible to image the information propagation and to extract critical phases.

**Vargas:** An EMI-based fault injection technique is presented by Vargas et al. in [VCG+05]. A GigaHertz Transverse Electromagnetic (GTEM) cell is employed to inject faults into the system under test. The number of injected faults depends on the radio frequency modulation and the electromagnetic field value. A typical experimental setup to inject EMI-based faults is depicted in Figure 2.1. A probe is used that can expose a small part of the board to the disturbance.

**MARS:** To validate the Maintainable Real-Time System (MARS) Karlsson et al. [KFA+95] evaluate three different hardware-based fault injection techniques: pin-



**Figure 2.1:** EMI injection setup

level, heavy ion, and electromagnetic injection. The goal of the authors is to compare fault injection techniques to identify similarities and differences in the error sets generated. Their evaluation results show fairly large differences of the implemented error detection mechanism depending on the applied fault injection technique. This suggests that the different methods generate different types of errors. Hence, the applied injection approaches are rather complementary. However, according to the authors, the heavy ion radiation method stresses the system the most while EMI-based injection generates only a very restricted set. Pin-forcing exercises error detection methods located outside of the chip under test more than the other techniques.

In general, hardware-based solutions have the advantage that no resources of the system under test itself are required to inject the fault. Hence, the real-time behavior of the system is not influenced by the injection process. However, hardware-based injection brings along a number of disadvantages. Either the hardware of the target system has to be modified or special expensive equipment, e.g., a radiation source or an ion-beam injector, is required. The statistical nature of the injection often results in a low precision of fault injection, since neither the exact location nor the exact point in time at which the injection should take place can be controlled.

### Software-Based Fault Injection

Software-based fault injection methods resolve some of the problems discussed above. However, depending on the specific approach used, new problems arise, which are discussed below.

**Off-line:** Different ways to inject faults exist. In an off-line approach, selected instructions are modified at compile time. This kind of injection generates a software image with hard-coded permanent faults. Due to these specific modifications, faults are only activated when the particular code locations are executed. An approach very similar to code modification is code insertion. Here, additional instructions are added to the target program that allow fault injection to occur before a particular instruction. A disadvantage is that the configuration used for testing the fault tolerance is different to the one deployed on the final system. Due to these

disadvantages, the remainder of this subsection only considers on-line approaches.

**Velazco:** In order to randomly inject faults into any process of a target application, an interrupt-based method is used by Velazco et al. [VRE00]. One of the simplest ways to trigger fault injection is to use a timer circuit which can generate external interrupts based on a specified time interval. If an interrupt request is received, a bit flip, called Code Emulated Upsets (CEU), will be injected concurrently with the execution of the program in accessible sensitive areas, like, e.g., registers and memory. The results obtained by the authors converge on experiments executed with radiation based injection. Thus, CEU-injection allows for prediction of error rates for a given application. The drawback of this approach, however, is the required modification of the interrupt handler code of the operating system.

**Xception:** In [CMS98], Carreira et al. describe Xception, an approach using debugging features of modern processors. The basic idea is to insert a breakpoint to stop the processor when accessing a specific memory area. On access, the target processor is stopped and the host debugger can insert a fault. By using the performance monitoring unit of the processor, Xception can furthermore record detailed information on the behavior of the processor after fault injection. Such information can include, for example, the number of executed instructions and the number of memory read and write cycles. If the exception triggers provided by the debugging hardware and the performance monitor features are combined, Xception will be able to detect memory areas accessed after fault injection. Hence, non-standard behavior of the application can be easily detected. However, target systems have to be modified to support Xception. Low-level exception handlers as well as the fault injection code have to be added.

**FIMBUL:** A further approach which also uses debug capability is FIMBUL [FSK98]. Here, fault injection is used to test error correction and detection of the Thor CPU. When a breakpoint condition is fulfilled, a fault will be injected. FIMBUL has the drawback that only a single bit flip is performed per run.

### Other Injection Approaches

Beside the 'classical' software-based and hardware-based fault injection techniques further approaches exist. Hybrid fault injection approaches comprise methods of software-based and hardware-based fault injection. One candidate in this area is the usage of the On-Chip Debugger (OCD) unit via a JTAG link. Another approach to inject faults is simulation based. Here, faults are injected into a simulation model of the target system.

**Garcia:** Like Xception, the approach proposed by Garcia et al. [PGLOGV+07] uses the debugging capabilities of the target processor. In contrast to Xception, no modification of the target systems is required, since the OCD is now controlled

externally via JTAG. The authors injected faults into an ARM7TDMI processor with a JTAG injection system running on an FPGA. With only 2 ms fault injection time the proposed solution is very fast. However, the main drawbacks of the FPGA solution are the low flexibility and low adaptability. Due to the complexity of FPGA programming, it is very difficult to implement complex fault models or to exchange the fault model. Furthermore, the simulation of real-world error reporting is hard to implement. Therefore, an instruction decoder running on the FPGA would be required to keep track of currently used hardware components. Implementing such a decoder for the FPGA is generally possible. However, due to the high complexity of FPGA programming, such a solution will be very expensive.

**FAIL\*:** Schirmeier et al. present FAIL\* [SHK+12] which tackles the problem of non-reusable fault injection code when using multiple system simulators. Especially if fault injection tools are highly platform specific, a tight coupling between simulator and injection code complicates exchanging of the tool's target backend. Therefore, FAIL\* allows for switching target backends. A framework API abstracting backend details and hence fostering injection experiment code reuse is proposed as well. FAIL\* currently supports Bochs [Law96] and Open Virtual Platform [Bai08] for x86 and ARM architecture, respectively.

In this dissertation a subset of the previously mentioned methods are used to inject faults. In particular, these will be a software-based, a hybrid, and two simulator-based approaches.

## 2.5 Scheduling of Real-Time Tasks

Scheduling strategies for embedded systems have to take real-time properties of the application into account. This typically includes the **relative deadline**  $d_i$  and the **period**  $p_i$  of each task  $T_i$ . In this thesis, a scheduling method will be called **optimal** if it will find a schedule if a feasible schedule exists.

Scheduling has a long history in computer science. Hence, a huge variety of scheduling methods exist. This section only provides an overview of classic real-time scheduling approaches and the scheduling strategy which is used later in this dissertation.

**EDF:** Earliest Deadline First (EDF) [Hor74] is a priority-based scheduling method for tasks. The priority of a task is a monotonically decreasing function of the task's deadline. This means that the task with the earliest deadline has the highest priority and hence will be scheduled to the CPU. EDF is an optimal scheduling strategy for independent tasks on a single SPU if preemption of tasks is allowed or if all tasks are ready at  $t = 0$ . Due to its optimality, EDF can determine a schedule even for processor utilizations of up to 100 %, if a schedule exists. However, EDF also has drawbacks. One is that EDF cannot give any guarantees of the task's la-

tency. Latency is the total time between the point in time where the task gets ready and its completion. Another drawback will be the domino-effect when a deadline is missed. Under EDF, it is very likely that other tasks will also miss their deadline.

**RMS:** Rate Monotonic Scheduling (RMS) was proposed by Liu and Layland in [LL73]. RMS is a priority-based scheduling strategy with preemption. The priority of a task is static and depends on the period. The shorter the period, the higher the priority. Typically, the relative deadline  $d_i$  of task  $T_i$  is considered equal to its period  $p_i$ . Furthermore, the execution time  $c_i$  of each task is assumed to be known and constant. It can be proven that RMS is optimal if the previous assumptions hold and the utilization of the processor does not exceed  $n(\sqrt[n]{2} - 1)$  for  $n$  tasks, on a single processor machine (alternative bounds also exist, like, e.g. [BBB01]). 100% CPU utilization can be achieved when the periods of the tasks are harmonic. This is provided if all task periods are multiples of the tasks having the next shorter period.

**QRMS:** To relax the strict constraint that each task scheduled under RMS requires a fixed known execution time, Hamann et al. introduce Quality Rate Monotonic Scheduling (QRMS) in [HRR+07]. QRMS divides a task into mandatory and optional parts. Mandatory parts have to fulfill the real-time requirements. For optional tasks, developers can specify a probability to which such tasks have to reach their deadline. For example, it is possible to specify that on average 90% of all optional parts of a task have to obey to their deadline. For each new task, the admission control part of QRMS checks whether or not the task can be scheduled and calculates the reservation time. Therefore, the probability distribution of all execution times is required. Such times can be determined experimentally, for example. QRMS allows for high processor utilization while keeping the admission overhead small. However, QRMS is not applicable in this dissertation since this thesis follows a data-centric approach. Hence, it is not possible to split the code in mandatory and optional parts.

**LST:** Least Slack Time First (LST) is a scheduling strategy shown by Liu in [Liu00]. The priority of a task under LST is inverse to its laxity. The less laxity, the higher the priority. Laxities can become negative. This is an early indication for a possible deadline miss in future. It can be shown that LST is an optimal scheduling strategy for single processor systems. However, due to the dynamically changing priority of the tasks, the priorities have to be recomputed during runtime. In contrast to EDF, LST needs to know the execution time of the executed tasks.

**Blazewicz:** A very early method to schedule dependent tasks with precedences and different arrival times is presented by Blazewicz in [Bla76]. In his paper EDF is used to schedule the task set. It was shown that preemption is not necessary if all tasks have the same arrival time. However, if arrival times differ, preemption will be necessary for an optimal scheduling algorithm. To handle the precedences it was proposed to change the deadline of each task in such a way that tasks which

have successors have earlier deadlines than all their successors.

**EDF\*:** EDF was extended by Henn [Hen78] and Chetto et al. [CSB90]. In addition to modifying deadlines, the authors propose to modify the release times too. The following algorithm, called EDF\*, is able to schedule sporadic tasks with precedence relations and different activation times. EDF\* consists of two basic principles.

1. *The relative deadline of a task depends on its deadline and on the deadline of its successors.*

The relative deadline  $d_i^*$  for a task  $S_i$  is calculated by taking the minimum of its deadline  $d_i$  and the relative deadlines of all its **successor** tasks  $d_j^*$  minus the execution time of the successor  $C_j$ :

$$d_i^* := \min(d_i, \min(d_j^* - C_j : S_i \rightarrow S_j)) \quad (2.1)$$

$S_i$  has to be completed by time  $d_j - C_j$  which represents the latest possible start time of its immediate successor

2. *The relative release time of a task depends on its release time and on the completion time of its predecessors.*

The relative release time  $r_i^*$  for a task  $S_i$  is calculated by taking the maximum of its release time  $r_i$  and the relative release times of all its **predecessor** tasks  $r_j^*$  plus the execution time of the predecessor  $C_j$ :

$$r_i^* := \max(r_i, \max(r_j^* + C_j : S_j \rightarrow S_i)) \quad (2.2)$$

By applying both equations, an equivalent scheduling problem is obtained with modified release times and deadlines. Whenever a task  $S_i$  has a successor  $S_j$  it holds that  $r_i^* < r_j^*$  and  $d_i^* < d_j^*$ . Hence, ordinary EDF can be used to schedule this modified task set.

Scheduling is not primarily in the focus of this dissertation. The aim is rather to provide mechanisms to allow for using well-known algorithms to schedule error correction. However, a scheduling strategy is required in this thesis as well. Due to the advantages of EDF\*, a slightly modified version of EDF\* is used to schedule error correction and the application.

## 2.6 Timing Estimations

Timing is an important characteristic of embedded systems. Although not all scheduling strategies require the execution time of all tasks, execution time estimations are still important to calculate CPU utilization. Especially for cyber-physical systems, to not overload the CPU is mandatory since overloading can lead to deadline misses. It is also required to know the execution time of all tasks to calculate

the global slack time. The slack time can be used later to schedule sporadic tasks, such as error corrections.

As already mentioned in the introduction, the most important time for scheduling is the estimated worst case execution time ( $WCET_{EST}$ ). Timing estimations are not in the focus of this dissertation. However, since the  $WCET_{EST}$  will be required later, a very short overview of the techniques used in this thesis is given in this subsection.

**aiT:** aiT [WEE+08] is a WCET analyzer developed by AbsInt. It is used to statically compute tight bounds of the  $WCET_{EST}$  of tasks in embedded real-time systems. Binary executables can be directly analyzed. Hence, the source code of an application is not required which enables to analyze also legacy code. However, sometimes an annotation file is required. For example, if loop-bounds cannot be determined statically, they have to be annotated manually. aiT also supports annotations specifying the values of registers and variables. This is, for example, useful to analyze software that is running in different modes depending on a value stored in a variable. To calculate the  $WCET_{EST}$ , aiT transforms the program code and the annotations into an annotated CRL (control-flow representation language). In the next step value analyses are performed to determine possible addresses and to detect infeasible code paths. The most challenging part of the WCET estimation is the cache analysis since aiT has to consider the cache replace strategy implemented by the processor. After the cache analysis, the processor pipeline behavior is predicted. Finally, the timing information is generated for each basic block and the worst-case execution path is calculated. The worst-case execution path determines the  $WCET_{EST}$ .

aiT is well suited for applications where the execution time is not too dependent on the input. For the H.264 video decoder application, which will be used as a demonstrator later in this dissertation, aiT calculates a WCET estimation of several seconds per frame. In reality, frame decoding takes under 100 ms. Due to this huge over-estimation, the  $WCET_{EST}$  computed by aiT cannot be used in this dissertation thesis.

**Huang:** To reduce the previously mentioned over-estimation, Huang et al. use a scenario-based approach in [HKB+14]. The basic idea of such a scenario-based approach is to have precalculated WCETs, so called scenario-WCETs, for different code paths the application can take. The key of this solution is to exclude infeasible code paths for the actually used scenario and to tighten the scenario-specific loop-bounds. Hence, if a certain input structure is known, like for example the sequence of I- and P-frames, different scenarios can be used to estimate the WCET for the different frame types. By using this approach, the authors achieved reductions of the over-estimation by 10-70%.

**Roitzsch:** In [Roi07] Roitzsch proposes to use slice-balancing to enable scalable decoding of H.264 videos on multi-core systems. The H.264 standard allows that



one frame<sup>1</sup> can be decoded from multiple slices<sup>2</sup>. Roitzsch added decoding time prediction on the encoder stage to create several slices for each frame. Due to the decoding time prediction, it is possible to balance the amount of work necessary to decode each slice. Apart from a decoder which has to be capable of slice-parallel decoding to gain decoding speed-up, no changes of the client system are required. The encoded bitstream is still full compliant to the H.264 standard. The methods presented in the paper to predict the decoding time look very promising. However, in this dissertation a prediction method on the decoder-side is required which is able to predict the execution time for arbitrary input.

**Bönninghoff:** The approach of Bönninghoff et. al (not published yet) extend the work of Roitzsch by moving the prediction to the decoder-side. The main goal of the authors is to predict the decoding time for arbitrary input. Since the input is not known, the prediction at the beginning of the decoding process is highly over-estimated. However, during decoding of a slice, the prediction becomes more and more accurate. Therefore, the authors divide the decoding process into four stages.

1. **Slice header parsing:** This stage only parses the slice header. After completion, the number of macro blocks to decode as well as the slice type is known.
2. **Slice parsing:** During slice parsing the whole slice input is read. The information is stored in the corresponding arrays which are required in later stages. After completion of this steps all macro block types, intra subdivisions and modes, and motion vector deltas are available. These are important data to significantly lower the over-estimation of the decoding time prediction.
3. **Slice deriving:** Since motion vectors have a resolution of a quarter pixel, it is not possible for the previous step to decide how many filtering steps will be necessary. In this stage the actual motion vectors are derived. After the slice deriving stage a good estimation for the execution time of the final slice rendering stage is available.
4. **Slice rendering:** This stage is responsible for writing the decoded frame.

The prediction of the execution time becomes more and more precise with every stage. Due to the fact that stage one till three can be executed very quickly, sufficient precise execution time estimation is available very early. Another advantage of the presented solution is that the operating system can query the predictor at any point in time, like, for example, when an error has occurred, for the actual execution time prediction. This allows the OS to get a good overview of the available slack which is important for scheduling error correction. Therefore, this dissertation thesis is using the approach of Bönninghoff to predict the execution time of the main H.264 decoder benchmark application.

---

<sup>1</sup>frame = the picture displayed on the screen

<sup>2</sup>slice = a part or the whole raw input for decoding a frame



# Models and Infrastructure

---

## Contents

---

<b>3.1</b>	<b>Reliable Computing Base . . . . .</b>	<b>31</b>
<b>3.2</b>	<b>RAP Model . . . . .</b>	<b>32</b>
<b>3.3</b>	<b>Transient Memory Fault Model . . . . .</b>	<b>34</b>
3.3.1	Thread-Based Fault Injection . . . . .	35
3.3.2	Fault Injection in CoMET . . . . .	35
3.3.3	Injection into Demonstrator Platform . . . . .	37
<b>3.4</b>	<b>Probabilistic Error Model . . . . .</b>	<b>42</b>
3.4.1	Supply Voltage Schemes . . . . .	43
3.4.2	Three-Stage Model . . . . .	43
3.4.3	Probabilistic ARM Platform . . . . .	47
<b>3.5</b>	<b>Task Model . . . . .</b>	<b>48</b>
<b>3.6</b>	<b>Quality of Service Metrics . . . . .</b>	<b>48</b>
3.6.1	$\Delta E$ . . . . .	49
3.6.2	Peak Signal-To-Noise Ratio . . . . .	49

---

This dissertation is based on several models which are explained in this chapter. In Section 3.1, the Reliable Computing base is introduced. It is a model which describes the susceptibility to transient faults for different classes of system components. The resilience articulation point model is discussed in Section 3.2. This model enables the usage of bit flips as a starting point to research the outcome of errors for higher architectural layers without respecting low-level physical effects. Section 3.3 discusses different approaches to inject transient bit flips into memory. An application of the resilience articulation point model is shown in Section 3.4. With HSPICE, low-level bit flip probabilities of probabilistic full adders are determined. These values are later used to inject bit flips in high-level models of multi-bit adders and multipliers. Section 3.5 clarifies the usage of different terms related to scheduling and task management. Last but not least, the used quality of service metrics are introduced in Section 3.6.

## 3.1 Reliable Computing Base

Handling errors in software has a serious weak spot: The software parts which handle errors are susceptible to errors as well. This can lead to situations in which

the error handling mechanism tries to handle an error which occurred during error handling. This may result in a livelock. To deal with this problem, guaranteed fault free hardware components are required to execute software-based fault-tolerance mechanisms. Together with the error handling software parts, these components form the Reliable Computing Base (RCB):

**Definition 3.1 (RCB)** *“The Reliable Computing Base is a subset of software and hardware components that implements all of or ensures the operation of software-based fault-tolerance methods and that we distinguish from a much larger amount of components that can tolerate errors without affecting the program’s desired results.”* (Engel and Döbel in [ED12])

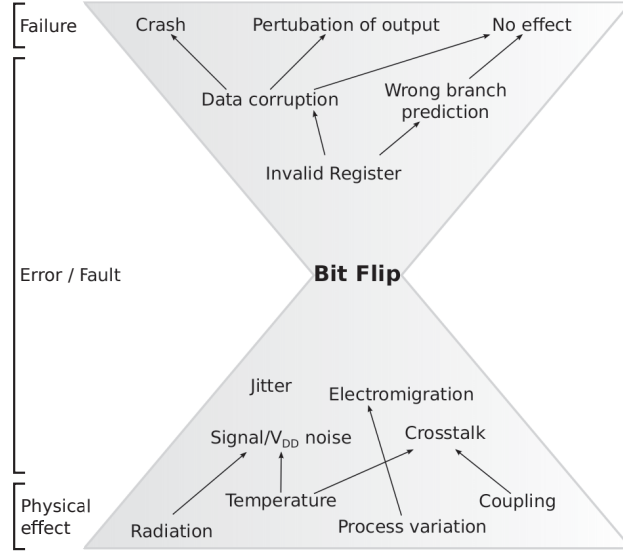
Basically, this definition divides the system into two parts. On the one hand, a small set of components exists – the RCB – which cannot tolerate errors at all. However, when these components are error-free, the software-based fault-tolerance methods are able to operate as intended. On the other hand, a large number of components can tolerate errors, since the error handling methods are operational, as ensured by the RCB components. Hence, as long as the RCB is intact, error handling in software is possible. To keep the RCB intact, the components within the RCB have to be protected from error propagation. Therefore, the following invariant has always to be fulfilled:

**Invariant 3.1** *Only (software) components which are part of the Reliable Computing Base are allowed to change the state of the Reliable Computing Base.*

To enforce this invariant, (para-)virtualization techniques are applied by the microvisor developed in this dissertation. The microvisor represents the software part of the RCB. All other software parts – including the guest operating system – are outside the RCB which leads to a small RCB size. The primary goal of the microvisor is to provide low-level error handling for components not part of the RCB.

## 3.2 RAP Model

The Resilience Articulation Point (RAP) [HAE+14] model serves as an abstraction for fault and error propagation. It is assumed that physically induced faults at the device layer will manifest as bit flips if not masked. In the RAP model, such single or multiple bit flips are the connection point of low-level fault origins and high-level error models of hardware/software system abstraction. This correlation can be visualized with an hour glass-model (cf. Figure 3.1). The bottom layer contains the physical effects, like, e.g., thermal hotspots, aging, and radiation. These effects can lead, for example, to noise which can be the cause for an erroneously interpreted signal. Hence, the originating physical effect will manifest itself as a bit flip if such an erroneous signal is not being masked. In such a situation, the bit flip is the outcome of the physical effect.



**Figure 3.1:** Bit flips as Resilience Articulation Point (based on [HAE+14])

In the upper half, bit flips have different levels of abstraction. An overview is depicted in Figure 3.2. Individual signals or bits stored in flip flops are the target of faults in the lower hardware levels. In [HAE+14], the probability of such an error is described as  $P_{bit}(\vec{x}, t)$  considering the space  $\vec{x}$  and time  $t$ . At the macro and architecture levels, several bits are typically grouped into words.  $P_{word}(\vec{x}, t)$  expresses the probability that a word is erroneous at time  $t$ . This probability can be derived from the error probabilities at bit-level. If operands are stored in memory,  $P_{word}(\vec{x}, t)$  will describe the probability of an error in the corresponding memory cells. In contrast,  $P_{word}(\vec{x}, t)$  can be different if the same data will be stored in a register. Compounds of words are referred to as interfaces at higher abstraction levels. Like  $P_{word}(\vec{x}, t)$ , the probability  $P_{interface}(\vec{x}, t)$  can also be derived from the error probabilities at word-level or bit-level and by taking into account the knowledge of the internal IP (intellectual property) block architecture. Above the architecture level, different software levels can be found. Software is typically operating on variables. Consequently,  $P_{variable}(\vec{x}, t)$  can be used to express the probability of an erroneous variable.

The RAP model is well suited for researchers working at different levels of abstraction. It allows for a clear and quantitative description of the error and fault relationships. For example, in an ECC protected memory cell, a flipped bit does not automatically mean an erroneous word. Only if the amount of flipped bits is larger than the number of bits correctable by the ECC mechanism, the stored word will be affected. In this case,  $P_{word}(\vec{x}, t)$  expresses the probability that at least  $k+1$  bits are flipped, where  $k$  is the amount of bits correctable by ECC. To use this high-level relationship, the knowledge of the lower level models is not necessary. It is sufficient to know the probability of a bit flip  $P_{bit}(\vec{x}, t)$ . The detailed technology

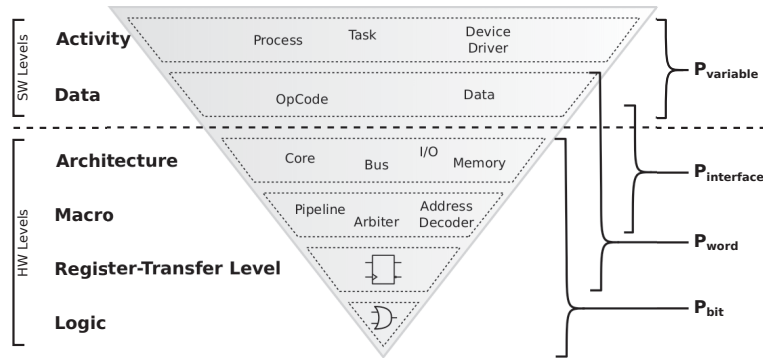


Figure 3.2: Bit flip abstraction in higher system levels (based on [HAE+14])

and implementation aspects of the system are considered in the underlying levels. This approach allows for deriving the probabilistic error functions at higher levels from the probabilistic error functions describing the low-level bit flips.

If the lower layers in the RAP model are modeled correctly, the higher layer SoC behavior – hardware and software – can be investigated without maintaining the complete set of low-level models, since, for example,  $P_{word}(\vec{x}, t)$  can serve as a good representation of the low-level fault model. In the next sections, the error models used in this dissertation are presented exploiting the previously mentioned advantage. To implement these models, different approaches will be used to inject faults directly into adequate positions, thereby replacing complex low-level models.

### 3.3 Transient Memory Fault Model

The main fault model considered in this thesis is a transient memory fault model. Within such a model, bits are flipped at random positions in memory. The bit flips are not permanent. Hence, the affected memory cells will normally operate again if they are overwritten by new data. One requirement of the transient memory fault model used is that the memory ranges where bits are flipping are freely configurable by the designer. This dissertation distinguishes between *high reliable silicon* and *low reliable silicon*. High reliable silicon denotes hardware components which operate fault-free. In contrast, low reliable silicon is susceptible to transient faults. Dividing the hardware into these two classes enables the developer to divide the software stack into two parts by applying a suitable mapping. The part which has to be tested under transient memory faults is mapped to low reliable components and the remainder is mapped to high reliable hardware components. This allows, for example, simulating ECC-protected reliable memory ranges without physically requiring the related ECC hardware.

Error detection is out of the scope of this dissertation. It is assumed that error detection is provided and that detected errors are reported to the operating system. Such an error reporting shall be as realistic as possible. Typically, real-

world hardware with error reporting capabilities, like memory protected by parity bits, will report an error only if the affected cell is accessed. Only on access the memory controller computes the checksum of the stored data and checks the stored parity information. Most likely, memory controllers which can detect errors cannot determine the exact location of the faulty bit. Otherwise, the correction would be very simple – just flipping the bit again. Depending on the implemented granularity of the memory controller’s check routine, the main processor is notified of an error in a memory range, e.g., with the granularity of a cache line. Within this range, one or more bits may be affected by transient errors.

The error reporting capabilities as well as the distribution of the faults depend on the realization of the transient memory fault model. The next subsections describe different ways to realize the fault module using fault injection.

### 3.3.1 Thread-Based Fault Injection

To get a first idea of different error impacts in applications, a fault injection method is desired which is easily applicable. A straightforward approach fulfilling this requirement is to inject transient memory faults at the application level. Hereby, an additional thread is executed concurrently to the application. This thread provides a reproducible, deterministic error scenario by injecting memory faults into different memory locations. The memory ranges where the errors are injected are freely configurable. All injected errors are registered by the injection thread and can be reverted on demand to simulate error correction taking place, which corresponds to a delayed error correction. However, this will not correct errors which already have been propagated to other locations. The error model employed is quite simple: a randomized uniform distribution of flipped bits. To deterministically execute the injection experiment, the fault injection sequence can be programmed. Additionally, error injection can be suspended by the application at any arbitrary point in time. To investigate the effects of errors, artificially high error rates can be used. Thus, fatal errors, like a crashed application, can be found in a short amount of time.

The advantage of the presented approach is its easy applicability. Since the additional thread is running in the same process as the other application threads, it is possible to directly access global symbols of the application. To inject faults in the stack, the corresponding addresses can be communicated to the injection thread via memory mapped interfaces. The main drawback of this approach is the limitation to the running process. It is not possible to inject faults into the operating system or into memory not owned by the process.

### 3.3.2 Fault Injection in CoMET

The main evaluation platform used in this dissertation is based on the Synopsys CoMET/METeor Simulator [Syn14]. CoMET/METeor provides fast and accurate models enabling hardware and software co-design. Such models include processor cores, peripheral devices, buses, and bus interface units. To simulate a complete

SoC, these models can be combined. Synopsys already provides a library of generic components. Furthermore, custom modules written in C, C++, or SystemC can be easily incorporated and simulated with CoMET and METeor. The latter option is used in this thesis to realize fault injection into memory by the implementation of an additional memory module. This module behaves like any other memory module in CoMET/METeor, except that the memory is erroneous. At each memory access the module determines if new faults have to be injected. To control the amount of faults to inject, a Poisson distribution with configurable parameter  $\lambda$  is used. The time base used for the Poisson distribution is memory bus ticks. Faults are randomly injected and are equally distributed over the memory. Hence, the locations of the accesses have no influence on the fault distribution.

The developed memory module also simulates error detection. Therefore, each injected memory fault is stored in a bitmap. On access, it is checked whether or not the actual used memory is affected. If an error is detected, a special error-detected pin will be set to '1' signaling error detection. This pin can be connected to the interrupt controller of the CPU to immediately stop normal program execution and to trigger error handling. The pin remains active until the error is acknowledged. The error will be acknowledged if any of the following three exported memory mapped registers is accessed:

1. **Error Type Register:** This register stores the type of the error and the amount of data possibly affected by the error. Possible error types are `ERROR_TYPE_MEMORY` and `ERROR_TYPE_REGISTER`. The CoMET based injection uses `ERROR_TYPE_MEMORY` since only memory errors are injected. In a later section, JTAG-based fault injection is described which can use both types.
2. **Error Location Register:** The base address of the last occurred error is stored in the location register. It is also possible to write to the type and the location register. In this case, error injection to the written address and size will be triggered.
3. **Configuration Register:** The configuration register can be used to get or set configuration settings of the memory module during runtime. The following settings can be changed:
  - (a) *EnableGeneration:* If activated, new faults will be generated based on  $\lambda$ .
  - (b) *EnableInject:* Bits will only be flipped in random locations if *EnableInject* is enabled. If this option is disabled, faults will be virtually injected. This means the affected memory location is marked as erroneous although no bit was changed by the injection. This mode is very useful to test error reporting.
  - (c) *EnableReport:* Faults will only be signaled (the error detection pin is set to "1") if this option is enabled. Otherwise, erroneous data is returned.



This option is introduced to test error detection mechanisms. However, as stated earlier, error detection is not part of this dissertation thesis.

Although the implemented error detection reports the location of the error, the exact bit position cannot be determined. To simulate the semantics of existing error detection and correction methods as close as possible, a minimal granularity exists in which errors are detected. In the implemented module, this is the size of a word (4 bytes on a 32 bit platform). This matches, for example, the granularity of parity bits or ECC which also operates on words or multiples of words. In addition to the minimal granularity of one word, the actual used detection granularity depends also on the requested amount of bytes of the corresponding bus transfer. Consequently, if, for example, a cache triggers a burst transfer to replace a complete cache line, the detection granularity will be the whole cache line. Hence, the error location register will contain the start address of the burst transfer and the error type register will contain the size of a cache line. By reading these report registers, the error handling method receives the base address of the fault but cannot make any assumption on the exact location of the faulty bit(s). Thus, it has to be assumed that all bits are erroneous.

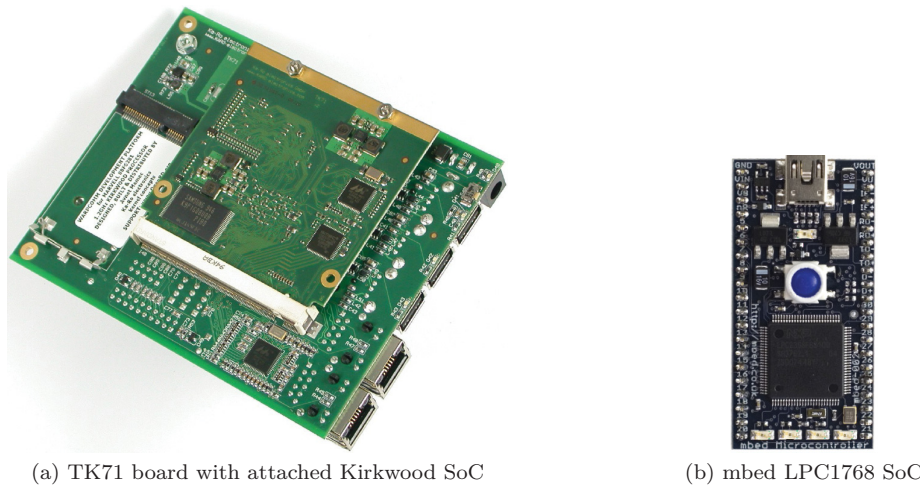
To sum up, errors are reported on access in the granularity of the corresponding bus request, but at least with 4 byte granularity. The main advantage of this implementation of a transient memory fault model is that the system under test is not disturbed by the injection process. Consequently, this memory model is used to study real-time impacts imposed by error handling.

### 3.3.3 Injection into Demonstrator Platform

To show the feasibility of the flexible error handling approach presented in the next chapter, a real hardware platform is used as demonstrator. The platform is based on an ARM926-based system [ARM14] implemented by a Marvell Kirkwood 88F681 SoC [Mar14]. To interface the Kirkwood SoC, a TK71 development board [Kar14] is employed (cf. Figure 3.3(a)). Injection of faults is realized by using the On-Chip Debugger (OCD) provided by the Kirkwood processor. The next subsection demonstrates interfacing the OCD with a Joint Test Action Group (JTAG) interface. Thereafter, the injection procedure is detailed.

#### 3.3.3.1 JTAG-based Fault Injection

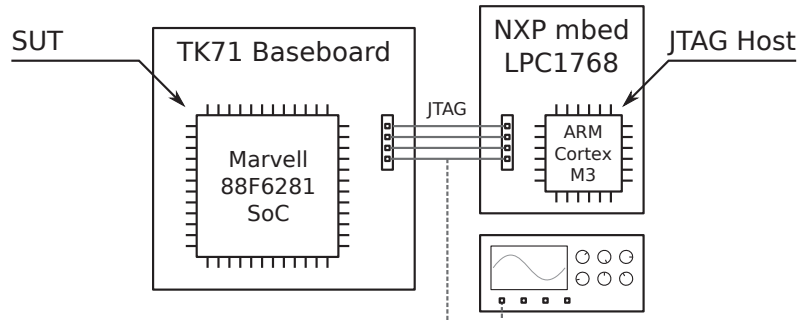
The IEEE 1149.1 Standard Test Access Port and Boundary-Scan Architecture, also known as JTAG – from the Joint Test Action Group created that standard – is a four-wire serial interface. Its original purpose was to facilitate automated connectivity test of populated printed circuit boards (PCBs). Due to the extensible design, JTAG is nowadays also used as a hardware interface for in-circuit programming and debugging of code running on microcontrollers and SoCs. Typically, JTAG consists of the following components:



**Figure 3.3:** Demonstrator platform and injection board

1. **Test Access Port (TAP):** The test access port, also called JTAG port, is the physical connector to the TAP controller.
2. **TAP Controller:** The TAP controller is a state machine controlling the test logic. It is controlled by the clock (TCK) and mode select (TMS) signals. By moving through the states, the JTAG host can select one of several shift registers. For reading and writing registers, the data out (TDO) and data input (TDI) signals are used. The shift registers are also clocked by TCK, but only when the state machine is in one of two specific states. The first state has to be used to write the instruction register. The instruction register selects a data register which will be accessed if the TAP controller is in the second state. Within those two states the state machine is only sensitive to changes on TMS. Hence, bits can be shifted at the frequency of the clock signal without any delays imposed by additional state changes. However, this also creates a challenge. Two signals have to be altered at the same point in time at the end of a data transfer: TMS has to signal a change of state while the last bit is moved into the shift register.
3. **Shift Registers:** The shift registers can be used to build a system that can arbitrarily access chip internals. The JTAG standard only specifies an interface used for boundary scans. This is direct access to the chip pins to check for short and open circuits on a PCB. However, proprietary extensions can be implemented as well. The bit width of the registers is not specified by the standard. In the case of the Kirkwood SoC it varies from 4 to 67 bits.

The first approach realized to inject faults was to use OpenOCD [Ope14] and a USB-JTAG adapter to interface a PC with the Kirkwood SoC. OpenOCD is an open source JTAG debugger software supporting a huge variety of different architectures and OCDs. Scripting OpenOCD is supported via Tcl. Injection scripts are used to



**Figure 3.4:** JTAG injection setup

regularly stop the target CPU to modify a value in memory. Due to the overhead imposed by USB transfers, those operations require 38.90 milliseconds every time the processor is accessed. This time is too long to meet real-time requirements, since the processor cannot execute any instruction during this period.

To speedup the injection process a microcontroller-based design is used. In this dissertation the microcontroller is a NXP mbed LPC1768 [NXP14] (cf. Figure 3.3(b)). This microcontroller, referred to as JTAG host for the remainder of this section, consists of an ARM Cortex M3 Core clocked at 96 MHz, 32 KiB RAM, and 512 KiB Flash. The injection setup is shown in Figure 3.4. The JTAG host is connected to the system under test (SUT) via JTAG. To measure the duration of a single fault injection, an oscilloscope is attached to the JTAG clock line. Most of the speedup is achieved due to direct connection of the microcontroller to the JTAG port without using USB. For fast data transfer, the SPI unit of the microcontroller is used. As mentioned previously, the TAP controller has to change the state while the last bit is moved into the shift register. This means to alter two signals at the same point in time. The SPI unit does not support such an operation. Hence, the last byte is transferred in software. Only in this way it is possible to change two signals at the same point in time.

The fault injection procedure is described in the next subsection. The performance of the microcontroller-based injection is evaluated afterwards.

### 3.3.3.2 Fault Injection Procedure

The basic idea of using debug units is to periodically interrupt the normal operation of the SUT to inject faults. However, the most challenging part is the implementation of error reporting due to the goal that the behavior of real world hardware should be simulated as close as possible. As described in the previous sections, hardware will typically report an error if the requested resource is affected. This results in two challenges. First, the used components have to be known. Second, the affected components have to be determined. To resolve the first problem, the currently executed instruction on the target processor is decoded as soon as the processor is interrupted by the OCD. By taking the content of the registers into

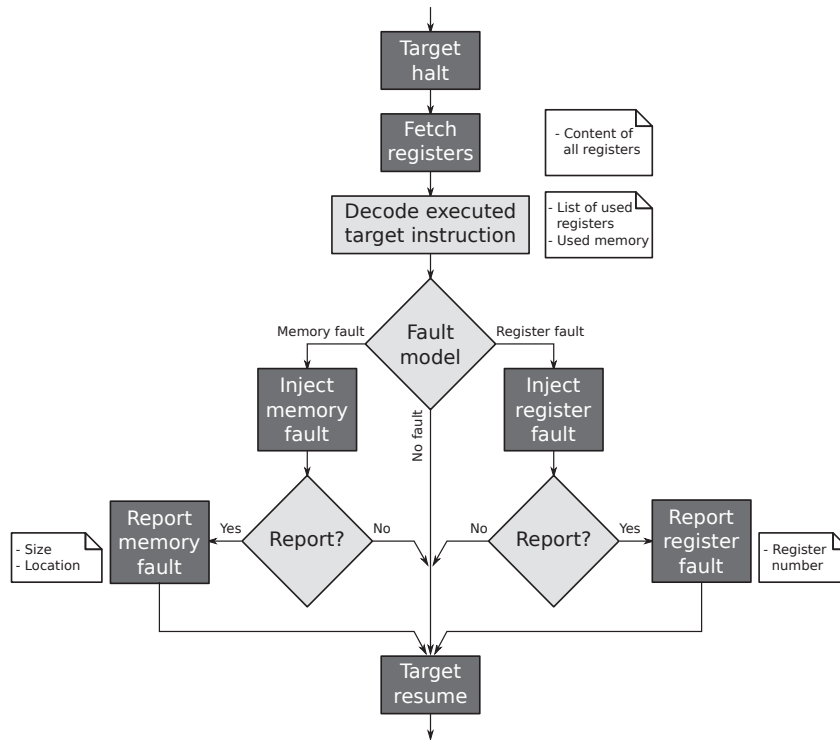


Figure 3.5: JTAG fault injection algorithm

account, it is possible to determine the set of used components. In case of load / store operations it is furthermore possible to compute the source / target addresses, respectively. The solution of the second problem could be a map which records all injected faults. Such a map would allow checking whether the target system accesses a faulty component or not. In the first case, a fault is reported. However, using such a map would lead to a huge disadvantage. It would require decoding and checking every executed instruction of the target system. On each write access, fault markings have to be cleared since overwriting would eliminate all transient faults in the written range. Whereas, on each read access the corresponding error markings will have to be checked to determine if the actually used components are erroneous. Decoding and checking every instruction means running the target system in single step mode. Consequently, the execution speed of instructions of the target system is defined by the execution speed of JTAG, the OCD, and the algorithms executed on the injection hardware.

To deal with this problem, the approach applied in this thesis, turns this order around. Instead of reporting previously injected faults when they are accessed, faults are reported when they are injected. In other words: *Fault injection is aware of the instruction currently executed.* Hence, after decoding, faults are directly injected into low reliable components used by the currently executed instruction. The corresponding fault injection procedure is depicted in Figure 3.5. All operations involving communication via JTAG are shaded dark. The whole injection procedure

Used path	Reporting	Time [ms]
No fault	–	1.55
Memory fault	yes	4.48
Memory fault	no	2.64
Register fault	yes	3.82
Register fault	no	1.94

**Table 3.1:** JTAG injection results

is repeated periodically at each timer interrupt of the JTAG host. The time between two interrupts is freely configurable. The first step is to stop the SUT by sending a debug request via JTAG. At this point in time, the OCD takes control over the CPU. Hence, no instructions of the SUT’s application are executed anymore until target resume is called in the last step. To keep jitter as low as possible, it is very important that all steps between target halt and target resume are executed as fast as possible. Directly after interrupting the target system, the register contents are fetched to completely decode the interrupted instruction. Based on the components used and decisions of the fault model, memory or register faults can be injected. It is also possible that no fault at all will be injected. This can be the case if, for example, the decoded operation is a co-processor instruction.

To inject a memory fault, the decoded address is used as base. Depending on the memory access granularity of the target instruction, the range in which bits are flipped is specified. In the used model, the minimum range is the size of a word. In cases of half word or byte loads, the memory address will be aligned to the closest lower possible word aligned address and the size will be set to word length. A register fault is injected by flipping bits in one of the used registers of the target instruction. Register bit flips are always performed on the full register, regardless of the instruction’s access pattern. Depending on the fault model the injected fault can be reported to the SUT via interrupt or not. In case of an injected memory fault, the address and injection range are reported. In the register fault case, the affected register is reported.

### 3.3.3.3 Fault Injection Performance

To evaluate the performance of the previously described approach, every path of the fault injection procedure is measured. The results are depicted in Table 3.1. Obviously, the fastest path is the one with no injected fault. This can be the case if the interrupted instruction only uses high reliable silicon or the desired low reliable component was not used. In contrast to injecting a register fault, injecting a memory fault requires additional read and write operations via JTAG. The read fetches the memory range in which bits have to be flipped. After flipping, the modified values have to be written back. This reflects in higher injection durations. Reporting the fault to the operating system also involves additional communication via JTAG. Consequently, the required time to execute the injection procedure will

increase as well.

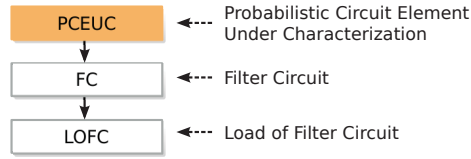
As mentioned earlier, OpenOCD requires 38.90 ms to perform a single fault injection with reporting. Decoding the instruction was not implemented. Compared to OpenOCD, the solution used in this thesis is more than 8.5 times faster. However, far more interesting is the comparison with the FPGA-based JTAG implementation of Garcia et al. in [PGLOGV+07]. For injecting one memory bit flip their implementation needs 2 ms. To be comparable, the microcontroller-based solution was modified by turning off error reporting and instruction decoding. Since instruction decoding is turned off, faults are injected into random locations inside the memory. With an injection time of 2.24 ms, the speed of the FPGA-based solution is nearly reached. However, the main drawbacks of the FPGA-based solution are low flexibility and low adaptability. Due to the complexity of FPGA programming, it is very difficult to implement complex fault models or to exchange the fault model. Furthermore, the simulation of real-world error reporting is hard to implement. Therefore, an instruction decoder running on the FPGA would be required to keep track of currently used hardware components. Implementing such a decoder for the FPGA is generally possible. However, due to the high complexity of FPGA programming, such a solution will be very expensive.

In contrast to expensive equipment, like particle beams, the approach used in this dissertation is very cheap. However, some drawbacks exist. Using the OCD limits the possible components to which faults can be injected. In fact, injection is possible on every component writable by the CPU using normal machine instructions. This includes the register file and the memory. Theoretically, peripheral devices can also be accessed if they are memory mapped. A further limitation is the fault coverage. Due to the instruction-aware injection, no guarantees can be given whether the whole memory and register space is covered or not. In one extreme, a program which idles most of the time is likely to never experience a fault injection. A `while(1)` loop in the idle body, for example, neither requires a register nor a memory transaction. The other extreme is a program permanently using the same memory range. With a very high probability, a huge amount of memory faults is injected in this range only. The presented approach has a further limitation. Due to generating faults on access, faults cannot accumulate.

However, since this injection approach is only used to inject faults into the physically available demonstrator platform, it is possible to cope with the limitations. To investigate the behavior of the flexible error handling approach under more realistic fault injection scenarios, the CoMET-based injection solution is used.

### 3.4 Probabilistic Error Model

The term probabilistic CMOS (PCMOS) was first introduced by Palem [Pal05] in the context of probabilistic bits (PBITS) and probabilistic computing [Pal03]. Instead of avoiding errors in ICs, PCMOS-based computing proposes to tolerate occasional errors in order to exploit the trade-off between correctness of circuit op-



**Figure 3.6:** Three-Stage Models for circuits (taken from [SBL+11])

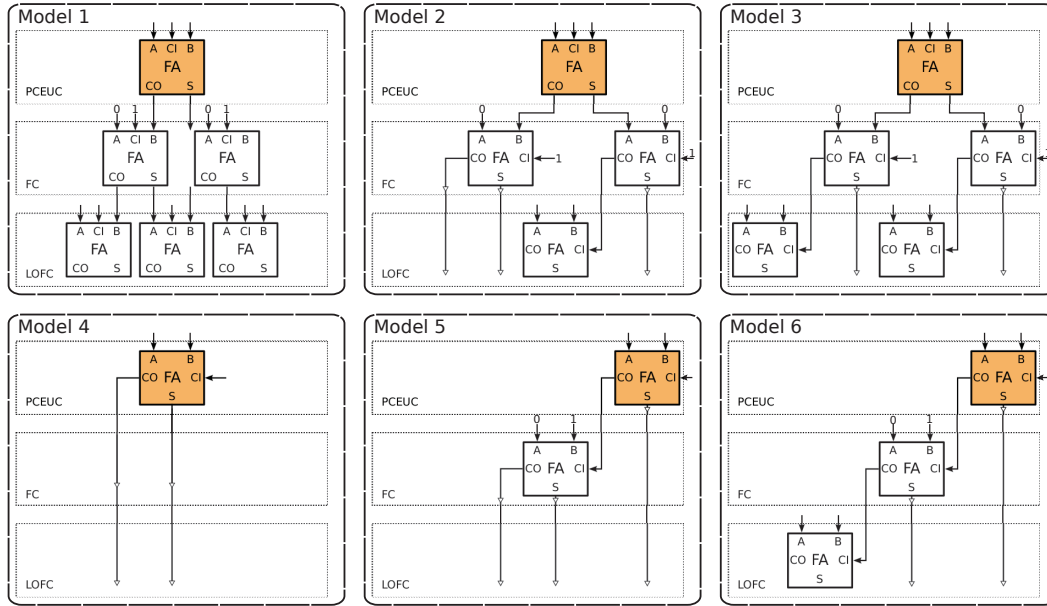
eration, energy consumption, and performance. The basic idea is to allow bits to have a *probability* of being zero or one. Consequently, logic functions have probabilistic outputs instead of deterministic outputs. In this section, probabilistic components considered in this thesis are described and their use in a system simulation environment is demonstrated.

### 3.4.1 Supply Voltage Schemes

In the context of silicon-based computation, noise is predicted to lead to PCMOS behavior in the future [SBL+11; BLL+10]. If the average noise level injected into such systems is constant, the probability  $p$  of jitter at the output of a chip depends on the supply voltage  $v$ . By lowering  $v$ ,  $p$  will increase. While the susceptibility of an erroneous output will increase, the power consumption will be quadratically reduced if the supply voltage is decreased. One important property of the probabilistic components is that each low-level probabilistic circuit element, like a single-bit probabilistic full adder (PFA), can be supplied with a different voltage. By combining single-bit PFAs to form larger circuits, this leads to various non-uniform *biased voltage scaling* (BIVOS) schemes. In such BIVOS schemes, more significant bits are typically provided with a higher supply voltage than less significant bits. Hence, the probability of noise-induced errors in more significant bits of a word is reduced. For small benchmarks, [CML+08] and [PCK+09] show that by using BIVOS, the accuracy of a probabilistic ripple carry adder can be increased compared to *uniform voltage scaling* (UVOS). In UVOS-based circuits, all bits are supplied with the same voltage. To be comparable, the power budgets for the UVOS-based and BIVOS-based adders are the same in all the experiments.

### 3.4.2 Three-Stage Model

Low-level circuit simulators like HSPICE are used to determine the susceptibility to noise of PCMOS-based hardware. The goal of the HSPICE simulation is to determine the probability of a bit flip at the circuit's output. The probabilities gained by such a simulation are very accurate. However, HSPICE simulations of whole ICs are complex and hence have a long runtime. Therefore, Singh et al. propose a divide and conquer strategy to cope with this problem [SBL+11]. The basic idea is to characterize subcomponents of the circuit with HSPICE and to compose the results in a mathematical model [LML+10]. The three-stage model



**Figure 3.7:** The used Three-Stage Models of different full adders (based on [SBL+11])

is used to characterize a probabilistic circuit element. Thereby, the filter effects of succeeding circuits are taken into account. In the model, three components play a central role (cf. Figure 3.6):

1. **Probabilistic Circuit Element Under Characterization (PCEUC):** This is the circuit that has to be characterized.
2. **Filter Circuit (FC):** The load of the PCEUC can be referred to as filter circuit due to its filter effect for errors generated by the PCEUC.
3. **Load of Filter Circuit (LOFC):** The load of the load of the PCEUC determines the filter capability of the FC.

It can be shown that the load of a circuit plays a major role for the propagation delay<sup>1</sup> of errors. Therefore, the load of the PCEUC has to be considered during characterization. The FC and LOFC have to be deterministic during simulation. Hence, only the filter effects are taken into account. Furthermore, the FC and LOFC have to be configured in a way that no logical masking can occur. For example, if the FC is an AND gate, the input not driven by the PCEUC has to be set to '1'.

This thesis considers probabilistic behavior of adders and multipliers. The three-stage models for probabilistic full adders (PFA) are used as basic components for building multi-bit adders and multipliers. The three-stage model yields fast simulation time while the accuracy is within 7–8% of more complex SPICE-based models

<sup>1</sup>The propagation delay is the duration between the point in time when the input of a logic gate becomes stable and the point in time when the output of that gate becomes stable.



Model	1.1 V		1.0 V		0.9 V		0.8 V	
	$p_s$	$p_c$	$p_s$	$p_c$	$p_s$	$p_c$	$p_s$	$p_c$
1	4.17e-08	0	4.17e-06	4.17e-06	8.33e-06	8.33e-06	3.75e-05	1.67e-05
2	4.17e-08	0	4.17e-06	0	8.33e-06	4.17e-06	4.17e-05	2.08e-05
3	4.17e-08	0	8.33e-06	0	1.67e-05	8.33e-06	4.17e-05	1.67e-05
4	4.17e-08	0	4.17e-07	0	1.25e-05	0	1.25e-05	3.75e-05
5	4.17e-08	0	4.17e-07	0	1.25e-05	0	5.00e-05	8.22e-06
6	4.17e-08	0	4.17e-07	0	1.25e-05	0	4.17e-05	4.17e-06

**Table 3.2:** Effective error rates

for such components. However, complex SPICE simulations take orders of magnitude more time to execute. The error rates calculated with the three-stage models of [SBL+11] are fairly accurate and fast to compute, which enables their use in a full system simulation of a complex application. The models required to build a probabilistic ripple carry adder and a probabilistic Wallace tree multiplier are depicted in Figure 3.7. As can be seen, the PCEUC is always a full adder. However, the filter circuit and the load of the filter circuit are different in each model. While Model 1 has a full adder as load for the sum as well as the carry output, Model 4 has only drivers as load. The load of the sum and carry output can be different as well. This is the case, for example, in Model 6. Due to the different filter effects, the probability of a wrong output is varying.

Table 3.2 shows the measured probabilities of a flipped bit using HSPICE. The depicted values show the probability of an erroneous sum output  $p_s(t, m, v, n)$  and carry output  $p_c(t, m, v, n)$  of a probabilistic full adder. The probability depends on the technology  $t$ , the model  $m$ , the supply voltage  $v$ , and the noise  $n$ . In all experiments,  $t$  is fixed to 90 nm technology and  $n$  is within 0 V and +0.12 V. This noise level equals ten percent of the nominal supply voltage of 90 nm ICs which is 1.2 V. The measured results for 1.2 V are omitted in Table 3.2 since the noise is too low to affect the circuit when operating at this voltage level. However, if the supply voltage is lowered while the maximum noise level remains at 0.12 V, the probability of an erroneous output will increase. The lower the supply voltage, the higher the probability of an error. Consequently, the FA operating at 0.8 V experiences the highest probability of an erroneous output. As described previously, the load of the PCEUC also influences its resilience against noise. This behavior can be seen in Table 3.2. For example, Model 1 is more susceptible to noise in the carry output than the other models.

In the next subsections the usage of the different three-stage models of a PFA is demonstrated for a probabilistic adder and a probabilistic multiplier.

### 3.4.2.1 Adder Implementation

The probabilistic adder considered in this thesis is a probabilistic ripple carry adder (PRCA). The PRCA simulation uses different models of the PFA, depending on the different output loads of each full adder in the overall circuit. For clarity,

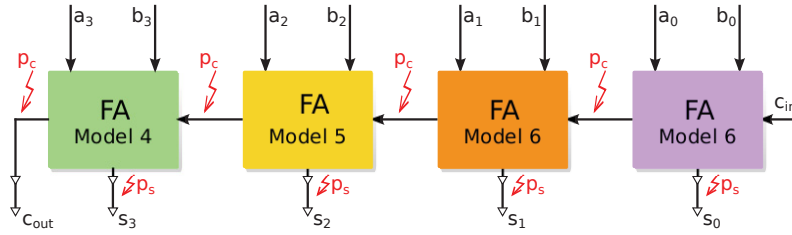


Figure 3.8: Probabilistic ripple carry adder (PRCA)

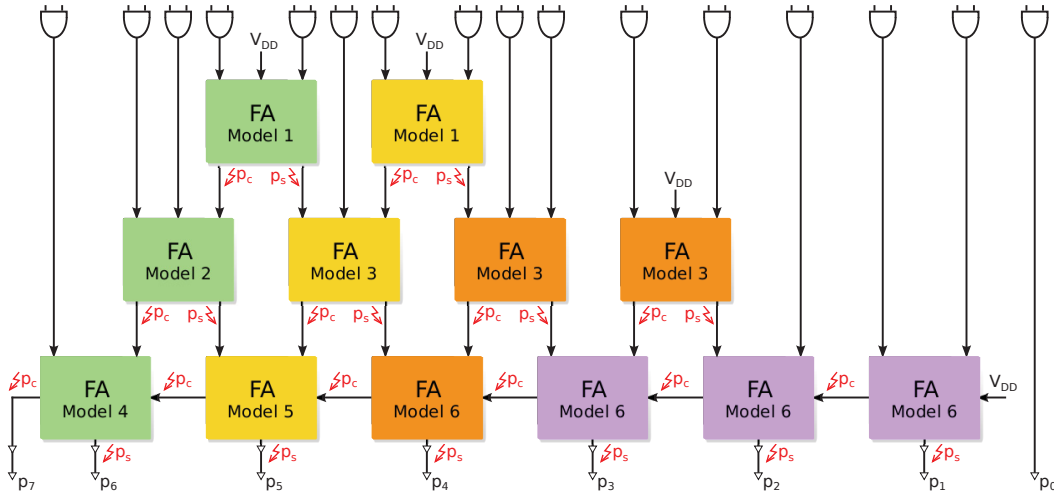


Figure 3.9: Probabilistic Wallace tree multiplier (PWTM)

Figure 3.8 shows a four-bit adder instead of the 32-bit adder actually used. In order to construct a PRCA using the three-stage-model, three different PFA models are required. The PRCA simulation starts with the PFA calculating the least significant bit  $s_0$  on the right hand side of Figure 3.8. The sum and carry bits are calculated deterministically. Thereafter, probabilistic behavior is modeled by bit-flips on the interconnections. These bit-flips will occur according to the error probabilities  $p_c$  and  $p_s$  as determined by the previous HSPICE simulation.

The different colors of the PFAs denote different supply voltages for a BIVOS scheme with four available voltages. In this configuration, for example, the PFA calculating  $s_0$  is supplied with the lowest voltage,  $s_1$  with the second lowest,  $s_2$  with the second highest, and the PFA for  $s_3$  with the highest voltage. Consequently, the likeliness for an erroneous output in more significant bits is lower than in less significant bits.

### 3.4.2.2 Multiplier Implementation

The multiplier used in this dissertation is a probabilistic version of a Wallace tree multiplier (PWTM) [Wal64]. A four-bit PWTM is depicted in Figure 3.9. Like the PRCA, the PWTM is constructed from multiple probabilistic full adders. For clarity, only a four bit multiplier is shown. Bit-flips occur analogously to the PRCA

case. In the multiplier case, all three-stage models depicted in Figure 3.7 are used.

Each PFA can be supplied with a different supply voltage enabling the analysis of different BIVOS configurations. Again, the coloring of the PFAs shows the distribution of four different supply voltages. Like in the PRCA case, more significant bits are supplied with higher voltages to decrease the probability of an erroneous output in higher bits.

### 3.4.3 Probabilistic ARM Platform

In order to perform an analysis of a complex real-world application, an execution platform for the application binary is required. In this dissertation, the MPARM ARMv3m architecture simulator [BBB+05] was extended to include PRCA and PWTM components in the CPU core in addition to the standard deterministic ALU and multiplier. MPARM uses SystemC to interface a software based ARM core, called SWARM [Dal03]. The probabilistic version of a ripple carry adder and a Wallace tree multiplier are added to SWARM. These probabilistic components are used by four new instructions. All other instructions continue to use deterministic components only. The new instructions are addition (padd), subtraction (psub), and reverse subtraction (prsb) using the PRCA, as well as multiplication (pmul) using the PWTM.

**Listing 3.1:** Function without probabilistic instructions

```

1 void enter(unreliable uchar *ptr, unreliable int q_delta) {
2   unreliable int i = *ptr + ((q_delta + 32) >> 6);
3   *ptr=Clip(i);
4 }
```

To support the different mnemonics of the probabilistic instructions, different functions exist. For example, to transform the code depicted in Listing 3.1, the add-operations have to be substituted by `__addsw` and `__addisw`. Hereby, `__addsw` means an addition of two signed words and `__addisw` performs a probabilistic add of a signed word with an immediate value. The transformed code is shown in Listing 3.2. This transformation can be automated by using a customized compiler [SHM+13]. Since `*ptr`, `q_delta`, and `i` are *unreliable*, this compiler can substitute all operations calculating `i` with probabilistic versions. Details of the compiler and the usage of *unreliable* are explained in detail in Section 5.1 and Section 4.2.4.1, respectively.

**Listing 3.2:** Code transformed to use the PRCA

```

1 void enter(uchar *ptr, int q_delta) {
2   int i = __paddsw(*ptr, (__paddisw(q_delta, 32) >> 6));
3   *ptr = Clip(i);
4 }
```

### 3.5 Task Model

The previous sections described all models which are important for this dissertation regarding faults and errors. This section briefly introduces the task model used later on. The task model is based on the task model of library operating systems, specifically RTEMS [OAR14]. A library OS is a special type of OS where the OS is provided in form of a library which is linked to the application by an ordinary linker. In such a system, only **one** *address space* and **one** *process* exist. An address space defines a range of discrete uniquely identifiable addresses to make an address unambiguous. A process is an entity to which resources are bound to. Such resources can be file handles or allocated heap data. A process can be comprised of **multiple** *tasks*. At least one task is required in a process to have an executable process. The terms *task* and *thread* are used equally in this dissertation. The tasks considered are periodic real-time tasks.

A task set  $T$  is defined as:

$$T = \{T_1, \dots, T_n\} \quad i = 1, \dots, n \quad (3.1)$$

where  $n$  is the number of tasks.

A task  $T_i$  is a tuple:

$$T_i = (c_i, d_i, p_i) \quad (3.2)$$

where  $c_i$  is the execution time,  $d_i$  is the relative deadline, and  $p_i$  is the period of task  $T_i$ .

Tasks can have dependencies. The resulting precedence system  $G$  is defined as directed graph:

$$G = (T, E) \quad (3.3)$$

where  $T$  is the previously defined task set. An edge  $e = (T_v, T_w)$  is element in  $E$ , iff  $T_w$  depends on  $T_v$ . An alternative writing is  $T_v \prec T_w$ .

### 3.6 Quality of Service Metrics

The flexible error handling approach presented in this dissertation is based on a best-effort approach. Hence, it is to be expected that some errors will propagate to the output. Nevertheless, it has to be evaluated whether the output is acceptable for the user. In order to distinguish between perceptible and non-perceptible errors, different metrics can be used. Since the main application used in this dissertation is a H.264-based video decoder, the next subsections describe quality of service metrics used in image compression.

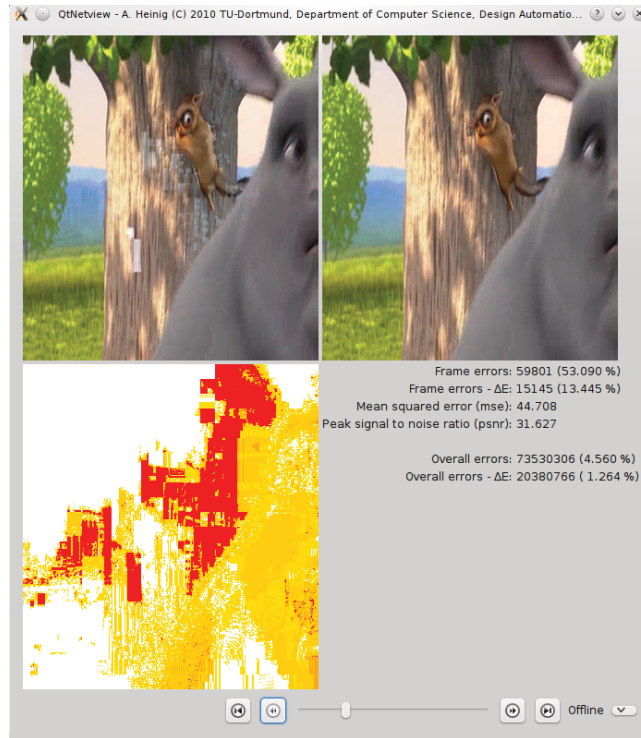


Figure 3.10: QTNNetview showing QoS metrics

### 3.6.1 $\Delta E$

The  $\Delta E$  metric is standardized in ISO 12647, which describes the distance between two colors.  $\Delta E$  is the Euclidean distance between two colors. Higher distance means higher deviation. In this dissertation a  $\Delta E$  value smaller than 5.0 is defined as a non-perceptible error. In Figure 3.10 a tool is shown which visualizes the  $\Delta E$  metric for each pixel. In the upper left corner the frame under error injection is shown. The correctly decoded frame is depicted on the right hand side. A difference frame shows perceptible and non-perceptible errors (indicated by the intensity of the colors). In addition, statistics giving the percentage of errors accumulated over the whole video and statistics of the current frame are displayed.

### 3.6.2 Peak Signal-To-Noise Ratio

The peak signal-to-noise ratio is most commonly used to measure the quality of reconstruction of lossy compression codecs. PSNR is defined as

$$\text{PSNR} = 10 \log_{10} \frac{2^B - 1}{MSE} \text{dB} \quad (3.4)$$

where  $MSE$  denotes the mean squared error between reconstructed and original frame, and  $B$  is the number of bits per sample. A higher PSNR indicates higher quality. The QTNNetview tool also displays the PSNR QoS metric (cf. Figure 3.10).



# Flexible Error Handling

---

## Contents

---

<b>4.1 Flexible Error Handling Strategy . . . . .</b>	<b>51</b>
<b>4.2 Error Classification . . . . .</b>	<b>53</b>
4.2.1 H.264 Application Analysis . . . . .	54
4.2.2 LEGO Mindstorms Application . . . . .	56
4.2.3 Further Examples . . . . .	59
4.2.4 Classification Annotation . . . . .	59
4.2.5 Implications . . . . .	61
<b>4.3 Real-Time Aspect . . . . .</b>	<b>62</b>
4.3.1 A kind of Priority Inversion Problem . . . . .	63
4.3.2 Subscriber Model . . . . .	63
4.3.3 Using the Subscriber Model to Schedule Error Correction . .	67
<b>4.4 Summary . . . . .</b>	<b>69</b>

---

In this chapter, the core subject of this dissertation – the flexible error handling approach – is discussed. The necessity of flexible error handling is justified by the observation that errors can have different impacts on the quality of service of an application. The differences are shown by several examples within this chapter. If every error was handled alike, the different impacts would not be exploited. However, just these differences together with flexibility are the keys to build a light-weight error handling solution which is applicable especially to embedded systems where resources like available memory and computing power are scarce. Flexibility in this sense means to decide how to cope with errors occurring during runtime based on the system state.

The key concepts of flexible error handling are depicted in this chapter. In the next chapter the realization and combination of these concepts is shown.

## 4.1 Flexible Error Handling Strategy

The idea of flexible error handling is shown in Figure 4.1. In the depicted scenarios an application is running and at some point in time an error occurs. This point in time is indicated by the flash. In a naive approach, every error would be handled alike without considering the available resources. This can lead to deadline misses.

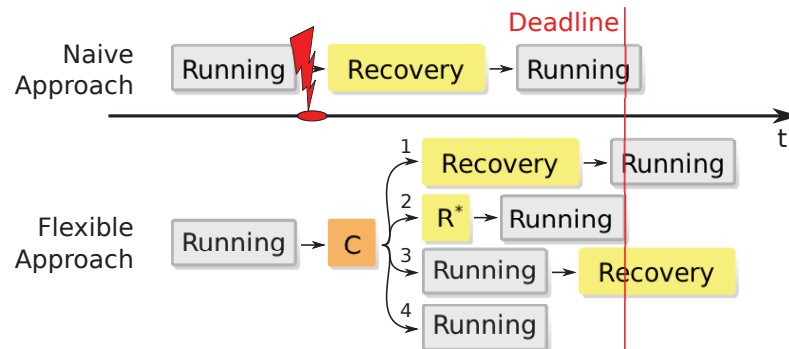


Figure 4.1: Flexible error handling

In the flexible error handling approach presented in this dissertation, the error is classified first, labeled as “C”. The objective of the classification is to determine **if**, **how**, and **when** errors have to be corrected.

(**if**) Whether errors have to be handled or not depends mainly on the error impact. Errors which lead to high impacts are, for example, errors which lead to exceptions due to wrong memory accesses. If an error has a high impact, error correction will be mandatory. In contrast, if an error has only low impact, e.g. an unused memory cell is affected, further handling is optional. In between these two extremes, errors exist which do not lead to crashes, but have an impact on the quality of service (QoS). Such a QoS impact can be multifaceted. Examples are jitter, deadline misses, lower signal-to-noise ratios, etc. To distinguish between errors with high impact potentially leading to crashes and errors which do not lead to crashes, static code analysis methods are used.

(**how**) Error handling depends on the available error correction methods, the error impact, and the available resources. To realize the flexible error handling approach it is assumed that two generic correction methods are always available. The first one is *checkpoint-and-recovery*. The second method is *ignore*, which is the empty recovery method (case 4 in Figure 4.1). In addition to these two methods, the application programmer can provide further error correction methods (labeled as “R\*” in Figure 4.1). An example for such a method can be a specialized correction of an erroneous motion vector in video decoders. Since the motion vector is responsible to copy a macro block into another location inside a frame, a corrupted vector can have fatal consequences, like copying the macro block outside of the frame memory location. However, if the motion vector is set to zero, no fatal behavior can be observed. Hence, an application specific method to transform a corrupted motion vector into a valid – but incorrect – state is to reset the corresponding vector components to zero. This avoids application crashes due to erroneous motion vectors, but also leads to degradation in the QoS.



Each correction method has different properties, like worst case correction duration and correction quality. In the case of *checkpoint-and-recovery*, the correction quality will be optimal if all data affected by errors are stored in the checkpoint are corrected. However, the execution time can be rather high. On the contrary, if *ignore* is used, the execution time will be nearly zero, but the error effect on the data will still remain. Other methods are located in between these two extremes. For example, setting an affected motion vector to zero would lead to a small disturbance in the video output. However, this method would be very fast and hence the probability is higher that the application adheres to its deadline.

**(when)** The scheduling algorithm has major influence on the question when an error correction method should be scheduled. Typically, the task with the highest priority is executed. Hence, if a high priority task is affected, error correction has to be scheduled immediately (cases 1 and 2 in Figure 4.1). If a low priority task is affected, the high priority task can continue execution and the error handling will be delayed (case 3).

By considering **if**, **when**, and **how** an error has to be corrected, the flexible error handling approach is able to select an error handling method which is suited for the current situation. In the worst case, the same error correction method as in the naive approach has to be scheduled immediately, since the highest prioritized task is affected by an error which has severe impact. Compared to the naive approach, this results in an additional overhead caused by the classification phase. In the best case, however, flexible error handling can completely ignore the error and hence save resources and adhere to the deadline. Between both extremes, the presented approach is able to schedule error correction by considering the available slack time and the QoS requirements of the application.

To be able to determine if, how, and when errors have to be corrected, error impact and mapping of tasks to errors have to be known during runtime. The next section shows different error impacts and shows how to specify possible error impacts in the source code of the application. Afterwards, mapping of errors to tasks is considered.

## 4.2 Error Classification

Faults can manifest themselves in different ways. It can be easily observed that not every fault has the same outcome. A fault can lead, for example, to errors responsible for application crashes, or just to disturbance in the output of the application. Consequently, a *classification* is required that provides information on the severity of detected errors. The goal of the classification is to partition possible errors according to their impact on the QoS of the application. Hence, such a classification can provide a basis to determine error correction precedence. If not enough time for error correction is available, errors with higher impact will be handled while

correction of low-impact errors can be skipped. Thus, the classification provides an important feature of the flexible error handling approach by providing support for the question **if** an error has to be handled or not.

The other important aspect of the classification is to specify a set of applicable error correction methods for the error classes. This set defines the basis to solve the question **how** errors can be handled. To specify possible correction methods, designers have to annotate the applications. Annotations are introduced in Section 4.2.4. In the next subsections, different kinds of applications are investigated to show a variety of possible error impacts and the resulting classifications.


### 4.2.1 H.264 Application Analysis

H.264 or ISO/IEC 14496-10 [Dra03] is a video compression standard used for Blu-ray discs, videoconferencing, digital video broadcasting, and many others. It is typically considered as an embedded application with soft real-time requirements. H.264 supports different profiles. The profile with the highest quality is the High 4:4:4 Predictive Profile (Hi444PP). It supports 4:4:4 chroma sampling, up to 14 bits per sample, and lossless region coding. The Constrained Baseline Profile (CBP) yields the lowest quality. In this dissertation, the source code of a simple CBP-based decoder is used [FB04]. The decoder consists of about 3000 lines of C code and 1000 lines of header files. It implements the following subset of CBP:

- I and P slices
- 4:2:0 chroma format
- 8 bit sample depth
- CAVLC entropy coding

Arbitrary slice ordering, redundant slices, and multiple reference frames are not supported in this implementation. More implementation details of the H.264 video application are shown in Chapter 6. To investigate different outcomes of errors, fault injection is used. For the injection purpose, thread based injection is used, as described in Section 3.3.1. Therefore, the H.264 decoder application is executed on a Linux system with an additional thread executed concurrently. This thread is responsible for fault injection.

The fault injection experiment reveals different behaviors of the decoder application. The resulting classification is depicted in Table 4.1. Program terminations (crashes) have the highest impacts on the QoS since the playback of the video is stopped. In such situations only restoring a checkpoint is feasible. The most critical part of decoding is the processing of header information. Here, a flipped bit can have fatal consequences for the decoding of the whole frame and even of subsequent frames. If, for example, the frame type is decoded incorrectly, the header bits will be misinterpreted, since different frame types use different layouts for header data. The majority of header and data words are integers encoded with

Impact	Possible cause	Error handling methods
 High	Program termination	Rollback
	Corrupted frame input	Rollback, Redisplay last frame
	Disturbance in frame header	Rollback, Spare frame, Redisplay last frame
	Disturbance in motion vector	Rollback, Set to zero
	Disturbance in macro block	Rollback, Copy neighbor block, Ignore
None	Disturbance in single pixel	Rollback, Copy neighbor pixel, Ignore
	Unused memory	Ignore

**Table 4.1:** Error classification for H.264 decoder

*Exponential-Golomb codes*, which can encode arbitrary positive integer numbers. The construction scheme favors small numbers by assigning them shorter codes. A value of  $x$  is represented by:

$$x = 2^n - 1 + v \quad v, n \in \mathbb{N} \quad (4.1)$$

$n$  and  $v$  are directly related to the encoding in the bit stream.  $n$  is the number of “0” bits in a continuous series of only “0” bits.  $v$  is a binary encoded offset represented by  $n$  bits. The separation of the  $n$  part and  $v$  part is a single “1” bit. For example, the decimal value “6” is represented as  $6 = 2^2 - 1 + 3$  which is encoded as “00111”. Errors affecting the bits representing  $n$  or the separator “1” bit not only change the decoded value. Due to the run-length encoding implemented by Exponential-Golomb codes, this leads to reading the input stream either not far enough or too far. Consequently, any subsequent read will automatically start at a wrong position in the stream. This leads to high deviations from the optimal video quality. Leaving such errors unattended will most likely causes crashes as well. Hence, error handling is mandatory. However, exact reconstruction of the original data is not always required. It is possible, for example, to redisplay the last frame and move forward to the next frame. On the one hand, this lowers the PSNR of the current frame and all following frames which are referencing the current frame. But, on the other hand, only a small impact on the real-time behavior of the application will be imposed, compared to restoring a complete checkpoint.

Disturbances in motion vectors can be classified as medium impact. In H.264 motion vectors are used to shift a macro block to a new location within a frame. Motion vectors are hence well suited to encode movements in the video. If motion vectors are corrupted, two possible outcomes can be observed: incorrectly placed macro blocks and application crashes. Misplacement of macro blocks will occur if the source macro block is moved to a wrong destination address within the frame.

But, it can also happen that the corresponding macro block gets shifted out of the frame. Hereby, other memory will be overwritten which can lead to application crashes. Consequently, motion vectors have to be corrected. Besides recovery of a checkpoint, setting the motion vector to zero resets corrupted motion vectors in a valid state. This corresponds to no movement of the macro block.

The last category includes faults which have low impact without the possibility to crash the application. It is obvious that errors in unused memory regions can be ignored. Other faults, leading to errors which result only in minor disturbance of the output, can be ignored as well. Such examples are flipped bits in the luminance or chrominance pixel data of a frame. However, if such errors are (optionally) corrected, the QoS can be increased.

To conclude, errors affecting a video decoder application can lead to loss of precision, wrong calculation results, or crashes. In the next section software controlling a robot is considered to show the applicability of error classification in other application domains as well.

#### 4.2.2 LEGO Mindstorms Application

The previously described H.264 decoder is a typical representative for multimedia applications. In the field of multimedia it is obvious that faults can have impacts reaching from fatal consequences, like application crashes, to only minor disturbances in the video output, like a wrong colored pixel. To show the applicability of the flexible error handling approach to control applications, a LEGO Mindstorm example is considered in this section. The application consists of two Mindstorm Robots referred to as Mindstorm A and Mindstorm B. In Figure 4.2 a robot is depicted. It consists of one NXT brick, one light sensor, one ultrasonic proximity detector, and two servo motors. The NXT brick is the processing unit of the Mindstorms. The light sensor measures the brightness. The measured values range from 0 (no light) to 1023 (very bright). To measure distances to other objects, the ultrasonic proximity detector emits pulses. With the integrated control unit the return interval of the reflected signal is measured and the resulting object distances are calculated. The two servo motors are responsible for locomotion of the robot. The motors also integrate a sensor measuring the rotation angle of the wheels. This allows querying the motor for the current position with the resolution of one degree.

The task of the robot A is to follow a line marked on the ground. The goal of B is to follow the line as fast as possible without colliding with A. Hereby, Mindstorm A acts as an obstacle for B. Mindstorm A follows the line but periodically randomizes its speed. Hence, B has to respond to the changing environment. The corresponding setup is depicted in Figure 4.3. Each Mindstorm has its own start and finish line at which the round trip times can be measured individually.

The maximum QoS can only be reached if neither the track of the line is lost nor collisions occur and B arrives in finish at the same moment in time as A arrives. The resulting classification is shown in Table 4.2. Like in the H.264 example, possible error handling methods are depicted as well. For the classification process, faults



Figure 4.2: LEGO Mindstorm-based robot

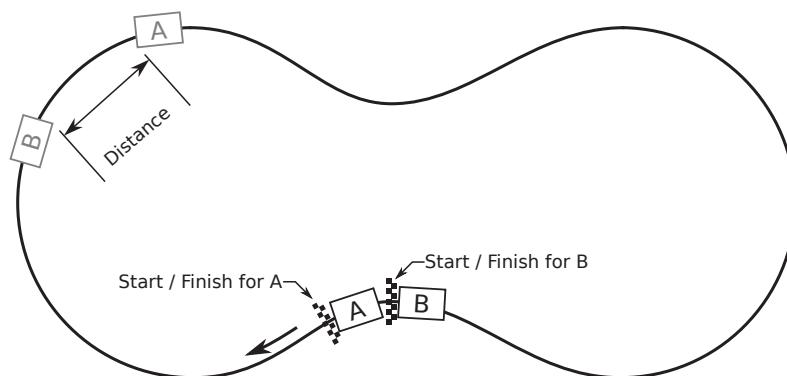


Figure 4.3: Experimental setup for robotic application

Impact	Possible cause	Error handling methods
High	Collision	-
	Lost track of line	Search for line
	Program termination	Rollback
	Over-steering	Rollback, Ignore
	Wrong acceleration	Rollback, Ignore
	Unnecessary braking	Rollback, Ignore
None	Unused memory	Ignore

**Table 4.2:** Error classification for robotic application

are only injected in Mindstorm B via a second thread.

Errors which only lead to longer round-trip times have low impact on the quality of service. This is the case, for example, if errors lead to unnecessary braking due to a wrongly calculated distance to Mindstorm A. In contrast, errors will have very high impact if they lead to application crashes, loss of line tracking, or collision. The latter outcome is worse. No error correction method exists in the case of a collision. Hence, errors which can lead to such an event have to be treated as fatal.

Although over-steering as well as wrong acceleration can lead to loss of line tracking or to collision, errors which lead to such effects are classified as medium impact. The reason therefore is that the application has an integrated resiliency against such errors due to the used proportional-integral-derivative (PID) controller paradigm. An overview of PID design is given by Ang et al. in [ACL05]. Briefly, the aim of a PID-based algorithm is to periodically adjust a control variable based on the measured state. However, the adjustment is not realized directly by using the difference to the expected value. Instead, the update  $u$  of control variable  $y$  is calculated by summation of a proportional part, integral part, and a derivative part. The PID formula considered in this dissertation is defined as follows:

$$u(t) = \underbrace{K_P * e(t)}_{\text{proportional}} + \underbrace{K_I * \int_{\tau=0}^t e(\tau)}_{\text{integral}} - \underbrace{K_D * \frac{d}{dt}y(t)}_{\text{differential}} \quad (4.2)$$

where  $e(t)$  is the deviation from the expected value for  $y(t)$ :  $e(t) = r(t) - y(t)$ , with  $r(t)$  is the expected value.  $K_P$ ,  $K_I$ , and  $K_D$  are constants used to adjust the controller.

One example of the application of a PID controller within the Mindstorm scenario is the distance control of B. The PID controller is used to adjust the motor speed according to the distance to A. In the following example it is assumed that an error affects the distance calculation of B. If the calculated distance is less than the real distance, B will brake unnecessary which will lead to no fatal consequences. However, if the wrong calculated distance is orders of magnitude higher than the real distance, B will accelerate to its maximum speed to close the gap. Without

a PID controller this leads most likely to a crash. In the case of the PID-based control algorithm, the robot is not immediately accelerated to its maximum speed due to the differential part of the PID equation. Only when the wrongly calculated distance is present over a longer period of time, the robot will accelerate to its maximum speed.

To sum up, errors affecting the presented robotic application also have different impacts depending on the affected component. Unlike H.264, errors exist which can lead to system states where recovery is not possible anymore. However, the robotic application inherits some kind of resilience against transient faults from the PID controller.

### 4.2.3 Further Examples

In the previous two subsections representatives from the domains of multimedia and control applications were presented. In addition, further examples exist where errors can be ignored or have only minor impact.

Li et al. give examples for artificial intelligence as well as compression algorithms in [LY07]. In an artificial intelligence example, it is shown by the authors that some errors only lead to small changes in the automated classification of web page types. In the case of compression algorithms, errors exist leading to larger sizes of the compressed data, but without errors in the data encoding.

All the examples show the potential of handling not every error alike. Hence, it is worthwhile to use the flexible error handling method presented in this dissertation. However, as shown in the previous section, the impact of errors on applications can be manifold. Consequently, a mechanism is required to specify the error impact classification individually. This mechanism is presented in the next subsection.

### 4.2.4 Classification Annotation

As mentioned earlier, the goal of the flexible error handling approach is to decide **if**, **how**, and **when** errors have to be corrected. The **if** and **how** part is covered by the classification. However, as the previous examples suggest, error impact and possible correction methods are highly application dependent. Hence, the semantics of the application has to be known. Unfortunately, this knowledge will be typically not available if not provided by the application designer. To provide the necessary information, annotation of the source code is required. Thereafter, the classification information can be extracted by a customized compiler for ANSI C-code, detailed in the next chapter. Source code can contain two kinds of annotations with which the application programmer can express application knowledge: Reliability type qualifiers and error correction annotations.

#### 4.2.4.1 Reliability Type Qualifiers

Reliability type qualifiers express the error impact. In the current version, the compiler supports two qualifiers called **reliable** and **unreliable** which correspond

**Listing 4.1:** Frame annotated with reliable and unreliable

```
1 typedef struct {
2     reliable int Lwidth, Lheight, Lpitch;
3     reliable int Cwidth, Cheight, Cpitch;
4     unreliable unsigned char * L, * C[2];
5 } frame_t;
```

to high and low impact classification, respectively. The type qualifiers are data-centric. Reliable data are expected to be error free, otherwise high impacts, like system crashes, cannot be excluded. Hence, if an error affects reliable data, error correction will be mandatory. It is only allowed for data to be annotated as unreliable if affecting errors cannot lead to crashes or unrecoverable system states. By definition, data classified as unreliable need not to be corrected. However, it is not precluded that uncorrected errors can lead to huge deviations in the application's output.

Listing 4.1 demonstrates the usage of both type qualifiers in the definition of the frame data type used in the considered H.264 video decoder application. The pixel color information stored in the L and C fields are annotated as unreliable since errors affecting these entries can only lead to wrong colored pixels. In contrast, the width, height, and pitch values have to be reliable. Otherwise, loops iterating over the pixel data can exceed the allocation which possibly leads to accessing unmapped memory locations or to overwriting other data.

#### 4.2.4.2 Correction Method Annotations

The other kind of annotation is used to specify and to assign possible error correction methods. This kind of annotation consists of two parts. The first part qualifies an arbitrary function as error correction method. In Listing 4.2 an example is depicted. In this example, the function `MoCompDefaultValue` is tagged as error correction method by the “`#pragma eca`” directive. Eca is an abbreviation for error correction annotation. Properties, like WCET, of the error correction method can be specified as well. In the depicted example, the purpose of the correction method is to correct a corrupted motion vector by assigning a default value. In this way, consequences, like a system crash due to corrupted motion vectors, can be prevented.

To assign a correction method to the actual data object, the second part of the error correction annotation has to be used. Hereby, the definition of a variable is annotated. An example for the assignment of `MoCompDefaultValue` to the corresponding variables is depicted in Listing 4.3. In lines 1 and 2, a mode prediction data structure is allocated with reliable assignment. Line 6 allocates memory for the X-vector component of all motion vectors of a frame. The annotation of `MoCompDefaultValue` as possible correction methods is done in line 4. Therefore, the “`#pragma eca ecm`” directive is used. Ecm is the abbreviation for error correc-



**Listing 4.2:** Annotated correction method

```

1 #pragma eca qos=... wcet=...
2 void MoCompDefaultValue(fault, object, start, private_data) {
3     int offset = fault2offset_int(fault, object, start);
4     mode_pred_info_t * mpi = (mode_pred_info_t *)private_data;
5     mpi->MVx[offset] = 0;
6     mpi->MVy[offset] = 0;
7 }

```

**Listing 4.3:** Correction method assigned to data

```

1 reliable mode_pred_info_t * mpi = (reliable mode_pred_info_t*)
2     malloc(sizeof(mode_pred_info_t));
3 mpi->MVx = (reliable int *)
4     #pragma eca ecm = MoCompDefaultValue;
5     #pragma eca private_data = (void *)mpi;
6     malloc(x * y * sizeof(int));

```

tion method. In line 5 an optional annotation is used to specify the `private_data` parameter for the correction method. This parameter can be used by the application designer to provide additional information for the correction method. In the case of the depicted example, it is used to specify the base address of the `mpi` structure.

### 4.2.5 Implications

The source code annotations come along with some implications. Obviously, application programmers have to annotate their source code. This process can be tedious and error-prone and wrong annotations can have severe consequences. Due to the fact that correction of errors affecting unreliable annotated data can be skipped during runtime, the application designer has to ensure that no error affecting such data can lead to system crashes or unrecoverable states. Fortunately, the customized compiler shown in Chapter 5 assists in annotating the application. On the one hand, this compiler checks whether or not unreliable annotations have influence on the control flow, pointers, or offset calculations. In such cases the annotations are rejected since errors affecting such kind of data are very likely to lead to system crashes due to the execution of wrong code paths or due to accessing wrong memory locations. On the other hand, the compiler is able to automatically annotate variables in the source code with reliable and unreliable type qualifiers. This automatic annotation is done with static code analysis techniques and type inferencing.

Another implication concerns customized correction methods, like used in the example depicted in Listing 4.2. If such methods correct reliable data, corrections have to be done in such a way that the corrected data cannot lead to system crashes. However, as shown in the previous example, the applied correction method is not

required to restore the erroneous data object to the same state as before affected by an error. Moreover, flexible error handling explicitly allows for setting erroneous data to arbitrary values *as long as the application remains in a valid state*. By setting the motion vectors to zero it is assured that no crashes can occur and the QoS impact stays very low. In this sense, the video decoder application is in a valid state.

To sum up, the classification is responsible for providing essential information required to decide **if** and **how** errors have to be handled. Therefore, reliable and unreliable type annotations divide data objects into two sets. Error handling for the set containing reliable data is mandatory. If an error affects the other set, error handling will be optional. However, handling errors in the latter case can increase the quality of service. To define a set of possible error correction methods for a data object, error correction methods can be annotated as well. This kind of annotation is optional. As described in Chapter 5, *checkpoint-and-recovery* will be always available as a fallback if no correction method is annotated.

So far, one aspect of flexible error handling is not described: the question **when** to handle errors. The foundation to answer this question is provided in the next section.

### 4.3 Real-Time Aspect

In embedded systems, real-time plays an important role. The correctness of a computation result depends on both, the logical result and the point in time it is delivered. Results delivered after the deadline may be considered as wrong, depending on the type of embedded system (exact definitions are provided by Kopetz in [Kop97]). To avoid such situations, tasks are planned using a real-time aware scheduling method. If a task has to be added to the application's task set, a feasibility algorithm will check whether the task can be scheduled without violating real-time constraints. If this feasibility test succeeds, the new task will be scheduled. In this thesis, it is assumed that the tasks of all considered applications pass the feasibility test. Hence, if no errors occur, it is guaranteed that no deadlines will be missed. In cases of errors affecting the application, the real-time behavior is unpredictable. This unpredictability is caused by the error correction overhead imposed by the handling routine. Basically, the error handling routine can be seen as an additional task which has to be executed during run-time. However, one exception to ordinary tasks exists. The result of the feasibility test is of no relevance for the error correction task since errors have to be handled anyway. Otherwise, the system can crash which would automatically fail every future deadline. Considering Figure 4.1 from the beginning of this chapter, this can lead to deadline misses even if the flexible error handling approach is used (case 1). Sometimes the situation occurs (case 3) that error handling can be delayed to a future point in time. This leads to the third question of flexible error handling: **When** shall an error be handled?

The time span in which an occurring error has to be handled can be bounded. Obviously, the lower bound is the time when the error occurred respectively is detected. However, error detection is not part of this dissertation. It is assumed that errors will be detected when the erroneous object is accessed. The upper bound is the point in time where the affected data objects are used again by the corresponding tasks. In this dissertation, error correction is always executed as a separate task. Hence, the question **when** error correction has to be executed can be reduced to a *task precedence problem*. Every task which is using the affected objects inherits a dependency to the correction task. In return, the correction task inherits the priority of the highest prioritized task affected by the corresponding error. Consequently, a scheduler supporting precedence can now answer the **when**-question.

However, to enable this setup a fine grained mapping of affected object to tasks is necessary. Otherwise, problems can occur where higher prioritized tasks are interrupted by low-priority error correction tasks. This phenomenon is detailed in the next subsection. To solve this problem the subscriber model is used which is described afterwards.

#### 4.3.1 A kind of Priority Inversion Problem

Before a process is started by the operating system, different resources are allocated. These consist of memory for data (.data, .bss) and, depending on the OS, a program stack for the initial task. During execution, a task can request additional resources from the OS via appropriate system calls. A resource is considered to be used as long as the process – or the processes sharing the resource – is not destroyed or a task within the process explicitly releases the resource to the OS.

If an error is reported to the OS, the affected resources are determined by traversing the allocation tables. However, the affected tasks cannot be determined, since resources are typically mapped to processes only. In other words: When a fault occurs, the OS will suspend **all** tasks of **all** affected processes until the error is handled. This can be problematic when unaffected high priority tasks are suspended as well. To solve this problem the subscriber model is introduced in the next subsection. With this model it is possible to determine the affected task within a process. Furthermore, it can also be decided whether or not the affected resources are currently in use and hence have live data stored.

#### 4.3.2 Subscriber Model

To get a very fine-grained mapping of resources to tasks, the subscriber model is used. The subscriber model defines a new programming paradigm where tasks have to explicitly `subscribe()` to resources prior usage. After usage, tasks can `unsubscribe()` from the resources. Hence, each data object possesses a set of tasks currently using the object. Before the semantics of the subscriber model are discussed, the terms object, existence, and liveness are defined.

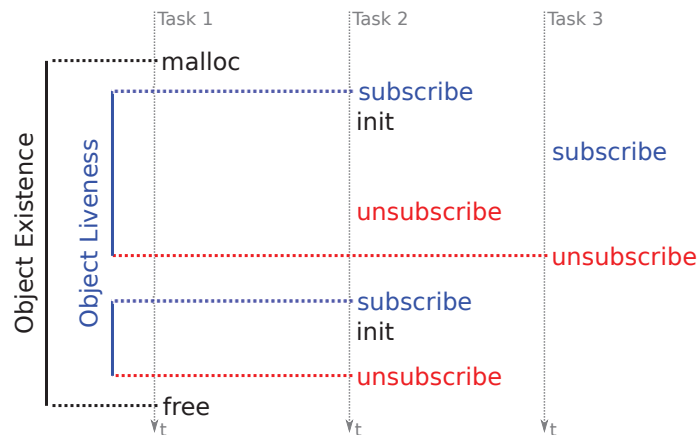
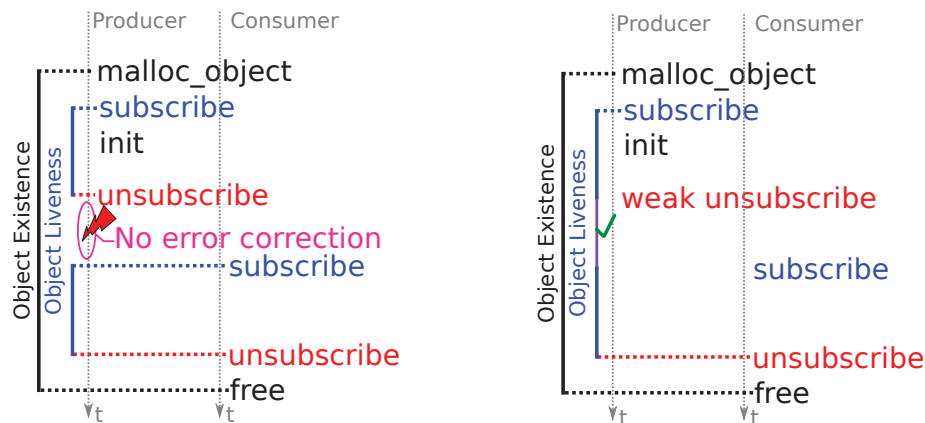


Figure 4.4: Object existence and liveness

#### 4.3.2.1 Objects and Liveness

In the subscriber model, an object is the representation of an arbitrary resource. For example, when the process requests additional heap memory by calling `malloc()`, a new object will be created by the OS. This new object now exactly represents the allocated memory. In addition to information like the allocation's base address and size, the object subscribers are stored as well. After creation the object existence begins (cf. Figure 4.4). Existence of an objects ends as soon as, for example, `free()` is called. A task can subscribe to an existing object using the `subscribe()` functionality. An object with at least one task subscribed is called *subscribed object*. Such a subscribed object is considered as live. This means valuable data are stored. As soon as the data represented by the object is not needed anymore, the corresponding task can use the `unsubscribe()` call. If no tasks are subscribed anymore, the object will be called *unsubscribed object*.

A special type of objects are *system objects*. System objects are always in the subscribed state. They are used to support legacy code. Therefore, it is also required that all system calls and standard library calls will produce system objects when called through the standard interfaces. To obtain an object which can be subscribed/unsubscribed, new interfaces have to be implemented. For example, the `malloc()` library call always returns system objects. In this way, legacy code as well as application code will run out of the box with the subscriber model. To support subscribable objects, a second interface has to be implemented. For allocation of subscribeable objects the functions `malloc_reliable()` and `malloc_unreliable()` can be used. These functions combine the reliable and unreliable type qualifiers and the subscriber model. The main difference of both functions is the return value which is `reliable void *` and `unreliable void *`, respectively. The memory is allocated usually on the same heap. However, if a memory hierarchy is available with different reliability characteristics, it is possible to use the allocation functions to allocate reliable objects on more reliable memory modules.



(a) Possible data corruption due to unsubscribed object

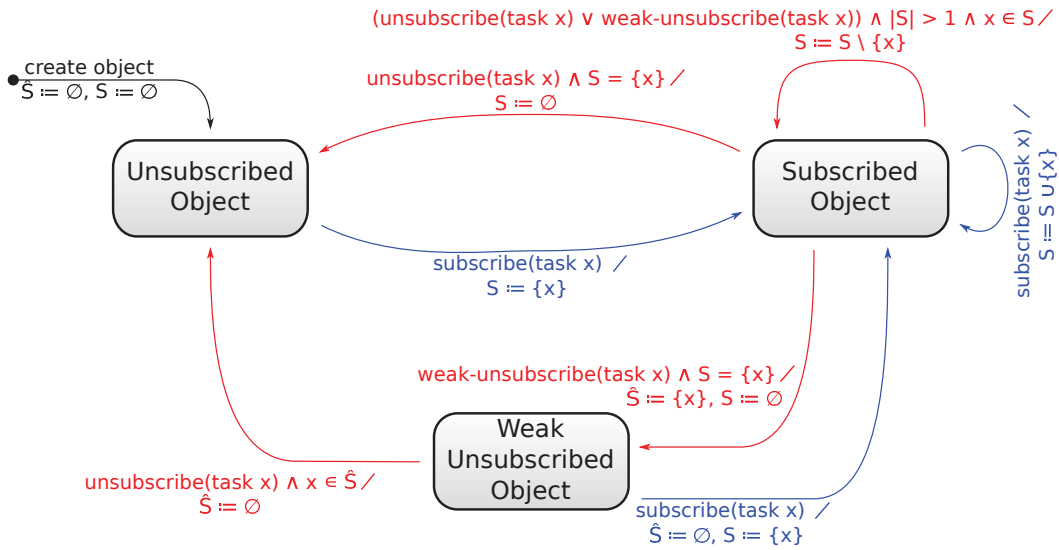
(b) Extended live time of object due to weak unsubscribe

**Figure 4.5:** Producer and consumer problem

#### 4.3.2.2 Producer and Consumer Problem

As mentioned earlier, the resource allocation list of a typical OS contains only information on the resource state – e.g., allocated or not – and the processes/address spaces using the resource. To extend this information, the subscriber model is employed to express liveness of an object and the set of tasks using this object. The basic idea is that **only subscribed objects are 'live'**. This is a very important point when handling errors, since errors need not to be handled in unused/not live resources. However, this has severe consequences for the data usage. If a task subscribes a previously unsubscribed object again, the task must not make any assumptions on the data stored in this object. This implies a new programming model. Before resources can be used, they have to be subscribed and initialized (cf. Figure 4.4). If an object is used by multiple tasks, each task – even in the same process – has to subscribe to the object. When a task does not need the object anymore, it can unsubscribe itself from the object. It is important here to stress the term “itself”. The subscriber model only allows for the subscribed task to unsubscribe itself. It is not possible to unsubscribe a different task. Unsubscribe generally means that the resource is of no interest for the task anymore. As soon as all tasks are unsubscribed, the liveness of the object ends. In the case of a crash, the operating system unsubscribes all subscribed objects.

Unfortunately, this model can provoke pitfalls. To detail this pitfall, a producer task P and a consumer task C is considered. To produce an object O, P allocates memory and subscribes the corresponding object. After finishing calculations, P adds the object O into a buffer and unsubscribes itself from O to produce the next object. The consumer C can now take O from the buffer to process the data. In the meantime, O is in an unsubscribed state which means that occurring errors will not be corrected (cf. Figure 4.5a). Hence, if data is exchanged between tasks in this way, they can be corrupted by errors. A solution to cope with this problem would be



**Figure 4.6:** Subscription states of an object

to insert extra synchronization, so that the producer has to delay the unsubscribe call until the consumer subscribes the object. Unfortunately, this requires either usage of semaphores or polling status variables. To solve this problem a third state is introduced: *weak unsubscribed object* (cf. Figure 4.5b). Weak unsubscribing an object will be only possible if the task already holds a subscription. Literally, weak unsubscribing means that data represented by the object have no relevance for the weak unsubscribing task anymore, but perhaps for other (unknown) tasks in future.

#### 4.3.2.3 Subscriber States

In Figure 4.6 all subscription states are depicted. Each object is associated with two sets  $S$  and  $\hat{S}$  tracking the subscribed and weak unsubscribed tasks, respectively. If  $S$  contains subscribers,  $\hat{S}$  will be empty and vice versa. As can be seen, a weak unsubscribed object has exactly one task in  $\hat{S}$ . If more than one task is in  $S$  ( $|S| > 1$ ) and a task performs a weak unsubscribe() operation, it will be interpreted just as normal unsubscribe() call. In the opposite case, if  $|\hat{S}| = 1$  and another task subscribes to the object, the weak unsubscribed task gets unsubscribed automatically. This behavior perfectly matches the producer and consumer problem. Before putting an object into a buffer, the producer weak unsubscribes the object. As soon as the consumer task subscribes to the object, the producer gets unsubscribed by the OS. With this methodology no periods of time exist where important data are unsubscribed. However, in the case that a mixture of weak unsubscribe and unsubscribe operations are used on the same object, the application programmer is responsible to appropriately synchronize the operations. Otherwise, race conditions can occur. An example for such a situation is an object subscribed by two tasks where the first task performs a weak unsubscribe operation and the second task concurrently uses normal unsubscribe. Depending on the execution order, the

Task	Period $p_i$	Deadline $d_i$	Execution Time $c_i$
$T_1$	4	4	1
$T_2$	5	5	3
$T_3$	40	40	3

**Table 4.3:** Example task set

object can be in the *weak unsubscribed* or in the *unsubscribed* state, respectively.

To summarize, an object – and hence the resource it represents – can be in exactly one of the following states:

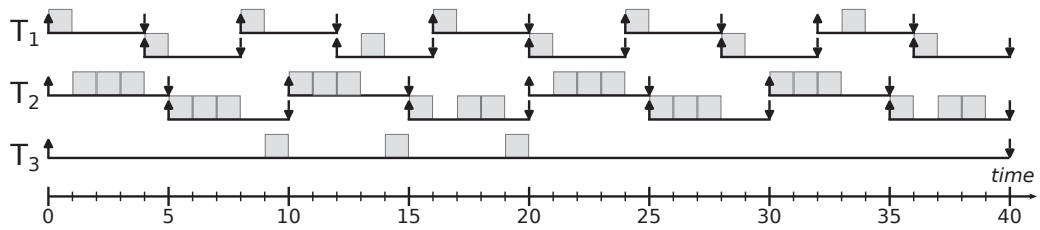
1. **Unsubscribed:** No task is using the resource. The resource contains no live data.
2. **Subscribed:** One or several tasks are using the resource. The resource contains live data.
3. **Weak unsubscribed:** No task is using the resource. The resource contains live data.

These states can directly be used in error handling. Only those tasks (which are in set  $S$ ) have to be suspended which are subscribed to the object that is affected by a fault. If the object is unsubscribed ( $S$  as well as  $\hat{S}$  are empty), no error correction will be necessary. Hence, the subscriber model complements the reliability type qualifiers. As discussed earlier, data annotated as reliable have to be corrected always. With the subscriber model this rule can be softened. Error correction is only mandatory for objects which are reliable **and** subscribed.

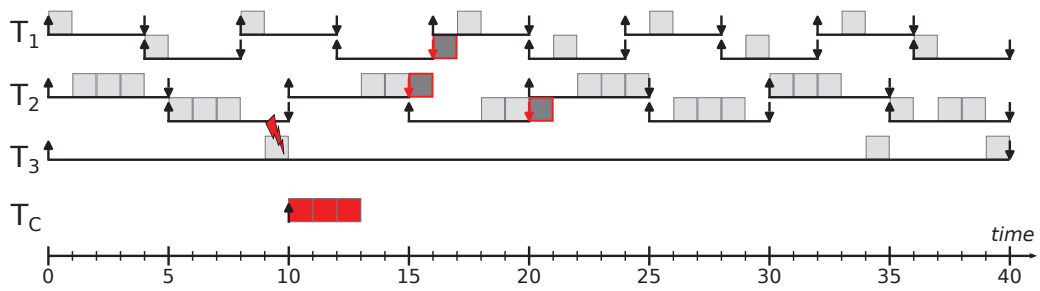
### 4.3.3 Using the Subscriber Model to Schedule Error Correction

To illustrate the advantages of the subscriber model the example task set defined in Table 4.3 is considered. All tasks belong to the same process and are assumed to be preemptable. As scheduling strategy Earliest Deadline First (EDF) is used. If two tasks have the same dynamic priority, the static priority will be used to determine the task for execution. The static priorities of the tasks are defined as:  $T_1 > T_2 > T_3$ . Figure 4.7(a) shows the corresponding EDF schedule for a complete hyper-period without faults. There are three idle slots available at tick 29, 34, and 39. The resulting CPU utilization is 92.5%. For the remaining scenarios (b), (c), and (d), a transient fault is assumed to occur at the end of time slot nine. The corresponding error correction method shall take three time units. Hence, the CPU load rises to 100%.

Figure 4.7(b) shows the situation without the subscriber model. Without the subscriber model no mapping of (affected) objects to tasks is available. Hence, a conservative error handling approach is required. Since every task has to be assumed to be affected by the fault, the error correction method has to be executed immediately. This means to switch to task  $T_C$  which executes the correction. In the depicted scenario, this leads to three deadline misses.



(a) Normal execution without fault



(b) Naive approach (always assume that all activities are affected by a fault)

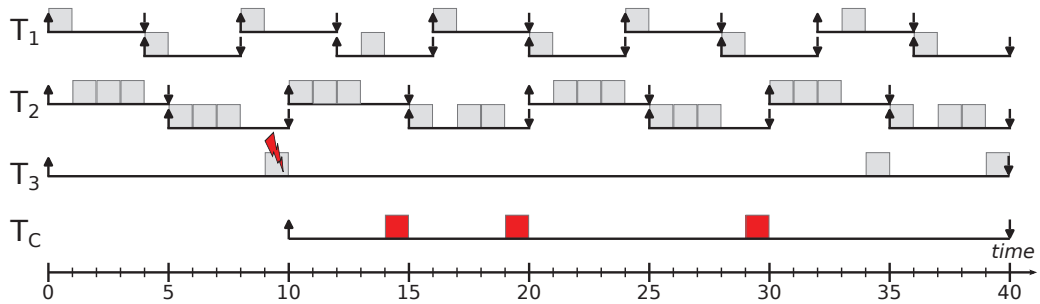
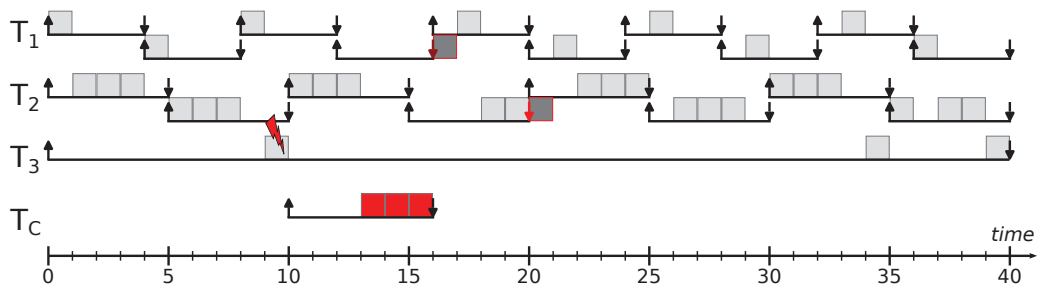
(c) Only  $T_3$  affected by fault(d)  $T_1$  and  $T_3$  affected by fault

Figure 4.7: Fault correction and real-time

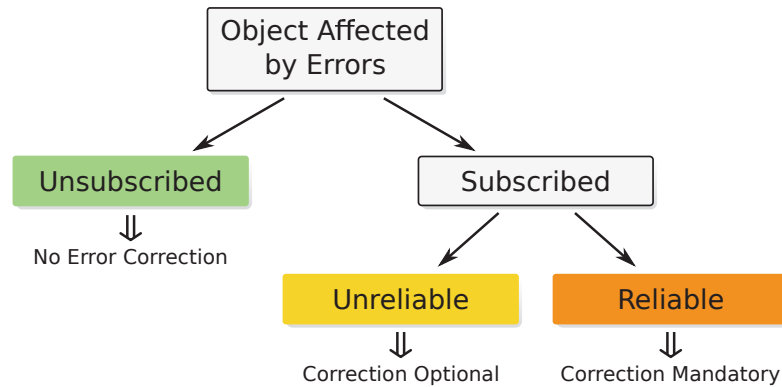


The key to adhere to deadlines under fault influence is to schedule error correction according to the currently used scheduling strategy. Therefore, a mapping of faulty objects to tasks using these objects is mandatory. If the affected tasks are known, it will be possible to schedule the error correction method with the maximum dynamic priority of the affected tasks. Hence, higher prioritized tasks can continue execution. The subscriber model developed in this dissertation can be used to provide the required mapping. In scenario (c), it is assumed that only  $T_3$  is subscribed to the objects affected by the fault. Consequently, error correction is scheduled with the priority of  $T_3$  which means that  $T_C$  inherits the deadline of  $T_3$ . In this scenario, all deadlines will be kept if the affected tasks can continue their execution without a restart.

An example case where two tasks ( $T_1$  and  $T_3$ ) are affected by an error is depicted in scenario (d).  $T_1$  has a higher priority than  $T_3$ . Thus, the error correction task  $T_C$  inherits the priority of  $T_1$ . Since  $T_2$  is fault-free and has the highest priority,  $T_2$  is executed immediately. After  $T_2$  finishes execution,  $T_C$  is scheduled to correct the error. In this scenario,  $T_1$  and  $T_2$  miss a deadline each. As can be seen at the end of time slot 20, it is possible that uninvolved tasks still miss their deadlines. This leads to the conclusion that the subscriber model cannot give any guarantees that no deadlines will be missed. In fact, the number of deadline misses depends on the slack time available for error correction. In the previously described scenarios, the EDF scheduling was chosen intentionally. The reason for choosing EDF is its optimality. By using EDF, it is automatically ensured that no valid scheduling exists, since if enough slack time had been available, no deadline misses would have been occurred. However, by using the subscriber model, a number of scenarios exists where the number of deadline misses can be reduced. With the knowledge of which task is using which faulty object, it is possible to schedule error correction with respect to task priorities. The possibility to schedule error correction guarantees that higher prioritized tasks not affected by faults can be executed before the actual error correction. While this approach can reduce the amount of deadline misses, it is not guaranteed that all tasks will keep their deadlines. However, the scheduling invariant – the task with the highest priority gets executed – is maintained all times.

## 4.4 Summary

In this chapter, the flexible error handling strategy was described. The flexible error handling strategy is a data-centric approach. It is based on the observation that the impact of errors on the quality of service of an application can be manifold. Indeed, the depicted examples showed impacts reaching from application crashes to negligible deviation in the application's output. By not handling every error alike, this observation is exploited by the proposed flexible approach. Hereby, errors are handled according to their impact on the application. The goal of flexible error handling is to decide **if**, **how**, and **when** errors have to be corrected. If the current



**Figure 4.8:** Will error correction be needed?

timing conditions do not allow for correcting every occurred error, only errors with severe impact are handled. To distinguish between errors with severe impact and errors with minor impact, a classification is used. Therefore, possible error impacts have to be analyzed and annotated in the source code. Two error classes are defined: **reliable** and **unreliable**. The reliable class contains all data objects where errors can lead to severe consequences if not handled. All other data can be classified as unreliable. However, data classified as unreliable automatically adds the option for the runtime system to ignore errors affecting the corresponding objects. To further reduce the number of errors where correction is mandatory the subscriber model is used. This model provides both, liveness information of an object and a mapping of objects to tasks. The former can be used to decide whether an error should be handled or not. There is no need to correct errors in unused objects. An object will be treated as alive/used if at least one task is **subscribed** or **weak-unsubscribed**. In both cases the object is called *subscribed object*. If all subscribed tasks **unsubscribe** from the object, the object is called *unsubscribed object*. In the latter case, the object will be treated as not alive/unused. In combination with the classification this leads to the decision tree depicted in Figure 4.8. Error correction will only be mandatory if *reliable subscribed objects* are affected. Correction of *unreliable subscribed objects* is optional. However, handling of errors affecting the latter object type can improve the quality of service.

Another important aspect of the flexible error handling strategy is to enable scheduling of error correction methods as ordinary tasks. This reduces the question **when** to handle errors to a *task scheduling with precedence* problem. Therefore, the object to task mapping provided by the subscriber model is used. With this mapping it is possible to map errors to the tasks which are affected. Hereby, the affected tasks get a dependency to the error correction task. The correction task inherits the priority of the highest prioritized affected task. Thus, unaffected tasks with higher priority can continue execution without interferences of error correction. A phenomenon comparable to priority inversion where a low-priority error correction suppresses the execution of high-priority tasks can be ruled out.

# FAME: Fault Aware Microvisor Ecosystem

---

## Contents

---

<b>5.1 REPAIR Compiler</b> . . . . .	<b>72</b>
5.1.1 Classification Parsing . . . . .	73
5.1.2 Subscriber Model Support . . . . .	75
5.1.3 Code Generation . . . . .	76
5.1.4 Summary . . . . .	77
<b>5.2 Runtime Overview</b> . . . . .	<b>77</b>
<b>5.3 Microvisor</b> . . . . .	<b>78</b>
5.3.1 RCB Components . . . . .	79
5.3.2 Virtualization Concepts . . . . .	79
5.3.3 Memory Management . . . . .	85
5.3.4 Subscriber Model Support and Data Object Management . .	88
5.3.5 Checkpoint-and-Recovery . . . . .	89
5.3.6 Summary . . . . .	92
<b>5.4 FAMERE</b> . . . . .	<b>93</b>
5.4.1 Para-Virtualization Support . . . . .	93
5.4.2 Checkpoint-and-Recovery . . . . .	94
5.4.3 Scheduling and Subscriber Model Support . . . . .	94
5.4.4 Memory Allocation . . . . .	95
5.4.5 Error Handling . . . . .	95
5.4.6 Summary . . . . .	95
<b>5.5 Para-Virtualizing RTEMS for FAME</b> . . . . .	<b>96</b>
5.5.1 RTEMS Overview . . . . .	96
5.5.2 Board Support Package and Infrastructure . . . . .	97
5.5.3 Context Switching . . . . .	97
5.5.4 Memory Management . . . . .	98
5.5.5 Summary . . . . .	100
<b>5.6 Flexible Error Handling - Realization</b> . . . . .	<b>101</b>
5.6.1 Stage 1: Low-Level Error Handling . . . . .	102
5.6.2 Stage 2: Mapping Errors to Tasks . . . . .	102
5.6.3 Stage 3: Classification and Handling Method Selection . . . .	103

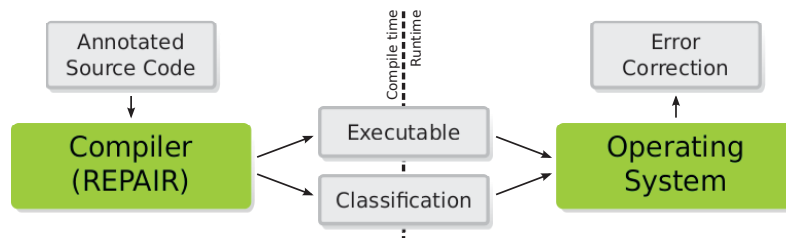
5.6.4	Stage 4: Error Correction and Acknowledgment . . . . .	103
5.7	Summary . . . . .	104

To enable flexible error handling, the concepts presented in Chapter 4 have to be combined. This means application knowledge gathered offline has to be incorporated with data only available during runtime. The *Fault Aware Microvisor Ecosystem (FAME)* realizes this incorporation. An overview of *FAME* is depicted in Figure 5.1. The *Reliable Error Propagation And Impact Restricting (REPAIR) compiler* [SHM+14], a customized compiler for ANSI C99, processes annotated source code. After processing, the application knowledge is encoded in a classification data base and a runtime classification library is created. The runtime components can efficiently extract error handling information with the help of this library. In the case of an error, the error correction method most suitable for the current runtime conditions can be selected and scheduled by the operating system. Section 5.1 briefly introduces the compiler components shown in Figure 5.1. The remainder of this chapter details the runtime components, since this is in the focus of the dissertation. An overview of the runtime components is shown in Section 5.2.

As the letter “M” in *FAME* suggests, the ecosystem is based on a microvisor. As already defined in Section 2.2, the term microvisor describes an approach in which features of a microkernel are combined with features of a hypervisor. The employed virtualization concepts and the *FAME* microvisor are detailed in Section 5.3. Afterwards, in Section 5.4, the *FAME Runtime Environment (FAMERE)* is described. *FAMERE* is embedded into the guest operating system and is responsible for error handling. In this dissertation, RTEMS is used as guest OS. The virtualization of RTEMS is depicted in Section 5.5. Finally, in Section 5.6, all components are integrated to realize the flexible error handling approach.

## 5.1 REPAIR Compiler

Flexible error handling would not be possible without interfacing knowledge gathered during compile time (offline) and information only available during runtime (online). The *Reliable Error Propagation And Impact Restricting (REPAIR) com-*



**Figure 5.1:** FAME - Fault Aware Microvisor Ecosystem

*piller* implements the offline part. By parsing the source code of the application, a classification of possible error impacts is created. After parsing, *REPAIR* creates a classification data base and a library, called *librecon*, which can be used by the operating system to access the created data base.

The next subsections describe the parsing of the classification, the subscriber model support, and the creation of *librecon*. However, these subsections are only intended to give a very coarse-grained overview. More details can be found in [SHM+13] and [SHM+14].

### 5.1.1 Classification Parsing

In the previous chapter, the concept of classification of data based on error impacts was described. To exploit this classification during runtime, the corresponding information has to be prepared during compile time. Therefore, the source code of the application is analyzed. The goal of this analysis is to determine error correction options for the individual data objects of the application. The necessary input has to be provided by the application designer via annotations. As shown in the previous chapter, the source code can contain two kinds of annotations: Reliability type qualifiers and error correction annotations.

The type qualifiers **reliable** and **unreliable** classify if error handling of affected data will be mandatory or not. Reliable data are expected to be correctly computed, otherwise it cannot be excluded that the system may crash. Hence, if an error affects reliable data, error correction will be mandatory. The *REPAIR compiler* automatically classifies data as reliable based on their use in the application and inserts the type qualifiers into the source code of the application [SHM+13]. Hence, reliability type qualifiers need not to be contained in the initial source code. The *REPAIR compiler* checks and automatically annotates reliable and unreliable type qualifiers based on the following two rules:

**Definition 5.1 (Prohibit Rules)** *Data must not be erroneous, if:*

- *they have influence on the control flow,*
- *they represent memory addresses, or*
- *they can lead to arithmetic exceptions.*

**Definition 5.2 (Propagation Rule)** *Data which can have influence on the calculation of reliable data have to be reliable too.*

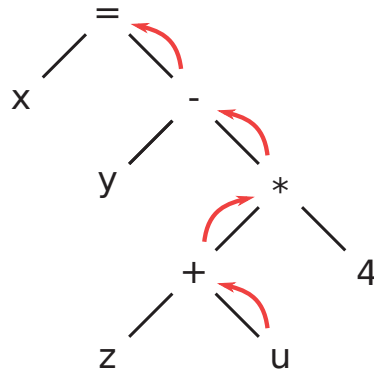
If at least one condition violates one of these rules, only reliable classification will be allowed. To check annotated source code, *REPAIR* propagates the unreliable attribute. An example is depicted in Figure 5.2. It shows the syntax tree for the program code in Listing 5.1. The arrows in Figure 5.2 demonstrate the propagation of the unreliable type qualifier of variable *u*. The propagation starts at the leaves. Since the root of the syntax tree is an assignment operator, the left branch has to

**Listing 5.1:** Example Code

```

1 unreliable int u, x;
2 int y, z;
3 x = y - (z + u) * 4

```

**Figure 5.2:** Propagation of *unreliable* attribute (taken from [SHM+13])

be unreliable. Otherwise, the compiler would raise an error. In the depicted example, the type annotations are correct. During the propagation of the annotation, *REPAIR* also determines whether an operation is unreliable or not. As soon as one input operand is unreliable the complete operation is considered as unreliable as well. In terms of the probabilistic error model presented in Section 3.4, this allows for the replacement of deterministic operations with probabilistic versions. In the depicted example, the reliability qualifier of variables *y* and *z* are of no importance since all operations can be marked as unreliable.

For automatic annotation of type qualifiers, type inference is used. To maximize the amount of unreliable data, the compiler assumes that all data are unreliable. Each time one of the prohibit rules is broken, a reliable annotation is inserted. Thereafter, this reliable annotation is recursively inherited to all operands to fulfill the propagation rule.

The second type of annotation – error correction information – is also processed by *REPAIR*. During parsing, *REPAIR* searches for `#pragma ecm` statements annotated to error correction functions. If such an annotation is discovered, the corresponding function will be added to a table containing all possible correction methods. Metadata, such as the  $WCET_{EST}$  and the QoS impact of the correction method, are also added to the table. With the table index of an error correction method and the object identifiers, *REPAIR* can efficiently encode the correction methods applicable to an object. Hence, during runtime, available correction methods and their parameters can be queried by the operating system.

### 5.1.2 Subscriber Model Support

Another important aspect of *REPAIR* is the support for the subscriber model. To support the subscriber model introduced in Section 4.3.2, applications have to be adapted. Each `malloc` call has to be converted into `malloc_reliable` or `malloc_unreliable` to allow for usage of the subscriber semantics. *REPAIR* can optionally rewrite all memory calls used by the application to match the reliability type qualifiers. In the case that a memory allocation is assigned to an unreliable or reliable variable, *REPAIR* rewrites the allocation to `malloc_unreliable` or `malloc_reliable`, respectively. In comparison to the standard `malloc` call, the `malloc_(un)reliable` calls provide a different API which is depicted in Listing 5.2.

**Listing 5.2:** `malloc_reliable/malloc_unreliable` API

```
1 reliable void * malloc_reliable(size_t size, objid_t object,  
    reliable void * private_data);  
2 unreliable void * malloc_unreliable(size_t size, objid_t object,  
    reliable void * private_data);
```

The API change is required to fit the needs of the *REPAIR compiler*. The second parameter is used to denote an object identifier which is freely assignable by the compiler. This identifier is important during runtime to map data allocated dynamically on the heap to the corresponding data type information extracted offline. The third parameter is the private data argument which can be used by the correction method. This parameter can only be stored by the runtime system since `private_data` can be dependent on the current program state at the time of the memory allocation. Listing 5.3 shows the automatically translated source code of the example depicted in Listing 4.3. All non-standard C expressions are replaced by expressions which can be parsed by ordinary C compilers. For example, the annotation of private data for the correction method is translated into the appropriate function parameter.

**Listing 5.3:** Source-to-source translated code of Listing 4.3

```
1 mode_pred_info_t * mpi = (mode_pred_info_t*)  
2     malloc(sizeof(mode_pred_info_t));  
3 mpi->MVx = (int *)  
4     malloc_reliable(x * y * sizeof(int), 42, (void *)mpi);
```

The replacement of `malloc` calls has a pitfall. The application designer has to assure the correct usage of the subscriber model by inserting `subscribe` and `unsubscribe` calls where appropriate. Otherwise, it can happen that reliable data will not be corrected even though they are used by the application. To avoid this problem, *REPAIR* will only replace `mallocs` if annotations are found. It is assumed that developers will only annotate source code if they can ensure the correct subscriber states. In Listing 5.3, only the second allocation is replaced since annotations are provided in lines four and five in Listing 4.3.

### 5.1.3 Code Generation

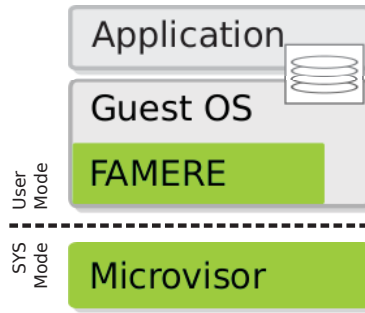
After extracting the data classification and assigning identifiers to dynamically created objects, *REPAIR* is able to integrate this knowledge into the binary. The knowledge is integrated in form of two entities. The first one is the classification database which comprises the following entries:

- **Object information:** For each object allocated on the data segment, bss segment, or the heap, an object identifier and type information are stored. Data objects allocated on the heap are called *dynamic objects*. Objects residing in a data or bss segment are called *static objects*. For dynamic objects, the object identifiers have to be specified when `malloc_(un)reliable` is used. In Listing 5.3, the object identifier is set to 42. During runtime only the operating system has the knowledge of the current allocations. Since the allocation of objects on the heap is not known during compile time, *REPAIR* cannot distinguish between different dynamic objects. The object identifiers bridge the gap between the runtime and compile time world. Now, the dynamic objects can be identified since the operating system stores the object identifier for each allocation.
- **Corrections methods:** Any supported correction method is listed in a table. For each method, the entry point and additional data, like the  $WCET_{EST}$  of the correction, is encoded.
- **Application tasks:** Another important task of *REPAIR* is to identify which static objects are used by which task. Therefore, *REPAIR* parses each task entry point of the application and stores the corresponding task information. Based on the entry points, static code analysis techniques are used to statically map each global object to all tasks which are using the object.
- **Classification table:** The core component of the classification data base is the classification table. For each object, this table stores possible error correction methods and the tasks which are using the object. To get the corresponding classification of an object, the object identifier is used as an index.

The second entity *REPAIR* creates is the **library for runtime error classification**, (**librecon**). This library provides a common API to access the classification data base. Functions are provided to query details of error correction methods and to map an address to a static object. The API is used by the operating system, specifically *FAMERE*, during error correction.

The most important functionality of *librecon* is to provide a possible correction solution for occurred errors. If several correction methods are available, multiple solutions will exist which are able to correct the errors. However, these solutions are likely to have different non-functional parameters, like, e.g.,  $WCET_{EST}$  or QoS impact of the correction. This leads to a multi-criteria optimization problem. Hence,





**Figure 5.3:** Software stack

a Pareto-optimal front containing possible error correction scenarios is required. An error correction solution will be part of the front, if it dominates other solutions for at least one optimization parameter. The front is created on request during runtime by *librecon* and is based on the currently occurred errors. Each point of the front is a Fault Correction Graph (FCG). A FCG is a directed graph  $G = (V, E)$ . By applying an FCG which is part of the generated Pareto-optimal front, all errors are corrected. A vertex  $V$  in an FCG represents a correction method that has to be applied. An edge  $e = (v_i, v_j)$  will be present if correction method  $v_i$  has to be executed prior correction method  $v_j$ . As long as the dependencies are respected, the operating system can apply the correction methods in any order. The selection of a suitable FCG is detailed in Section 5.6.3.

#### 5.1.4 Summary

In this section, the compiler aspects of the flexible error handling approach were introduced. The compiler is responsible for transforming the application knowledge into a form which can be efficiently parsed during runtime. For extracting information how to handle occurred errors, *REPAIR* creates *librecon* during compile time. The operating system can later use *librecon* to obtain possible error correcting solutions.

The remainder of this chapter is focused on the runtime system. From now on, the compiler is treated as a black box and only the services provided by *librecon* are used to access application knowledge.

## 5.2 Runtime Overview

Figure 5.3 provides an overview of the runtime components. An application with integrated classification information is running on a virtualized guest OS. The guest OS is linked against the *FAME Runtime Environment (FAMERE)*. *FAMERE* is responsible for the flexible error handling as well as the interfacing with the microvisor. The microvisor runs low-level error correction and ensures the feasibility of software-based error handling. In the next sections, the aspects of the runtime sys-

tem are described in more detail. Currently, *FAME* is designed for single processor systems and implemented for ARM926-based platforms. Hence, some concepts are based on the peculiarities of single-core ARM926 processors.

### 5.3 Microvisor

The *FAME* microvisor is the foundation to enable software-based flexible error handling. The main purpose of the microvisor is to isolate critical system components from possible error propagation. Critical components in this context are resources required to keep error detection and correction running. Depending on the underlying hardware, the actual critical resources vary. If, for example, errors are signaled via interrupts, the interrupt controller will be part of the critical resources set. The microvisor itself is also part of the critical resources set. Like for all software-based fault-tolerance mechanisms, this is a weak spot, since the microvisor cannot protect itself. If software parts critical for error handling are susceptible to errors as well, situations will occur where the error handling routine has to cope with errors affecting the error handling itself. This can result in a livelock. To deal with this problem, some guaranteed fault free hardware components are required to execute software-based fault-tolerance mechanisms. Those reliable hardware components and the microvisor form a minimal set, the so called Reliable Computing Base (RCB) [ED12]. In the next subsection, the components which have to be reliable are detailed.

As already mentioned, virtualization plays an important role in *FAME*. Virtualization is essential to shield the RCB against error propagation and to keep the low-level error handling components of *FAME* running. The employed virtualization concepts are described in Subsection 5.3.2.

Another important task of the microvisor is the realization of the subscriber model (cf. Section 4.3.2). Due to the fact that the subscriber model stores important information for error correction, the implementation inside the microvisor is compulsory. Otherwise, if the subscriber information is corrupted, it would be possible that errors would not be corrected. This would be the case, if errors changed a subscribed to an unsubscribed object. In Subsection 5.3.3, memory management and functionality used to realize the subscriber model is described.

A further important functionality the *FAME* microvisor provides is *checkpoint-and-recovery*. Three major reliability concerns force checkpointing to be implemented inside the microvisor. First, the recovery of a checkpoint will be the last resort if error handling is mandatory and no application-specific correction method is available. Second, even if an application-specific method error correction method exists, it can happen that this method will be not executable due to other errors which render the guest OS or *FAMERE* unusable. In such a case, the microvisor is still able to recover a checkpoint. Third, the checkpoint has to store and restore also data belonging to the RCB. While the storing process can in theory be executed by the guest operating system, the restore-part would violate RCB Invariant 3.1.

This invariant defines that only components inside the RCB are allowed to change the state of the RCB. Checkpointing is discussed in detail in Subsection 5.3.5.

### 5.3.1 RCB Components

As described in Section 3.1, the RCB classifies all components of a system into two sets based on the importance for error handling. Components will be important if they implement or ensure the operation of software-based fault-tolerance methods. In both cases, such components are part of the RCB. The RCB should be as small as possible. This avoids invoking – potentially expensive – error handling routines.

The microvisor has to be part of the RCB software-side since the microvisor is a single point of failure. Without the microvisor, handling errors in software would not be possible. The hardware components of the RCB are defined as all components required for execution of the microvisor. The following components have to be part of the RCB:

- **CPU:** The majority of the CPU components have to be reliable except for the probabilistic error model where the ALU can support probabilistic operations. Otherwise, it cannot be guaranteed that the microvisor executes correctly. The memory management unit (MMU) has to be reliable as well. Without a reliable MMU, memory virtualization on ARM926-based systems would be impossible.
- **ROM/Flash:** Since the program image including the application and the initial data are stored in ROM or FLASH, it has to be reliable.
- **Reliable Memory:** Very important data, like the page table, is stored in reliable memory. In contrast to main memory or ROM, the size of the reliable memory can be rather small (128 KiB for ARM926-based systems). Therefore, reliable memory can also be implemented with ECC-protected scratch pads directly inside the processor.
- **Timer and IRQ:** The timer is typically not a first-class concern for reliability. However, since embedded systems are considered, timing errors have a huge probability to directly affect the program's desired results. To receive timer ticks and fault notifications, the interrupt controller has to be reliable as well.
- **Other Components:** There are also further components which are not covered so far. These are, for example, the power supply or the clocking system.

### 5.3.2 Virtualization Concepts

To shield the RCB from error propagation, the developed microvisor uses different virtualization concepts. Compared to other virtualization solutions, like, e.g. Xen [BDF+03], the *FAME* microvisor is tailored to the needs of embedded systems and fault tolerance. Therefore, features most commonly found in other solutions

are not supported. In contrast, specialized features are integrated complementing flexible error handling. The used virtualization concepts are detailed in the following subsections.

### 5.3.2.1 Para-Virtualization

Para-virtualization is a technology providing a software interface which is similar to the real underlying hardware. However, the interface is not identical. For example, execution of privileged operations is not allowed. Hence, the guest operating system has to be ported to be runnable under this kind of virtual machine. This is contrary to pure virtualization where a guest OS can run out of the box. Like shown in [BDF+03], para-virtualization allows for higher performance by reducing the virtualization overhead due to the specialized software interface and the porting of the guest OS.

Para-virtualization is the key concept to keep the code size of the microvisor small. The guest OS can be granted direct access to specific hardware components. Hence, the microvisor can out-source the implementation of device drivers to the guest. This is another advantage of para-virtualization since only device drivers important to secure the RCB have to be provided by the microvisor. All other drivers are implemented by the guest OS running outside the RCB. In the case of *FAME*, the microvisor implements device drivers for the IRQ and timer hardware as well as the power and clock manager.

### 5.3.2.2 Library-Based Guest OS

To keep the virtualization overhead even lower, the *FAME* microvisor supports only one guest operating system during runtime. In contrast to other hypervisors, this releases from the burden to provide virtual CPUs and CPU multiplexing. Furthermore, caches and TLB entries need not to be switched between different OS instances.

The target guest operating system is a library based embedded real-time operating system. The idea behind a library OS is that the entire OS runs in the same address space as the application. The OS is linked to the application as an ordinary library. OS functionalities are provided as normal API calls. Library operating systems usually have better performance than other OS approaches since they can be easily tailored to application demands. Due to the single address space, using a library OS also has advantages for virtualization. Once the address space is initialized, there is typically no need to change address mappings anymore. Hence, the microvisor only has to provide simple mechanisms to support page manipulations. In the case of the *FAME* microvisor, it is sufficient to provide functionalities for I/O mapping and for cache management.

Together with the limitation to execute only one guest operating system, the required code base in the microvisor can be reduced further. In the next two subsections the communication between guest OS and microvisor is described.

### 5.3.2.3 Hypercalls: Communication between Guest OS and Microvisor

Like other hypervisors, the *FAME* microvisor can be interfaced via hypercalls. Hypercalls have to be used whenever a privileged operation or a protected operation has to be performed by the guest operating system. Examples for such operations are changing processor status registers or changing the page table, respectively. A complete list of all available hypercalls can be found in Appendix A. Performing a hypercall also implies that data is submitted to the microvisor. Since the microvisor is part of the RCB, submitting data to the microvisor from components outside the RCB violates Invariant 3.1. However, communicating with the microvisor is essential for para-virtualized operating systems. To allow for the passing of parameters without harming the RCB invariant, hypercalls are divided into two phases. This division is an important design principle for fault tolerance. Hereby, the problem of the transition from an unreliable to a reliable state is solved.

The first phase of hypercall progressing is stateless, hence it can be aborted at any point in time without the risk of leaving data leaks or other inconsistencies behind. The main task of this phase is to copy all arguments to a temporary reliable location and to check the hypercall arguments for consistency. Depending on the hypercall parameters, consistency checking includes checking for valid arguments, pointers addresses, and the right object type of the object pointed to. For example, if the hypercall expects a reference to a fault entry but the pointer points to an address which is not a fault entry, an error will be raised and the hypercall will be abort.

Another possible reason to abort a hypercall can be the occurrence of a transient fault while processing the hypercall arguments, since these arguments are not necessarily allocated on high reliable silicon. In such an abort case, the microvisor will return the `EINTR`<sup>1</sup> return code. After error handling, the guest OS can try to re-execute the hypercall. Since memory allocations would change the state of the RCB, the stateless phase can only use the stack or special global variables as temporary buffer. Theoretically, both are part of the RCB and a write would change the RCB's state. However, when errors occur during the first hypercall phase, the stack can be easily rolled back and those special global variables, dedicated only to hypercall argument copy, have no influence on the overall state of the microvisor. Hence, error propagation into the microvisor is prevented.

As soon as all arguments are checked and copied into reliable regions, the second hypercall phase is allowed to alter the RCB state. Since only memory belonging to the RCB is used now and all parameters are checked for consistency, no faults can occur during the RCB state change. Thus, the microvisor state can be safely modified. While this approach suits well to protect the RCB, it implies double copy of data. However, the amount of data to copy is very small, typically, only a few words.

Hypercalls of the *FAME* microvisor are non-interruptible and only one hypercall can be in execution at any point in time. Only one exception exists: The first phase

---

<sup>1</sup>`EINTR` is a standard UNIX return code for an interrupted system call (in this case a hypercall)

of hypercall processing where hypercalls can be interrupted by detected errors.

The presented hypercall approach enforces a further design principle: *isolation between different memory domains*. If the microvisor does not access memory which does not belong to the RCB, accessing erroneous data will be avoided. Hence, it can be assumed that all data used by the microvisor are error free.

#### 5.3.2.4 ECI: Communication between Microvisor and Guest OS

The *FAME* microvisor is reactive. If the guest OS requests services via hypercalls, the microvisor will respond directly with the appropriate result. This is *synchronous* behavior. However, there are events requiring interaction of the microvisor, like, e.g., IRQs. In such a case, no direct request of the guest operating system exists. These events are *asynchronous* from the perspective of the guest OS. After internal handling in the microvisor, the guest OS is notified of the occurrence. The microvisor implements an *event callback interface (ECI)* for this purpose. To setup the interface, the guest operating system has to register an event handler during initialization. In the case of a new event, the microvisor places the event into a ring buffer and calls the previously registered handler. To avoid error propagation into this buffer, it is allocated in reliable memory and mapped read-only for the guest OS. The ring buffer is used to avoid explicit acknowledge via an extra hypercall. Instead, the guest OS implicitly acknowledges an ECI message by advancing the ring buffer read offset. Therefore, the read offset is mapped writable into the address space of the guest OS. In the case that the capacity of the ring buffer is exceeded or the read offset points to an invalid buffer element, the microvisor assumes a serious problem in the guest OS. Consequently, a checkpoint will be restored or a guest reset will be performed.

Currently, three different event types are defined:

1. **ECI\_MESSAGE\_IRQ**: This event type signals an IRQ.
2. **ECI\_MESSAGE\_TICK**: At each clock tick, the guest OS is informed.
3. **ECI\_MESSAGE\_FAULT**: If an error occurs, it will be signaled by this message type.

Typically, the ECI handler of the guest OS is executed within the context of the currently executed task. To avoid interrupting the ECI handling, the microvisor automatically disables interrupts. The previous interrupt status is stored within the ECI message and can be restored after processing the message. As long as interrupts are disabled no further ECI messages are delivered, except error messages which are always delivered.

#### 5.3.2.5 Timer

In embedded systems time plays an important role. The correctness of a task is determined by the correct logical result as well as the correct timing. To maintain

control of the system time, the microvisor virtualizes the system clock. On each timer tick the guest OS is informed via an ECI message. Maintaining the system time within the microvisor has huge advantages. On the one hand, the system time will be maintained continuously even if a previous system state is restored or if the guest OS is reset. On the other hand, the microvisor can detect stalls of the guest OS by implementing timeouts. The latter case can happen, for example, in the case of a fault where the guest OS hangs in an invalid state.

The *FAME* microvisor exports the global system time in the read-only symbol `fame_systime`. The system time counts the clock ticks since power on. The tick interval is freely configurable by the guest OS in granularity of milliseconds.

### 5.3.2.6 Interrupts

To protect short critical code sections on the same processor, an often used concept is to disable interrupts. This suppresses timers and other interrupts which may cause the OS to schedule a different task. However, if interrupts are disabled, faults reported by IRQs would not be reported anymore and also the system time would not be updated. Another problem when handling interrupts is that interrupts are typically handled in a privileged processor mode. Thus, directly executing the interrupt handling in the guest OS would allow the guest OS for altering the RCB state.

To solve these problems, interrupts are also virtualized. Therefore, interrupts are divided into two classes. The first class consists of interrupts managed by the microvisor. These are the timer IRQ and fault reporting IRQs. The second class consists of all remaining interrupts. If the guest OS requests to disable interrupts, the *FAME* microvisor will physically disable interrupts belonging to the second class. Interrupts of the first class still interrupt execution and are handled by the microvisor. However, they are not reported via an ECI message anymore. Hence, first class interrupts are only virtually disabled. As always, the exception exists that error handling has higher priority than the disabled interrupts of the guest OS. Thus, ECI fault messages are still delivered to the guest OS.

Not reporting first class interrupts has an important implication for the guest OS. After re-enabling interrupts, the state has to be synchronized with the state of the microvisor. For example, without synchronization, the clock of the guest OS can be several ticks behind the clock of the microvisor if a timer ticks occur during disabled interrupts.

### 5.3.2.7 Execution

In contrast to other hypervisors, the *FAME* microvisor explicitly does **not** multiplex the CPU by providing virtual CPUs. While this is theoretically feasible for the microvisor, it is not necessary, since only one guest operating system is supported anyway. Execution time abstraction, such as threads (like in L4 [KEH+09; LW09]) or scheduler activations (like in K42 [KAR+06]) are also **not** part of the microvisor.

**Listing 5.4:** RTEMS context switch code for ARM

```

1 ;r0 pointer to context store location
2 ;r1 pointer to context load location
3 _CPU_Context_switch:
4     mrs     r2,     cpsr
5     stmia   r0,     {r2, r4-r11, r13, r14}
6     ;context stored to memory pointed to by r0 -> load now via r1
7     ldmia   r1,     {r2, r4-r11, r13, r14}
8     msr     cpsr,   r2

```

Task and thread management is solely the concern of the guest OS. However, some support of the microvisor is needed, especially for context switching. A typical context switch for the ARM processor is depicted in Listing 5.4 (taken from RTEMS [OAR14]).

Due to virtualization, this code is now executed in unprivileged mode which is uncritical for all instructions, except for `msr` (line 8). This instruction will silently fail. Silent fail means that the instruction does not show the intended behavior but also does not trap or return an error code. In the example, the instruction is expected to restore the complete program status word. However, due to the execution in unprivileged mode, the instruction only changes the condition codes of the program status word. To enable changes of the whole program status word, the microvisor provides the hypercall `fame_hypercall_cpsr_set`.

The `cpsr` instruction is an example where a sensitive instruction is not in the set of privileged instructions. This violates a formal requirement for virtualization (Theorem 1: Popek and Goldberg in [PG74]). Thus, virtualization without modifying the guest OS is not possible. Hence, either para-virtualization or binary patching is required.

### 5.3.2.8 Devices and I/O

Frequently, devices are virtualized by exporting a simplified interface to the guest OS while the actual device driver is implemented in the hypervisor. Another option is to directly export the device to the guest. Therefore, the device registers are mapped into the guest's address space. The *FAME* microvisor uses the latter method. All devices that are not part of the RCB are directly mapped into the I/O address space accessible by the guest OS. However, directly mapping devices which have DMA capabilities is problematic. Such hardware can overwrite memory at arbitrary addresses. A platform comprising an I/O MMU can prevent such a behavior. If an I/O MMU is not available, the DMA interface of the device will be virtualized by the microvisor. Some devices use interrupts to signal status changes. In such a case, the ECI method of the microvisor is used to forward such interrupts.

Another problem is the granularity of the mapping. In some cases device registers are not page aligned. An example is the Marvell Kirkwood 88F6281 SoC [Mar14] presented earlier in this dissertation. In this SoC, important system regis-



ters, general purpose I/O (GPIO), and real-time clock (RTC) registers reside on the same page. If this page was mapped with write permission to allow the guest OS to program the GPIO pins, the guest would also have the possibility to reprogram the flash or to change the CPU clock. This would be a severe violation of Invariant 3.1. In such a situation, the corresponding registers are mapped read-only. For write operations, a hypercall (`fame_hypercall_register_set`) is provided which checks the register access on a fine grained base. In the rare case that registers are read-sensitive, the registers are not mapped at all for the guest. Instead, `fame_hypercall_register_get` has to be used by the guest OS to read those registers. This method shields registers from accidental read operations initiated by the guest OS due to occurring errors.

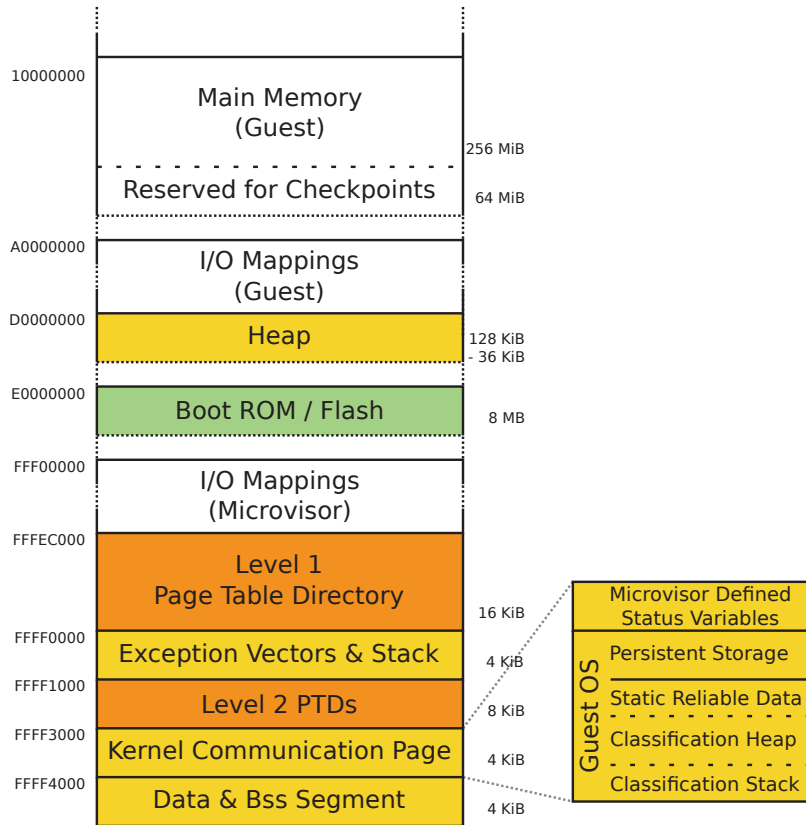
Two further hypercalls complement device support. The first hypercall, `fame_hypercall_cache_parameter_set`, can be used to change the caching parameters. Some devices, mostly DMA controllers, can only operate on uncached memory regions. The second hypercall, `fame_hypercall_map_io`, supports mapping of large I/O ranges into the address space of the guest OS. In this dissertation, the mapping feature is used to map the frame buffer of graphic cards.

### 5.3.3 Memory Management

In the previous section, the basic virtualization strategies are depicted. In this section, the memory management subsystem of the *FAME* microvisor is described. The microvisor distinguishes between memory belonging to the reliable computing base and other memories. The former type of memory has to be equipped with some type of protection against transient faults. Protection of this *reliable memory* has to be transparent for the *FAME* microvisor. Or, in other words, the microvisor assumes that all data stored within this memory are always correct. If the microvisor uses other memory, it will take care of protecting the data itself.

All data structures of the microvisor are stored in the reliable memory and, hence, are part of the reliable computing base. Since the RCB is protected externally, the microvisor only has to ensure protection against unintended status changes. Such changes can be caused by faults propagating from the guest OS into the microvisor. To prevent this propagation, the majority of the reliable memory is write-protected. Figure 5.4 depicts the virtual memory layout of the microvisor for the CoMET-based ARM926 platform. Shaded parts, with exception of the ROM, are mapped to the reliable memory. In the remainder of this subsection, each mapping is detailed.

**ROM:** In this thesis, the ROM is assumed to be error free. The program code of the *FAME* microvisor and the application is directly executed out of the ROM. It is feasible that the ROM can be affected by faults. In this case, the data stored in ROM have to be equipped with redundancy. The microvisor would then copy the currently used code pages into reliable memory correcting all errors. Since it is expected that the amount of available reliable memory is smaller than the size of



**Figure 5.4:** Virtual memory layout (CoMET-based ARM926 platform) the ROM, demand paging would be used to map additional code pages.

**Heap:** The reliable heap consists of all the remaining memory not used for static allocations. It is mainly used to create new errors entries and to allocate object information data structures. To minimize the waste of memory, two allocation strategies are operated in parallel. A simple page allocator is used to map consecutive 1 KiB pages<sup>2</sup> for allocation requests larger than 512 Bytes. If an allocation request is larger than the size of a page, the simple page allocator will search for a free virtual address range and will map free physical pages into this range. There is no guarantee that the physical pages are adjacent. For allocations smaller than or equal to 512 Bytes a buddy allocator is used. The guest is also allowed to allocate memory on the heap to protect data structures with a very high susceptibility to faults. Therefore, the hypercalls `fame_hypercall_mem_alloc` and `fame_hypercall_mem_free` have to be used. Although reliable memory allocations made for the guest OS are stored in fault-free memory provided by the RCB, they are not part of the RCB itself. RTEMS, which is used as guest OS, does not rely on reliable memory. However, by providing access to reliable memory, the guest is able to protect critical data structures. This can increase the probability that the guest OS can handle faults

<sup>2</sup>1 KiB is the smallest page size (tiny pages) on ARM926-based systems

in an application-specific way.

**Exception Vectors and Stack:** According to the architectural specification, the high-level exception vectors have to start at the virtual address 0xFFFF0000. The *FAME* microvisor implements the exception vectors as jump table to the actual exception handling functions. This jump table is very small. Hence, the majority of this small page mapping<sup>3</sup> is available as stack for the microvisor.

**Page Table Directories:** The page table directories (PTD) are the largest part of the static reliable data (24 KiB of 38 KiB). The level one PTD is always required on ARM when paging (and caching) is used. Only sections with the size of 1 MiB can be mapped with the level one PTD. To map pages with a finer granularity, second level PTDs are necessary. For a fine-grained mapping of the last megabyte in the virtual address space (starting at 0xFFF00000), the microvisor uses a level two PTD. Another PTD is used to map the heap. If the heap is larger than 1 MiB, further second level PTDs are used.

**I/O Mappings:** Two virtual address ranges exist to map device memory. The address range starting at 0xA0000000 is used to map devices exporting large address ranges, like frame buffers. The guest OS can map devices here by calling `fame_hypercall_map_io`. Mapping is only possible in section granularity since no second level PTDs are available for this address range. The second address range resides within the last megabyte and is covered by a second level PTD. Hence, fine-grained mappings are possible. The guest cannot map devices here since this address range is exclusively used by the device drivers integrated into the microvisor. However, depending on the device driver, some devices are accessible by the guest OS. An example for such a device is the UART.

**Kernel Communication Page:** The Kernel Communication Page(s) (KCP) is the only memory region where the guest OS has complete write permission. Some writable status variables are exported through the KCP mapping. These variables are used to communicate the state of the guest OS to the microvisor. For example, one status variable exists which signals that the guest is in a critical code section where error handling is not possible. In this case, the microvisor will handle errors directly (by recovering a checkpoint) and will not delegate them to the guest.

The KCP also contains a reliable memory location which is excluded from checkpointing. This location can be used to store a persistent application state. The remainder of the KCP can be freely used by the guest OS. For the ARM configuration of the microvisor, this remainder is guaranteed to be at least 3 KiB in size. In the current implementation, this area is used for statically allocating data and as temporary heap and stack for the error classification task. Another purpose of this area is to store data which have to be passed to the microvisor in a reliable way. Although most hypercalls accept unreliable data as arguments, it can

---

<sup>3</sup>Small pages have a size 4 KiB on ARM926-based systems

be beneficial to allocate arguments in reliable memory to decrease the probability of a fault when passing arguments to the microvisor. Only two hypercalls exist, namely `fame_hypercall_fault_objects_map` and `fame_hypercall_fault_objects_classified`, which require parameter passing in reliable memory. Both hypercalls are used during fault handling. The main reason which drives this design decision is to avoid the occurrence of new faults during fault handling.

**Data and BSS segments:** The last reliably stored memory location includes the data and bss segments of the microvisor. Like the KCP, the data segments also contain a permanent storage area. The *FAME* microvisor uses this area to store the system time, status information, statistics, and checkpoint information.

**Main Memory:** The main memory is dedicated to the guest, except for a part used for checkpointing. All writable data of the guest OS and the application, like `.data`, `.bss`, `heap`, and `stack`, are stored here. Once mapped, the main memory is not directly controlled by the microvisor anymore. Instead, a *cooperative model* is used. The microvisor acts as a helper for managing allocated data objects. The main purpose of this cooperation is to keep track of data usage within the microvisor to efficiently provide checkpointing support. When the guest OS needs to allocate new memory, it decides where the new memory shall be allocated. Thereafter, it requests an object from the microvisor representing the new memory. The microvisor reliably stores the allocation information as well as metadata for the object. To free memory, the guest OS simply request the deletion of the associated objects. More details of this cooperative model is provided in Section 5.5.4

Overall, the footprint of the *FAME* microvisor is very small. In the case of the CoMET-based ARM926 platform (cf. Section 7.1.1) only 2.48 % of the overall system memory has to be reliable. If ROM is not considered, the amount of reliable memory will only be 128 KiB in total which corresponds to 0.04 % of all memories. This amount fits into ECC-protected scratch pads found in modern processors.

### 5.3.4 Subscriber Model Support and Data Object Management

As described in the last subsection, the microvisor keeps track of all dynamic data allocations of the guest OS and the application. On the one hand, this enables optimizing checkpoint creation. On the other hand, this allows implementing the subscriber model within the microvisor. As shown in Figure 4.8, the subscriber model is very important to decide whether or not errors have to be corrected. If an error affects the subscriber information, it will not be possible to decide how to handle the error. Hence, per Definition 3.1, the implementation of the subscriber model has to reside within the RCB.

The subscriber model is tightly coupled with the object management. The first step is to create an object via the hypercall `fame_hypercall_object_add`. This hypercall expects the object specification as parameters. In the case of memory

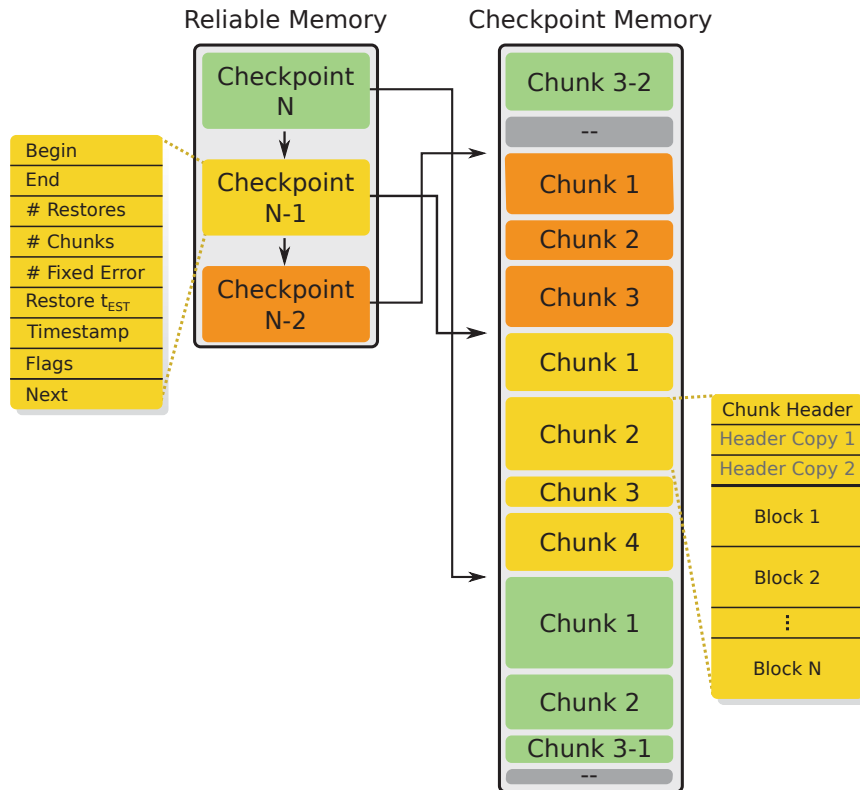
objects, this is the virtual address, the size, the object identifier, and a task identifier. The address and the size can be set arbitrarily as long as no objects overlap. The task identifier specifies the allocating task in the guest OS. In the subscriber model implementation of the microvisor, the object is automatically subscribed to the requesting task. In contrast to the description of the subscriber model in Figure 4.6, this behavior directly creates subscribed objects. This releases *FAMERE* from the burden to execute a second hypercall to explicitly subscribe the calling task. To destroy an object, `fame_hypercall_object_remove` is provided. Modification of already created objects is possible with `fame_hypercall_object_modify`. However, modification is limited. The base address of an object cannot be changed. Changing the object size will be only possible if the resulting object does not overlap with other objects. Other properties which can be changed are the reliability information, object flags, and the applied checkpointing strategy.

To subscribe another task the hypercalls `fame_hypercall_object_subscribe` and `fame_hypercall_object_subscribe_multiple` can be used. The latter hypercall subscribes the corresponding task to multiple objects. As long as the requested object is not erroneous, subscribing automatically unsubscribes weak-unsubscribed tasks. If the object is erroneous, weak-unsubscribing will be delayed until the corresponding object is corrected. This is an exception to the subscriber model shown in Figure 4.6. However, this behavior is necessary to avoid inconsistencies during error classification. To unsubscribe an object the three hypercalls `fame_hypercall_object_unsubscribe`, `fame_hypercall_object_unsubscribe_multiple`, and `fame_hypercall_object_unsubscribe_all` are provided. The first two hypercalls are the counterparts to the previously mentioned subscribe hypercalls. The latter hypercall can be used to unsubscribe a task from all the objects it is currently subscribed to. This hypercall is used by the guest OS to destroy a task.

All the hypercalls presented in this subsection are intended to be used only by *FAMERE* and the guest OS. It is not safe for applications to directly call these hypercalls. Especially the subscription-related hypercalls are exclusively provided for *FAMERE*. Since the microvisor does not know the tasks of the guest OS, *FAMERE* has to perform further actions after object subscription to synchronize the task states. For example, if tasks subscribe to an erroneous object, they have to be set to a blocking state. Otherwise, they remain in the ready queue and continue execution on erroneous data. More details of the subscriber model support of *FAMERE* are provided in Section 5.4.3

### 5.3.5 Checkpoint-and-Recovery

*Checkpoint-and-recovery* is an important strategy in the flexible error handling approach. It is used as a last resort and in the case that no application specific error correction method is available. At some points, the application or guest OS may fail to handle a fault. This is mainly the case if the fault handling routines themselves are affected by faults or very important data of the guest OS are affected which cannot be recovered easily. In such a case, the guest OS can instruct the microvisor



**Figure 5.5:** *FAME* Checkpoints

to restore the last checkpoint. A checkpoint in *FAME* consists of the whole system state. This includes the state of the microvisor, guest OS and application. Since data of the microvisor – and hence RCB data – are stored in checkpoints, checkpoint creation and recovery can only be realized by the microvisor. Due to the limited amount of reliable memory, checkpoints are stored in main memory. Theoretically, this breaks Invariant 3.1, because data of the RCB is stored on hardware which is not part of the RCB. Since this hardware is susceptible to faults this can lead to changes of RCB data from external sources. To circumvent this problem, the microvisor applies forward error correction, like, e.g., TMR, to store checkpoints.

The microvisor manages checkpoints in a ringbuffer. A checkpoint consists of metadata stored in reliable memory and the checkpointed data stored in unreliable memory. Figure 5.5 depicts an overview of checkpointing. The metadata contain pointers to the beginning and the end of the checkpoint data in the unreliable memory, various counters for statistics, the estimated time required for restoring, and various flags. The estimated restore time is not a  $WCET_{EST}$ . It is rather a prediction of the restore time without any guarantees.

The actual data of a checkpoint is stored in one or several chunks. A chunk represents a continuous memory range without any gaps. Each chunk has a header containing the base address of the stored memory range and the size of the range. In addition, the applied block encoder is stored in the header too. The header is

stored with TMR since the checkpoint memory is susceptible to transient faults. Data is stored in blocks of fixed size. This size depends on the used block encoder. Available block encoders are TMR, Reed-Solomon (RS) code [RS60], LDPC (Low-Density Parity-Check) code [Gal62], and *plain* encoding. However, due to the long encoding time of RS and LDPC, only TMR and *plain* are used. If TMR is used, data will be stored three times. In contrast, *plain* stores data without any encoding. Consequently, *plain* can only be used to store data classified as unreliable. All blocks in a chunk are encoded with the same strategy. As mentioned previously, the size of a block is fixed. This allows for high performance block encoding since size comparisons can be omitted in the encoder implementation. However, if the size of the memory range to store is not a multiple of the block size, the last block has to be encoded differently. Therefore, the *FAME* microvisor copies the corresponding data to a temporary location with the necessary block size. Afterwards, this temporary location is stored into the checkpoint. Restoring of such blocks is analogously realized. As shown in Figure 5.5, chunks are stored in a cyclic buffer. At the end of the checkpoint memory, storing starts again from the beginning. Wrapping is only possible at the end of a block. As soon as a chunk overwrites a chunk of an older checkpoint, that checkpoint is deleted. One exception to the previous rule exists. If the chunk would overwrite the last available checkpoint, the current checkpoint creation will be aborted. This exception is very important. Otherwise, the last complete checkpoint will be deleted. In the case that an error occurs while creating the new checkpoint, no recovery would be possible anymore. Such errors are likely to happen if erroneous data are read by the copy routine.

To avoid wasting time and memory space, the microvisor selects the data which are stored. In general, only memory ranges with allocated data are stored in a checkpoint. Unused memory is generally excluded. To decide which dynamically allocated objects have to be stored, the subscriber model and reliability annotations are used. Storing of reliable and subscribed objects is mandatory. Optionally, the microvisor can also store all subscribed objects, whether they are reliable or not. It is also possible to store all reliable objects, whether they are subscribed or not. The guest OS (or application) can instruct the microvisor to use any of the optional methods per checkpoint creation request. The block encoder can also be separately chosen by the application for each data object. However, data annotated as reliable are only allowed to be stored by TMR, RS, or LDPC. Other important data which have to be included in the checkpoint are the `.data` and `.bss` segment. During initialization of the guest OS, the hypercall `fame_hypercall_checkpoint_static_area_modify` has to be used to submit the corresponding information to the microvisor.

Since embedded real-time systems are considered in this dissertation, timing estimations for checkpoint creation are also provided by the *FAME* microvisor. This estimation is based on the size of the static memory regions and the size of all objects. Therefore, the microvisor maintains counters of the size of every allocated object class. By calling `fame_checkpoint_predict`, the prediction for static areas and the prediction for the objects are summarized and returned to the caller. This

function expects the desired checkpointing mode as input parameter. The mode specifies the optional checkpoint parts, like, e.g., additional checkpointing of all unreliable subscribed data. It is worth to note, that the prediction function is not a hypercall. Hence, this function can be directly called by the guest OS or the application. It is not necessary to provide a hypercall here since calling `fame_checkpoint_predict` does not change the status of the RCB.

To summarize, a checkpoint is constructed of several chunks. Each chunk consists of several blocks depending on the size of the checkpointed memory region. The following chunks are stored in order:

1. **Architecture chunk:** This chunk stores the hardware state, like, e.g. the currently enabled IRQs.
2. **Reliable data chunks:** After the architecture chunk, the reliable data is stored. This includes the stack, data, bss and heap segments of the microvisor as well as the PTDs and the KCP.
3. **Static data (guest OS):** The next chunks store the data and bss segments of the guest OS.
4. **Dynamically allocated memory objects:** The remaining chunks store dynamically allocated objects. The selection which objects to store here is guided by the subscriber model and reliability annotations.

Recovering a checkpoint works straightforward. The chunks are decoded in the creation order. After restoring the chunk containing the reliable data, the caches as well as the TLB are flushed. This step is necessary to activate possibly changed page table entries.

### 5.3.6 Summary

In this section, the most important concepts of the *FAME* microvisor were introduced. The main purpose of the microvisor is to provide the foundation for the flexible error handling approach. This means to protect the reliable computing base which is necessary to build software-based error handling solutions. For the protection, para-virtualization is used. With a very specific hypercall implementation, the microvisor avoids the modification of the RCB by erroneous data. Hereby, it is ensured that the hypercall parameters are within the specification and error-free. Compared to traditional virtualization techniques, the goal of the *FAME* microvisor is not to multiplex hardware or to create the illusion of a virtual machine which behaves like the physical hardware. Instead, the guest OS is still in charge of most of the operations it would perform without virtualization. When virtualized, the microvisor just inserts a new layer which ensures that error detection and correction in software continues even in case of faults.

So far, one question is not resolved: Can the *FAME* microvisor be called microvisor? The “-visor“-part can be easily answered. Except for virtual CPU and



multiple guest support, the *FAME* microvisor implements all aspects important for para-virtualization. The “micro“-part is harder to answer. Microkernels, like, e.g. L4, typically provide support for IPC, virtual memory, and scheduling. The *FAME* microvisor only provides basic virtual memory support. Scheduling is solely the task of the guest OS. Another related operating system concept is the exokernel. An important feature of exokernels is that the untrusted operating system running on top of the exokernel has to be a library operating system. This is also the case in *FAME*. Furthermore, in exokernels, applications can directly interact with the hardware. This is partially the case in *FAME*. All hardware components related to EDAC are only exclusively accessible by the microvisor. Other components, like, e.g., the graphics card can be directly used by the guest OS. However, the *FAME* microvisor integrates components an exokernel would never include. These are the object management, subscriber model, checkpointing, and low-level error handling support. According to the minimal principle formulated by Liedtke (cf. Definition 2.1), integrating such components into a microkernel will be tolerated if they cannot be implemented outside. In the case of *FAME*, these are core features to enable flexible error handling which cannot be implemented outside the RCB.

In fact, the *FAME* microvisor is located somewhere between exokernels and microkernels. However, in the view of the author, there are more common points with microkernels.

In the next sections, *FAMERE* is described. *FAMERE* is an additional important component in building the flexible error handling approach.

## 5.4 FAMERE

The *FAME Runtime Environment (FAMERE)* is the counterpart of the *FAME* microvisor. It is a library embedded in the guest operating system. Unlike the *FAME* microvisor, *FAMERE* is guest-specific. *FAMERE* is the component where all information – compile time as well as runtime – are combined to implement the flexible error handling approach. However, the task of *FAMERE* is not only to handle errors. In addition, *FAMERE* provides support for other aspects related to virtualization and reliability. In the next subsections, all major aspects are detailed.

### 5.4.1 Para-Virtualization Support

In contrast to full system virtualization, where a guest OS can be executed without modifications, para-virtualization requires porting efforts. Typically, the communication with the virtual machine monitor – in the case of *FAME*, the microvisor – is implemented with hypercalls. Instead of providing emulated hardware, like an interrupt controller, the microvisor exports such functionalities directly as hypercalls. To abstract the implemented hypercall mechanics, *FAMERE* provides C-bindings enabling C code to directly execute hypercalls. If, for example, the guest OS requires interrupts to be disabled, the appropriate hypercall (`fame_hypercall_irq_disable`) can be used.

*FAMERE* also acts as the communication endpoint for the opposite direction. When the microvisor signals an event – interrupt, timer tick, or error – *FAMERE* receives the event message and triggers the appropriate functions of the guest OS.

### 5.4.2 Checkpoint-and-Recovery

Basic checkpointing support is provided by the *FAME* microvisor. However, the microvisor only stores memory ranges and the state of hardware supported directly within the microvisor. To create a full system checkpoint, the register file and other status information have to be stored as well. Storing the registers is task of *FAMERE*.

If the application needs to create a checkpoint, the function `famere_checkpoint_create` can be called. To create a checkpoint, *FAMERE* spawns a dedicated checkpoint task responsible for the creation. The checkpoint task is initially created with background priority. Thereafter, a dependency from the calling task to the checkpoint task is added. Due to the added dependency, the priority of the calling task gets inherited. If another task also initiates checkpoint creation, only a further dependency will be created. Hereby, the priority will be inherited to the checkpoint task as well if the priority of the calling task is higher.

### 5.4.3 Scheduling and Subscriber Model Support

Despite virtualization, the guest OS remains in charge of scheduling and task management. From the perspective of the microvisor, the guest OS with all running tasks is just one single binary. The microvisor is unaware of application tasks. However, some aspects of the subscriber model and flexible error handling require modification of the guest OS scheduler. Therefore, *FAMERE* hooks into the guest OS scheduler. The following behaviors have to be enforced:

- **Task creation:** If the created task uses a static object which is affected by an error, this task has to be suspended. In the case of RTEMS as guest OS, *FAMERE* adds a new task blocking state, `STATES_WAITING_FOR_ERROR_HANDLING`. Hence, affected tasks are set to his new state.
- **Task destroy:** As soon as a task is destroyed, all of its subscribed objects have to be unsubscribed. Therefore, *FAMERE* calls the hypercall `famere_hypercall_object_unsubscribe_all` for the task.
- **Task state updates:** The new blocking state remains active until all errors affecting the task are corrected. *FAMERE* hooks into task state updating to unblock a task after successful error correction.

To complete the subscriber model, *FAMERE* provides the functions `famere_object_subscribe[_multiple]` and `famere_object_unsubscribe[_multiple]`. These functions abstract the corresponding hypercalls which must not be called

directly by the application. A further abstraction of the subscribe hypercall is necessary to properly update the state of the subscribing task. If, for example, tasks subscribe to an erroneous object, the task states have to be set to `STATES_WAITING_FOR_ERROR_HANDLING`.

*FAMERE* also provide its own scheduler which optionally can replace the scheduler of the guest OS. Flexible error handling is orthogonal to the used scheduling strategy. However, the scheduler has to support precedence relations. The *FAMERE* scheduler is based on EDF\* [Hen78; CSB90] which fulfills this requirement. However, the EDF\* realization implemented in this dissertation is slightly modified. Deadlines and release times are still transformed using the original equations 2.1 and 2.2 shown in Section 2.5. In contrast to the original EDF\*, the dependencies are not dropped and still reported to the operating system. This behavior is beneficial in cases where errors affect the system. In *FAME*, error correction methods are executed as separate tasks. By considering the dependencies of all affected tasks, the operating system can appropriately assign new task priorities based on the used scheduling strategy. For example, in the case of EDF\*, the error correction method inherits the earliest deadline of all affected tasks.

#### 5.4.4 Memory Allocation

To support the subscriber model, the heap allocator of the guest OS has to be modified. After an allocation, the corresponding memory has to be registered to the microvisor. If memory is freed, this registration has to be revoked.

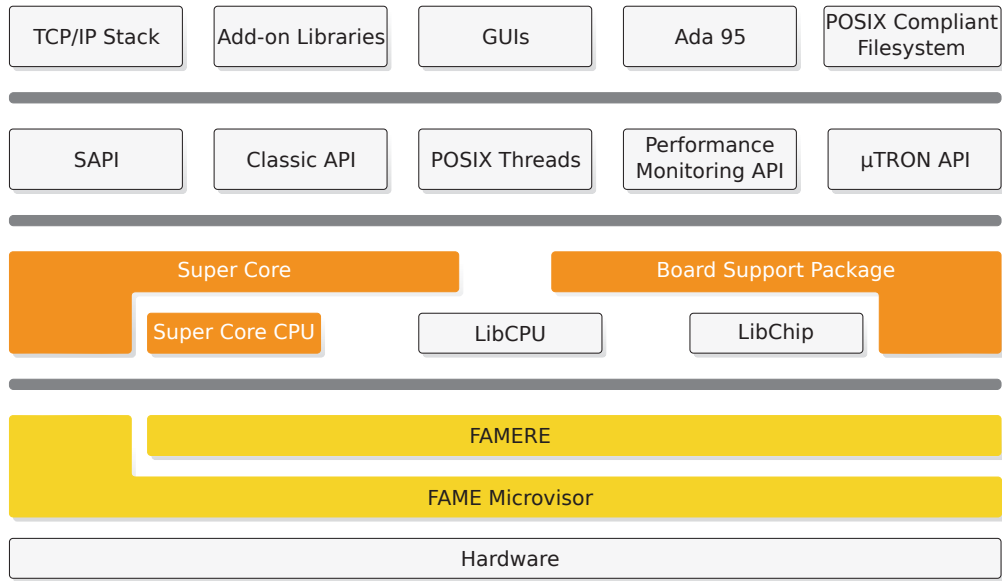
To ease porting the guest OS, *FAMERE* provides allocators for stack and heap. Therefore, *FAMERE* uses the object information exported by the *FAME* microvisor. The metadata of objects can be directly accessed. To allocate new memory, *FAMERE* searches for the first suitable gap between two objects or for free space after the last allocated object. If a free memory range is found, *FAMERE* will instruct the microvisor to create the corresponding new object. As a positive side effect, all allocation information are stored in the RCB. Hence, the attack surface for errors affecting the guest OS is getting smaller.

#### 5.4.5 Error Handling

By far, the most important task of *FAMERE* is error handling. The flexible error handling approach is realized by interfacing with the compiler-generated library *librecon*. This library is primarily used to query information on statically allocated data objects and to query possible correction methods for a given set of occurred errors. *FAMERE* contributes all necessary runtime information. The complete error handling procedure is described in Section 5.6.

#### 5.4.6 Summary

*FAMERE* is a very important component of the *Fault Aware Microvisor Ecosystem*. Information collected during compile time and information on the current system



**Figure 5.6:** Overview of RTEMS and *FAME*

state are incorporated to form the flexible error handling approach described in Chapter 4. Moreover, porting the guest OS to *FAME* is eased by *FAMERE* due to the provided support for interfacing the microvisor, scheduling, and memory allocation.

In the next section, some aspects of porting RTEMS to *FAME* are detailed.

## 5.5 Para-Virtualizing RTEMS for FAME

The Real-Time Executive for Multiprocessor Systems or RTEMS is a full featured real-time operating system. It supports a variety of open API and interface standards including POSIX-1003.1b. In this section, RTEMS is introduced first. Thereafter, the virtualization efforts are depicted.

### 5.5.1 RTEMS Overview

In Figure 5.6, the RTEMS architecture with *FAME* is depicted. Without *FAME*, RTEMS would run directly on the hardware. In the para-virtualized case, it has to interface with *FAME*. RTEMS itself consists of three layers. In the upper layer, support for various system services is provided. This includes, for example, file systems and additional libraries, like, e.g., *libz*. The middle layer consists of the implementation of various APIs. In this dissertation, the classic API and POSIX are used. The classic API represents the native RTEMS interface. Architecture and platform specific parts as well as the RTEMS kernel, called *super core (score)*, are located at the bottom layer. In this layer, porting is required to support *FAME*.

The components which have to be ported are colored orange in Figure 5.6. In the next subsections, the porting is detailed.

### 5.5.2 Board Support Package and Infrastructure

Platform specific adaptations and drivers are provided by the board support packages (BSPs). In this dissertation, BSPs are implemented for MPARM-based and CoMET/METeor ARM926-based platforms, and for the TK71 development board. MPARM implements the ARMv3m architecture without MMU support. Hence, the *FAME* microvisor is not compatible with this platform. Virtualization is only provided for the used CoMET-based platform and for the TK71 board.

The virtualization of the BSP is straightforward. The clock driver is reduced to its minimum which is to configure the *FAME* microvisor to the right tick interval. Handling of IRQs is ported to the appropriate hypercalls. The UART driver is nearly unchanged except for the replaced base address of the memory mapping.

To support virtualization, the RTEMS build system is modified. To enable *FAME* support, RTEMS has to be configured with `--enable-fame`. Without this configuration option, RTEMS is built bare-metal. This allows for comparisons between virtualized RTEMS and native RTEMS based on nearly the same code base.

### 5.5.3 Context Switching

Switching between two tasks requires execution of privileged instructions. These are required to update status registers. Typically, these are the program status words and the page table entry. Since RTEMS consists only of one address space, page table switches are not necessary. However, updating the program status words is still required.

In Listing 5.4, the context switch code for ARM926 of native RTEMS was already depicted. The `msr` instruction on line 8 is problematic due to the silent fail. This instruction is used to overwrite the current processor status. Executed in user-mode, it is only allowed to change the condition bits. All other bits are read-only. However, writing to these bits does not lead to an exception. Moreover, writes are ignored. Consequently, the context switch code has to be ported. The ported version is depicted in Listing 5.5. Instead of writing directly to the program status word, the hypercall `fame_hypercall_cpsr_set` is used (line 13). To avoid using a second hypercall for restoring the IRQ disable level, the previously mentioned hypercall expects the program status word and the IRQ disable level as input parameters. The IRQ disable level, or short `IRQ-level`, represents an internal counter of RTEMS for nested IRQ disable requests. The counter avoids activation of IRQs too early if more than one subsystem requires disabled IRQs. Only if all subsystems enable IRQs again (the counter is zero) IRQs will be activated. The `IRQ-level` is additionally stored in the task's context store location within the thread control block of RTEMS. In lines 5 and 6, the `IRQ-level` is fetched. The hypercall responsible

**Listing 5.5:** RTEMS context switch code for ARM ported to *FAME*

```

1 ;r0 pointer to context store location
2 ;r1 pointer to context load location
3 _CPU_Context_switch:
4  mrs    r2,    cpsr
5  ldr    r3,    fame_irq_disabled_ptr
6  ldr    r3,    [r3]
7  stmia  r0,    {r2, r3, r4-r11, r13, r14}
8  ;context now stored to r0 -> load context from r1
9  ldmia  r1,    {r2, r3, r4-r11, r13, r14}
10 push  {r2, r3, lr}
11 restore_cpsr:
12  ldmia  sp,    {r0, r1}
13  bl     fame_hypercall_cpsr_set
14  cmp    r0,    #EINTR
15  beq    restore_cpsr
16  ldr    lr,    [sp, #(2 * 4)]
17  add    sp,    sp, #(3 * 4)

```

for setting the IRQ-level and the program status word is executed in a loop (lines 11-15). This is necessary since hypercalls can be interrupted by transient faults. Hence, the hypercall return value has to be checked.

### 5.5.4 Memory Management

RTEMS implements several heaps. One heap is the RTEMS work space. The work space contains all major operating system data, like, e.g. task information and semaphore status. The size of this heap is rather small. During compilation of the application, the size is determined by evaluating the configuration statements in the source code. In all the examples used in this dissertation, the size never grows beyond 64 KiB. Among the work space two other heaps exist. One heap for the stacks of the tasks and another heap for the dynamic memory allocations (mainly for mallocs). Depending on the configuration, these heaps also can be united. In the next subsection, the drawbacks of the RTEMS heap implementation are depicted. In consequence, the heaps for the stacks and mallocs are replaced by a customized heap implementation.

#### 5.5.4.1 RTEMS Heap Implementation

In Figure 5.7 the RTEMS specific heap implementation is depicted. Each heap is represented by a data structure named `Heap_Control`. The memory range handled by the heap is stored in `area_begin` and `area_end`. Allocations as well as free memory are managed in `Heap_Blocks` (yellow blocks in Figure 5.7). To quickly find free memory, the free blocks are chained in a double linked list. In the depicted example, two free blocks (green) and three used blocks (red) are available. As can be seen, the validity of the entries in `Heap_Block` strongly depends on whether

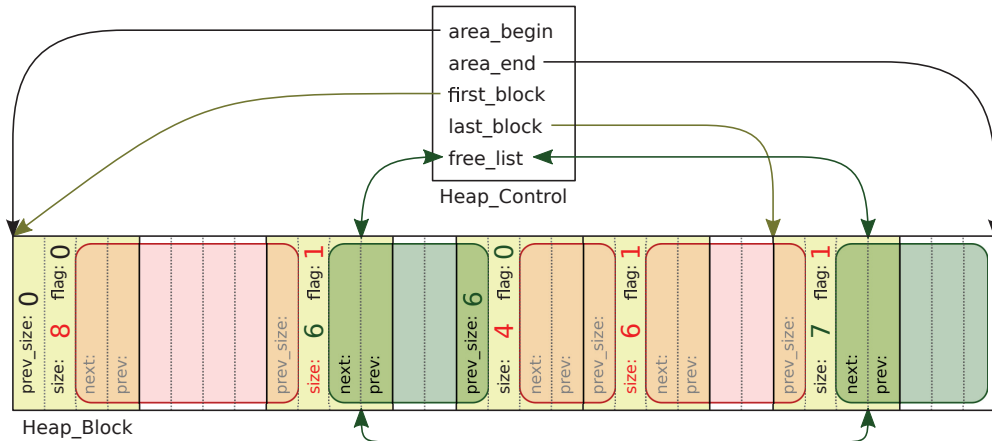


Figure 5.7: RTEMS native heap implementation

the block is used or free. Only the size and flag fields are always available. The information on the size of the previous block (`prev_size`) will only be valid, if the flag is set to '0'. The pointers to the next and previous free block (`next` and `prev`) will only be valid if the directly adjacent block has the flag set to '0'. To determine this directly adjacent block, the size data field can be used. The size of the block data structure itself plus the corresponding size of the memory area after that block is stored in the size field. Hence, it is sufficient to add the size to the beginning of a block to reach the next block on the heap. This highly optimized data structure significantly reduces the imposed overhead for heap management. For example, between the second and the third allocation in Figure 5.7 only one word is wasted.

However, the drawback of this heap implementation is the mixture of management information and user data. On the one hand this is a security issue: the application can (accidentally) overwrite heap management data due to memory over- or underflows. On the other hand, there are reliability issues. The heap stores a mixture of data classified as reliable or unreliable. Hence, a `Heap_Block` can be partially reliable and unreliable at the same time. This is not supported by the *REPAIR compiler*. Solutions showing how to cope with the reliability issues are described in the next subsection.

#### 5.5.4.2 Hardening the RTEMS Heap

One obvious method to protect the metadata of the heap is *checkpoint-and-recovery*. In the case of an error, the complete system state – and hence also the heap – is restored by recovering the last checkpoint. This solution is applicable to all kinds of heaps. However, with respect to flexible error handling, this solution is only feasible for heaps storing solely reliable data without annotated correction methods. In such a constellation, recovering a checkpoint is the only applicable correction method anyway. The RTEMS work space perfectly matches these specifications. Consequently, the native RTEMS heap implementation is used and the complete

heap is stored in a checkpoint.

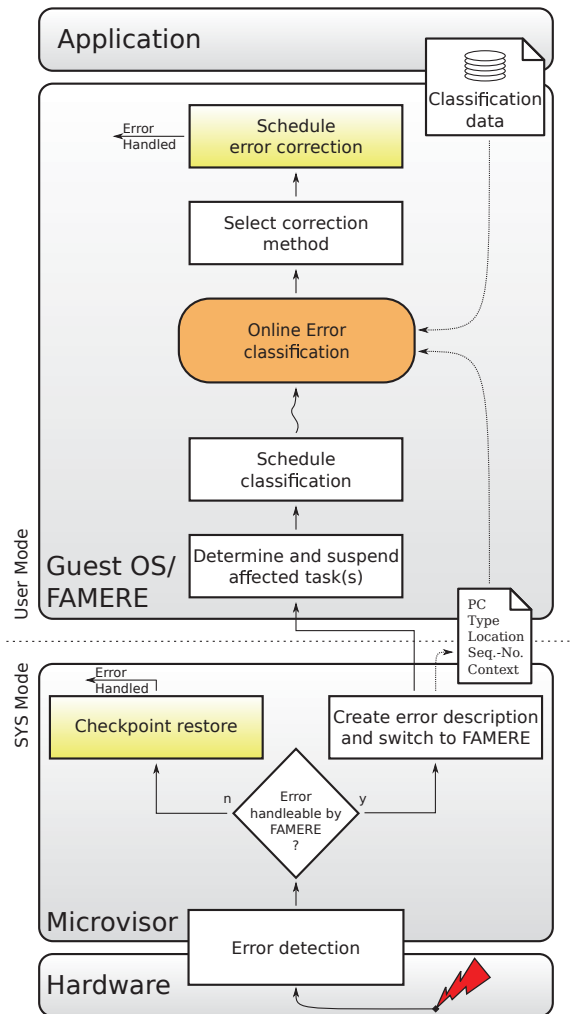
The next possible method would be checkpointing only the metadata. Due to the complex semantics of a heap block, this method is not feasible. Depending on the current block type, a rollback would partially overwrite the application data which leads to an inconsistent state. Another method to protect the metadata would be to apply forward error correction, like, e.g. TMR. In the case of an error, the first step is to vote about the block type and size. Thereafter, the valid fields can be corrected. One drawback of this solution is the overhead. Currently, the overhead is only one word per block. With TMR, the overhead grows to nine words. The other drawback would be that data classified as reliable has to be stored as well. Therefore, checkpointing has to be used. However, this creates a situation where the metadata is stored with TMR and partially within checkpoints due to the mixture of metadata and application data on RTEMS-managed heaps. This would increase the overhead even further.

To eliminate the disadvantages of the previously mentioned methods the heap allocator of *FAMERE* is used. Therefore, the heap blocks are replaced by microvisor managed objects. These objects are required anyway to store information for *librecon*. From the perspective of RTEMS, the overhead with respect to main memory is zero since all metadata are stored in reliable memory. The separation of the management data leaves the application the choice which algorithm to implement to protect valuable data. For example, it is possible to store data classified as unreliable completely unprotected. This is a significant advantage of the virtualized heap implementation. Another advantage is that the application cannot corrupt the heap anymore by over- or underflows. However, it is still possible that other application data gets overwritten. Protection can be enabled by insertion of guards and periodically checks or by using page table based protection. However, both solutions have drawbacks. Since page table entries are too coarse grained, a huge amount of memory is wasted. Furthermore, due to the additionally required page table entries, the amount of reliable memory needed to store the page table entries would dramatically increase. However, protection against over- or underflows will not be necessary if *FAME* is used. In this dissertation, it is assumed that applications will work correctly and will be free of software bugs if not affected by transient faults. This means that over- or underflows will normally not occur. Due to the *REPAIR compiler*, wrong offsets or pointers can be also excluded under the influence of transient faults. To recapitulate, *REPAIR* enforces the prohibit rule (Definition 5.1) and propagation rule (Definition 5.2). Consequently, the correct values in offset and pointers can be guaranteed as long as erroneous data classified as reliable are corrected prior to usage.

### 5.5.5 Summary

Due to the fact that *FAMERE* implements the whole interfacing with the *FAME* microvisor, the virtualization and porting of RTEMS to *FAME* is straightforward. The BSPs are modified to use hypercalls to access hardware components protected





**Figure 5.8:** *FAME* Error Handling Procedure

by the microvisor. Clock management as well as dynamic allocation of data is replaced by services provided by *FAMERE*.

At this point, all components are described which are necessary to realize the flexible error handling approach. In the next section, the realization is detailed.

## 5.6 Flexible Error Handling - Realization

An overview of the error handling process is depicted in Figure 5.8. Error handling starts in the microvisor (bottom of Figure 5.8) and ends with the acknowledgment of the successful execution of the selected error handling method (not depicted in the figure). All stages of the complete error handling procedure are detailed in the next subsections.

### 5.6.1 Stage 1: Low-Level Error Handling

Error handling starts with the detection of the error and a context switch to the microvisor. As already mentioned, error detection is outside the scope of this thesis. Hence, it is assumed that some kind of error detection is present. If an error is detected, the microvisor will be notified and the error handling will begin (bottom of Figure 5.8). After storing the current context, the microvisor checks whether or not the fault can be handled outside the RCB. Errors, for example, which affect *FAMERE*, are very unlikely to be handled by *FAMERE* itself – again, this is the previously described chicken-and-egg problem. In such a case, the microvisor automatically restores the last system checkpoint or, if no checkpoints are available, resets the complete user space as a last resort. If *FAMERE* is not affected, error handling is delegated by sending a message to *FAMERE*. Before jumping to the *FAMERE* message handler, the microvisor creates an error description containing information about the occurred error as well as the user space context. With switching to *FAMERE*, the second stage begins.

### 5.6.2 Stage 2: Mapping Errors to Tasks

After switching to *FAMERE*, the tasks affected by errors have to be determined. To find these tasks, the subscriber model is used. In the subscriber model, objects carry the information which task is subscribed. Hence, the objects affected by errors have to be located. Dynamic objects are maintained by the microvisor in a red-black tree sorted by the base address. Hence, the lookup is very fast. In the case of static objects, *librecon* is queried. To unify further error handling, the returned static objects are added to the red-black tree of the microvisor. After error handling, these objects are automatically removed.

When all affected objects are determined, the corresponding tasks are suspended and the objects are attached to the error description created during stage one. Therefore, *FAMERE* uses the hypercall `fame_hypercall_fault_objects_map`. After all errors have been mapped to the corresponding tasks, the classification stage is scheduled. The classification stage is executed as a separate task with a priority equal to the highest priority of all affected tasks. Hence, if higher prioritized tasks are not affected by faults, error handling will be delayed until all higher prioritized tasks finish their execution.

Stage one and two are immediately executed after error detection. Interrupts are deactivated. To lower the probability that additional errors occur during execution, stage two operates on a temporary heap and stack allocated in the KCP section of the reliable memory. Nevertheless, errors can still occur. In such a case, error handling will be reset. Thus, it is very important to limit the possible state changes of this stage. Only two state changes are allowed during this stage. The first one is to submit the error to task mapping to the microvisor which is able to reliably add the mapping to the corresponding error description data structures. The second allowed status change is the scheduling of stage three. To avoid creation of several

classification tasks, the task is constrained to only one copy. To realize this, one classification task is created during the initialization of *FAMERE*. After finishing a classification, the task is suspended. If a new classification has to be performed, stage two just resets the classification task and sets the appropriate priority. Hence, stage three also is designed to be (nearly) stateless.

### 5.6.3 Stage 3: Classification and Handling Method Selection

The task of the third stage is to classify all errors which are not classified so far and to provide at least one error correction solution which handles every error. The classification is realized by *librecon*. Therefore, *FAMERE* calls the exported function `recon_fcg_generate`. This function returns a Pareto-optimal front of possible solutions. As described in Section 5.1.3, a solution is represented by a Fault Correction Graph (FCG). By construction, each solution element of the returned FCG is able to correct all errors which are detected at the point in time the FCG is generated.

In the next step, one solution is selected. The selection is based on the available slack time and other parameters annotated on the FCG. Thereafter, *FAMERE* creates a task for each correction method referenced in the FCG and stores the task ID to the corresponding FCG node. However, some exceptions exist. If the correction method is *ignore*, *FAMERE* will directly execute stage four. If checkpoint recovery is part of the FCG, *FAMERE* will immediately trigger the recovery.

As soon as all tasks are created, *FAMERE* notifies the microvisor using the hypercall `fame_hypercall_objects_classified`. This hypercall expects the selected FCG as parameter. Since *FAMERE* stores the task IDs of each correction method in the FCG, the microvisor can determine how many correction methods per object respectively error have to be executed. The former is stored as *correction count* within the object metadata. The correction count is important in the next stage for acknowledgment of the finished error correction.

Unfortunately, the time period between creation of the first error handling task and the notification of the microvisor is a critical section since the system state is changed. If a fault occurs in this period of time, tasks will remain which possibly perform invalid corrections with respect to the new classification triggered by the additional faults. Consequently, *FAMERE* sets the actual system state to critical which forces the microvisor to restore the last checkpoint when a new fault occurs. To avoid checkpoint restore-loops, the creation of new checkpoints is blocked, as long as classification of errors is pending.

### 5.6.4 Stage 4: Error Correction and Acknowledgment

In the last stage, the actual error correction is performed by calling the application specific handler routines. Hereby, each error correction routine is scheduled according to their priority. The priority of the correction method is the maximum priority of all tasks affected by the corresponding error. If dependencies to other correction

methods exist, the appropriate dependencies will be added.

After the correction method finishes, the corrected error has to be acknowledged to the microvisor by calling `fame_hypercall_ack_single_correction_done`. This hypercall decreases the correction count of the affected object. If the count reaches zero, the object will be considered as completely corrected. If the object was a static object temporarily added by *librecon*, the object will be deleted. A blocked task can continue execution as soon as all used objects are corrected.

As soon as all objects mapped to an error are corrected, the error entry is deleted by the microvisor. At this point in time, error handling is completed.

## 5.7 Summary

In this chapter the *Fault Aware Microvisor Ecosystem (FAME)* was presented. *FAME* realizes the flexible error handling approach. The ecosystem consists of an offline part, executed during compile time, and an online part, comprising the runtime components. During compile time, the semantics of applications are extracted by the *REPAIR* compiler. The semantic is important to answer the question **if** and **how** errors have to be handled. Since transient faults leading to errors occur during runtime, the semantic has to be available to the online components of *FAME*. Therefore, *REPAIR* encodes the semantic in a classification of possible error outcomes. Together with the compiler-generated library *librecon*, the runtime system can query the classification data base for possible error handling solutions.

To get a more fine-grained resource usage model and to be able to map errors to tasks, the subscriber model is used. The mapping is important to answer the question **when** errors have to be handled. Due to the mapping, it is possible to add dependencies between the error correction tasks and the tasks waiting for error correction. Hence, the problem can be reduced to an ordinary task scheduling problem with precedence. Implementing the subscriber model involves many software layers. It starts in the compiler which provides support for rewriting allocations and ends in the *FAME Runtime Environment* which applies the semantics to the scheduler. Thereby, the subscriber model connects the offline and online worlds.

However, not only the subscriber model affects several software layers. The error handling itself involves the *FAME* microvisor, *FAMERE* and *librecon*. Only the microvisor is protected against faults due to execution in the RCB. All other components are susceptible to faults. Hence, the microvisor has to ensure that *FAMERE* and *librecon* are fault-free prior delegating error handling. If this is not the case, a checkpoint will be immediately recovered. To protect the state of the microvisor, virtualization is used. To minimize the code base required to be executed within the RCB, the applied virtualization concepts are tailored to the needs of fault-tolerance and embedded systems.

In the next chapter the main benchmark application is detailed. Afterwards, *FAME* is evaluated.

# Demonstrator: H.264 Video Decoder Application

---

## Contents

---

<b>6.1</b>	<b>H.264 Decoder Library – <i>libh264</i></b>	<b>105</b>
<b>6.2</b>	<b>Simple Decoder Application</b>	<b>106</b>
<b>6.3</b>	<b>Real-Time Decoder Application</b>	<b>106</b>
6.3.1	Timing	107
6.3.2	Scheduling	110
6.3.3	RTEMS Modifications	112
<b>6.4</b>	<b>Summary</b>	<b>113</b>

---

In Chapter 4, the feasibility of flexible error handling is described for different kinds of application fields. However, writing executable benchmark applications with a sufficiently large workload and precisely specified real-time constraints is difficult. Furthermore, the applications have to be ported to RTEMS. Consequently, only the H.264 decoding benchmark is fully evaluated in this dissertation.

In the next subsections, two different versions of the decoder are investigated. The first variant is a simplified version which is used to evaluate the error classification. In this version, the video is decoded as fast as possible without respecting real-time constraints. The second variant is an application running under RTEMS with real-time constraints. The latter version is used to evaluate the real-time behavior of the flexible error handling approach. Both decoder applications are based on *libh264* which is described in the next section. Afterwards, the decoder applications are detailed.

## 6.1 H.264 Decoder Library – *libh264*

ISO/IEC 14496-10 or H.264 [Dra03] is a video compression standard used widely in the area of multimedia applications. H.264 supports different profiles defining the video and compression quality. The profile with the highest quality is the High 4:4:4 Predictive Profile (Hi444PP). It supports 4:4:4 chroma sampling, up to 14 bits per sample, and lossless region coding. The Constrained Baseline Profile (CBP) yields the lowest quality.

In this dissertation, the H.264 codec is provided by a library called *libh264*. It is based on the source code provided by Martin Fiedler [FB04]. The library consists of about 3000 lines of C code and 1000 lines of header files. It supports the following subset of CBP:

- I and P slices
- 4:2:0 chroma format
- 8 bit sample depth
- CAVLC entropy coding

Arbitrary slice ordering, redundant slices, and multiple reference frames are not implemented.

## 6.2 Simple Decoder Application

The simple decoder application can be seen as a wrapper around *libh264*. It controls *libh264* and feeds it with input data. The decoded video is sent via a network socket to a (remotely) running application, like, e.g., QtNetview (cf. Section 3.6) which can analyze and display the frames. The simple H.264 decoder application is implemented for different platforms running under different operating systems. For example, ports are available for Linux (x86) and RTEMS (ARM).

The simple decoder is running completely without timing constraints. It decodes a given H.264 encoded video as fast as possible. The main purpose of the simple decoder is to study possible impacts of transient faults. Thus, it is used in Section 4.2.1 to create a classification of possible error impacts. The simple decoder application is also used in Section 7.2.2 to evaluate the probabilistic error model.

## 6.3 Real-Time Decoder Application

The real-time decoder application is based on *libh264* as well. In contrast to the simple decoder variant, this decoder is designed for embedded real-time systems. This means the real-time decoder application maps the decoding process to several tasks with corresponding real-time characteristics, such as release time, execution time, deadline, and dependencies.

Determining the execution time for decoding H.264 video material is not trivial. In the next section, execution time estimation is detailed. When knowing the execution time, scheduling is possible. In Section 6.3.2, the splitting of the application into tasks and the scheduling is sketched. The target operating system is RTEMS. For a better support of *FAME* and the application, RTEMS is modified. The modification mainly affects the task management of RTEMS and the scheduling subsystem. In Section 6.3.3, more details on the RTEMS modifications are given.

	Video	Resolution	# Frames
A	TV series	320 × 352	15499
B	Computer animated movie	320 × 352	14314
C	Music video	640 × 360	9083
D	TV movie	620 × 476	62952
E	Computer animation	848 × 480	21244

Table 6.1: Analyzed videos

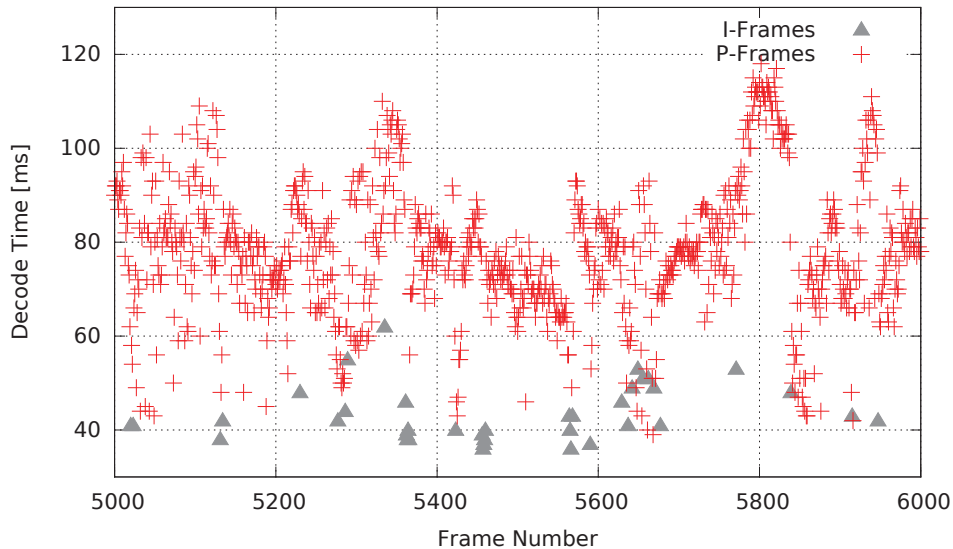
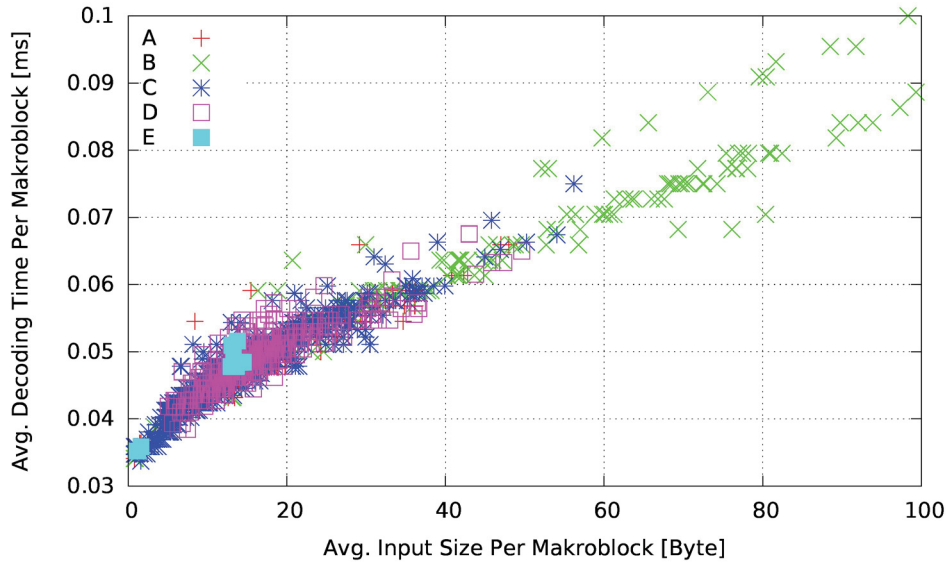


Figure 6.1: Frame decoding time for successive frames

### 6.3.1 Timing

The decoding time of a frame is required for scheduling. Only if the execution times of tasks are available, calculating the slack time is possible. Using the code of *libh264*, an analysis of the decoding function was performed using the aiT [WEE+08] WCET analysis tool for an ARM7-based platform. Compared to the measured results on this platform, the estimation yields a  $WCET_{EST}$  that is ten times higher than the highest observed frame decoding time. Obviously, such an overestimation is useless to calculate the available slack time. This leads to the conclusion that better estimations are only possible during runtime.

In a first approach, the relations between subsequent frames are analyzed by considering typical H.264 encoded videos. The used videos are summarized in Table 6.1. In Figure 6.1, the decoding time is depicted for a sequence of thousand frames. The experiment was executed on the CoMET-based ARM926 platform with video C (cf. Table 6.1) The decoding times vary between 35 ms and 120 ms. The figure shows that it cannot be expected to find a dependency of execution times between subsequent frames. However, it also shows that it is useful to distinguish



**Figure 6.2:** Decoding time for I-Frames in relation to average size of macro blocks

between P- and I-frames. While I-frames are self-contained, P-frames can reference previously decoded frames. In the case of *libh264*, only the last decoded frame can be referenced. In contrast to P-frames, I-frames are decoded faster and the number of I-frames is smaller by an order of magnitude or more.

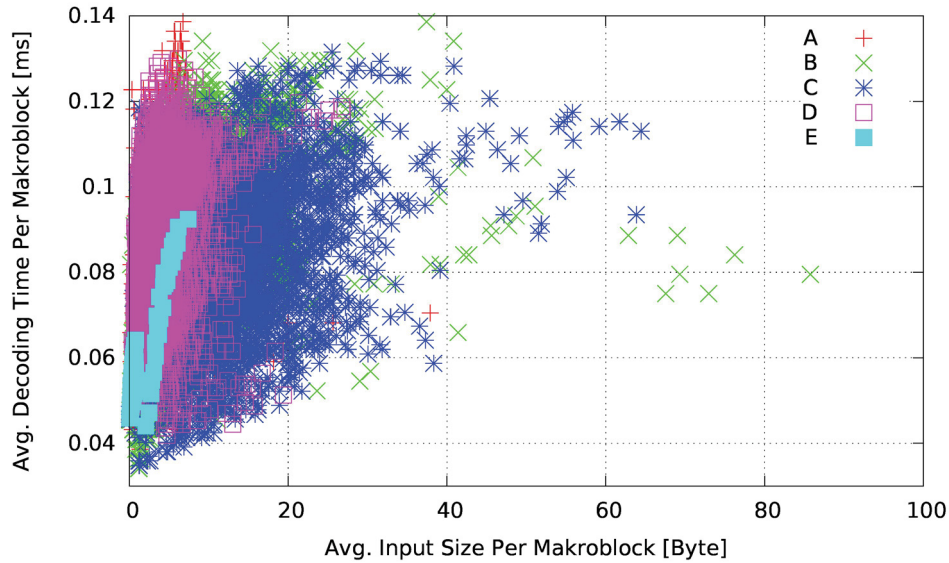
To determine the frame type, reading the header of the frame is necessary. By reading the header, it is also possible to get the resolution and hence the number of macro blocks. In H.264, a frame is constructed of macro blocks. Since neither static WCET analysis nor knowledge about frame sequences are a suitable basis to estimate the decoding time of a frame, the next approach tries to use the input size and frame header information for estimation. In the case of I-frames it is feasible to define a probabilistic upper bound. The average decoding time seems to be related to the average macro block size. In Figure 6.2, the measured values are plotted. The outcome for the same measurement made with P-frames is depicted in Figure 6.3. Unfortunately, the decoding time of such frames are not primarily depending on the size of the input alone. An upper bound for P-frames cannot be defined. However, since P-frames are the frame type most frequently used, estimations purely based on the input size are not meaningful.

To cope with this problem more information has to be gathered. Therefore, Bönninghoff<sup>1</sup> proposes to divide *libh264* into several stages. At the beginning of the first stage, the prediction of the decoding time is highly over-estimated. However, during decoding of a frame, the prediction becomes more and more accurate. The stages are as follows:

1. **Slice header parsing:** This stage only parses the slice header. After com-

<sup>1</sup>Not published yet



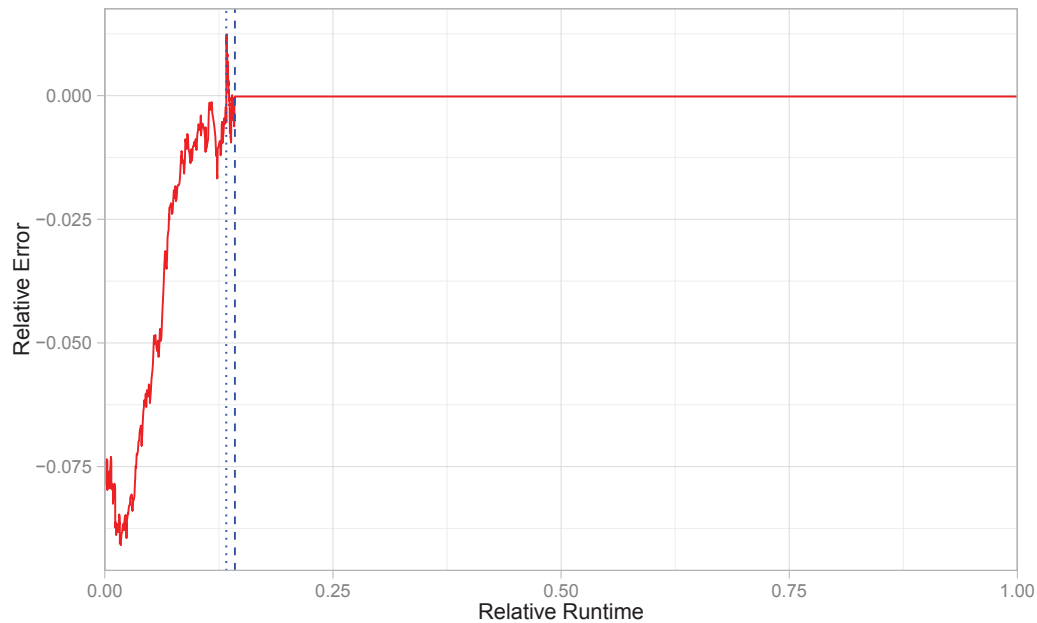


**Figure 6.3:** Decoding time for P-Frames in relation to average size of macro blocks

pletion, the number of macro blocks to decode as well as the slice type is known. The term slice denotes the encoded input of a frame. H.264 allows for multiple slices per frame. However, *libh264* is constrained to only one slice per frame.

2. **Slice parsing:** During slice parsing the whole slice input is read. The information is stored in the corresponding arrays which are required in later stages. After completion of this step, all macro block types, intra subdivisions and modes, and motion vector deltas are available. These are important data to significantly reduce the over-estimation of the decoding time prediction.
3. **Slice deriving:** Since motion vectors have a resolution of a quarter pixel, it is not possible for the previous step to decide how many filtering steps will be necessary. In this stage, the actual motion vectors are derived. After the slice deriving stage, a good estimation for the execution time of the final slice rendering stage is available.
4. **Slice rendering:** This stage is responsible for writing the decoded frame. Therefore, all transformations are executed and the kernels are applied.

The prediction of the execution time becomes more and more precise with every stage. Figure 6.4 shows the relative error of the prediction versus the measured runtime of a randomly selected frame. The x-axis shows the normalized runtime of the decoding from the beginning (0.00) to the end of the decoding (1.00). The dotted blue vertical line denotes the end of the slice parsing step and the dashed blue line denotes the end of slice deriving. After each decoded macro block, a new

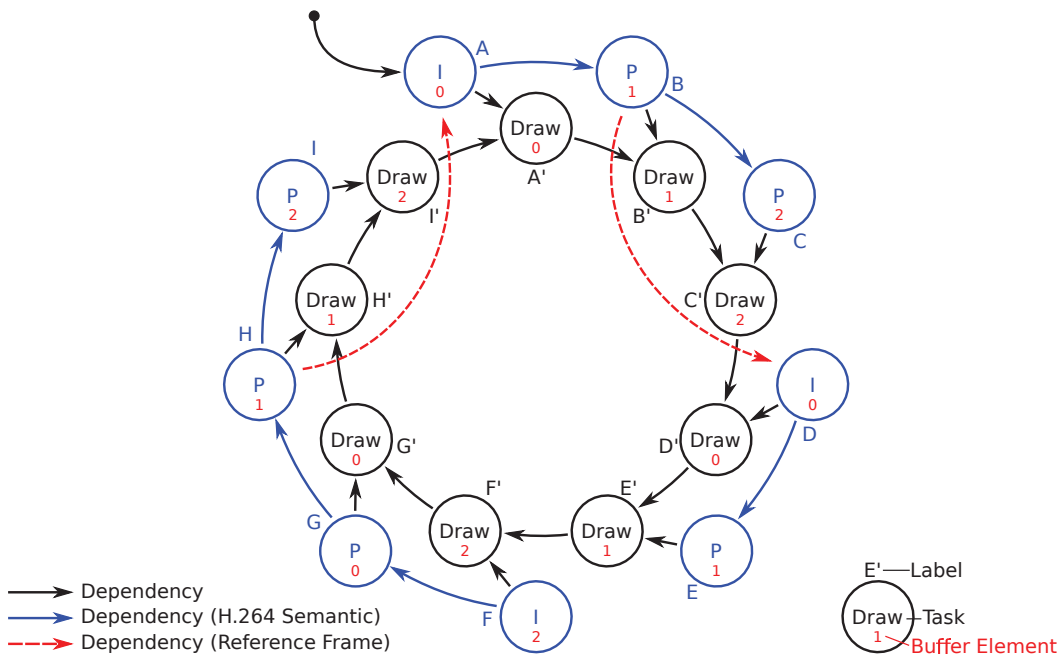


**Figure 6.4:** Relative error of decoding time estimation (with kind permission of Björn Bönninghoff)

estimation is made. The relative error of this estimation is plotted in red. The estimation of the frame decoding time is fairly accurate after the slice deriving step is finished. The runtime of stages one to three is very short and is mainly determined by the amount of input data. Hence, (over-)estimation of the execution time for those stages is easily possible at the very beginning of decoding. After execution of the first three stages, a sufficiently precise execution time estimation is available which can be used for scheduling the last stage. Hence, it is useful to divide *libh264* into two parts. Part one comprises the first three stages and the second part consists only of the last stage. In the next section, the execution time estimation and the partitioning is considered for scheduling.

### 6.3.2 Scheduling

Before scheduling and task partitioning can be described, another important aspect has to be considered. In Figure 6.1, high fluctuations of the decoding times within a video were already shown. This has severe consequences for decoding. For example, if a frame rate of ten frames per second (fps) is assumed, the available time for decoding a frame will be 100 ms. As can be seen in Figure 6.1, frames exist where the decoding time is above 100 ms. Hence, it will not be feasible to decode the video in real-time if frames are decoded in periods of 100 ms. However, it should be noted that the majority of frames can be decoded within the 100 ms boundary. If the remaining time not used for decoding the current frame is used to decode the next



**Figure 6.5:** Task dependencies of the H.264 decoding application

frame in advance, it will be possible to cope with frames requiring more decoding effort. In typical video decoders, several frames are decoded in advance and stored into a ring buffer. Only if the buffer is full, the system will idle. Otherwise, computation time is used to decode frames in advance. In the real-time aware version of the H.264 application, a ring buffer is used as well. The maximum number of frames stored in this buffer is limited to eight. This allows for executing the decoder on all supported platforms without modifications.

In the implementation of the H.264 decoding application created in this thesis, every frame is handled by two separate tasks. Per frame, one task is responsible for decoding and one task for displaying. The execution order of all tasks is modeled via dependencies. In Figure 6.5, a dependency graph is depicted. The nodes represent the tasks. The outer (blue) nodes are decoding tasks and the inner (black) nodes are the output tasks. For simplicity reasons in Figure 6.5, a ring buffer size of three and the continuous example sequence IPPPIPPP of I- and P-frames are assumed. The red numbers denote the buffer index. For example, task A decodes an I-frame in buffer element 0 and task A' displays the decoded frame stored in buffer element 0. It is obvious that A has to be finished before A' can display the frame. Hence, a dependency edge is inserted. Each P-frame is dependent on the previously decoded frame. There is also a dependency between the draw tasks. The next draw task has to wait until the previous draw task finishes execution. Otherwise, two frames will be drawn on the screen. The last kind of dependency is shown with red dashed arrows. This dependency is required for each I-frame to avoid possible overwriting of the reference frame used by another P-frame. For example, without

the dependency edge from task B to task D, D can be scheduled before B. However, since D operates on buffer element 0, D would override the frame which B requires as reference. Consequently, each task decoding an I-frame is dependent on the task decoding the  $n - 1$ 'st last buffer element relative to the current position, where  $n$  is the buffer size. There is only one exception. If the  $n - 1$ 'st last buffer element represents an I-frame, the dependency can be omitted. This is the case for the dependency edge from D to F.

In Figure 6.5, the dependencies seem to be circular. This is not the case. The tasks are only depicted as a circle to show all dependencies for the assumed frame sequence. In the real implementation, only seven tasks exist in parallel. These are three decoding tasks and four draw tasks. In the considered example, the tasks A, A', B, B', C, and C' are created at startup and execution begins with task A. After A finishes, task A' can display the frame stored in buffer position 0. If A' finishes drawing, new tasks have to be created to refill the buffer at position 0. Therefore, A' creates the tasks D and D'. To get all necessary parameters, A' already executes part one of the frame decoding process. Thereafter, the execution time for task D can be estimated. The execution time for task D' is composed of the time required for drawing the frame and the time which will be required to execute part one of the decoding. Both execution times can be estimated by considering the frame resolution only. The new release time  $r_{D'}$  and deadline  $d_{D'}$  of task D are calculated as  $r_{D'} = r_{A'} + n \cdot \text{fps}^{-1}$  and  $r_{D'} = d_{D'} + \text{fps}^{-1}$ , respectively. The release time of the new decoding task D is set to "now". Specifying a deadline for D can be omitted since it is automatically inherited by setting the dependencies.

The previously described procedure is repeated until the end of the video stream every time a draw task finishes the frame output. This means two new tasks are created. One task as replacement for the current draw task and one task responsible to execute part two of the frame decoding. All tasks are assigned with the appropriate real-time parameters, either directly or indirectly via dependencies. The complete system is scheduled with EDF\*Henn [Hen78; CSB90].

### 6.3.3 RTEMS Modifications

To implement the real-time aware H.264 decoding application on top of RTEMS, it is necessary to add some mandatory features. In RTEMS, task release times, dependencies, and execution times ( $\text{WCET}_{\text{EST}}$ ) are not supported. Deadlines are only implemented for some schedulers and are only usable via specialized services.

To improve the RTEMS support for release times, execution times, and deadlines, the function `rtems_task_create_realtime` is introduced. In Listing 6.1, its declaration is shown. The first six parameters are equal to the standard `rtems_task_create` function. The parameter `t_activation` specifies the absolute release time of the task measured in ticks since boot. The estimated execution time is specified in `t_wcet`. To specify the deadline of the task, the parameter `t_deadline` has to be used. The deadline is specified relative to the release time. In RTEMS, task start and creation is realized by two separate functions. It is always the first step to

**Listing 6.1:** New RTEMS real-time task create API

```

1 rtems_status_code rtems_task_create_realtime(
2     rtems_name          name,
3     rtems_task_priority initial_priority,
4     size_t              stack_size,
5     rtems_mode          initial_modes,
6     rtems_attribute     attribute_set,
7     rtems_id            *id,
8     uint32_t             t_activation,
9     uint32_t             t_wcet,
10    uint32_t             t_deadline
11 );

```

**Listing 6.2:** New RTEMS task dependency API

```

1 rtems_status_code rtems_task_depend(
2     rtems_id            id,
3     rtems_id            id_dependending_on
4 );

```

create a task with the previously mentioned function. Thereafter, the task can be started by calling `rtems_task_start`. If `rtems_task_start` is called prior to the release time, the new implementation will automatically delay the start. The deadline and the `WCETEST` are attached to the RTEMS thread control block managing the task. This means the selected scheduler has to take care of the deadline.

To support task dependencies the function `rtems_task_depend` is added (cf. Listing 6.2). If the task denoted by `id` is in the ready queue or currently executing, it will be blocked immediately. Tasks are blocked until all tasks they depend on are finished. Dependencies are directly enforced by RTEMS regardless of the used scheduler.

## 6.4 Summary

Decoding H.264 videos is a very good application to evaluate the flexible error handling approach. It provides enough complexity to achieve a high and varying CPU load over a long period of time. Due to the varying CPU load, the slack time varies at well. In this highly dynamic scenario, *FAME* has to handle errors with respect to the current system condition. This is especially important when the real-time version of the H.264 video decoder application is used.



# Evaluation

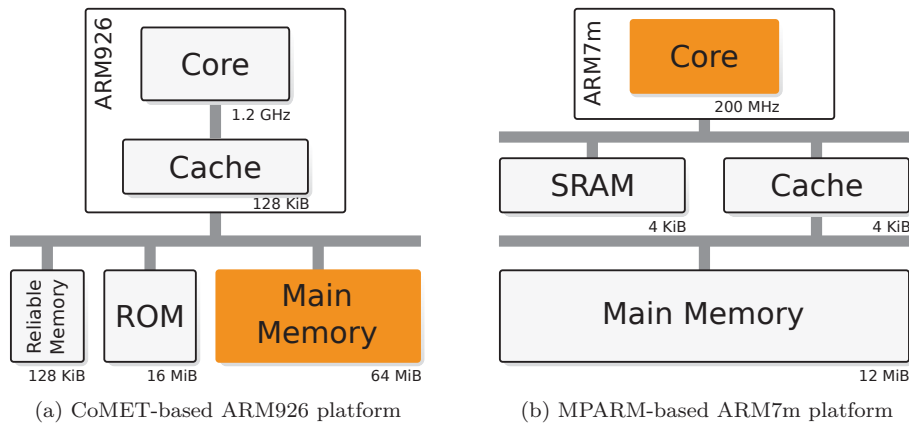
---

## Contents

<b>7.1</b>	<b>Evaluation Platforms</b>	<b>116</b>
7.1.1	CoMET-based ARM926 Platform	116
7.1.2	MPARM-based ARM7 Platform	117
<b>7.2</b>	<b>QoS Impact of Classification</b>	<b>117</b>
7.2.1	Transient Errors in Memory	118
7.2.2	Probabilistic Error Model	120
7.2.3	Summary	123
<b>7.3</b>	<b>Overhead</b>	<b>124</b>
7.3.1	Interrupt Latency	124
7.3.2	Hypercalls	125
7.3.3	Timing	126
7.3.4	Summary	127
<b>7.4</b>	<b>Checkpointing</b>	<b>128</b>
7.4.1	Encoder Type and Implementation	128
7.4.2	Block Size	130
7.4.3	Reducing the Number of Data Objects	130
7.4.4	Summary	132
<b>7.5</b>	<b>Flexible Error Handling and Transient Faults</b>	<b>132</b>
7.5.1	Naive Error Handling	133
7.5.2	Flexible Error Handling	134
7.5.3	Flexible Error Handling with Application Specific Error Correction Methods	135
7.5.4	Quality of Service Impact of Flexible Error Handling	135
7.5.5	Summary	137
<b>7.6</b>	<b>Summary of Evaluation</b>	<b>137</b>

---

In this chapter, the flexible error handling approach as well as *FAME* are evaluated. All evaluation scenarios are based on the H.264 benchmark application described in Chapter 6. The evaluation starts with a description of the used platforms in Section 7.1. Afterwards, in Section 7.2, the impact of ignoring errors is shown for different error models. This section also proves that the classification approach – which is one of the basic principles of flexible error handling – can be applied



**Figure 7.1:** Used evaluation platforms

to different error models. Embedded systems are in the focus of this dissertation. Since the available resources in such systems are scarce, the runtime overhead of the flexible error handling approach should be as small as possible. The overhead imposed by *FAME* is measured in Section 7.3.

*Checkpoint-and-recovery* is the default method in *FAME* to handle errors affecting data classified as reliable. In Section 7.4, the creation and the recovery of checkpoints are evaluated. Finally, in Section 7.5, the real-time behavior under the influence of transient faults is evaluated.

## 7.1 Evaluation Platforms

The flexible error handling approach is evaluated on two platforms. The first platform is based on CoMET [Syn14] and is used to evaluate *FAME* with the transient memory fault model. MPARAM [BBB+05] is the second evaluation platform. It is used for evaluating the flexible error handling approach with the probabilistic error model.

### 7.1.1 CoMET-based ARM926 Platform

The main evaluation platform used in this dissertation is based on the Synopsys CoMET/METeor Simulator [Syn14]. CoMET/METeor provides fast and accurate models enabling hardware and software co-design. The setup used in this dissertation is depicted in Figure 7.1a. The processor core configured in this platform is an ARM926 to match the demonstrator platform shown in Figure 3.3 on page 38. The simulated platform is equipped with 64 MiB main memory, 16 MiB ROM and 128 KiB reliable RAM. The memory bus is clocked at 400 MHz. The main memory is based on a customized module implementing the transient error model described in Section 3.3.1. This model does not inject faults continuously. Instead, faults will be injected when the memory is accessed since this is the relevant point in time



where a fault can be noticed. To determine the amount of faults which have to be injected between two consecutive memory accesses, a Poisson distribution with configurable parameter  $\lambda$  is used. Faults are randomly injected and are equally distributed over the memory. Hence, the location of the access has no influence on the fault distribution. The fault detection functionality of the memory module is enabled. Thus, it is checked on access whether or not the requested memory cell is affected by an error. If an error is detected, a Fast Interrupt Request (FIQ) will be triggered to notify the processor.

### 7.1.2 MPARM-based ARM7 Platform

The MPARM-based platform is used to show the feasibility of the flexible error handling approach also for error models apart from memory. MPARM [BBB+05] was developed at the University of Bologna. It is a multi-processor cycle accurate architectural simulator. Several SWARM [Dal03] (SoftWare ARM) processor cores implemented in C++ are connected by an AMBA bus implemented in System C. The SWARM cores simulate ARM7m processors. SRAM based memory and caches are connected with the ARM core by an internal bus. The SRAM is not considered in this thesis. Except for the size of the main memory, the default configuration of MPARM is used. The amount of main memory had to be increased from 1 MiB to 16 MiB to allow for execution of the H.264 benchmark application. MPARM is configured to use one CPU with the default frequency of 200 MHz. An overview of the used configuration is depicted in Figure 7.1b.

In this dissertation, MPARM is used to implement a probabilistic error model. Therefore, a probabilistic version of a ripple carry adder and a Wallace tree multiplier are added to SWARM. These probabilistic components are used by four new instructions. All other instructions continue to use deterministic components only. The new instructions are addition (`padd`), subtraction (`psub`), and reverse subtraction (`prsb`) using the PRCA (cf. Section 3.4.2.1), as well as multiplication (`pmul`) which uses the PWTM (cf. Section 3.4.2.2).

## 7.2 QoS Impact of Classification

The classification of data objects regarding their susceptibility to faults is one of the basic principles of the flexible error handling approach. Classification consists of two parts: reliability annotations and possible error correction methods. This section is dedicated to the former part. The usage of different error correction methods is evaluated in Section 7.5.

The *REPAIR* compiler supports two error impact classes. If an error affecting an object can lead to severe consequences, like, e.g., application crashes, the object will belong to the *high impact* class. Hence, such an object has to be annotated as **reliable**. All other objects are mapped to the *low impact* class and are annotated as **unreliable**. In this section, a static mapping of error correction methods to error classes is defined. Errors affecting data objects annotated as reliable are corrected

Video resolution	Memory size of reliable data	Memory size of unreliable data
176 x 144	92,908 bytes (55 %)	76,184 bytes (45 %)
352 x 288	228,784 bytes (43 %)	304,280 bytes (57 %)
1280 x 720	1,623,104 bytes (37 %)	2,764,952 bytes (63 %)

**Table 7.1:** Ratio of reliable to unreliable memory (taken from [SHM+13])

by restoring the last checkpoint. If an object annotated as unreliable is affected, the error will be ignored.

The purpose of the evaluation in this section is to show the applicability of the error classification by performing a qualitative analysis and to show the impact on the quality of service by a quantitative analysis. Both analyses are performed for the transient memory and probabilistic error model.

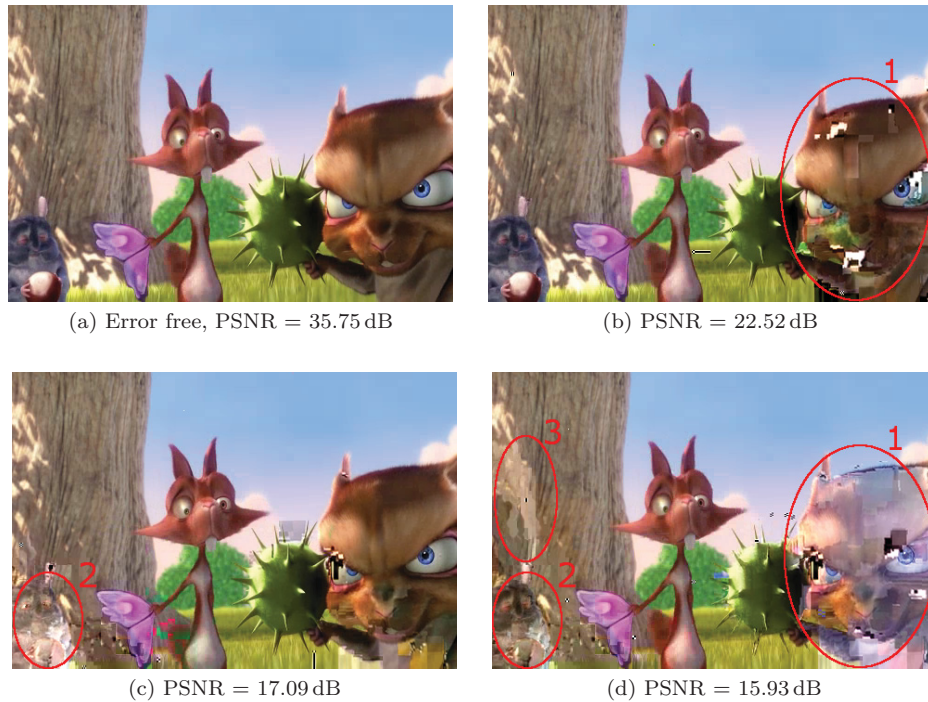
## 7.2.1 Transient Errors in Memory

To evaluate the flexible error handling approach for the transient memory error model, the CoMET-based platform described in Section 7.1.1 is used. The executed software load is the real-time version of the H.264 video decoder (cf. Section 6.3) executed on top of *FAME*. The qualitative and quantitative analyses follow in the next two subsections.

### 7.2.1.1 Qualitative Analysis: Applicability of Unreliable Memory

In the qualitative analysis it is investigated whether a significant share of the data objects of the H.264 decoder can be annotated as unreliable. The flexible error handling approach will only be applicable if this is the case. By applying the static code analysis of the *REPAIR* compiler and manual source code annotations, it is possible to identify unreliable data objects. Hence, errors affecting such objects can be safely ignored without severe consequences. The size of the memory that is necessary to store reliable respectively unreliable data objects is depicted in Table 7.1. The numbers in this table are estimations from a high-level source code analysis of *libh264* only. To determine the sizes of the memory, the sizes of individual data objects in the C source code were summed up. Of course, the values are not accurate. However, they provide a rough idea of the relation between video resolution and amount of reliable versus unreliable memory.

To determine the real sizes, the complete H.264 benchmark is measured during runtime. The tested video had a resolution of 352x288 pixels. The resulting amounts of allocated reliable and unreliable memory are 2,277,024 bytes (38%) and 3,649,536 bytes (62%), respectively. This measurement includes all writable data. These are the heap, .data segment, and .bss segment. The complete stack is considered as reliable since *FAME* does not support the unreliable attribute for data objects residing on the stack. The measured memory sizes are an order of magnitude larger than the sizes estimated by the high-level source code analysis. This result



**Figure 7.2:** Different impacts of transient memory faults

is expected since it includes the whole operating system and the application which uses a ring buffer to decode up to eight frames in advance.

This evaluation clearly shows the applicability of the flexible error handling approach since enough data objects exist which can be classified as unreliable.

### 7.2.1.2 Quantitative Evaluation: Signal-to-Noise Ratio

The purpose of the quantitative evaluation will be to reveal the impact on the quality of service if all errors affecting unreliable data are ignored. In Figure 7.2, the same frame is shown with different error outcomes. For comparison, the error free decoded frame is depicted in (a). As metric for the quality of service metric PSNR is used. To calculate the PSNR value of a frame, the original image of the frame is used as reference. Due to the fact that the used constrained baseline profile only supports lossy compression, the error free decoded frame in (a) does not exactly match the source image. Depending on the error rate and the affected objects, different disturbances in the video output can be observed. For example, in (b) some artifacts on the squirrel (1) are visible. In (c) the chinchilla (2) is not recognizable anymore. The last picture (d) has visible disturbances on the chinchilla (2) and the squirrel (1). Furthermore, the shadows of the leaves on the tree trunk are blurred (3). These disturbances clearly result in a low PSNR value.

A measurement of the PSNR values dependent on the error rate is shown in Table 7.2. For each error rate, the table depicts the average PSNR values of all

Effective Error Rate [ $s^{-1}$ ]	Avg. PSNR [dB]	Min PSNR [dB]
–	36.20	30.07
0.16	36.19	25.59
1.77	36.18	24.78
48.04	28.82	8.17

**Table 7.2:** QoS dependent on the effective error rate (flexible error handling + minimal checkpointing)

	Video	Resolution	# Frames
A	TV series	$176 \times 144$	320
B	Computer animated movie	$192 \times 144$	100
C	Music video	$176 \times 144$	320
D	TV movie	$320 \times 144$	320
E	Computer animation	$176 \times 144$	320

**Table 7.3:** Analyzed videos

frames and the minimal observed PSNR value. In the case of the lower error rates of 0.16 and 1.77 errors per second, the QoS of the error-free run is nearly reached. If the highest error rate is used, the average PSNR is 20% lower. However, the minimal observed PSNR value is getting worse. The applied error correction methods also have an influence on the QoS. A detailed evaluation considering the correction methods is presented in Section 7.5.4

The conclusion for the flexible error handling approach is that the average disturbance of the video is tolerable. However, periods of time exists where the video is not recognizable anymore.

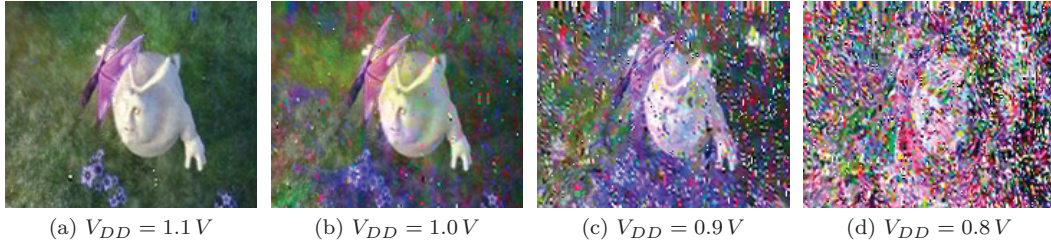
## 7.2.2 Probabilistic Error Model

In the transient memory fault model evaluated in the previous section, all errors affecting unreliable objects are ignored. This implied that the complete runtime subsystem of *FAME* is required to decide during execution whether or not an object is unreliable. The probabilistic error model has the advantage that this decision can already be taken during compile-time. As soon as one operand of an operation is annotated as unreliable, the whole operation allows for unreliable execution. Hence, in the probabilistic error model, the annotations indicate if deterministic behavior is required or not. Only in the latter case the unreliable annotation is allowed. To avoid fatal consequences and unintentional propagation of errors to reliable data objects, the use of unreliable annotations is restricted by the *REPAIR* compiler (cf. Section 5.1).

To perform quantitative and qualitative analyses of the flexible error handling approach using the probabilistic error model, the MPARM-based platform described in Section 7.1.2 is used. The simple version of the H.264 decoder (cf. Section 6.2) is used as benchmark application. The input videos are depicted in Table 7.3.

	add	sub	rsb	mul	overall
Executed using PRCA/PWTM	18.59 %	18.60 %	43.01 %	76.27 %	<b>13.36 %</b>

**Table 7.4:** Instructions executed using probabilistic components



**Figure 7.3:** Quality of service degradation due to lowered supply voltages using UVOS

### 7.2.2.1 Qualitative Analysis: Applicability of Probabilistic Arithmetics

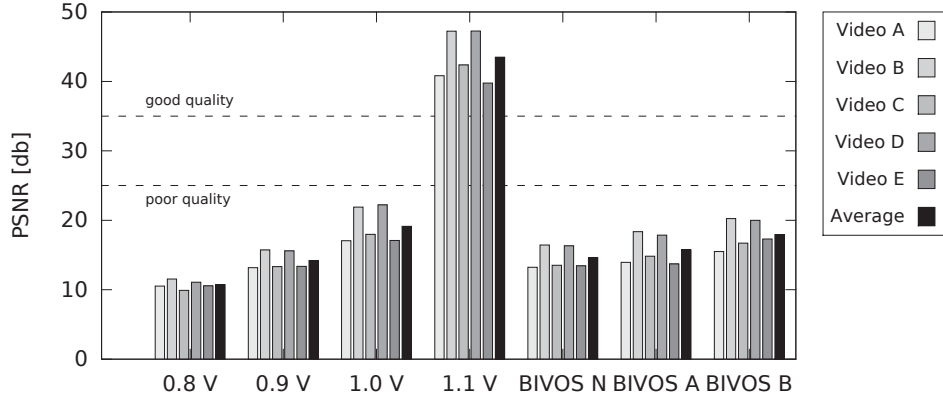
As a first step, it is evaluated whether a significant percentage of the program instructions can be safely executed using the probabilistic adder or multiplier. Using MPARM, the number of instructions executed are counted to determine which of these could tolerate an imprecise result. Table 7.4 shows the average relative frequencies for all videos. Here, 76.27% of the mul instructions means that about three quarters of all multiplications were computed using probabilistic components, whereas all other multiplications were computed using the deterministic ALU. In total, 13.36% of all operations were executed on probabilistic arithmetic components. This is a significant result, since this percentage considers all operations executed by the ALU including logic and compare instructions.

In all experiments, no application crash could be observed. This result is expected since the static code analysis techniques of *REPAIR* ensure that operations requiring reliable computations are executed deterministically.

### 7.2.2.2 Quantitative Evaluation: Signal-to-Noise Ratio

In the last subsection, the applicability of the flexible error handling approach under the probabilistic error model was shown. A significant fraction of the H.264 decoder was executed on a probabilistic processor unit. In this subsection, the impact on the output quality is evaluated. The quality is evaluated using peak signal-to-noise ratio (PSNR) values for each decoded frame. The calculated PSNR values are based on the correctly decoded frames. A higher PSNR value indicates better quality. A perfectly decoded video has a PSNR value of infinity. In contrast to the previous section, the baseline for the calculation of PSNR values are changed. The reason for this is that the tools used to evaluate MPARM runs only support comparisons of H.264 encoded videos.

Figure 7.3 shows results for test video E of Table 7.3 using different voltages. The nominal supply voltage is 1.2V and the noise level is up to  $\pm 0.12V$  relative

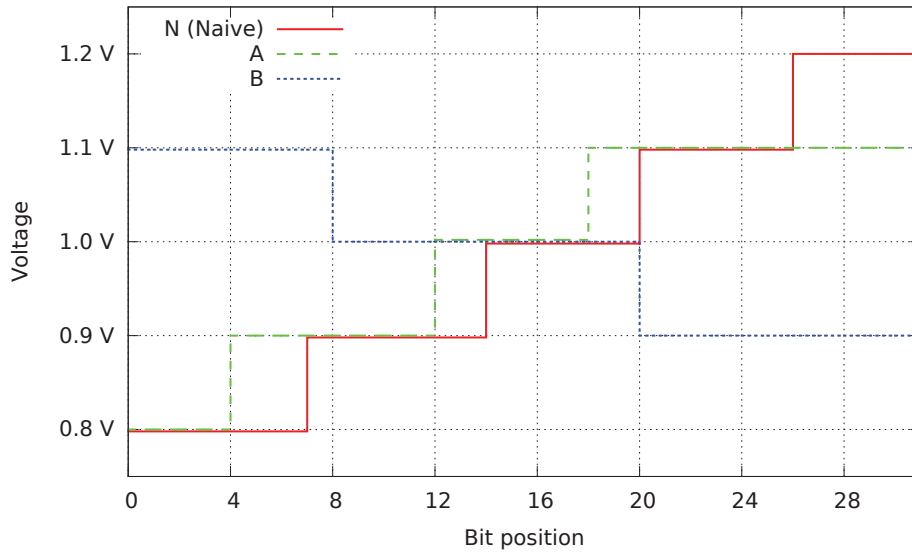


**Figure 7.4:** PSNR values for different videos

to the supply voltage. In this evaluation, a *PSNR* value of at least 35 dB is defined as good quality. In contrast, a value of less than 25 dB indicates very poor quality. However, the interpretation of video quality and *PSNR* values depend on the perception of the viewer and the output quality requirements. The values used here are commonly accepted for consumer video applications (e.g. [LC07]). If the probabilistic components are operated close to the nominal voltages, only minor disturbances can be noticed (Figure 7.3a). However, when uniformly lowering the supply voltage, noise effects are increasingly visible, leading to garbled pictures at 0.9 V (c) and 0.8 V (d). Using UVOS, *PSNR* values for 1.0 V are already below the acceptable limit of 25 dB. Detailed *PSNR* values are shown in Figure 7.4. It can be easily seen that a higher supply voltage yields better quality. However, only the 1.1 V UVOS version yields good quality.

Due to the low QoS results achieved using UVOS, it will be interesting to analyze if employing BIVOS schemes provide better quality. Since the 1.0 V version is very close to the 25 dB margin, the goal is to improve the QoS by using a BIVOS scheme with the energy budget equivalent to the 1.0 V UVOS scheme. The UVOS and BIVOS energy consumptions are calculated with the energy model used by MPARM based on [PLS01]. Three BIVOS models are considered in this dissertation. These models are shown in Figure 7.5 and the measured *PSNR* values are depicted in Figure 7.4. Naive BIVOS (N) supplies less significant bits with a low voltage and the most significant bits with the nominal supply voltage. This BIVOS scheme also achieved only a low *PSNR* value.

Since 1.1 V UVOS yields good *PSNR* values, another BIVOS scheme, referred to as A, was constructed. BIVOS A reduces the supply voltage of the most significant bits to 1.1 V. The gained energy savings are spent to increase the supply voltage of less significant bits. As shown in Figure 7.4, this version achieves improved *PSNR* values compared to BIVOS N using the same amount of energy as the 1.0 V UVOS



**Figure 7.5:** Used BIVOS setups

scheme. However, the PSNR value is still quite poor. Analyzing the H.264 code revealed that most of the code only uses less significant bits of the 32 bit probabilistic adders and multipliers. Consequently, BIVOS scheme B was constructed, which supplies the least significant bits with a higher voltage than the most significant bits. Again, the same energy budget is used. In fact, the PSNR values of this version are better compared to the other BIVOS schemes. However, they are still worse than the 1.0 V UVOS scheme.

Using BIVOS, it is not possible to increase the output quality under identical energy budgets. One reason for this phenomenon can be revealed by considering the H.264 specification. If 32-bit integers are transformed into an 8-bit value to be stored in the frame buffer, clipping code will be used which implements saturation to a maximum value of 255. For BIVOS scheme B this implies that if, e.g., bit eleven flips from '0' to '1', the precision of the less significant eight bits are irrelevant due to wrong clipping. In the opposite case, e.g., using BIVOS scheme A, correct clipping is performed, but the least significant eight bits are too imprecise. For operations like the selection of luminance and chrominance values for macro blocks or larger frame parts, this effect is even worse.

### 7.2.3 Summary

In general, the flexible error handing approach is applicable to both the transient memory and the probabilistic error model. However, as soon as the error rate increases respectively the supply voltage is lowered, huge disturbances are noticeable. Increasing the QoS in the case of the transient memory error model is possible by applying error correction also to unreliable annotated data objects.

Interrupt	Average Execution Time [ns]
RTEMS Native	182.6
ECI/IRQ	1365.3
ECI/Timer	1974.8
ECI/Fault	3408.5

**Table 7.5:** Interrupt measurement

To also increase the QoS if the probabilistic error model is used, the static code analysis provided by *REPAIR* has to be extended. The analysis has to consider the number of unused bits of probabilistic variables. Using this information under a BIVOS distribution, additions and subtractions can be performed by shifting the parameters by the unused number of bits minus one to the left. For multiplications, the result may in general require twice as many bits as the largest operand. Hence, multiplications are expected to have a lowered potential to improve the QoS. However, due to the shifting, errors in the lower bits only have minor influence on the QoS.

## 7.3 Overhead

This section is dedicated to the overhead imposed by the virtualization of RTEMS. The overhead is evaluated in terms of IRQ latency, hypercall processing time, and runtime impact. For all measurements, the CoMET-based ARM926 platform is used. The real-time version of the H.264 decoder is executed. In each experiment, 600 frames with a resolution of 480x320 pixels are decoded with 10 fps. If not stated otherwise, error injection will be turned off. To measure execution times, intrinsics of CoMET are used.

### 7.3.1 Interrupt Latency

Interrupt latency is the time that elapses between interrupt generation and servicing the interrupt. In Table 7.5, the measured interrupt latencies of native RTEMS and virtualized RTEMS are shown. To determine the latency, the time measurement is started directly after the CPU switches to the IRQ mode. The measurement is stopped right before invoking the corresponding service routine.

If virtualization is used, interrupts are reported by the *FAME* microvisor using the *event callback interface (ECI)* (cf. Section 5.3.2.4). Depending on the type of the ECI message different latencies can be observed. In RTEMS, the latencies for different types of interrupts are the same since all interrupts are handled by the same basic code which later calls the appropriate handlers. Compared to the IRQ handling of RTEMS, the microvisor is 7.5 times slower in routing second class interrupts<sup>1</sup>. The overhead is the accumulation of additional mode switches and context save operations the microvisor has to perform.

<sup>1</sup>Second class interrupts are interrupts not handled directly by the microvisor.



Hypercall	Avg. Exec. Time [ns]	Std. Deviation	Share
fame_hypercall_object_unsubscribe_all	757,193	51,759	75.08 %
fame_hypercall_object_subscribe_multiple	32,260	14,685	9.80 %
fame_hypercall_irq_disable	1,360	49	7.04 %
fame_hypercall_irq_enable	728	19	4.21 %
fame_hypercall_object_subscribe	4,908	606	1.27 %
fame_hypercall_object_add	5,323	2,160	0.68 %
fame_hypercall_cpsr_set	824	56	0.68 %
fame_hypercall_object_remove	4,623	283	0.55 %
fame_hypercall_object_unsubscribe	4,337	412	0.50 %

**Table 7.6:** Hypercall measurement

In the case of timer IRQs, the *FAME* microvisor has to update the system clock and has to process watchdog operations which impose additional overhead. If an error occurs, the microvisor will start low-level error handling (cf. Section 5.6.1). At the end of low-level processing, an ECI fault message is generated. However, before such a message can be sent, an entry specifying the error has to be created and the microvisor has to check whether handling the error in the guest OS is possible. These operations delay the delivery of the ECI message. Error injection was enabled to measure the execution time of ECI fault messages.

### 7.3.2 Hypercalls

Hypercalls are very important for the para-virtualization concept realized in *FAME*. They are used whenever the guest operating system requires a service of the microvisor. Each hypercall represents a specific service. The *checkpoint-and-recovery* service is separately evaluated in Section 7.4. Compared to all other services, this is the most compute intensive service. The idle hypercall represents a special service. It can be issued by RTEMS if the ready task queue is empty. For ARM926-based systems, this hypercall stops the CPU until the occurrence of an external event, like an interrupt.

Table 7.6 shows the measured processing time of hypercalls with a fraction of at least 0.50% of the accumulated total hypercall execution time. The idle and checkpoint hypercalls are omitted. For the measurement, a timer is started before performing the hypercall operation (svc instruction on ARM). After the return from the svc instruction the timer is stopped. The hypercalls are sorted by their accumulated share of the overall time spent in hypercall processing. The share is depicted in the last column. By ignoring the idle and checkpoint hypercalls, the depicted hypercalls account for over 99% of the total runtime spent in hypercalls.

The second column of Table 7.6 shows the average execution time of a hypercall and the third column shows the corresponding standard deviation. The hypercall `fame_hypercall_object_unsubscribe_all` has a huge share of the execution time of the microvisor. This hypercall is used whenever a task is destroyed to clean up its subscriptions. Therefore, the microvisor has to traverse the complete object tree,

	Execution Time [ms]	Idle Time [ms]	Avg. CPU Load [%]
Native	60,611	14,750	75.7
Virtualized	60,614	19,196	68.3

**Table 7.7:** Execution time comparison

since the subscribers of an object are stored in the object data structure. A reverse mapping is not provided to reduce the amount of required reliable memory. Due to the fact that the object tree is being changed with every new allocation or free operation, the hypercall execution time has a high variation. This is also true for all other hypercalls which perform operations on the object tree.

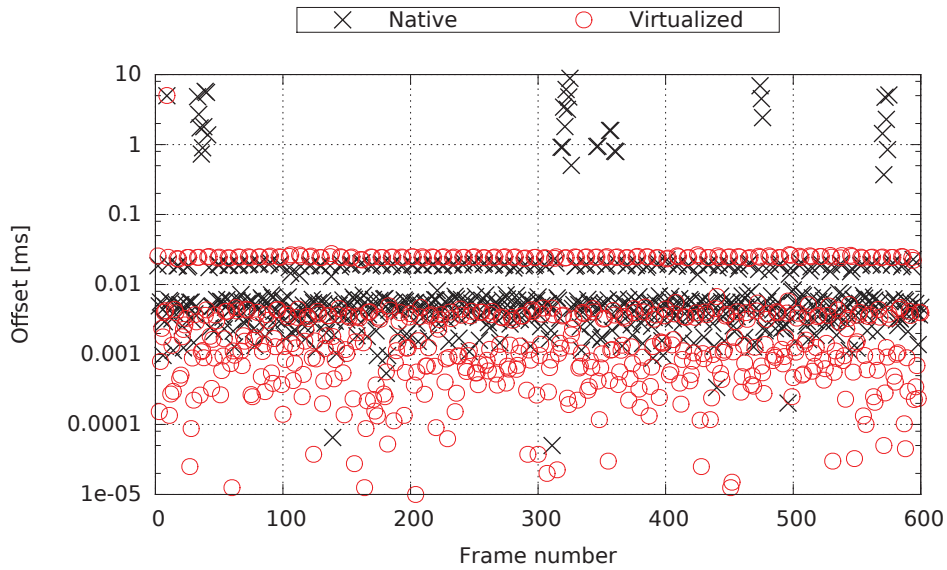
The hypercalls for enabling and disabling IRQs are frequently used by RTEMS to secure critical sections in the RTEMS super core. The variation of the execution time of these hypercalls is very low in contrast to hypercalls used for object management and the subscriber model. Virtually disabling IRQs takes more time than enabling the IRQs. The time difference is mainly caused by an additional parameter check required in the IRQ disable hypercall. This parameter is a pointer used to return the IRQ state to the caller. Since this pointer is an address generated outside the microvisor, its validity has to be checked. Validation includes tests for alignment as well as for write permission. The latter check requires page table lookups which is the main performance impact.

Overall, performance penalties are observable whenever input parameters have to be validated or dynamic data structures are accessed. However, the former case cannot be avoided, otherwise the RCB invariant would be violated.

### 7.3.3 Timing

Virtualization can have an impact on the timing behavior of the application. In this subsection, the influence of the *FAME* microvisor on the H.264 video decoder benchmark is investigated. A comparison of the execution time of the native and virtualized versions is shown in Table 7.7. The difference of 3 ms in the execution time is the initialization overhead of the microvisor and *FAMERE*. The measured idle time of the virtualization version is higher than the idle time of the native version. Consequently, the CPU load of the virtualized version is lower. At the time of the development of *FAME*, this was completely unexpected. However, other research groups also achieved higher performance of virtualized benchmarks compared to their native versions. For example, in [HSH+08], the authors measured higher throughput in certain configurations. An evaluation of the Xen hypervisor is shown in [YWG+06]. The authors also showed increased performance for the Linpack [Don87] LU decomposition benchmark. Hence, it is not completely unusual that a virtualized system is faster than the corresponding native version. However, this clearly shows the high performance of the hypercall and ECI implementation of the *FAME* microvisor.

In this dissertation, the focus is based on embedded real-time systems. Hence,



**Figure 7.6:** Jitter comparison

an important evaluation aspect is the influence of the virtualization on the jitter. To measure the jitter, the H.264 decoder application was slightly modified. Unmodified, the decoder clears the complete frame buffer periodically to wipe erroneous pixels outside the video display area of the screen. This clearing leads to remarkable jitter and hence is disabled in this evaluation to better visualize the basic jitter. Figure 7.6 shows the jitter for the native and the virtualized version. The decoder is configured to 10 fps. Hence, every 100 ms a new frame has to be displayed.

In this evaluation, jitter is defined as the absolute difference of the expected point in time where the display refresh occurs and the measured point in time of the display refresh. Since the decoder is configured to 10 fps, every 100 ms a new frame has to be displayed. The absolute difference to these 100 ms is depicted in Figure 7.6. The figure shows that the jitter of the virtualized and the native version is similar. If the average jitter is considered, the jitter of the virtualized version will be lower. This clearly proves a good design of *FAME* with respect to real-time.

The jitter reaching the 5 ms offset is in conjunction with the system timer which is configured to 5 ms. In the case of such high jitter, the draw task was scheduled one tick earlier or later relative to the last displayed frame. In the virtualized version, this phenomenon is only observable one time. The native version is affected more often.

### 7.3.4 Summary

In this dissertation, one of the goals is to design a virtualization solution which is applicable to embedded real-time systems. The current section clearly proved that this goal is reached. The overhead imposed by *FAME* is low and the real-time

behavior of the used test application is not disturbed. Moreover, it can be measured that the performance increases and the jitter decreases if the virtualized version is used. In the next section, *checkpoint-and-recovery* is evaluated.

## 7.4 Checkpointing

*Checkpoint-and-recovery* is an important error correction strategy in the flexible error handling approach. It is used whenever an error has to be corrected and no specialized error correction method is available. This is the case for all objects which are not explicitly annotated with applicable correction methods. Since checkpoints play an important role, they have to be highly optimized to minimize the impact on the real-time behavior of the application. The runtime of checkpoint creation is determined by the size of the data which have to be stored in the checkpoint and the selected block encoder. The runtime for recovering a checkpoint depends on the block encoder and the checkpoint size as well.

As described in Section 5.3.5, a checkpoint created by *FAME* consists of several chunks. The chunks consist of blocks of equal size. To copy data to a block, a block decoder can be selected. Implemented block encoders are TMR, Reed-Solomon (RS) code, LDPC (Low-Density Parity-Check) code, and PLAIN<sup>2</sup> encoding. RS and LDPC take orders of magnitude longer than TMR and PLAIN. Hence, only the latter two block encoders are considered in this dissertation.

To measure the performance of the two encoders dependent on the block size, the CoMET-based ARM926 platform is used. Fault injection is turned off. In the measurement setup, 2.8 MiB of data have been arbitrarily selected for storing in a checkpoint. The execution times of the checkpoint create and checkpoint recover procedure is measured using CoMET intrinsics, called TSPI. These intrinsics allow for precise measurements with an overhead of less than ten processor cycles. Hence, the measurement overhead is negligible. In Figure 7.7 and Figure 7.8, the results of the measurements of the creation and recovery procedures are shown, respectively. All results include the whole procedure starting at the point where the application calls the appropriate services.

### 7.4.1 Encoder Type and Implementation

In Figure 7.7 and Figure 7.8, execution times for different versions of the encoders are shown. The dotted curves denote an implementation in C code where a simple for-loop is responsible for encoding the data. For the dashed curves, this loop was manually unrolled to eliminate compare and branch instructions necessary for the loop implementation. A manually written assembler-based version is shown with solid lines. This version is also unrolled and additionally uses load multiple and store multiple instructions available on ARM processors to enforce burst transfers on the data bus. Using such instructions also minimizes the number of instruc-

<sup>2</sup>PLAIN implements a simple memcpy operation

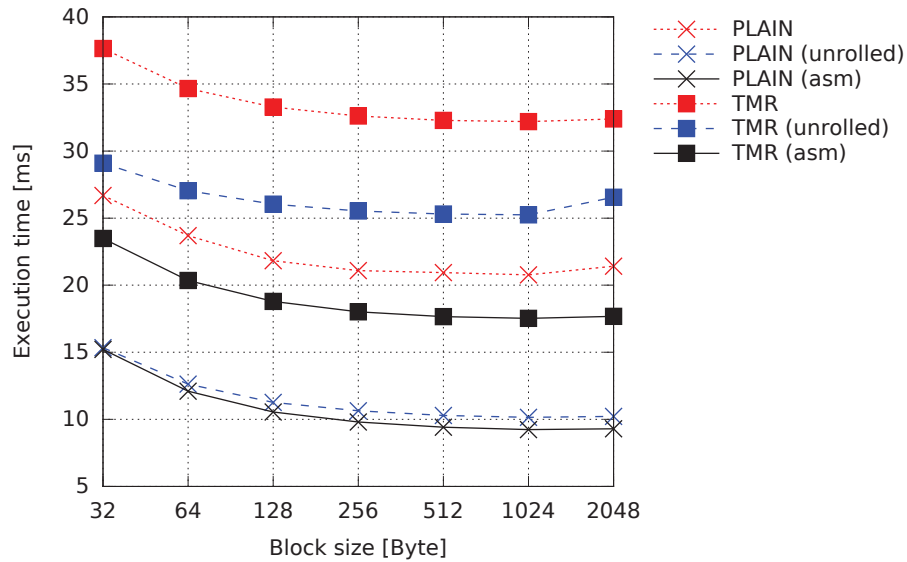


Figure 7.7: Checkpoint creation time dependent on the used block size and encoder

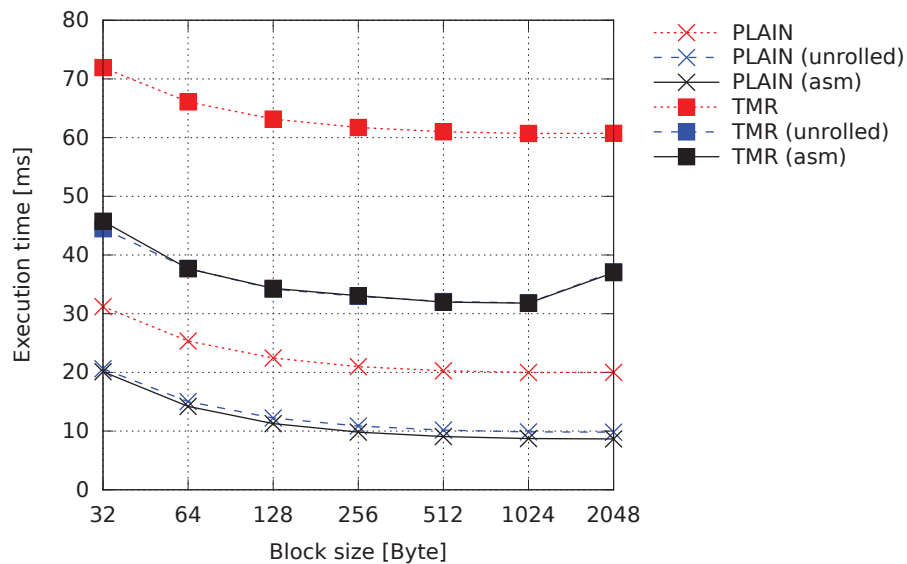


Figure 7.8: Checkpoint recover time dependent on the used block size and encoder

tions to execute, since one load multiple or store multiple instruction can load or store several words in one instruction, respectively. Due to these optimizations, the assembler-based encoder versions yield huge performance improvements for checkpoint creation, especially in the case of the TMR encoder. For checkpoint recovery, the performance improvement is lower.

Not surprisingly, the PLAIN encoder is faster than TMR. PLAIN is just the implementation of a simple memcopy. TMR has to store the data three times to create a checkpoint. During the recovery procedure, TMR has to load the data at least two times and a comparison of the loaded data has to be performed. If the comparison for equality fails, the third copy has to be loaded and compared too. With a majority vote, the final result is determined. However, while TMR imposes a huge overhead compared to PLAIN, TMR is able to correct errors affecting the checkpoint.

### 7.4.2 Block Size

Regarding the block size, it can be seen that larger blocks lead to a reduction of the execution time. However, if the blocks are too large, the execution time will increase. This is the case for a block size of 2KiB. With higher block sizes it is more likely that the block size is not a multiple of the chunk size anymore. In such a case, the last block is stored from a temporary location matching the block size. To create this location, the *FAME* microvisor has to copy the data first (cf. Section 5.3.5). At a certain point, this copy overhead cannot be amortized anymore by the block encoding speedup gain.

### 7.4.3 Reducing the Number of Data Objects

The most obvious parameter which influences the runtime of checkpoint creation and recovery is the size of the data which have to be stored in a checkpoint. To reduce the amount of data, the subscriber model and the reliability annotations can be used.

To evaluate the checkpoint performance the real-time version of the H.264 decoder application is used. The decoder is configured to create a checkpoint every frame. If not stated otherwise, the TMR encoder will be used to create checkpoints. In all experiments, 600 frames are decoded in total at a rate of 10 frames per second. The frame resolution is 480x320 pixels. The subscriber model is applied to all major dynamic data structures, such as frame buffers, slice buffers, and residuals. All other parts of the application as well as libraries and RTEMS are using the legacy system object support.

The results are depicted in Table 7.8. The first row shows the results without checkpointing enabled. In this configuration, the overall CPU load is 63.85%. This means in average 36.15% slack time remains for checkpointing and error recovery. In the second experiment, no optimizations are applied to reduce the amount of data. In this scenario, all allocated data of the operating system, the H.264 decoder, and

	Avg. Size of Checkpoint	CPU Load	Chkpt. t [ms]	Chkpt. Speedup
Chkpt. disabled	-	63.85 %	-	-
No optimization	25.12 MiB	111.16 %	47.78	1.00
Only subscribed obj.	8.67 MiB	81.04 %	17.36	2.75
Only reliable obj.	8.90 MiB	81.44 %	17.76	2.69
Minimal	1.79 MiB	68.34 %	4.53	10.55
Hybrid	4.03 MiB	74.48 %	10.73	4.45

**Table 7.8:** Checkpoint measurement

the microvisor are stored in the checkpoint. The resulting average checkpoint size is 26 MiB and the CPU load including checkpointing is increased to 111.16%. Hence, without optimizations the system is overloaded.

In the third row, results are shown for checkpointing with enabled subscriber model. Checkpoints contain the complete state of the microvisor, the guest OS and only live data of the application. Unused application memory is excluded using the subscriber model. For this experiment, the reliability qualifiers of the objects are ignored. As can be seen, the speedup compared to the unoptimized checkpointing is more than a factor of two. The fourth experiment considers only reliable data for checkpointing. Here, the subscriber state is ignored. That means, data is stored in the checkpoint regardless of whether it is used by any task or not. Similar to the previous experiment the speedup is still more than a factor of two. Both results look very similar. Hence, it could be supposed that the set of reliable objects and the set of subscribed objects are overlapping in huge parts. However, the next experiment will show the opposite.

The fifth row depicts the result if both, the reliability and the subscriber state, are considered in checkpointing. Hereby, only objects are checkpointed which store reliable data and are subscribed by at least one task or are system objects. With this strategy, significant savings are achieved. The checkpoint size is reduced to less than  $1/14$ -th and the creation time is speed up by a factor of more than ten. This clearly shows that there is only a subset of objects which are in the reliable subscribed or reliable weak unsubscribed state leading to the following implications; (1) not all used data are reliable, and (2) not all reliable data are used in the time interval between two checkpoints. Concerning the CPU load, using the minimal checkpointing approach the CPU load is increased by less than 5% compared to a version without checkpointing at all. Hence, on average more than 30% slack time is still available for error handling.

Considering the system state which has to be saved, it can be observed that the necessary data can be stored into a checkpoint with the average size of 1.79 MiB protected by TMR. This system state only contains information which has to be error free to guarantee the correct control flow of the application without system crashes. This leads to the drawback of the approach. Since unreliable data are not stored, a checkpoint recovery can result in a huge disturbance in the output. Subscribed unreliable data which were part of the working set at the time of the

checkpoint creation are not rolled back. Hence, the system state after checkpoint recovery does not match the state at the creation. In the case of H.264, this results in frame outputs with very high noise. To mitigate this problem a *hybrid strategy* is used (last row of Table 7.8). The *hybrid strategy* is similar to the experiment where only subscribed objects are stored. However, this time the reliability annotations are considered. All objects which are subscribed and annotated as unreliable are stored using the PLAIN encoder and the remaining subscribed objects are stored using the TMR encoder. Compared to the *minimal* version, the *hybrid* approach is only 2.4 times slower. The average remaining slack time is 25%. The QoS improvement is shown in Section 7.12.

#### 7.4.4 Summary

In *FAME*, *checkpoint-and-recovery* is highly dependent on the used block encoder and the block size. The highest performance is achieved by using the assembler-based versions of TMR and PLAIN with a block size of 1 KiB. Since all objects not annotated as unreliable have to be stored with protection, the support of TMR is mandatory. The PLAIN encoder is usable to quickly store unreliable objects.

To allow for applying *checkpoint-and-recovery* on the H.264 benchmark application, reduction of the amount of data which have to be stored in a checkpoint is necessary. Otherwise, the CPU will be overloaded. By applying the subscriber model as well as the reliability type annotations, significant improvements can be achieved.

### 7.5 Flexible Error Handling Approach under Influence of Transient Faults

In the previous sections of this evaluation chapter, single aspects of *FAME* are evaluated. To evaluate the flexible error handling approach, the complete system has to be considered. The CoMET-based ARM926 platform is used for this evaluation. To evaluate the real-time behavior under influence of transient faults, the customized CoMET memory module is configured to insert faults. For each memory access, error detection in hardware is simulated. If the processor accesses an erroneous word, an interrupt will be raised. The occurrence of faults to be injected is modeled by a Poisson distribution with configurable parameter  $\lambda$ . Five different parameters  $\lambda$  are used. Not all injected faults are visible by the application, since faults – and hence the resulting errors – are only detected when the affected memory cell is accessed. In the second column of Table 7.9, the observed average error rates (of detected faults) are depicted. The injection rates range from several errors per minute to an artificially high error rate of 48 faults per second.

As software load the real-time H.264 decoder application is used. The decoder creates a checkpoint after every displayed frame. The microvisor is configured to allow a maximum of eight restores per checkpoint. Hence, if *FAMERE* requests



$\lambda$	Effective [ $s^{-1}$ ]
1e-16	0.18
1e-15	2.44
2.5e-15	9.37
5e-15	20.92
1e-14	48.04

**Table 7.9:** Effective average error rates

to restore the same checkpoint the ninth time, the checkpoint will be dropped and the previously created checkpoint will be restored. To show the impact of the applied checkpointing strategy, two different strategies are used, namely *hybrid* and *minimal*. Using *hybrid*, all data objects which are in a subscribed state are stored in a checkpoint. However, if such an object is annotated as unreliable, the PLAIN encoder will be used. Otherwise, TMR is used to encode the object in the checkpoint. If the *minimal strategy* is used, only subscribed reliable objects are stored in the checkpoint encoded by the TMR block encoder. Both checkpoint strategies store the OS state and the microvisor state with TMR.

In all experiments, 600 frames of the Big Buck Bunny video [Bbb] are decoded at a rate of 10 frames per second. The frame resolution is fixed to 480x320 pixels. Although resolution and frame rate seem quite low, this setup leads to a CPU utilization of more than 65%, since decoding H.264 in software is slow. However, higher resolutions and frame rates will be possible if more computing power is available.

By using CoMET with disabled fault injection, the execution of one decoder run takes 20 minutes on an Intel<sup>®</sup>Xeon<sup>®</sup>E5630 CPU clocked at 2.53 GHz. Every run which takes longer than 2 hours is automatically terminated, since such a run is very likely to violate any real-time constraint due to numerous checkpoint restores. If a system reset is necessary since no checkpoint is available that can be recovered, the run is also terminated. Each of the following experiments is executed 100 times for each error rate.

### 7.5.1 Naive Error Handling

In this scenario, the flexible error handling approach is turned off. Thus, no classification information of errors is available. Consequently, each occurring error will be handled alike by immediately recovering a checkpoint. Theoretically, this setup would be not feasible for *naive error handling* if the *minimal checkpointing strategy* is used. If an error affects unreliable data, a checkpoint will be restored although the affected data is not part of the checkpoint. This behavior is equal to ignoring the error and performing checkpoint recovery. However, this setup is chosen to set a baseline for the comparison with the flexible approach.

Table 7.10 depicts the average number of missed deadlines and Table 7.11 shows the average time by which a deadline is missed. In the second column, the results for *naive error handling* using the *hybrid checkpointing strategy* are shown. For

$\lambda$	Naive Error Handling		Flexible Error Handling		Flexible Handling App. Specific	
	(Hybrid)	(Minimal)	(Hybrid)	(Minimal)	(Hybrid)	(Minimal)
1e-16	0.01	0.00	0.01	0.00	0.00	0.00
1e-15	1,202.91	3.80	21.59	0.53	23.63	0.59
2.5e-15	2,051.94	660.70	817.80	42.54	829.53	22.89
5e-15	-	1,727.29	1,590.73	728.56	1,537.78	646.41
1e-14	-	-	-	1,955.78	2,452.54	1,917.27

Table 7.10: Average deadline miss count

$\lambda$	Naive Error Handling		Flexible Error Handling		Flexible Handling App. Specific	
	(Hybrid)	(Minimal)	(Hybrid)	(Minimal)	(Hybrid)	(Minimal)
1e-16	0.35	0.00	0.20	0.00	0.00	0.00
1e-15	6,904.38	14.91	334.46	7.83	559.35	7.58
2.5e-15	18,281.37	4,678.06	4,469.62	235.42	4,898.65	121.10
5e-15	-	11,356.24	11,144.60	3,604.88	11,681.34	4,663.03
1e-14	-	-	-	10,785.46	33,863.69	9,308.65

Table 7.11: Average time period by which deadlines are missed in milliseconds

the lowest error rate, only a few deadline misses are observable. If the error rate increases by an order of magnitude, a significant high number of deadline misses occurs. The time by which deadlines are missed is increased alike. Considering the two highest error rates, it can be noticed that no experiment run terminates within the two hour time limit.

By reducing the amount of data which have to be stored in a checkpoint (*minimal strategy*), significant reductions are achieved (cf. Table 7.10 and Table 7.11 columns three). For example, reductions of 99.7% can be observed for the 2.44 errors per second rate.

## 7.5.2 Flexible Error Handling

In columns four and five of Table 7.10 and Table 7.11, the measured results for the flexible error handling approach are depicted. All errors affecting data classified as unreliable or data which are not subscribed are ignored. For the remaining errors *checkpoint-and-recovery* is used.

The flexible error handling approach reduces the number of deadline misses by up to 98% and 94% using the *hybrid* and *minimal strategy*, respectively. The average reduction is 74% respectively 68%. The time by which a deadline is missed is also reduced (up to 95% for both checkpointing strategies). On average, reductions of 74% respectively 76% are achieved. However, most importantly, flexible error handling allows for coping with much higher error rates. By using the flexible error handling approach, the *hybrid checkpointing strategy* becomes feasible. Still, the number of missed deadlines is, unfortunately, very high and the system cannot cope with the highest error rate and *hybrid* checkpointing.

### 7.5.3 Flexible Error Handling with Application Specific Error Correction Methods

In the last setup depicted in columns six and seven, the annotations shown in Listing 4.2 and 4.3 are used to enable the application specific error correction method `MoCompDefaultValue`. This method is able to transfer a corrupted motion vector into a valid state. In H.264, motion vectors are used to shift a macro block to a new location within a frame. Motion vectors are hence well suited to encode movements in the video. However, if a motion vector is corrupted, it can happen that the corresponding macro block gets shifted out of the frame. Hereby, other memory will be overwritten. Consequently, motion vectors have to be reliable. Anyway, if a motion vector only moves the corresponding macro block to a location inside the frame, no fatal consequences will happen. Therefore, `MoCompDefaultValue` provides a valid correction method by just setting the motion vector to zero. This corresponds to no movement of the macro block. By applying this application specific error method to erroneous motion vectors, the deadline misses are further reduced. Relative to naive error handling the average reduction of the number of deadline misses is 72% and the highest reduction is 97% in the case of the *minimal strategy*. The times by which deadlines are missed are reduced by up to 9%. By applying the *hybrid strategy* the number of deadline misses is reduced by 73% and the time by which deadlines are missed is reduced by 81%.

The comparison of the flexible error handling without and with the application specific error correction methods reveals the potential of the approach. Even though only one application specific error correction method is used, the average reduction of the number of deadline misses and the time by which deadlines are missed is 5% and 4%, respectively (*minimal strategy*). The highest observed reductions are 46% and 49%, respectively.

To get a better understanding of these numbers, Figure 7.9 shows the evaluation of the ratio between errors which can be ignored, errors which require checkpoint restore, and errors which can be handled by `MoCompDefaultValue`. The bars are normalized for better comparison. On average, 59% of all detected errors can be ignored. The application specific error correction method `MoCompDefaultValue` is applied on average to 4% of all errors. Considering the highest error rate, 6% of all errors are corrected by `MoCompDefaultValue`. Or in other words, the amount of required checkpoint restores is reduced by 6% which results in the decreased number of deadline misses. This leads to the conclusion that it is worthwhile to write application specific error corrections methods.

### 7.5.4 Quality of Service Impact of Flexible Error Handling

Setting the motion vector to zero and ignoring faults only affecting unreliable data has no impact on the control flow of the application. However, it will definitely have an impact on the quality of service. Table 7.12 shows the measured PSNR values of the different scenarios. Only successfully executed runs are considered. To

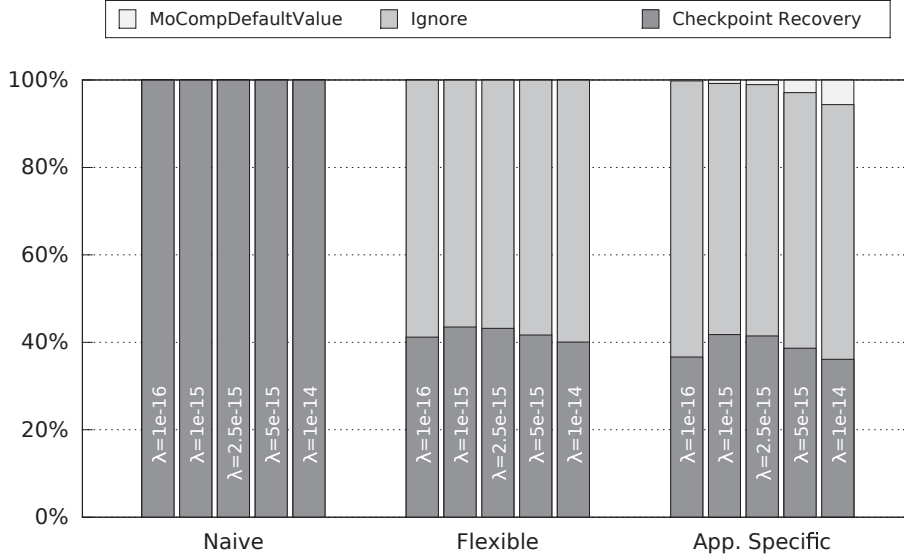


Figure 7.9: Applied error correction methods (*minimal checkpointing strategy*)

$\lambda$	1e-16	1e-15	2.5e-15	5e-15	1e-14
Naive Handling (Minimal)	36.19	36.15	35.36	34.19	-
Flexible Error Handling (Minimal)	36.19	36.18	36.08	34.21	28.82
Flexible + Application Specific (Minimal)	36.20	36.15	36.08	34.45	29.06
Naive Handling (Hybrid)	36.20	36.10	35.56	-	-
Flexible Error Handling (Hybrid)	36.19	36.14	35.55	34.37	-
Flexible + Application Specific (Hybrid)	36.19	36.17	35.54	34.19	31.32

Table 7.12: Average peak signal to noise ratio (in dB) with flexible error handling

obtain the PSNR values, the decoded frames are compared with the original source images of the video. The average PSNR ratio achieved by a golden run is 36.20 dB. As expected, higher fault rates lead to lower PSNR and hence to a decreased QoS.

Using the naive error handling approach, no errors are ignored since the *FAME* microvisor automatically triggers checkpoint recovery for each detected error. Even though, the PSNR values are decreasing with higher error rates. In the case of the *minimal strategy*, this is attributable to the fact that data will be excluded from a checkpoint if they are neither reliable nor subscribed. By using the *hybrid strategy*, errors affecting the PLAIN encoded parts of a checkpoint are not corrected which leads to the QoS degradation.

In the second scenario, the flexible error handling approach is applied. Hereby, errors affecting unreliable data or unsubscribed data are ignored. If an error affects a reliable and subscribed object, checkpoint recovery is used for correction. Intuitively, it is expected that the QoS is decreased due to ignoring a huge amount of errors. However, this is not the case. Ignoring errors causes a decreased number of checkpoint recoveries which amortizes the imposed QoS degradation. To recapitulate, if a checkpoint – created with the *minimal strategy* – is recovered, the state of the used unreliable data after recovery does not necessarily match the state at the

point in time where the checkpoint was created. Hence, the key is to reduce the number of checkpoint recoveries or to include subscribed unreliable data into the checkpoint.

The third experiment shown in Table 7.12 is based on the former approach. In addition, the application specific error correction method `MoCompDefaultValue` is used. Due to the fact that this method corrects errors not exactly, it is expected that the deviation from the best achievable QoS is higher compared to the previous experiment. This is not the case, since the number of the necessary checkpoint recoveries is reduced. Again, fewer restored checkpoints lead to fewer disturbances in unreliable data. Hence, the number of deadline misses is reduced and simultaneously the QoS is increased. This leads to the conclusion that it is worthwhile to provide application specific error correction methods.

In the last three experiments, the *hybrid checkpointing strategy* is used. This means that subscribed unreliable data is included in the checkpoint but only encoded using the PLAIN encoder. An improvement of the QoS can be noticed in the high fault rates. However, as shown in Table 7.10, the number of deadline misses is significantly increased.

### 7.5.5 Summary

In the evaluation of the flexible error handling approach, it is shown that the real-time behavior of the H.264 application is improved compared to a naive version where every error is handled alike. Flexible error handling allows to efficiently handle high error rates. However, it cannot be guaranteed that all deadlines are kept.

The price of the proposed approach is a decreased quality of service, especially if the *minimal checkpointing strategy* is used. It can be shown that, by using hybrid checkpoints, the QoS is improved for higher error rates. However, the drawback is an increased number of missed deadlines. Hence, there is a trade-off between QoS and real-time. Embedded system designers have to decide which solution suites best for a given environment.

## 7.6 Summary of Evaluation

The goal of this thesis is to provide an error mitigation technology which is able to cope with high error rates and which is applicable to embedded real-time systems. To be applicable to embedded real-time systems, the overhead of a technique should be as small as possible and the influence on the real-time behavior should be as low as possible. However, increasing the resiliency against transient faults always requires some kind of redundancy. Redundancy increases the required resources which leads to overhead. To lower the overhead, the approach presented in this dissertation proposes to handle errors in a flexible way by deciding **if**, **how**, and **when** to correct errors. Using flexible error handling, the evaluation clearly showed significant overhead reductions. The size of a checkpoint is reduced by up to 84%

and the creation time is reduced by a factor of more than ten. The overhead of the *FAME* microvisor is negligible. Moreover, for the fault-free case, a negative overhead can be observed. Concerning the real-time influence of the virtualization for the fault-free case, it can be seen that the virtualization does not lead to deadline misses. Furthermore, the average jitter is reduced when RTEMS is virtualized. These results clearly show that the flexible error handling approach is applicable to embedded real-time systems.

Furthermore, the evaluation shows that the developed approach is able to cope with very high error rates of up to 48 errors per second. Compared to a naive approach where every error is handled alike, the flexible error handling approach is able to reduce the number of missed deadlines by up to 97%. However, the price of this reduction are deviations in the output of the application.

# Conclusions and Future Work

---

This dissertation tackled the problem of growing transient error rates in future semiconductor devices [Int13]. By observing the outcome of errors injected into different applications, it was identified that errors can have different impacts on the quality of service dependent on the application and the affected data. This observation can be exploited by handling not every error alike. Hence, this thesis proposed a software-based flexible error handling approach. The goal of flexible error handling is to decide **if**, **how**, and **when** errors have to be corrected. In the thesis, the applicability of the flexible error handling approach was shown for a probabilistic error model and a transient memory error model. To realize flexible error handling, the *Fault Aware Microvisor Ecosystem (FAME)* was developed. *FAME* incorporates application knowledge gathered during compile time with information only available at runtime to realize the following main concepts:

1. **Software-based error handling:** Hardware-based error correction methods, like, e.g., ECC protection applied to a memory module, have the inherent problem that application semantics can not be considered. All data stored on a module would be protected. Thus, a software-based approach was realized to exploit application knowledge to handle not every error alike.
2. **Virtualization tailored to the needs of embedded systems and error-tolerance:** Software-based fault-tolerance mechanisms have a weak spot. If software parts critical for error handling are susceptible to errors as well, situations will occur where the error handling routine has to cope with errors affecting the error handling itself. This can result in a livelock. To deal with this problem, some guaranteed fault free hardware components are required to execute software-based fault-tolerance mechanisms. Those fault free hardware together with the necessary software components form a minimal set, the so called Reliable Computing Base (RCB) [ED12]. To protect the RCB this theses proposed to use virtualization. The virtualization was designed with respect to embedded systems and fault-tolerance. Hereby, features seen in other vitalization solution, like, e.g., CPU multiplexing in Xen [BDF+03], are omitted to minimize virtualization overhead. Correct timing is mandatory for embedded real-time systems. Consequently, the virtualization of the timing subsystem was carefully designed.

The evaluation results show clearly that virtualization overhead is negligible. Moreover, *FAME* runs faster than the native system and the jitter is reduced as well.

3. **Classification of the worst case error impact:** To decide whether or not errors have to be handled, data objects are classified regarding their susceptibility to errors. If an error affecting an object can lead to severe impact on the quality of service, like, e.g., an applicaiton crash, the object has to be classified as **reliable**. Otherwise, the object can be classified as **unreliable**. Error correction of reliable objects is mandatory and error correction of unreliable objects is optional. Optional in this context means that errors can be *ignored*, for example.

In the case of the H.264 video decoding benchmark, it was shown that up to 63% of all data objects can be classified as unreliable. This enabled the flexible error handling approach to ignore on average 59% of all occurred errors. Hence, the efforts which have to be spent on error correction are significantly reduced.

4. **Reduction to a task precedence problem:** The point in time where an error correction method is executed is crucial to an embedded real-time system. If correction is immediately executed, the error correction method could interrupt a task with higher priority which is not affected by errors. Thus, a kind of priority inversion problem occurs. To eliminate this problem, it was proposed to schedule error correction as an ordinary task. The erroneous tasks inherit a dependency to the correction task. The correction task inherits the highest priority of all affected tasks. This concept reduces error handling to a task precedence problem. Hence, an ordinary scheduling strategy which is able to cope with task precedences can be utilized.
5. **Subscriber-based programming model:** To be able to determine the necessary task dependency, a mapping of errors to tasks is required. Therefore, this dissertation introduces a subscriber-based programming model where the usage of resources has to be explicitly announced by subscribing to the appropriate objects. In this way, *FAME* can keep track of which task is using which object. The subscriber model was also used to determine whether an object is used since error correction for unused objects is not necessary. This liveness information is also advantageous for checkpoint creation. All unused objects can be excluded from a checkpoint.

In H.264, this yielded an average reduction of the checkpoint size of 65%. In combination with the reliability classification, checkpoint were reduced in average by up to 93% which results in a checkpoint creation speedup of more than 10.

By applying these concepts, the flexible error handling approach becomes feasible. Compared to a naive error handling approach where every error is handled alike, the approach presented in this thesis reduced the efforts needed to correct errors significantly. This resulted in the reduction of missed deadlines by up to 98%. Furthermore, it was shown that the approach was able to cope with very high error rates of nearly 50 errors per second.



## Future Work

The current realization of the flexible error handling approach is provided by *FAME*. All concepts and techniques described in this thesis are fully integrated. *FAME* can be used to automatically protect applications running under the RTEMS operating system. However, there is always space for improvements. This section finalizes the thesis by giving directions for further research.

### Application Specific Error Correction Methods

The evaluation of the flexible error handling approach clearly showed the improvements achievable if application specific error correction methods are used. Each applied application specific method avoids the recovery of a checkpoint. This leads to fewer deadline misses and to an improved quality of service. Hence, future work is to provide more application specific error correction methods.

One possible additional error correction method for H.264 is to replace an erroneous macro block type with a “skip” block type. This should lead to minor disturbance in the output but prevents checkpoint recovery.

### Benchmarks

Writing benchmark application with sufficient workload and precisely specified real-time constraints is difficult. Hence, only the H.264 decoding benchmark is fully evaluated in this dissertation. However, the applicability of flexible error handling to other domains was already shown in this thesis. Consequently, a further task is to port such applications to RTEMS and optionally to provide annotations in the source code to guide the classification process of the *REPAIR* compiler.

### Classification Improvement

Currently, the classification of data objects is black and white. A data object is classified as either reliable or as unreliable for its entire existence. By using data-flow analyses, a more fine-grained classification over the program execution should be possible. Only when an object is used in a context that requires high dependability, the reliable classification will be necessary. In the remaining time, the object can be classified as unreliable. This would allow for the reduction of cases where error correction is mandatory.

Another improvement could be the definition of further dependability classes to introduce some gray scale. This will be beneficial if multiple objects are affected by errors and not enough time for correcting every object is available. In this case, objects mapped to higher dependability classes can be corrected first.

### Error Detection

In this thesis, it is assumed that error detection is present. A future goal can be to integrate error detection into *FAME*. A possible error detection mechanism could be

software-based ECC. Due to the fact that the guest OS is virtualized, software-based ECC can be introduced without modifying the guest OS or the application. In this scenario, the main memory is not directly accessible by the application. Instead, demand paging is used. On access, the requested data is copied into a temporary reliable location by checking the software ECC. If the application accesses another block, the previously fetched block is written back and the new block is fetched.

Another possible error detection method could be redundant multi-threading (RMT). In such an approach, multiple instances of the program code are executed and the results are compared. If the results differ, a majority vote will be used.

### Multi-Core Support

Currently, *FAME* only supports single-core systems. By using multi-core systems more sophisticated EDAC methods can be realized. For example, multi-core systems allow handling errors which affect an entire CPU.

However, to enable multi-core support in *FAME*, the microvisor and RTEMS have to be extended. For the latter, R<sup>2</sup>G was already developed [Hei10]. The extension of the microvisor is not trivial. Hypercalls which do not allow for concurrent execution have to be identified and an appropriate synchronization has to be provided.

# Hypercall Table

The following table shows the hypercalls provided by the *FAME* microvisor. To shorten the table, the hypercall prefix `fame_hypercall_` is omitted. This means that, for example, the hypercall `fame_hypercall_config_modify` is abbreviated with `config_modify`.

No.	Name	Description
1	<code>config_modify</code>	Modification of different run-time parameters of the microvisor: system tick interval, ECI handler callback, and verbosity debug levels.
2	<code>irq_disable</code>	Enable the ECI interface (IRQ and timer tick messages).
3	<code>irq_enable</code>	Disable the ECI interface (IRQ and timer tick messages)
4	<code>irq_vector_disable</code>	Enable an interrupt vector in the virtualized IRQ controller.
5	<code>irq_vector_enable</code>	Disable an interrupt vector in the virtualized IRQ controller.
6	<code>irq_vector_isenabled</code>	Check whether an interrupt vector is enabled or not in the virtualized IRQ controller.
7	<code>cpsr_set</code>	Write to the program status word. (This hypercall is platform specific).
8	<code>register_get</code>	Write to a memory mapped I/O register.
9	<code>register_set</code>	Read from a memory mapped I/O register.
10	<code>cache_parameter_set</code>	Set caching parameters for a specific memory range. Different caching methods are support by this hypercall: caching on/off and writeback on/off.
11	<code>map_io</code>	Map an I/O register range into the guest OS address space.

**Table A.1:** Hypercall table (Part 1/2)

No.	Name	Description
12	object_add	Add a new object.
13	object_remove	Remove an object.
14	object_subscribe	Subscribe or weak-unsubscribe an object.
15	object_subscribe_multiple	Subscribe or weak-unsubscribe multiple objects.
16	object_unsubscribe	Unsubscribe an object.
17	object_unsubscribe_multiple	Unsubscribe multiple objects.
18	object_unsubscribe_all	Unsubscribe from all subscribed objects for a specific task. This hypercall is used on task deletion.
19	object_modify	Modify an object. This hypercall is used by the FAMERE heap implementation to implement realloc functionality.
20	reliable_malloc	Allocate reliable memory for the guest OS.
21	reliable_free	Free reliable memory for the guest OS.
22	reliable_avail	Returns the remaining heap space available for dynamic allocations of reliable memory.
23	reset_guestos	This hypercall is provided to enable the guest OS to reset itself.
24	checkpoint	Creates a full system checkpoint.
25	checkpoint_recover	Restores the latest created system checkpoint.
26	checkpoint_drop	Deletes the latest created system checkpoint.
27	checkpoint_static_area_modify	This hypercall is used to register the static memory areas of the guest OS which have to be checkpointed additionally to the dynamic objects.
28	idle	Signals the microvisor that the guest OS is idle.
29	debug1	Internal debug hypercall.
30	debug2	Internal debug hypercall.
31	fault_test	Simulates the occurrence of an fault (for debug purpose).

Table A.2: Hypercall table (Part 2/2)

# Bibliography

- [ACL05] K. H. Ang, G Chong, and Y. Li, PID control system analysis, design, and technology, *Control Systems Technology, IEEE Transactions on* 13.4 (2005), pp. 559–576.
- [ACL89] J Arlat, Y Crouzet, and J. C. Laprie, Fault injection for dependability validation of fault-tolerant computing systems, *Fault-Tolerant Computing Symposium* (1989), pp. 348–355.
- [Aus99] T. Austin, DIVA: a reliable substrate for deep submicron microarchitecture design, *32nd Annual International Symposium on Microarchitecture*, 1999, pp. 196–207.
- [Bai08] B. Bailey, *System level virtual prototyping becomes a reality with OVP donation from imperas*, White Paper, 2008.
- [Bbb] *Big Buck Bunny*, 2008, URL: <http://www.bigbuckbunny.org/>.
- [BBB+05] L. Benini, D. Bertozzi, A. Bogliolo, et al., MPARM: Exploring the Multi-Processor SoC Design Space with SystemC, *Journal of VLSI Signal Processing Systems* 41.2 (Sept. 2005), pp. 169–182.
- [BBB01] E. Bini, G. Buttazzo, and G. Buttazzo, A Hyperbolic Bound for the Rate Monotonic Algorithm, *ECRTS '01: Proceedings of the 13th Euromicro Conference on Real-Time Systems*, June 2001, p. 59.
- [BDF+03] P. Barham, B. Dragovic, K. Fraser, et al., Xen and the art of virtualization. *SOSP* 37.5 (2003), pp. 164–177.
- [Ber88] P. A. Bernstein, Sequoia: a fault-tolerant tightly coupled multiprocessor for transaction processing, *Transactions on Computers* 21.2 (1988), pp. 37–45.
- [Bla76] J. Blazewicz, *Scheduling Dependent Tasks with Different Arrival Times to Meet Deadlines*, North-Holland Publishing Co., Oct. 1976.
- [BLL+10] A Bhanu, M. S. K. Lau, K.-V. Ling, et al., A More Precise Model of Noise Based PCMOs Errors, *Electronic Design, Test and Application*, 2010, pp. 99–102.
- [CHM+11] D. Cordes, A. Heinig, P. Marwedel, et al., Automatic Extraction of Pipeline Parallelism for Embedded Software Using Linear Programming, *Parallel and Distributed Systems (ICPADS), 2011 IEEE 17th International Conference on*, 2011, pp. 699–706.

- [CML+08] L. N. B. Chakrapani, K. K. Muntimadugu, A. Lingamneni, et al., Highly energy and performance efficient embedded computing through approximately correct arithmetic: a mathematical foundation and preliminary experimental validation, *International conference on Compilers, architectures and synthesis for embedded systems*, New York, New York, USA, Oct. 2008, pp. 187–196.
- [CMS98] J. Carreira, H. Madeira, and J. G. Silva, Xception: a technique for the experimental evaluation of dependability in modern computers, *Software Engineering* 24.2 (1998), pp. 125–136.
- [CSB90] H. Chetto, M. Silly, and T. Bouchentouf, Dynamic scheduling of real-time tasks under precedence constraints, *The Journal Real-Time Systems* 2.3 (1990), pp. 181–194.
- [Dal03] M. Dales, *SWARM 0.44 Documentation*, 2003, URL: <http://www.cl.cam.ac.uk/~mwd24/phd/swarm.html>.
- [DHE12] B. Döbel, H. Härtig, and M. Engel, Operating system support for redundant multithreading, *the tenth ACM international conference on embedded software (EMSOFT'12)* (Oct. 2012), pp. 83–92.
- [Don87] J. Dongarra, The LINPACK Benchmark: An Explanation. *ICS* 297.Chapter 27 (1987), pp. 456–474.
- [Dra03] I. Draft, Recommendation and final draft international standard of joint video specification (ITU-T Rec. H. 264| ISO/IEC 14496-10 AVC), *Joint Video Team (JVT) of ISO/IEC MPEG and ITU-T VCEG, JVTG050* 33 (2003).
- [ED12] M. Engel and B. Döbel, The Reliable Computing Base-A Paradigm for Software-based Reliability. *Software-Based Methods for Robust Embedded Systems*, 2012.
- [EHS+11] M. Engel, A. Heinig, F. Schmoll, et al., Temporal properties of error handling for multimedia applications, *Electronic Media Technology*, 2011, pp. 1–6.
- [EKO95] D. Engler, F. Kaashoek, and J. O’Toole, Exokernel: an operating system architecture for application-level resource management, *Operating systems principles (SOSP)*, New York, USA, Dec. 1995, pp. 251–266.
- [ELD+92] R. Ecoffet, M. Labrunee, S. Duzellier, et al., Heavy ion test results on memories, *Radiation Effects Data Workshop* (1992), pp. 27–33.
- [ESH+11] M. Engel, F. Schmoll, A. Heinig, et al., Unreliable yet useful—reliability annotations for data in cyber-physical systems, *Workshop on Software Language Engineering for Cyber-physical Systems*, 2011.

- [FB04] M. Fiedler and R. Baumgart, *Implementation of a basic H. 264/AVC Decoder*, seminar paper at Chemnitz University of Technology, 2004.
- [FSK98] P. Folkesson, S. Svensson, and J. Karlsson, A comparison of simulation based and scan chain implemented fault injection, *Fault-Tolerant Computing* (1998), pp. 284–293.
- [Gal62] R. G. Gallager, Low-density parity-check codes, *Information Theory, IRE Transactions on* 8.1 (1962), pp. 21–28.
- [Gla] *Google Glass*, URL: <http://www.google.com/glass/start/>.
- [HAE+14] A. Herkersdorf, H. Aliee, M. Engel, et al., Resilience Articulation Point (RAP): Cross-layer dependability modeling for nanometer system-on-chip resilience, *Microelectronics Reliability* 54.6-7 (Feb. 2014), pp. 1066–1074.
- [Ham50] R. W. Hamming, Error detecting and error correcting codes, *Bell System Technical Journal, The* 29.2 (1950), pp. 147–160.
- [Hei10] A. Heinig, *R2G: Supporting POSIX like semantics in a distributed RTEMS system*, tech. rep. 836, TU Dortmund, Faculty of Computer Science 12, Dec. 2010.
- [Hen78] R. Henn, Antwortzeitgesteuerte Prozessorzuteilung unter strengen Zeitbedingungen, *Computing* 19.3 (1978), pp. 209–220.
- [HES+10a] A. Heinig, M. Engel, F. Schmoll, et al., Improving transient memory fault resilience of an H.264 decoder, *Embedded Systems for Real-Time Multimedia (ESTIMedia)* (2010), pp. 121–130.
- [HES+10b] A. Heinig, M. Engel, F. Schmoll, et al., Using application knowledge to improve embedded systems dependability, *Workshop on Hot Topics in System Dependability* (Oct. 2010).
- [HKB+14] C.-W. Huang, T. Kelter, B. Boenninghoff, et al., Static WCET Analysis of the H.264/AVC Decoder Exploiting Coding Information, *International Conference on Embedded and Real-Time Computing Systems and Applications*, IEEE, Chongqing, China, Aug. 2014.
- [HKS+13] A. Heinig, I. Korb, F. Schmoll, et al., Fast and Low-Cost Instruction-Aware Fault Injection, *Software-Based Methods for Robust Embedded Systems*, 2013.
- [HL10] G. Heiser and B. Leslie, The OKL4 microvisor: convergence point of microkernels and hypervisors, *Proceedings of the asia-pacific workshop on systems*, New York, USA, Aug. 2010, pp. 19–24.

- [HMS+12] A. Heinig, V. J. Mooney, F. Schmoll, et al., Classification-Based improvement of application robustness and quality of service in probabilistic computer systems, *ARCS'12: Proceedings of the 25th international conference on Architecture of Computing Systems*, Feb. 2012.
- [Hor74] W. A. Horn, Some simple scheduling algorithms, *Naval Research Logistics Quarterly* 21.1 (1974), pp. 177–185.
- [HRR+07] C. J. Hamann, M. Roitzsch, L. Reuther, et al., Probabilistic Admission Control to Govern Real-Time Systems under Overload, *Real-Time Systems, 2007. ECRTS '07. 19th Euromicro Conference on*, 2007, pp. 211–222.
- [HSB+15] A. Heinig, F. Schmoll, B. Bönninghoff, et al., FAME: Flexible Real-Time Aware Error Correction by Combining Application Knowledge and Run-Time Information, *Workshop on Silicon Errors in Logic - System Effects*, Mar. 2015.
- [HSH+08] J.-Y. Hwang, S.-b. Suh, S.-K. Heo, et al., Xen on ARM: System Virtualization Using Xen Hypervisor for ARM-Based Secure Mobile Phones, *Consumer Communications and Networking Conference, 2008. CCNC 2008. 5th IEEE*, 2008, pp. 257–261.
- [HSL78] A. L. J. Hopkins, T. B. I. Smith, and J. H. Lala, FTMP: A highly reliable fault-tolerant multiprocess for aircraft, *Proceedings of the IEEE* 66.10 (1978), pp. 1221–1239.
- [HSM+14] A. Heinig, F. Schmoll, P. Marwedel, et al., Who's using that memory? A subscriber model for mapping errors to tasks, *The 10th Workshop on Silicon Errors in Logic - System Effects (SELSE'10)*, Stanford, Apr. 2014.
- [HTI97] M.-C. Hsueh, T. K. Tsai, and R. Iyer, Fault injection techniques and tools, *Transactions on Computers* 30.4 (1997), pp. 75–82.
- [KAR+06] O. Krieger, M. Auslander, B. Rosenburg, et al., K42: building a complete operating system, *Proceedings of the European Conference on Computer Systems*, New York, New York, USA, Apr. 2006, pp. 133–145.
- [KEH+09] G. Klein, K. Elphinstone, G. Heiser, et al., seL4: formal verification of an OS kernel, *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, New York, New York, USA, Oct. 2009, pp. 207–220.
- [KFA+95] J Karlsson, P Folkesson, J Arlat, et al., Application of three physical fault injection techniques to the experimental assessment of the MARS architecture, *Dependable Computing for Critical Applications*, 1995.



- [Kop97] H. Kopetz, „*Real-Time Systems-Design Principles for Distributed Real-Time Systems*“, tech. rep., 1997.
- [Law96] K. P. Lawton, Bochs: A Portable PC Emulator for Unix/X, *Linux Journal* 1996.29es (Sept. 1996).
- [LC07] X. Li and J. Cai, Robust Transmission of JPEG2000 Encoded Images Over Packet Loss Channels. *ICME* (2007), pp. 947–950.
- [Lee07] E. A. Lee, *Computing foundations and practice for cyber-physical systems: A preliminary report*, tech. rep. UCB/EECS, Berkley, 2007.
- [Lie95] J. Liedtke, *On micro-kernel construction*, vol. 29, ACM, Dec. 1995.
- [Lie96] J. Liedtke, Toward Real Microkernels, *Communications of the ACM* 39.9 (Sept. 1996), pp. 70–77.
- [Liu00] J. W. S. Liu, *Real-Time Systems*, 2000.
- [LL73] C. L. Liu and J. W. Layland, Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment, *Journal of the ACM (JACM)* 20.1 (Jan. 1973), pp. 46–61.
- [LML+10] K.-V. Ling, V. J. Mooney, M. S. K. Lau, et al., Error rate prediction for probabilistic circuits with more general structures, *Synthesis And System Integration of Mixed Information technologies* (2010).
- [LW09] A. Lackorzynski and A. Warg, Taming subsystems: capabilities as universal resource access control in L4, *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems*, New York, New York, USA, Mar. 2009, pp. 25–30.
- [LY07] X. Li and D. Yeung, Application-Level Correctness and its Impact on Fault Tolerance, *High Performance Computer Architecture, 2007. HPCA 2007. IEEE 13th International Symposium on* (2007), pp. 181–192.
- [Mar11] P. Marwedel, *Embedded System Design*, New York, Heidelberg: Springer, 2011.
- [MEF+04] S. S. Mukherjee, J. Emer, T. Fossom, et al., Cache scrubbing in microprocessors: myth or necessity?, *Dependable Computing, 2004. Proceedings. 10th IEEE Pacific Rim International Symposium on*, 2004, pp. 37–42.
- [Moo65] G. E. Moore, Cramming more components onto integrated circuits, *Electronics* 38.8 (Apr. 1965), 114 ff.
- [Nah12] D. Nahberger, Fehlerbehandlung in echtzeitfähigen Steueranwendungen, MA thesis, TU Dortmund, 2012.

- [OSM02] N. Oh, P. Shirvani, and E. McCluskey, Error detection by duplicated instructions in super-scalar processors, *IEEE Transactions on Reliability* 51.1 (2002), pp. 63–75.
- [Pal03] K. V. Palem, Energy aware algorithm design via probabilistic computing: from algorithms and models to Moore’s law and novel (semiconductor) devices, *International conference on Compilers, architecture and synthesis for embedded systems*, New York, New York, USA, Oct. 2003, pp. 113–116.
- [Pal05] K. V. Palem, Energy aware computing through probabilistic switching: a study of limits, *Computers* 54.9 (2005), pp. 1123–1137.
- [PBK+94] J. S. Plank, M. Beck, G. Kingsley, et al., Libckpt: Transparent checkpointing under unix (1994).
- [PCK+09] K. V. Palem, L. N. B. Chakrapani, Z. M. Kedem, et al., Sustaining moore’s law in embedded computing through probabilistic and approximate design: retrospects and prospects, *CASES ’09: Proceedings of the 2009 international conference on Compilers, architecture, and synthesis for embedded systems*, New York, New York, USA, Oct. 2009, pp. 1–10.
- [PG74] G. J. Popek and R. P. Goldberg, Formal requirements for virtualizable third generation architectures, *Communications of the ACM* 17.7 (July 1974), pp. 412–421.
- [PGLOGV+07] M Portela-Garcia, C Lopez-Ongil, M Garcia-Valderas, et al., A Rapid Fault Injection Approach for Measuring SEU Sensitivity in Complex Processors, *IEEE International On-Line Testing Symposium*, 2007, pp. 101–106.
- [PLF03] V Pouget, D Lewis, and P Fouillat, Time-resolved scanning of integrated circuits with a pulsed laser: application to transient fault injection in an ADC, *Instrumentation and Measurement Technology Conference*, 2003, pp. 1376–1380.
- [PLS01] J. Pouwelse, K. Langendoen, and H. Sips, *Dynamic voltage scaling on a low-power microprocessor*, New York, New York, USA: ACM, July 2001.
- [RCV+05] G. A. Reis, J. Chang, N. Vachharajani, et al., SWIFT: Software Implemented Fault Tolerance, *CGO ’05: Proceedings of the international symposium on Code generation and optimization*, Mar. 2005.
- [Roi07] M. Roitzsch, Slice-balancing H.264 video encoding for improved scalability of multicore decoding, *Proceedings of the International Conference on Embedded Software*, New York, New York, USA, Sept. 2007, pp. 269–278.

- [RS60] I. S. Reed and G Solomon, Polynomial Codes Over Certain Finite Fields, *Journal of the Society for Industrial and Applied Mathematics* 8.2 (June 1960), pp. 300–304.
- [SAC+99] T. Slegel, R. Averill, M. Check, et al., IBM’s S/390 G5 microprocessor design, *IEEE Micro* 19.2 (1999), pp. 12–23.
- [SBL+11] A Singh, A. Basu, K.-V. Ling, et al., Modeling multi-output filtering effects in PCMOs, *VLSI Design, Automation and Test* (2011), pp. 1–4.
- [SHK+12] H. Schirmeier, M. Hoffmann, R. Kapitza, et al., Fail\*: Towards a versatile fault-injection experiment framework, *ARCS Workshops (ARCS), 2012* (2012), pp. 1–5.
- [SHM+13] F. Schmoll, A. Heinig, P. Marwedel, et al., Improving the fault resilience of an H.264 decoder using static analysis methods, *Transactions on Embedded Computing Systems* 13.1s (Nov. 2013), pp. 1–27.
- [SHM+14] F. Schmoll, A. Heinig, P. Marwedel, et al., *Passing error handling information from a compiler to runtime components*, tech. rep., 2014.
- [SKA+04] S. M. Srinivasan, S. Kandula, C. R. Andrews, et al., Flashback: A lightweight extension for rollback and deterministic replay for software debugging, *USENIX Annual Technical Conference* (2004).
- [SKS13] H. Schirmeier, I. Korb, and O. Spinczyk, Inderscience Publishers - Log In - Resource Secured, *International Journal of Critical Computer-Based Systems* (2013).
- [Sor09] D. J. Sorin, Fault Tolerant Computer Architecture, *dx.doi.org* 4.1 (May 2009), pp. 1–104.
- [SPH+06] L. Singaravelu, C. Pu, H. Härtig, et al., Reducing TCB complexity for security-sensitive applications: three case studies, *Proceedings of the European Conference on Computer Systems*, New York, New York, USA, Apr. 2006, pp. 161–174.
- [SSF99] J. S. Shapiro, J. M. Smith, and D. J. Farber, EROS: a fast capability system, *SOSP ’99: Proceedings of the seventeenth ACM symposium on Operating systems principles*, New York, New York, USA, Dec. 1999, pp. 170–185.
- [VCG+05] F Vargas, D. L. Cavalcante, E Gatti, et al., On the proposition of an EMI-based fault injection approach, *IEEE International On-Line Testing Symposium*, 2005, pp. 207–208.
- [VHM03] R. Venkatasubramanian, J. Hayes, and B. Murray, Low-cost online fault detection using control flow assertions, *On-Line Testing Symposium, 2003. IOLTS 2003. 9th IEEE* (2003), pp. 137–143.

- [VKC+92] R Velazco, S Karoui, T Chapuis, et al., Heavy ion test results for the 68020 microprocessor and the 68882 coprocessor, *Nuclear Science* 39.3 (1992), pp. 436–440.
- [VRE00] R Velazco, S Rezgui, and R Ecoffet, Predicting error rate for microprocessor-based digital architectures through C.E.U. (Code Emulating Upsets) injection, *Nuclear Science* 47.6 (2000), pp. 2405–2411.
- [Wal64] C. S. Wallace, A Suggestion for a Fast Multiplier, *Electronic Computers, IEEE Transactions on* 1 (1964), pp. 14–17.
- [WEE+08] R. Wilhelm, J. Engblom, A. Ermedahl, et al., The worst-case execution-time problem—overview of methods and survey of tools, *Transactions on Embedded Computing Systems* 7.3 (Apr. 2008), pp. 1–53.
- [WKG+06] L. Wang, Z. Kalbarczyk, W. Gu, et al., An OS-level Framework for Providing Application-Aware Reliability, *12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, Dec. 2006, pp. 55–62.
- [Yeh98] Y. Yeh, Design considerations in Boeing 777 fly-by-wire computers, *Third IEEE International High-Assurance Systems Engineering Symposium* (1998), pp. 64–72.
- [YWG+06] L. Youseff, R. Wolski, B. Gorda, et al., Evaluating the Performance Impact of Xen on MPI and Process Execution For HPC Systems, *Virtualization Technology in Distributed Computing, 2006. VTDC 2006. First International Workshop on* (2006), pp. 1–1.
- [AMD06] AMD, *BIOS and Kernel Developer's Guide for AMD Athlon64 and AMD Opteron Processors*, Feb. 2006, URL: <http://support.amd.com/TechDocs/26094.pdf>.
- [ARM14] ARM, *ARM Architecture Reference Manual*, Oct. 2014, URL: <http://infocenter.arm.com/help/topic/com.arm.doc.ddi0406b/>.
- [Int12] International Telecommunication Union (ITU), *Parameter values for ultra-high definition television systems for production and international programme exchange*, Aug. 2012, URL: [http://www.itu.int/dms\\_pubrec/itu-r/rec/bt/R-REC-BT.2020-0-201208-I!!PDF-E.pdf](http://www.itu.int/dms_pubrec/itu-r/rec/bt/R-REC-BT.2020-0-201208-I!!PDF-E.pdf).
- [Int13] International Technology Roadmap for Semiconductors, Process Integration, Devices, and Structures (PIDS), *ITRS* (2013).
- [Kar14] Karo Electronics, *TK71 Development Board*, Oct. 2014, URL: <http://www.karo-electronics.com/tk71.html>.

- 
- [Mar14] Marvell, *Marvell Kirkwood series*, Oct. 2014, URL: <http://www.marvell.com/embedded-processors/kirkwood>.
- [NXP14] NXP, *NXP mbed LPC1768 rapid prototyping board*, Oct. 2014, URL: <http://mbed.org/platforms/mbed-LPC1768/>.
- [OAR14] OAR Corporation, *RTEMS: Real-Time Executive for Multiprocessor Systems*, Oct. 2014, URL: <http://www.rtems.com>.
- [Ope14] OpenOCD, *Free and Open On-Chip Debugging, In-System Programming and Boundary-Scan Testing*, Oct. 2014, URL: <http://openocd.sourceforge.net/>.
- [Syn14] Synopsys Corporation, *CoMET/METeor Models*, Oct. 2014, URL: <http://www.synopsys.com/Systems/VirtualPrototyping/VPMODELS/Pages/CoMET-METeor.aspx>.



# List of Figures

1.1	Definition of laxity, period, and activation time of a task . . . . .	6
1.2	WCET-related terms (based on [Mar11]) . . . . .	7
2.1	EMI injection setup . . . . .	23
3.1	Bit flips as Resilience Articulation Point (based on [HAE+14]) . . .	33
3.2	Bit flip abstraction in higher system levels (based on [HAE+14]) . .	34
3.3	Demonstrator platform and injection board . . . . .	38
3.4	JTAG injection setup . . . . .	39
3.5	JTAG fault injection algorithm . . . . .	40
3.6	Three-Stage Models for circuits (taken from [SBL+11]) . . . . .	43
3.7	The used Three-Stage Models of different full adders (based on [SBL+11])	44
3.8	Probabilistic ripple carry adder (PRCA) . . . . .	46
3.9	Probabilistic Wallace tree multiplier (PWTM) . . . . .	46
3.10	QTNetview showing QoS metrics . . . . .	49
4.1	Flexible error handling . . . . .	52
4.2	LEGO Mindstorm-based robot . . . . .	57
4.3	Experimental setup for robotic application . . . . .	57
4.4	Object existence and liveness . . . . .	64
4.5	Producer and consumer problem . . . . .	65
4.6	Subscription states of an object . . . . .	66
4.7	Fault correction and real-time . . . . .	68
4.8	Will error correction be needed? . . . . .	70
5.1	FAME - Fault Aware Microvisor Ecosystem . . . . .	72
5.2	Propagation of <i>unreliable</i> attribute (taken from [SHM+13]) . . . . .	74
5.3	Software stack . . . . .	77
5.4	Virtual memory layout (CoMET-based ARM926 platform) . . . . .	86
5.5	<i>FAME</i> Checkpoints . . . . .	90
5.6	Overview of RTEMS and <i>FAME</i> . . . . .	96
5.7	RTEMS native heap implementation . . . . .	99
5.8	<i>FAME</i> Error Handling Procedure . . . . .	101
6.1	Frame decoding time for successive frames . . . . .	107
6.2	Decoding time for I-Frames in relation to average size of macro blocks	108
6.3	Decoding time for P-Frames in relation to average size of macro blocks	109
6.4	Relative error of decoding time estimation (with kind permission of Björn Bönninghoff) . . . . .	110
6.5	Task dependencies of the H.264 decoding application . . . . .	111

---

7.1	Used evaluation platforms . . . . .	116
7.2	Different impacts of transient memory faults . . . . .	119
7.3	Quality of service degradation due to lowered supply voltages using UVOS . . . . .	121
7.4	PSNR values for different videos . . . . .	122
7.5	Used BIVOS setups . . . . .	123
7.6	Jitter comparison . . . . .	127
7.7	Checkpoint creation time dependent on the used block size and encoder	129
7.8	Checkpoint recover time dependent on the used block size and encoder	129
7.9	Applied error correction methods ( <i>minimal checkpointing strategy</i> ) .	136



# List of Tables

3.1	JTAG injection results . . . . .	41
3.2	Effective error rates . . . . .	45
4.1	Error classification for H.264 decoder . . . . .	55
4.2	Error classification for robotic application . . . . .	58
4.3	Example task set . . . . .	67
6.1	Analyzed videos . . . . .	107
7.1	Ratio of reliable to unreliable memory (taken from [SHM+13]) . . .	118
7.2	QoS dependent on the effective error rate (flexible error handling + minimal checkpointing) . . . . .	120
7.3	Analyzed videos . . . . .	120
7.4	Instructions executed using probabilistic components . . . . .	121
7.5	Interrupt measurement . . . . .	124
7.6	Hypercall measurement . . . . .	125
7.7	Execution time comparison . . . . .	126
7.8	Checkpoint measurement . . . . .	131
7.9	Effective average error rates . . . . .	133
7.10	Average deadline miss count . . . . .	134
7.11	Average time period by which deadlines are missed in milliseconds .	134
7.12	Average peak signal to noise ratio (in dB) with flexible error handling	136
A.1	Hypercall table (Part 1/2) . . . . .	143
A.2	Hypercall table (Part 2/2) . . . . .	144

# List of Code Listings

3.1	Function without probabilistic instructions . . . . .	47
3.2	Code transformed to use the PRCA . . . . .	47
4.1	Frame annotated with reliable and unreliable . . . . .	60
4.2	Annotated correction method . . . . .	61
4.3	Correction method assigned to data . . . . .	61
5.1	Example Code . . . . .	74
5.2	malloc_reliable/malloc_unreliable API . . . . .	75
5.3	Source-to-source translated code of Listing 4.3 . . . . .	75
5.4	RTEMS context switch code for ARM . . . . .	84
5.5	RTEMS context switch code for ARM ported to <i>FAME</i> . . . . .	98
6.1	New RTEMS real-time task create API . . . . .	113
6.2	New RTEMS task dependency API . . . . .	113

# List of Abbreviations

<b>ACET</b>	average case execution time
<b>ADC</b>	analog-to-digital converter
<b>BCET</b>	best case execution time
<b>BCET<sub>EST</sub></b>	estimated best case execution time
<b>BSP</b>	board support package
<b>CEU</b>	Code Emulated Upsets
<b>CPS</b>	cyber-physical system
<b>DMR</b>	Double Modular Redundancy
<b>DSP</b>	Digital Signal Processor
<b>ECC</b>	Error Correction Codes
<b>ECI</b>	event callback interface
<b>EDAC</b>	error detection and correction
<b>EDF*</b>	Earliest Deadline First with precedence
<b>EDF</b>	Earliest Deadline First
<b>FAME</b>	Fault Aware Microvisor Ecosystem
<b>FAMERE</b>	FAME Runtime Environment
<b>FCG</b>	Fault Correction Graph
<b>FEHLER</b>	Flexible Error Handling in Embedded Real-Time Systems
<b>fps</b>	frames per second
<b>IC</b>	integrated circuit
<b>IP</b>	intellectual property [block/core]
<b>IT</b>	Information Technology
<b>ITRS</b>	International Technology Roadmap for Semiconductors
<b>JTAG</b>	Joint Test Action Group
<b>LET</b>	linear energy transfer
<b>LST</b>	Least Slack Time First
<b>MMU</b>	Memory Management Unit
<b>MPSoC</b>	Multiprocessor System-on-Chip
<b>MTBF</b>	Mean Time Between Failures
<b>MTTF</b>	Mean Time To Failure
<b>OS</b>	Operating System
<b>OCD</b>	On-Chip Debugger
<b>PC</b>	Personal Computer
<b>PCB</b>	printed circuit board
<b>prob-cc</b>	probabilistic C compiler
<b>PSNR</b>	peak signal-to-noise ratio
<b>QoS</b>	quality of service
<b>QRMS</b>	Quality Rate Monotonic Scheduling
<b>R<sup>2</sup>G</b>	RTEMS and RTLib Glued together
<b>RAP</b>	Resilience Articulation Point

<b>RCB</b>	Reliable Computing Base
<b>REPAIR</b>	Reliable Error Propagation And Impact Restricting [Compiler]
<b>RMS</b>	Rate Monotonic Scheduling
<b>SEU</b>	single event upset
<b>SoC</b>	system on a chip
<b>SUT</b>	system under test
<b>TLB</b>	Translation Lookaside Buffer
<b>TMR</b>	Triple Modular Redundancy
<b>VMM</b>	virtual machine monitor
<b>WCET</b>	worst case execution time
<b>WCET<sub>EST</sub></b>	estimated worst case execution time

# Index

- A**
- address space ..... 48
- C**
- cache scrubbing ..... 14
- capability ..... 16
- checkpoint
  - block encoder ..... 91, 128
  - create ..... 19, 89, 94, 128
  - full-system checkpoint ..... 21
  - hybrid strategy ..... 132, 134
  - incremental ..... 20
  - minimal strategy ..... 132, 134
  - recovery ..... 19, 92, 128
  - time estimation ..... 91
- classification ..... 59
  - automatic annotation ..... 61, 74
  - correction method annotation 60, 74
  - impact ..... 117–124
  - parsing ..... 73
  - prohibit rule ..... 73
  - propagation rule ..... 73
  - reliable type qualifier . 59, 73, 117
  - unreliable type qualifier .. 59, 73, 117
- copy-on-write ..... 20
- cyber-physical systems ..... 5
- D**
- dependability ..... 7
- dynamic object ..... 76
- E**
- ECC ..... 14
- embedded systems ..... 5
  - efficiency ..... 7
- error
  - application-specific handling . 135
  - classification *see* classification, 53
  - definition ..... 4
  - flexible error handling .. 3, 51–70, 101–104, 132, 134
  - ignore ..... 52
  - masking ..... 4
  - naive error handling ..... 133
  - propagation ..... 32
  - soft errors ..... *see* transient fault
- event callback interface ..... 82, 124
- execution time ..... 6
  - BCET<sub>EST</sub> ..... 6
  - WCET<sub>EST</sub> ..... 6
  - BCET ..... 6
  - upper bound ..... 6
  - WCET ..... 6
- exokernel ..... 19
- F**
- failure
  - definition ..... 4
- FAME Runtime Environment .. 93–96
- fault
  - definition ..... 4
  - duration ..... 5
  - injection ..... 21–25, 35–42, 117
  - masking ..... 4
  - permanet fault ..... 5
  - propagation ..... 32
  - transient fault ..... 5, 118
- Fault Aware Microvisor Ecosystem 71–104, 118
- Fault Correction Graph ..... 77
- fault tolerance
  - proactive ..... 13
  - reactive ..... 13
- H**
- H.264 ..... 54, 105–113, 118, 127
- libh264 ..... 105
- real-time decoder application 106–112
- simple decoder application ... 106

- timing ..... 28, 29, 107–110
  - high reliable silicon ..... 34
  - hour glass-model ..... 32
  - hypercall ..... 17, 81, 125
  - hypervisor ..... 17
- I**
- interrupt latency ..... 124
- J**
- jitter ..... 127
- K**
- kernel communication page ..... 87
- L**
- library OS ..... 19, 80
- librecon ..... 73, 76
- low reliable silicon ..... 34
- M**
- masking
  - application level ..... 4
  - architectural ..... 4
  - arithmetical ..... 4
  - logical ..... 4
- memory management ..... 85, 98
  - cooperative ..... 88
- microkernel ..... 16
  - L4 ..... 17, 18
  - seL4 ..... 18
- microvisor 17, 19, 32, 78–93, 102, 124, 130
- minimality principle ..... 17
- P**
- para-virtualization 17, 80, 93, 96, 125
- probabilistic error model . 42–47, 117, 120
  - biased voltage scaling .... 43, 122
  - probabilistic CMOS ..... 42
  - ripple carry adder ..... 45, 117
  - three-stage model ..... 43–47
  - uniform voltage scaling ... 43, 122
  - Wallace tree multiplier ... 46, 117
- process ..... 48
- Q**
- quality of service metric ..... 48
  - delta E ..... 49
  - peak signal-to-noise ratio 49, 119, 121, 135
- R**
- real-time systems ..... 5
  - hard real-time system ..... 5
  - soft real-time system ..... 5
- redundancy
  - double modular redundancy .. 14
  - spatial ..... 13
  - temporal ..... 13
  - triple modular redundancy .. 14
- reliable computing base ... 31, 78, 126
  - components ..... 79
  - definition ..... 32
  - invariant ..... 32
- REPAIR compiler ... 72–77, 100, 118, 120
- resilience articulation point ..... 32
- ring buffer ..... 111
- RTEMS ..... 96–101
  - modification ..... 112
- S**
- scheduling ..... 25–27, 94, 110–112
  - earliest deadline ..... 25, 67
  - EDF\* ..... 27, 95, 112
  - least slack time first ..... 26
  - optimal scheduler ..... 25
  - quality rate monotonic ..... 26
  - rate monotonic ..... 26
- single event upset . *see* transient fault
- static object ..... 76
- subscriber model .. 63–69, 88, 94, 130
  - automatic malloc rewrite .... 75
  - liveness ..... 64
  - object definition ..... 64
  - object management ..... 88
  - subscribed object ..... 64
  - subscriber states ..... 66
  - system object ..... 64

- 
- unsubscribed object .....64
  - weak unsubscribed object .....66
- T**
- task
    - WCET<sub>EST</sub> .....28
    - available .....6
    - deadline .....5, 48, 112
    - deadline interval .... *see* deadline
    - execution time .....48
    - laxity ..... *see* slack
    - period .....6, 48
    - precedence ..... 48, 63, 112
    - real-time task .....6
    - release time .....6, 112
    - slack .....6, 28
    - task model .....48
    - task set .....48
  - transient memory fault model . 34–42
  - trusted computing base ..... 17
- U**
- ubiquitous computing ..... 1
- V**
- virtual machine .....17
  - virtual machine monitor ..... 17

<http://www.andreasheinig.de/permalink/15051>