Thesis for the Degree of Doctor of Philosophy

Layered Fault Tolerance for Distributed Embedded Systems

Raul Barbosa

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2008

Layered Fault Tolerance for Distributed Embedded Systems
Raul Barbosa
ISBN 978-91-7385-209-8

Doktorsavhandlingar vid Chalmers tekniska högskola
Ny serie 2890
ISSN 0346-718X

Technical Report No. 52D
Department of Computer Science and Engineering
Dependable Real-Time Systems Group

Department of Computer Science and Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
Telephone: +46 (0)31–772 1000

Printed by Chalmers Reproservice
Göteborg, Sweden 2008

Abstract

This thesis deals with principles and techniques of fault tolerance for distributed embedded systems. A layered approach is taken to achieve high dependability by structuring error detection and recovery mechanisms into three layers. The first layer consists of mechanisms implemented in hardware, either at the circuit or the micro-architectural level. Many integrated circuits, especially microprocessors, are provided with such mechanisms in order to mask transient hardware faults and to detect permanent ones. To prevent software faults and hardware faults not captured at the hardware layer from causing node failures, it is desirable to introduce node-layer mechanisms. While they may depend on hardware support such as memory protection, they are mostly implemented in software. For this second layer, the thesis proposes techniques for building robust operating systems, addressing software and hardware faults in a comprehensive manner. The goal is to guarantee the integrity of tasks in a multithreaded environment by preventing undesired interactions among tasks and by providing them with recovery services. Some of these techniques were added to an existing real-time kernel and assessed experimentally. To this end, an experimental platform, with an associated fault injection tool, was developed. Following a methodology for fault removal, the tool revealed two design flaws in the kernel extension. Even though the goal of node-layer mechanisms is to make computer nodes highly dependable, nodes may still fail. This motivates the development of system-layer mechanisms that can deal with node failures. Accordingly, the thesis investigates methods for distributed redundancy management and proposes a protocol for guaranteeing consistent diagnosis of node failures in synchronous systems. Due to its importance as a building block, the protocol was formally verified using model checking. An important goal of the proposed framework and the associated node-layer and system-layer mechanisms is to reduce the cost of fault tolerance in distributed embedded systems.

List of Publications

This thesis is partly based on the following publications:

I Raul Barbosa and Johan Karlsson, "On the integrity of lightweight checkpoints", to appear in *Proceedings of the 11th IEEE High Assurance Systems Engineering Symposium (HASE 2008)*, Nanjing, China, December 2008.

II Raul Barbosa and Johan Karlsson, "Formal specification and verification of a protocol for consistent diagnosis in real-time embedded systems", in *Proceedings of the 3rd IEEE International Symposium on Industrial Embedded Systems (SIES'2008)*, Montpellier – La Grande Motte, France, pp. 216–223, June 2008.

III Raul Barbosa, "Operating system services for recovering errant applications", in *Proceedings Supplemental Volume of the 7th European Dependable Computing Conference (EDCC-7)*, Kaunas, Lithuania, pp. 91–96, May 2008.

IV Raul Barbosa, António Ferreira and Johan Karlsson, "Implementation of a flexible membership protocol on a real-time Ethernet prototype", in *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, Melbourne, Australia, pp. 342–345, December 2007.

V Raul Barbosa and Johan Karlsson, "Analysis of robust partitioning mechanisms", Technical Report No. 2007:13, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, October 2007.

VI Raul Barbosa and Johan Karlsson, "Flexible, cost-effective membership agreement in synchronous systems", in *Proceedings of the 12th Pacific Rim International Symposium on Dependable Computing (PRDC'06)*, Riverside, California, USA, pp. 105–112, December 2006.

VII Raul Barbosa, Jonny Vinter, Peter Folkesson and Johan Karlsson, "Assembly-level pre-injection analysis for improving fault injection efficiency", in *Proceedings of the 5th European Dependable Computing Conference (EDCC-5)*, Budapest, Hungary, LNCS 3463, pp. 246–262, April 2005.

Acknowledgements

I would like to express my deepest gratitude to Professor Johan Karlsson for his invaluable advice and knowledge shared throughout my studies.

Very special thanks are owed to Professor Emeritus Jan Torin, Professor Simin Nadjm-Tehrani, Dr. Thomas Lundqvist, Dr. Kristina Forsberg and Sam Nicander for insightful discussions on the partitioning problem and for substantial comments on several parts of the thesis.

I wish to thank Professor Bengt Jonsson for his guidance on model checking. I am grateful also to Professor Andreas Steininger for the extensive comments provided halfway through the studies.

I would like to show my appreciation to Professor Mário Rela, who's enthusiasm influenced my decision to pursue my studies, for the introduction to the field of dependability.

Special thanks are due to Daniel Skarin, with whom I had the pleasure to collaborate and exchange valuable insights. Special thanks go also to Jorge Alçada, António Ferreira and Mikael Hedén for their inspiration and remarkable dedication during their studies.

I also thank Professor Jan Jonsson and Professor Philippas Tsigas for the regular follow-up meetings to discuss the direction of my studies.

For her constant support, encouragement, dedication and for being the best companion I could wish for, I thank Filipa.

Last and most importantly, I am grateful to my parents Ângelo and Marta for stimulating my curiosity and for their everlasting support in pursuing my goals in life. Obrigado!

This work was supported by the Portuguese Fundação para a Ciência e a Tecnologia through doctoral grant SFRH/BD/18126/2004.

Contents

1 Introduction **1**

2 The Architectural Framework **5**
2.1 Terminology 5
2.1.1 Faults, Errors and Failures 6
2.1.2 Dependability Attributes 8
2.1.3 The Means to Dependability 9
2.2 System Model 10
2.3 Layered Fault Tolerance 11
2.4 Objectives 13
2.5 Main Contributions 14

3 Separation of Integrated Functions **17**
3.1 Theoretical Motivation 18
3.1.1 Modeling Hardware Failures 21
3.1.2 Modeling Software Failures 25
3.2 Requirements for Partitioning 30
3.3 Mechanisms for Partitioning 35
3.3.1 Spatial Partitioning 35
3.3.2 Temporal Partitioning 42
3.4 Summary and Discussion 43

4 Robust Operating Systems **47**
4.1 SECERN: An Extension to μC/OS-II 49
4.1.1 Design Principles of SECERN 50
4.1.2 Error Detection and Fault Handling 53
4.1.3 Scheduler 54
4.2 Robustness Testing for Partitioned Systems 54
4.3 Focused Fault Injection 57
4.3.1 Methodology 58
4.3.2 Results 60
4.3.3 Limitations 64
4.4 Recovering Errant Applications 65
4.4.1 A Comprehensive Recovery Strategy 65
4.5 Lightweight Checkpoints 67
4.5.1 Context and Applicability 68
4.5.2 Failure Modes and Error Detection Latency 69
4.5.3 Assuring the Integrity of Checkpoints 70
4.5.4 Implementation Aspects 72
4.5.5 Verification using Model Checking 74
4.6 Related Research 80
4.7 Summary and Discussion 82

5 On the Efficiency of Fault Injection **85**
5.1 Related Research 87
5.2 Fault-space Optimization Method 89
5.2.1 Optimization Input 89
5.2.2 Optimization Output 90
5.2.3 Performing the Optimization 90
5.3 Experimental Setup 91
5.3.1 Fault Injection Tool 92
5.3.2 MPC565 Microcontroller 93
5.3.3 Workloads 93
5.3.4 Fault Model and Fault Selection 95
5.4 Experimental Results 96
5.4.1 Fault Injection in Registers 96
5.4.2 Fault Injection in Memory 102
5.4.3 Fault-space Considerations 102
5.5 Summary and Discussion 104

6 Distributed Redundancy Management 107
6.1 System Model and Assumptions . . . 109
6.1.1 Failure Modes . . . 110
6.1.2 Rationale . . . 111
6.1.3 Node Restarts . . . 112
6.2 The Membership Protocol . . . 113
6.2.1 Notation and Definitions . . . 113
6.2.2 Agreement on Exclusion . . . 115
6.2.3 Inclusion Ordering . . . 117
6.2.4 Agreement on Inclusion . . . 120
6.2.5 Integration with Node-Layer Fault Tolerance . . . 121
6.2.6 Tuning the Protocol . . . 122
6.3 Prototype Implementation . . . 123
6.3.1 Network Configuration . . . 124
6.3.2 Network and Membership Performance . . . 125
6.4 Related Research . . . 127
6.5 Summary and Discussion . . . 128

7 Formal Verification of Consistent Diagnosis 131
7.1 Formal Specification of the Protocol . . . 132
7.2 System and Protocol Models . . . 134
7.2.1 The Broadcast Channel . . . 135
7.2.2 The Processor Nodes . . . 135
7.2.3 Modeling Failures . . . 138
7.2.4 Modeling Restarts . . . 139
7.2.5 Specifying the Correctness Properties . . . 140
7.2.6 Parametrization of the Model . . . 141
7.3 Verification Results . . . 142
7.3.1 Further Considerations . . . 143
7.4 Summary and Discussion . . . 145

8 Interoperability between Layers 147
8.1 Advantages of Fail-Report Semantics . . . 148
8.2 Multiple Transmission Slots . . . 150
8.3 Application-Process Membership . . . 151
8.4 Summary and Discussion . . . 152

9 Conclusions 153

References **159**

List of Figures

2.1 The dependability tree. 6
2.2 Structural elements of the architectural framework. 10
2.3 Layered fault tolerance for distributed embedded systems. 13

3.1 State transition diagram, regarding hardware failures, for a 1-out-of-n-resilient federated system. 22
3.2 State transition diagram, regarding hardware failures, for a 2-out-of-n-resilient federated system. 22
3.3 State transition diagram, regarding hardware failures, for a 1- or 2-resilient integrated non-DMR system. 23
3.4 State transition diagram, regarding hardware failures, for a 1- or 2-resilient integrated DMR system. 23
3.5 Comparison of federated and integrated systems regarding hardware failures. 24
3.6 Sensitivity of integrated systems to hardware failure rate. 25
3.7 State transition diagram, regarding software failures, for a 1-out-of-n-resilient federated system. 27
3.8 State transition diagram, regarding software failures, for a 2-out-of-n-resilient federated system. 27
3.9 State transition diagram, regarding software failures, for a 1-out-of-n-resilient integrated system. 28

3.10 State transition diagram, regarding software failures, for a 2-out-of-n-resilient integrated system. 28
3.11 Sensitivity of integrated systems to the coverage of partitioning mechanisms (1-resilient systems with $\lambda_{si} = 10^{-6}$ failures/h and $\lambda_{pm} = 0$). 29
3.12 Sensitivity of integrated systems to the coverage of partitioning mechanisms (2-resilient systems with $\lambda_{si} = 10^{-6}$ failures/h and $\lambda_{pm} = 0$). 29
3.13 Sensitivity of integrated systems to the failure rate of partitioning mechanisms (1-resilient systems with $\lambda_{si} = 10^{-6}$ failures/h and $c = 99\%$). 30
3.14 Sensitivity of integrated systems to the failure rate of partitioning mechanisms (2-resilient systems with $\lambda_{si} = 10^{-6}$ failures/h and $c = 99\%$). 30

4.1 μC/OS-II extended with SECERN. 50
4.2 Context switching time measurements. 52
4.3 Evaluation platform for μC/OS-II and SECERN. 55
4.4 Main routine of a workload thread. 57
4.5 Manual instrumentation of the low priority thread to corrupt the stack pointer and wait for a context switch. . . . 63
4.6 Logical checkpoint area (visible to the application) mapped to one of three physical checkpoints. 73
4.7 Application and exception handler models. 75
4.8 Model of the application's errant behaviour. 76
4.9 Model of the checkpointing service. 77
4.10 Error injector and error detector processes. 78

5.1 Example of the optimization procedure. 92
5.2 Evaluation platform for the jet engine application. 93
5.3 Exception distribution in the non-optimized quicksort campaign (83 faults in registers). 98
5.4 Exception distribution in the optimized quicksort campaign (744 faults in registers). 98
5.5 Exception distribution in the non-optimized jet engine controller campaign (200 faults in registers). 99
5.6 Exception distribution in the optimized jet engine controller campaign (466 faults in registers). 99

5.7 Number of faults injected in each register (1559 faults in the optimized jet engine controller campaign). 101
5.8 Exception distribution in the non-optimized jet engine controller campaign (40 faults in memory). 103
5.9 Exception distribution in the optimized jet engine controller campaign (166 faults in memory). 103

6.1 Round number signaling by a node in the membership, using the *i-flag* of its messages (one message per round). . 119
6.2 The experimental real-time Ethernet network. 123

7.1 Data structures for the broadcast channel. 135
7.2 The broadcast process. 136
7.3 The membership views of all processor nodes. 136
7.4 Structure of the processor nodes, where the comments represent the code in Algorithms 6.1 and 6.2. 137
7.5 Failure injection routine (inline), called by the broadcast process. 139
7.6 Assertion for verifying the *agreement* property. 141

8.1 Failure injection model, modified to inject only node errors that are detected (for a system with seven nodes). 149

List of Tables

4.1 Activation of the fault injection breakpoint. 61
4.2 Outcome of the fault injection experiments. 61

5.1 Distribution of outcomes of fault injection in registers. . . 97
5.2 Error detection coverage estimations (faults injected in registers). 97
5.3 Distribution of outcomes of fault injection in memory. . . 102
5.4 Error detection coverage estimations (faults injected in memory). 102
5.5 Comparison of fault-space sizes (registers). 104
5.6 Comparison of fault-space sizes (memory). 104

6.1 Mapping of component failures to failure modes. 110
6.2 Configuration of the real-time Ethernet network and resulting clock skew. 125
6.3 Node departure and node reintegration latencies (worst case). 126

7.1 Exhaustively verified protocol configurations with respect to safety properties. 143
7.2 Exhaustively verified protocol configurations with respect to liveness properties. 143

List of Abbreviations

CAN	Controller Area Network
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
DMA	Direct Memory Access
DMR	Dual Modular Redundancy
ECC	Error-Correcting Code
EDF	Earliest Deadline First
FCR	Fault Containment Region
FMEA	Failure Modes and Effects Analysis
IMA	Integrated Modular Avionics
LTL	Linear Temporal Logic
MMU	Memory Management Unit
MPU	Memory Protection Unit
RMS	Rate-Monotonic Scheduling

SEU Single Event Upset

SIL Safety Integrity Level

TDMA Time Division Multiple Access

TLB Translation Look-aside Buffer

TMR Triple Modular Redundancy

WCET Worst-Case Execution Time

WCRT Worst-Case Response Time

CHAPTER 1

Introduction

We often depend on a computer system without being aware of its existence. Whether it is our mobile phone or the airplane we're flying, there's frequently a part of our life which we trust directly or indirectly to a computer. Naturally, we expect product developers to weigh the consequences of a failure against the cost of reducing the risk of such an event. Thus, we are willing to pay for reliability and safety along with the functional benefits of a system.

From the designer's viewpoint, dependability and functional features impose conflicting requirements. The constant demand for improved functionality increases hardware and software complexity – a major obstacle to creating dependable systems. Nevertheless, society craves for new products with enhanced customer value. The increased dependence placed on computers – a steady trend in most economic sectors (transportation, health, finance, telecommunication, etc.) – demands strict attention to their reliability, availability, safety and other attributes of dependability.

In critical applications, computers are usually embedded into the devices they control. Users seldom perceive the presence of these computers and their operation is limited to the scope of the application. Though most embedded systems are unlikely to harm anyone, their failure can

sometimes be extremely harmful. A faulty system can cause great human and economic losses in avionics control, air and rail traffic control, telecommunications and industrial applications. Due to the distributed nature of these applications, embedded computer systems are usually distributed as well. Thus, the concerns with faults and errors go beyond a single computer node. Moreover, embedded systems are often expected to function correctly for a number of years, possibly without maintenance or repair. Fault tolerance is fundamental to assure that those systems are trustworthy.

This thesis deals with principles and techniques of fault tolerance for distributed embedded systems. The overall goal is to improve the cost-effectiveness and flexibility of such systems by developing an architectural framework and supporting services which allow both critical and non-critical functions to be executed on the same processor node. The framework provides a model for implementing fault tolerance using a layered approach which combines hardware-, node- and system-layer mechanisms.

The core idea is to ensure that processor nodes can handle a majority of the errors themselves, without any involvement of the other nodes in the system. Thus, the mechanisms at the hardware and node layers should jointly allow a node to detect and recover from errors autonomously. However, even with such mechanisms in place, the possibility of node failures cannot be disregarded completely. System-layer mechanisms are therefore provided to deal with errors that cannot be corrected by the nodes themselves. These mechanisms are also necessary for dealing with errors that occur in the communication network. While all three layers are important for achieving fault tolerance, the main contributions of this thesis focus on the node and system layers.

The hardware layer consists of mechanisms implemented in hardware at the circuit and micro-architectural levels. Techniques such as pipeline flushing and instruction retry can be used for masking transient hardware faults transparently to the software. With the increasing scale of integration, we can also expect that more integrated circuits will utilize on-chip redundancy techniques for tolerating permanent hardware faults, although such techniques are not widely used today. The proposed framework relies on the existence of hardware mechanisms, but assumes that their fault coverage is imperfect, and hence there is the

possibility for hardware faults to affect program execution.

Regarding the node layer, the thesis investigates techniques for building robust operating systems capable of guaranteeing the integrity of tasks in a multithreaded environment. The goals are to facilitate composability within computer nodes, by preventing undesired interactions among software components, and to detach recovery mechanisms from applications, so as to promote reusability of fault tolerance services. One guiding principle is to tolerate, in a comprehensive manner, software and hardware faults affecting application processes.

An existing real-time kernel was extended with the objective of experimentally assessing these techniques. To this end, an experimental platform, with an associated fault injection tool, was developed and used for testing the implementation. Following a methodology for fault removal, which consists in focusing fault injection experiments according to the properties that are to be verified, the tool exposed two vulnerabilities in the kernel extension.

With respect to the system layer, the thesis investigates redundancy management techniques for distributed real-time systems. Two primary goals of a system-layer recovery are to isolate any faulty nodes and to reconfigure the remaining working nodes. Thus, the working nodes must maintain a consensus on the nodes that should, and those that should not, participate in service delivery. This key service is provided by a group membership protocol which serves as a building block for system-layer fault tolerance. The proposed protocol was formally verified using model checking.

The verification of fault tolerance is one of the facets of this thesis. The motivation for this is that mechanisms that provide fault tolerance have the potential to generate severe failure modes when poorly designed, even though they are created for improving system dependability. This means that they should be thoroughly verified using appropriate methods. To this end, fault injection was used for testing the robustness of the kernel extension, whereas model checking was chosen for formally verifying the correctness of the design of the group membership protocol.

The remainder of the thesis is organized in eight chapters. Chapter 2 describes the dependable computing background and sets the architectural framework. The node layer is addressed first, starting with a discussion in Chapter 3 on requirements and techniques for safely integrating

functions in critical environments. Chapter 4 addresses the construction of fault-tolerant operating systems for embedded applications. Chapter 5 focuses on improving the efficiency of fault injection by reducing the number of experiments required for assessing node-layer mechanisms. System-layer issues are discussed in Chapter 6, which proposes methods for distributed redundancy management, and in Chapter 7, which describes the formal verification of the group membership protocol using model checking. Chapter 8 unifies the building blocks proposed in the other chapters by looking into interoperability between fault tolerance layers. Finally, the conclusions are presented in Chapter 9.

CHAPTER 2

The Architectural Framework

This chapter introduces the architectural framework for layered fault tolerance in distributed systems. First, some background to the field of dependable computing is given, followed by a description of the framework and the contributions of the thesis.

2.1 Terminology

Safety can be defined as "a property of a system that it will not endanger human life or the environment" [1]. According to the taxonomy of dependable and secure computing [2], a *system* is the basic entity which interacts with other systems (*i.e.*, hardware, software, humans or the physical world). Systems always interact by providing and/or receiving some service. A system is safety-critical if safety cannot be ensured when it fails to provide correct service.

Product developers must therefore be thorough in addressing the dependability of safety-critical systems. Generally speaking, a system is dependable if one can assure that the frequency and the consequences of its failure are adequate for a particular application. However, assurance and adequacy are often subjective terms. Figure 2.1 shows the dependability tree. The figure was adapted from [2] by including only the

attributes of interest for dependability. The following sections describe the *threats*, *attributes* and *means* to attain dependability.

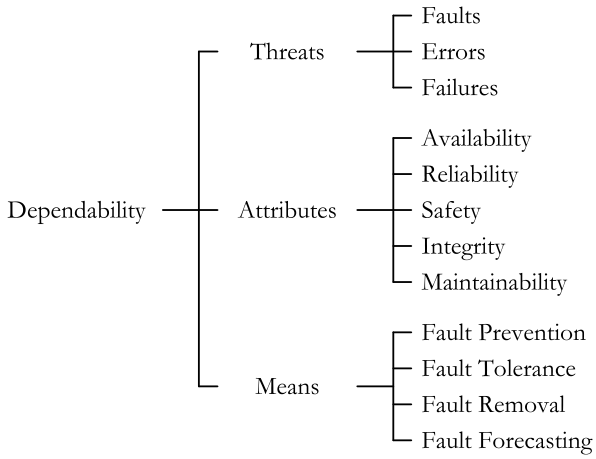

Figure 2.1: The dependability tree.

2.1.1 Faults, Errors and Failures

The *threats* to dependability are faults, errors and failures. The relationship between these threats is:

- A *failure* occurs when the delivered service deviates from what is considered correct.
- An *error* is an incorrect system state that may affect the external behaviour, thereby causing a failure.
- A *fault* is the adjudged or hypothesized cause of an error [2].

Faults can have diverse origins and may be classified into three partially overlapping groups:

- *Development faults* are introduced in the system during the development phase. These include software bugs, hardware production defects, etc.

- *Physical faults* include all hardware faults. These can be caused, for instance, by physical deterioration, design flaws or by external disturbances.

- *Interaction faults* are all faults that originate outside the system. These faults are usually the result of human action or physical interference during the system's use phase.

A service failure occurs when the delivered service deviates from the correct service. The service *failure modes* characterize the different ways in which failures are manifested. Failures can be described in terms of four characteristics:

- The *failure domain* distinguishes between content failures and timing failures. A service can fail in respect to content and timing simultaneously.

- The *detectability* of a failure describes whether or not the service failure is signaled to service users.

- The *consistency* of failures refers to the way users perceive failures. A failure is consistent when all users observe the same failure. If any two users observe different results from a component, then the failure is inconsistent.

- The *consequences* of a failure can range from minor to catastrophic and therefore grade the impact that a failure can have in the complete system.

Faults, errors and failures form a *causality chain*, where a failure of one component may cause a fault in another component. Understanding the failure modes of all components is essential to ensure the cost-effectiveness of fault tolerance mechanisms. Knowing, for instance, the consistency of failures in a distributed system determines the complexity of the communication algorithms. If the nodes can produce inconsistent failures then the Byzantine generals result [3] dictates that $3f+1$ nodes must participate and $f+1$ communication rounds must be completed to tolerate f faulty nodes. On the other hand, if the nodes are known to exhibit only consistent failures, simple majority voting among $2f+1$ nodes suffices to ensure agreement with f faulty nodes.

2.1.2 Dependability Attributes

According to Figure 2.1 there are five main *attributes* of dependability. The *reliability* of a component describes its ability to provide correct service continually, for a given period of time [4]. If X is a random variable which represents the lifetime of a component, then the reliability function for that component is

$$R(t) = P(X > t).$$

The *availability* of a system is also important in many situations. It describes the on-demand probability of correct service. A system that can be repaired after a failure will have, at least, two states: functional and failed. The availability at time t is therefore

$$A(t) = P_{functional}(t).$$

Availability is often represented by a number (*e.g.*, stating that a system is available 99.999% of the time). This number reports the *steady-state availability*, which is the expected fraction of time that the system would be available after an infinite operation time. Thus,

$$A = \lim_{t \to \infty} A(t).$$

Safety describes the absence of catastrophic failures. In addition to the functional and failed states, some systems are able to find a safe state even under faulty conditions. A train which stops in the event of a fire is an example of a system capable of safe shutdown. Airplanes and satellites are examples of systems which do not have this property. The safety function is thus

$$S(t) = P_{functional}(t) + P_{safe\text{-}state}(t).$$

It should be emphasized that we consider the dependability attributes from the probabilistic (or quantitative) point of view. However, it is also viable to use the same concepts qualitatively. Safety, for instance, can be attained without the assignment of probability figures. This is typical in a standard-following industry, where safety is ensured by using state-of-the-art development methods. Doing so ensures that the product is as safe as possible at the time of development.

2.1.3 The Means to Dependability

The *means* to attain dependability consist of methods and techniques to achieve the previously described attributes of dependability. The dependability tree in Figure 2.1 classifies those means into four groups.

- *Fault prevention* is applied during the development phase to prevent the occurrence of faults. Development faults are prevented through good development processes such as software testing, formal methods, hardware design rule checking, etc. Physical faults are prevented by protecting the hardware, usually via radiation shields, increasing the signal-to-noise ratio, etc. Interaction faults are commonly prevented by controlling the users' access to the system.

- *Fault tolerance* techniques are the means to allow a system to provide correct service even when faults occur. Such techniques use diverse forms of redundancy to detect and recover from faults. To identify erroneous conditions, one can use hardware redundancy, software redundancy, time redundancy or information redundancy. The subsequent recovery process relies on the remaining fault-free parts of the system to correct the errors and prevent them from reappearing.

- *Fault removal* is applied during the development and use phases of a system. During development, fault removal consists in verifying the correctness of the system and validating the specification. During the use phase of a system, fault removal is applied either by corrective or preventive maintenance. It usually requires human intervention to replace faulty units or to correct software defects.

- *Fault forecasting* methods provide assurance with respect to frequency and consequences of faults. These methods combine qualitative evaluation of failure consequences, *e.g.*, conducting a Failure Modes and Effects Analysis (FMEA), with quantitative techniques such as Markov models to measure the attributes of dependability. Essentially, qualitative analysis defines, for instance, the *safe states* and quantitative analysis evaluates the probability of remaining in those states.

2.2 System Model

The structural elements of the architectural framework are nodes, networks, services and tasks. A *node* is essentially a computer with a processor, memory and I/O interfaces which provide the access to the network and peripherals (*e.g.*, storage, sensors and actuators). Each node is able to support the execution of multiple tasks.

A *task* is a computer program, which consists of code, data and all the information relevant to its execution. In the operating systems literature a task is referred to as a process or a thread [5]. Tasks are logically grouped into *services* when they collaborate in providing a system function. In a car a service can, for example, implement a brake-by-wire function, whereas in an aircraft a service can implement an autopilot function.

Tasks that jointly provide a service can be distributed across different nodes by using the network for information exchange. Different services are also allowed to exchange information, thus creating dependencies among services. The definition of service is therefore only introduced to reason about the dependability of a given function (which may depend on other functions). Figure 2.2 depicts the structure of the system. It should be noted that a complete system can include several networks of processing nodes, which form independent clusters.

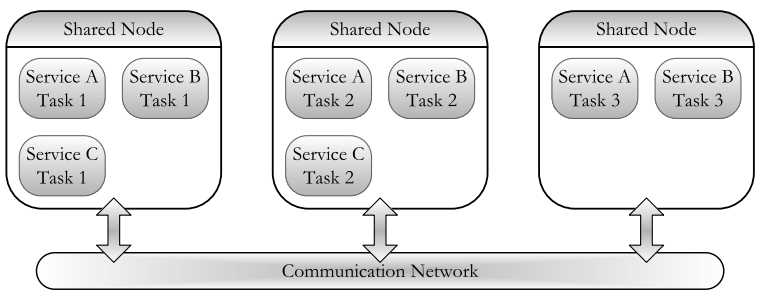

Figure 2.2: Structural elements of the architectural framework.

2.3 Layered Fault Tolerance

In the distributed system depicted in Figure 2.2 fault tolerance can be viewed as a set of mechanisms that provide error detection and recovery. Those mechanisms can be structured into three different layers, based on where they are implemented and what parts of the system they involve:

- *Hardware-layer* mechanisms provide the basic fault tolerance implemented in hardware. Most hardware units include some forms of fault tolerance. Examples are the ability of most microprocessors to detect exceptional conditions (*e.g.*, invalid instructions and erroneous memory accesses), cache protection with parity checks and main memory protection with error-correcting codes (ECCs). Triple modular redundant (TMR) logic at the transistor-level [6] is an example of a more advanced hardware-layer technique.

- *Node-layer* mechanisms are executed locally in a computer node. Additional hardware or software is used to detect errors and, if possible, recover from them. Executing, for example, a task twice allows transient errors to be detected; triplicated time-redundant execution of a task and voting provides effective transient error masking. Other examples of node-layer fault tolerance techniques include checkpointing, watchdog timers, runtime assertions, etc.

- *System-layer* techniques aim at tolerating node failures and communication network failures. They rely on the use of redundant nodes. These can operate in *static redundancy*, which uses majority voting, or in *dynamic redundancy*, which utilizes error detection and reconfiguration.

It is important to realize that these layers are not working in isolation from one another. Fault tolerance mechanisms often require different layers to cooperate. To exemplify, consider a fault in one of the tasks of a brake-by-wire system. A memory access outside its memory address space may be detected at the hardware layer by a Memory Management Unit (MMU). An exception is raised and, at the *node layer*, the exception handling routine can delete the faulty task. This, in turn, causes the node to exhibit a silent failure. At the *system layer* all remaining

fault-free nodes detect the omission and may switch to an alternate braking algorithm which takes into account that one of the wheel nodes is not braking. This allows the system to provide degraded service while remaining in a safe state by preventing the car from moving sideways. This exemplifies a scenario where mechanisms at all layers cooperate to tolerate a fault.

To minimize the cost of fault tolerance, it is important to find an appropriate combination of fault tolerance mechanisms at the different layers, even when there is no explicit cooperation among them. In theory one should try to ensure that *distinct fault tolerance mechanisms don't overlap*, *i.e.*, they should not detect or handle the same faults. This is often difficult to ensure in practice. A second guideline is that *the lower fault tolerance layers should restrict the failure modes exhibited to the upper layers*. This restriction aims at simplifying the fault tolerance mechanisms by allowing only increasingly benign failure modes to be observed at each layer. With respect to the characteristics of the failure modes, signaled failures are more benign than unsignaled failures; consistent failures are more benign than inconsistent failures; and so on.

The second guideline is important since the cost of handling complex failure modes at the upper layers is much higher than detecting and handling them earlier in the *causality chain*. An activated fault causes an error, which may cause a failure; this failure may then cause a fault in another component. Allowing, for instance, nodes to exhibit inconsistent failures requires complex Byzantine agreement algorithms at the system layer. Therefore, a majority of the errors should be handled at the hardware and node layers in order to minimize the likelihood of inconsistent failure modes. Figure 2.3, adapted from [7] and similar to the one portrayed in [8], illustrates the three layers of fault tolerance mechanisms.

Figure 2.3 shows a possible combination of failure modes observed at the different layers. It should be noted that the figure is intended to depict the layers where faults are treated. Thus, the figure does not indicate that development, physical and interaction faults occur at the hardware layer. A fault is assumed to occur anywhere in the system. The fundamental design decisions are where (*i.e.*, at which layer) and how to detect and recover from them.

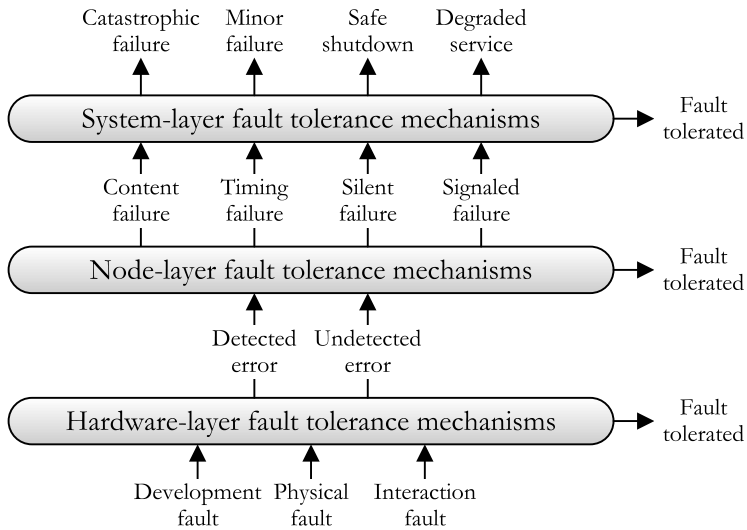

Figure 2.3: Layered fault tolerance for distributed embedded systems.

2.4 Objectives

A hardware fault, such as a Single Event Upset (SEU) in an integrated circuit, may be detected by mechanisms of the system layer by using, for example, a TMR configuration. This is, however, a costly approach to fault tolerance. Mechanisms of the system layer are likely to exclude an entire node from the set of operational nodes (*i.e.*, the *processor-group membership*) in order to prevent the fault from being re-activated. A more cost-efficient combination of fault tolerance mechanisms would first attempt to mask errors at the node layer. This could be achieved with hardware redundancy [9] or with software and time redundancy [10].

This thesis aims to study methods that allow the task to be the elementary unit of failure. However, hardware faults have the potential to disrupt entire nodes. Thus, system-layer mechanisms must also be provided to detect and recover from errors that cannot be handled locally at the nodes. The overall goal of the thesis is to develop and validate a set of mechanisms that support a cost-effective implementation of fault tolerance in distributed real-time systems. Those mechanisms are characterized by the following features:

- Achieve fault tolerance with a layered approach, which combines hardware-layer, node-layer and system-layer mechanisms.
- Ensure strong fault containment within nodes by using robust partitioning among tasks to tolerate software development faults.
- Allow both critical and non-critical functions to be executed on the same processing node.
- Provide redundancy at the node layer to tolerate a majority of the transient hardware faults. The principal concern here is to use mostly software, time and information redundancy, in order to minimize the hardware redundancy and thereby the system cost.
- Provide redundancy and consensus mechanisms at the system layer to tolerate node failures and network failures.
- Support time-triggered execution for critical tasks and event-driven execution for non-critical tasks and recovery mechanisms.

2.5 Main Contributions

The main contributions of this thesis focus on the node and system layers. An overview of the contributions of each chapter is presented below.

- *Chapter 3* examines the requirements of partitioned systems in the light of declassification – a computer security notion that we found useful for specifying partitioning requirements. Moreover, it surveys the existing mechanisms for safely integrating functions in critical environments and presents a probabilistic analysis of the reliability of federated and integrated architectures.
- *Chapter 4* describes SECERN – an approach for implementing partitioning and fault tolerance in real-time kernels. Several fault tolerance mechanisms were implemented as extensions to the μC/OS-II kernel. We developed a fault injection tool with the goal of experimentally assessing these mechanisms and conducted a series of preliminary tests. In addition to the mechanisms implemented in the extended real-time kernel, SECERN includes a lightweight mechanism for checkpointing and rollback recovery of real-time tasks.

The lightweight checkpointing scheme allows applications to save snapshots to main memory while providing them with a service for locking the checkpoint area using memory protection. We used the SPIN model checker to verify the design of this mechanism.

- *Chapter 5* describes a pre-injection analysis technique aimed at reducing the cost of fault injection campaigns. The technique eliminates faults that have no possibility of activation by using knowledge of program flow and resource usage, before any faults are injected. The chapter compares the results of selecting faults randomly with those obtained when using the pre-injection analysis.

- *Chapter 6* proposes a group membership protocol for guaranteeing consistent views of failures and restarts among nodes in a distributed system. The protocol is intended to serve as a building block of distributed redundancy management for time-triggered systems. It provides designers with the ability to configure the reliability of the protocol according to the available resources. Furthermore, the protocol supports inclusion of restarted nodes under the same failure assumptions as exclusion.

- *Chapter 7* describes the usage of the SPIN model checker to formally verify the correctness of the group membership protocol. The chapter specifies the correctness properties and describes the PROMELA models of the protocol and the time-triggered communication channel. Moreover, it presents the results of the exhaustively verified protocol configurations.

- *Chapter 8* unifies the building blocks presented in the other chapters by considering the issue of interoperability between fault tolerance layers. In addition to extending the protocol with support for nodes that execute multiple tasks, the chapter shows that using fail-report instead of fail-silent semantics improves the reliability of the group membership protocol.

CHAPTER 3

Separation of Integrated Functions

Embedded systems have traditionally been implemented by dedicating a computer node to each software component or function. This architecture, which is usually referred to as *federated*, has the advantage of providing clear fault containment boundaries in the design. Each software component executes independently on its own processor and resource sharing is reduced to message passing through a communication infrastructure. The need for fault tolerance is satisfied with the introduction of redundant computer systems as well as redundant networks. This approach makes it simple to contain hardware and software faults in the processor where they originate.

The main drawback of federated architectures is that they lead to a proliferation of hardware as the number of functions grows. The trend to increase the number of subsystems, designed to add new and enhance existing features, demands a large number of microcontrollers – one per major function. The consequence of such designs is the reliability and cost problems currently faced by the manufacturers of embedded systems. The use of many independent computer systems increases the cost of acquisition, space and maintenance, as well as the power consumption. Moreover, a larger number of hardware units leads to a higher fault rate, that may reduce the system's reliability.

To address these problems, there are several initiatives underway aiming at simplifying the sharing of computer resources among different functions in distributed real-time systems. Examples of such initiatives are the development of the Integrated Modular Avionics (IMA) concept [11] and the ARINC 653 standard [12] for the aerospace industry; and the AUTOSAR project [13] launched by the automotive industry. One goal of these initiatives is to integrate different functions and software components into a common hardware platform with few but powerful processing elements. Such *integrated architectures* have a great potential to reduce cost and improve reliability, since they require fewer hardware components than federated architectures. Furthermore, these initiatives favour the integration of Commercial Off-The-Shelf (COTS) software in order to reduce development and maintenance costs.

However, to achieve these improvements, it is necessary to equip the system with robust partitioning mechanisms. Such mechanisms prevent faults in the design of one function from disrupting the operation of other coexisting functions. Robust partitioning mechanisms should therefore ensure fault containment *within nodes* – between different application processes, and between the application processes and the operating system. These mechanisms must prevent processes from writing into each other's memory space – *spatial partitioning* – as well as ensuring that there is no interference in the time domain – *temporal partitioning* –, which encompasses both task scheduling and concurrency control.

This chapter examines the requirements for robust partitioning and identifies existing approaches to provide a computing platform which achieves those requirements. Section 3.1 provides a probabilistic analysis to understand the impact of integrated architectures on a system's reliability. Section 3.2 identifies the requirements for partitioning and Section 3.3 discusses the existing mechanisms to fulfill those requirements. Section 3.4 summarizes the main conclusions.

3.1 Theoretical Motivation

In this section we analyze the effort necessary to assure the reliability of federated and integrated architectures. In our probabilistic analysis, the main assumption is that hardware and software components have a failure rate and that in order to reduce it, the development effort has

to be increased. Furthermore, we assume that the development process follows a standard that assigns criticality levels to components. Setting a lower target failure rate implies a higher criticality level which, in turn, requires a higher development effort.

If a processing node does not contain robust partitioning mechanisms then all its software is required to be developed and certified at the *criticality ceiling* of that node. The criticality ceiling of a node is the criticality level of the most critical software running on it. Since a fault in less critical software can cause a failure of the most critical function, its criticality must be raised to that of the most critical function.

The problem with this approach is that, without partitioning, the failure rate of the less critical software must be decreased to zero in order to ensure that the reliability of the most critical software remains as high as if the two tasks were running on two distinct nodes. In fact, there are only three possibilities to assure the reliability of the most critical software resulting from the integration of less critical software:

1. Reduce the failure rate of the less critical software to zero.
2. Decrease not only the failure rate of the less critical software but also the failure rate of the most critical task to a suitable level.
3. Equip the node with partitioning mechanisms that provide 100% coverage of application errors.

Clearly, there is no process by which we can ensure that the failure rate of software is zero. Decreasing the failure rate of highest criticality software would require even more strict development processes than those available today. Hence the most promising approach is to develop a computing platform with robust partitioning mechanisms that contain faults in the faulty partitions, even if all software is of the same criticality.

It is also viable to combine the different integration possibilities in situations where partitioning exists but is not 100% effective. Moving from a federated architecture to an integrated one will require either very strong partitioning mechanisms or a higher development effort to prevent failures from occurring in the first place. As we will see next, there is a trade-off between development effort and partitioning effort, which allows an integrated system to be built with, for instance, 99% effective partitioning mechanisms (by assuring a slightly lower task failure rate).

It should be emphasized that we are referring to the effectiveness of the partitioning mechanisms in terms of error detection and assume that detected errors are handled correctly. Thus, we define the error detection coverage of the partitioning mechanisms as the conditional probability

$$c = P(\textit{partitioning is not violated} \mid \textit{partition has failed}).$$

If λ partition failures occur every year, then the rate at which such failures result in partitioning violations is $\lambda(1-c)$. Thus, if partitioning mechanisms are only 99% effective ($c = 0.99$) and, for instance, $\lambda = 10^{-6}$ failures/year, partitioning violations would occur at a rate of 10^{-8} per year.

An orthogonal problem to partitioning coverage is the failure rate introduced by the partitioning mechanisms themselves. The partitioning mechanisms must be implemented in either software or hardware. Both approaches have the potential to add new failure modes and increase the existing failure rate. An example would be the failure of a memory protection mechanism which prevented *fault-free* tasks from accessing their own memory spaces. The partitioning failure rate must therefore be reduced to a suitable degree through strict development processes. Clearly, we would like the failure rate of the partitioning mechanisms to be as low as possible and their coverage as high as possible. These are, however, two separate issues.

In following sections we derive continuous-time Markov models to compare the reliability of federated and integrated architectures. The goal is to compare the two design alternatives with regards to their resilience to hardware and software faults. We consider two different benchmarks in this analysis: 1-out-of-n-resilient systems and 2-out-of-n-resilient systems.

Definition 1. *A system is said to be f-out-of-n-resilient if it can can tolerate the failure of any f components from a total of n components. For short, we call these systems f-resilient unless n is relevant.*

The rationale for using 1- and 2-resilient systems as benchmarks is to capture the non-functional requirements of safety-critical systems. Such systems are designed to compensate for errors by having enough redundancy to mask errors or to enter a degraded mode of operation in the event of a failure.

However, if we consider 0-resilient systems, which cannot tolerate the failure of any component, it is simple to draw the conclusion that federated architectures are less reliable than the integrated approach. By using less hardware, the overall hardware failure rate of integrated systems is lower. Assuming that the software is the same in both architectures, the resilience to software failures is the same (no failures are tolerated). Thus, we turn to studying 1- and 2-resilient systems, for which less is known *a priori*. We begin by modeling federated and integrated systems with respect to hardware failures.

3.1.1 Modeling Hardware Failures

We consider the problem of integrating two or more tasks (software components), which were previously granted their independent hardware units, into a single one. These tasks compose a 1-resilient or a 2-resilient system – we will discuss both cases. The symbol λ_{hf} denotes the failure rate of each hardware unit in a federated system, which we assume to be constant during the useful life period.

In federated systems, each task has one dedicated hardware unit. Each hardware unit is a Fault Containment Region (FCR) and there are n such units, failing at a λ_{hf} rate. Figures 3.1 and 3.2 show the state transition diagrams of 1- and 2-resilient federated systems, respectively, which have the following states:

State 0 – The n hardware units are functioning correctly;

State 1 – One hardware unit has failed and the remaining $n-1$ are functional;

State 2 – A second hardware unit has failed (a 2-resilient system tolerates the second failure whereas a 1-resilient system does not);

State F – This state represents a system failure (the third failure in a 2-resilient system or the second failure in a 1-resilient system).

Let $R(t)$ denote the system's reliability, *i.e.*, the probability that the system is functional in $[0, t]$. We derive the reliability of the two federated systems by finding $P_0(t)$, $P_1(t)$ and $P_2(t)$ in the Markov model of Figure 3.2, where $P_S(t)$ is the probability of being in state S at time t. The reliability, regarding hardware failures, of the 1-resilient federated

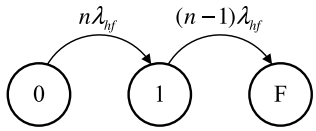

Figure 3.1: State transition diagram, regarding hardware failures, for a 1-out-of-n-resilient federated system.

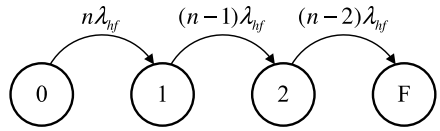

Figure 3.2: State transition diagram, regarding hardware failures, for a 2-out-of-n-resilient federated system.

system is $R_{hf\text{-}1r}(t) = P_0(t) + P_1(t)$ and the reliability of the 2-resilient system is $R_{hf\text{-}2r}(t) = P_0(t) + P_1(t) + P_2(t)$. From Figure 3.2 we obtain the transition rate matrix

$$Q = \begin{bmatrix} -n\lambda_{hf} & n\lambda_{hf} & 0 & 0 \\ 0 & -(n-1)\lambda_{hf} & (n-1)\lambda_{hf} & 0 \\ 0 & 0 & -(n-2)\lambda_{hf} & (n-2)\lambda_{hf} \\ 0 & 0 & 0 & 0 \end{bmatrix}.$$

We know that $\bar{P}'(t) = \bar{P}(t) \cdot Q$ and $\bar{P}(0) = \begin{bmatrix} 1 & 0 & 0 & 0 \end{bmatrix}$, so we obtain the system of differential equations

$$\begin{aligned} P_0'(t) &= -n\lambda_{hf}P_0(t), \\ P_1'(t) &= n\lambda_{hf}P_0(t) - (n-1)\lambda_{hf}P_1(t), \\ P_2'(t) &= (n-1)\lambda_{hf}P_1(t) - (n-2)\lambda_{hf}P_2(t), \\ P_F'(t) &= (n-2)\lambda_{hf}P_2(t), \end{aligned}$$

which can be solved by applying the Laplace transform. We omit this step and present the reliability functions of the federated systems:

$$R_{hf\text{-}1r}(t) = (1-n)e^{-n\lambda_{hf}t} + ne^{-(n-1)\lambda_{hf}t}, \tag{3.1}$$

$$R_{hf\text{-}2r}(t) = \frac{n^2-3n+2}{2}e^{-n\lambda_{hf}t} + n(2-n)e^{-(n-1)\lambda_{hf}t} + \frac{n(n-1)}{2}e^{-(n-2)\lambda_{hf}t}. \tag{3.2}$$

In integrated systems, multiple tasks share the same hardware unit, which is vulnerable to failures – each hardware unit is a FCR with respect to hardware faults. We analyze two alternative integrated systems: one where all tasks share a hardware unit with no redundancy and one

where the hardware unit and the tasks are replicated using Dual Modular Redundancy (DMR) with perfect error detection. Figures 3.3 and 3.4 show the state transition diagrams for the two integrated systems. The symbol λ_{hi} denotes the failure rate of each hardware unit.

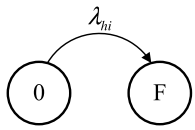

Figure 3.3: State transition diagram, regarding hardware failures, for a 1- or 2-resilient integrated non-DMR system.

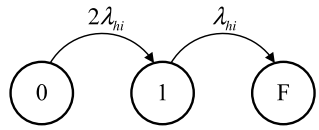

Figure 3.4: State transition diagram, regarding hardware failures, for a 1- or 2-resilient integrated DMR system.

The integrated non-DMR system has an exponentially distributed reliability

$$R_{hi}(t) = e^{-\lambda_{hi}t} \tag{3.3}$$

and the integrated DMR system's reliability can be obtained by replacing n with 2 in Equation (3.1), giving

$$R_{hi\text{-}dmr}(t) = 2e^{-\lambda_{hi}t} - e^{-2\lambda_{hi}t}. \tag{3.4}$$

The plots in Figure 3.5 compare, using Equations (3.1) through (3.4), the reliability of federated and integrated systems with respect to hardware failures.

The first conclusion one can draw from Figure 3.5 is that the reliability of federated systems decreases substantially with the number of hardware units. Increasing the number of hardware units from 5 to 10 leads to more than a three-fold increase in unreliability $(1 - R(t))$, both for 1-resilient and 2-resilient federated systems, over the considered period of time. Hence, integrated architectures are a promising alternative by reducing the number of hardware parts.

The second and perhaps most important conclusion is that integrated architectures are not beneficial in all situations. It is only when the number of hardware units exceeds a certain threshold – between 5 and 10 – that we can benefit from integration. This number has already been surpassed by the industry as there can be as many as 70 processors in a

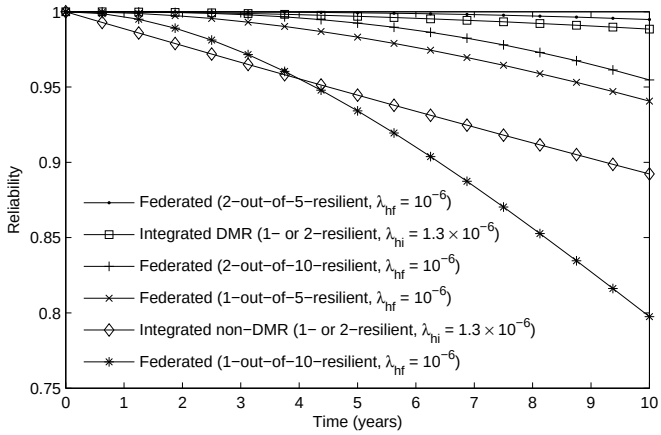

Figure 3.5: Comparison of federated and integrated systems regarding hardware failures.

high-end road vehicle and 50 in a modern airplane, with the consequent penalty in safety and reliability.

Third, considering the 2-resilient federated system, a non-DMR integrated system is not competitive with respect to reliability. In other words, there is a price to pay – in reliability – for using less hardware. Hence, structural redundancy is needed to protect integrated systems against hardware failures. As we can see from the plot of the integrated DMR system, redundancy helps in providing a similar level of reliability to that of the most resilient federated system.

Sensitivity of Integrated Systems to Hardware Parameters

Figure 3.5 compares the different designs when the hardware failure rate of integrated systems is 30% higher than the assumed 10^{-6} failures/h for federated systems. There are several reasons for this. For one, integrated systems require more powerful microcontrollers, built using more recent manufacturing processes. This makes the hardware more sensitive to both transient and permanent faults [14]. Moreover, those microcontrollers are likely to be more complex, potentially increasing the failure rate. Finally, since several tasks are running on a processor, its load is likely greater – a factor which is known to increase fault activation [15].

However, there is no evidence that the failure rate will only be 30% higher. Figure 3.6 shows how the reliability of the two integrated systems (DMR and non-DMR) is affected when the failure rate increases to 2.0×10^{-6} failures/h.

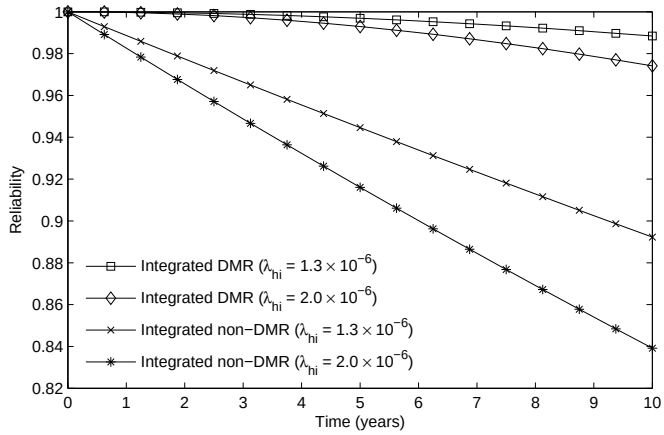

Figure 3.6: Sensitivity of integrated systems to hardware failure rate.

One can draw the conclusion, from Figure 3.6, that the hardware failure rate is one of the determinant factors for the resilience of integrated architectures. When the failure rate increases by ∼54%, from 1.3×10^{-6} to 2.0×10^{-6} failures/h, the unreliability of the system increases by approximately the same factor.

3.1.2 Modeling Software Failures

This section focuses the reliability assessment regarding software failures. We apply continuous-time Markov modeling [4] to compare the federated architecture with the integrated architecture. We explicitly use software failure rates as transition rates in our models. Most software reliability modeling techniques [16] use software failure rates to predict reliability and number of faults (*i.e.*, bugs) in software systems. Moreover, it is commonly assumed that the software failure rate is proportional to the number of faults in the system. Under these assumptions, predicting the number of software faults can be done before software deployment.

An approach is to use field failure data from previous releases or products [17].

However, the statistical approach to software reliability assessment is not always used in practice. The DO-178B [18] standard for avionics software development does not require the assignment of a failure rate for software of any level of criticality. Instead, this approach aims to assure a high level of confidence that the software is free from faults. This is usually achieved by using the best existing systems engineering practices. Reference [19] scrutinizes the differences between the *statistical* and the *perfectionist* approach, and clarifies the relationship between statements of software failure rates and about software correctness.

We assume the existence of a software failure rate in order to reason about the dependability of the two architectures. Furthermore, we assume that reducing the failure rate of a software component implies setting a higher criticality level (and therefore a greater development effort). The IEC 61508 [20] international standard for functional safety defines four Safety Integrity Levels (SILs) for safety-related functions. To each range of failure probabilities corresponds an integrity level. Lower probabilities of failure (specified either in terms of probability of failure per hour or probability of failure on demand) impose higher integrity levels. (Note that the converse is not true, *i.e.*, development at a certain integrity level does not guarantee the target failure rate.) Under these assumptions we can relate the development effort to the software failure rate. Thus, we can compare the development effort in the two architectures by comparing the failure rates of their components.

In federated systems, each hardware unit is a FCR also for software failures. We are assuming that all tasks are functionally independent and that the only pathways for fault propagation result from shared resources – inexistent in the modeled federated systems. There are n tasks which have a failure rate of λ_{sf} failures/h. Figures 3.7 and 3.8 show the state transition diagrams of 1- and 2-resilient federated systems, respectively, which have the following states:

State 0 – The n tasks are functioning correctly;

State 1 – One task has failed and the remaining $n-1$ are functional;

State 2 – A second task has failed (a 2-resilient system tolerates the second failure whereas a 1-resilient system does not);

State F – This state represents a system failure (the third software failure in a 2-resilient system or the second software failure in a 1-resilient system).

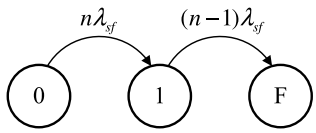

Figure 3.7: State transition diagram, regarding software failures, for a 1-out-of-n-resilient federated system.

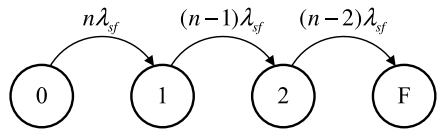

Figure 3.8: State transition diagram, regarding software failures, for a 2-out-of-n-resilient federated system.

The state transition diagrams for federated systems concerning software failures are equal to those in Figures 3.1 and 3.2, derived for hardware failures. Thus, Equations (3.1) and (3.2) give us also the reliability of 1- and 2-resilient federated systems regarding software failures, by replacing λ_{hf} with λ_{sf}. This similarity between the effects of software and hardware faults made it possible for airplane and car manufacturers to assume that software is fault-free; they could implicitly take software faults into account by assuming a conservative hardware failure rate and obtain safe reliability estimates for the entire system.

Unfortunately, the same cannot be said for integrated architectures. To enable resource sharing among multiple tasks, robust partitioning mechanisms should enforce temporal and spatial protection. There are, therefore, two new parameters which influence the resilience of integrated systems: the software failure rate introduced by the partitioning mechanisms themselves, denoted by λ_{pm}, and their coverage.

The state-transition diagrams for integrated systems are shown in Figures 3.9 and 3.10. They have the same states as federated systems but there are direct transitions to the failed state: a fault which is not contained by the partitioning mechanisms (with probability $1 - c$) or a failure of the partitioning mechanisms.

One can immediately draw the conclusion that the reliability functions, concerning software, of integrated and federated systems are equal when the coverage of the partitioning mechanisms is perfect ($c = 100\%$) and the failure rate of the partitioning mechanisms is zero ($\lambda_{pm} = 0$).

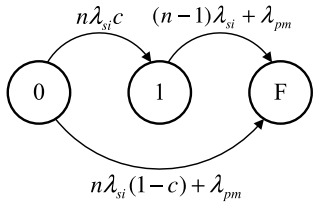

Figure 3.9: State transition diagram, regarding software failures, for a 1-out-of-n-resilient integrated system.

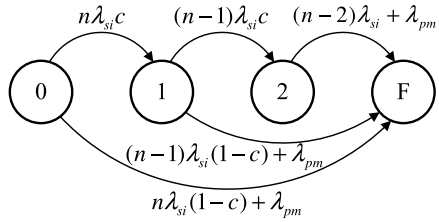

Figure 3.10: State transition diagram, regarding software failures, for a 2-out-of-n-resilient integrated system.

Thus, if the software failure rate of tasks is the same, integrated systems can only be less resilient than federated systems. Since software faults are design faults, redundancy (*e.g.*, using a DMR configuration) does not increase the reliability. To achieve that, one would have to consider decreasing the software failure rate of the tasks or using design diversity – both options are costly and demand a greater development effort. The alternative endorsed by ongoing efforts such as AUTOSAR and IMA is to place the development effort into designing reusable platforms that provide robust partitioning.

Sensitivity of Integrated Systems to Software Parameters

The same technique used to determine $P_0(t)$, $P_1(t)$ and $P_2(t)$ in the Markov models of the preceding section (hardware failures) can be applied to Figures 3.9 and 3.10. We obtain the reliability of 1-resilient integrated systems with respect to software failures

$$R_{si\text{-}1r}(t) = (1 - nc)e^{-(n\lambda_{si}+\lambda_{pm})t} + nce^{-((n-1)\lambda_{si}+\lambda_{pm})t} \tag{3.5}$$

and the reliability of 2-resilient integrated systems regarding software failures

$$R_{si\text{-}2r}(t) = \frac{n(n-1)c^2 - 2nc + 2}{2}e^{-(n\lambda_{si}+\lambda_{pm})t} + nc(1 - nc + c)e^{-((n-1)\lambda_{si}+\lambda_{pm})t} + \frac{n(n-1)c^2}{2}e^{-((n-2)\lambda_{si}+\lambda_{pm})t} \tag{3.6}$$

To understand the sensitivity of integrated systems to the coverage of partitioning mechanisms we fix $\lambda_{pm} = 0$, $\lambda_{si} = 10^{-6}$ failures/h and compare systems with 5 and 10 tasks with a coverage of 99% and 95%. Figures 3.11 and 3.12 show the resulting reliability curves for 1- and 2-resilient systems, respectively.

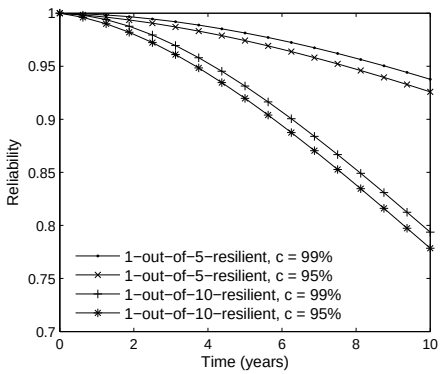

Figure 3.11: Sensitivity of integrated systems to the coverage of partitioning mechanisms (1-resilient systems with $\lambda_{si} = 10^{-6}$ failures/h and $\lambda_{pm} = 0$).

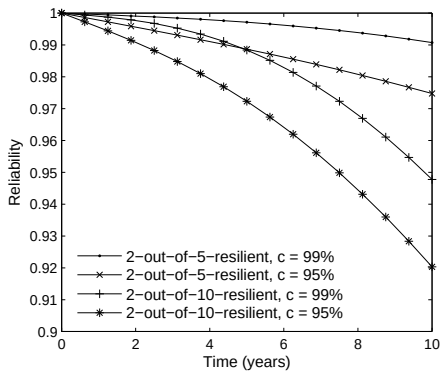

Figure 3.12: Sensitivity of integrated systems to the coverage of partitioning mechanisms (2-resilient systems with $\lambda_{si} = 10^{-6}$ failures/h and $\lambda_{pm} = 0$).

We can conclude from Figure 3.11 that 1-resilient systems are somewhat sensitive to variations of the coverage of partitioning mechanisms – a 4% decrease in covered faults results in a 5-20% increase of unreliability. Regarding Figure 3.12, we can draw the conclusion that 2-resilient systems are very sensitive to the coverage factor. The same 4% decrease in coverage leads to an increase of ∼50% in unreliability for 2-out-of-10-resilient systems and a ∼160% increase in unreliability for 2-out-of-5-resilient systems. The greater sensitivity of 2-resilient systems is due to the fact that c appears as a squared factor in Equation (3.6).

The other parameter is λ_{pm} – the failure rate potentially introduced by faults in the software designed to detect errors, isolate faulty partitions and recover the system through rollback, rollforward or compensation. To understand the impact of this parameter we fix $c = 99\%$, $\lambda_{si} = 10^{-6}$ failures/h and compare systems with 5 and 10 tasks with $\lambda_{pm} = 10^{-7}$ failures/h (an order of magnitude lower than each individual task) and

$\lambda_{pm} = 10^{-6}$ (the same failure rate as one task). Figures 3.13 and 3.14 show the sensitivity of integrated systems to the software-related failure rate of partitioning mechanisms.

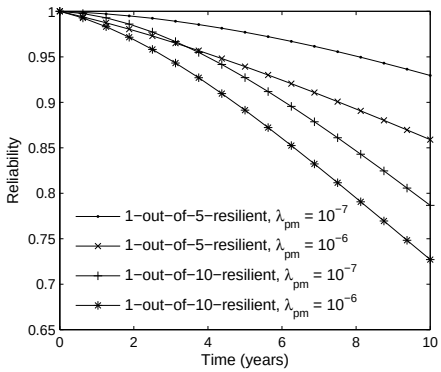

Figure 3.13: Sensitivity of integrated systems to the failure rate of partitioning mechanisms (1-resilient systems with $\lambda_{si} = 10^{-6}$ failures/h and $c = 99\%$).

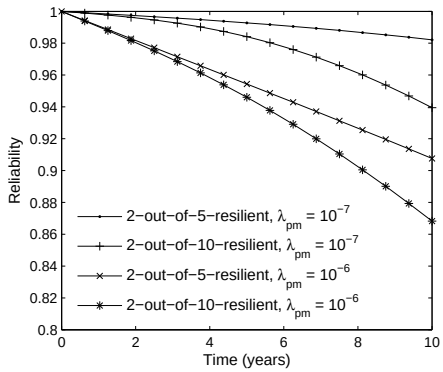

Figure 3.14: Sensitivity of integrated systems to the failure rate of partitioning mechanisms (2-resilient systems with $\lambda_{si} = 10^{-6}$ failures/h and $c = 99\%$).

Figure 3.13 shows that when λ_{pm} increases from 10^{-7} to 10^{-6} failures/h, the unreliability of the system increases by 25-100% in 1-resilient systems and by 130-200% in 2-resilient systems, depending on the number of tasks, over the considered period of time. This is a significant impact on the system's reliability, justified by the fact that a failure of the partitioning mechanisms may disrupt all partitions on a hardware unit. Hence, a great development effort must be placed into avoiding design faults in platforms supporting integrated systems.

3.2 Requirements for Partitioning

So far, we have discussed partitioning in abstract terms. We see it as a set of mechanisms that behaves like a firewall, preventing faults from propagating among components. We have implicitly assumed that tasks are executing according to a model and that partitioning would be unnecessary if the tasks always behaved according to this model. The task

model may include, for instance, a deadline which must be met in every execution. Furthermore, one may schedule tasks according to their priorities and design them to call a `DelayUntil` primitive to release the CPU as soon as their computations are finished. When all tasks follow this model we are trusting them to complete their execution and call the `DelayUntil` primitive on time.

The main reason for using partitioning is that the arguments collected during the certification of one component only assess its ability to provide correct service – which includes calling the `DelayUntil` primitive on time. For cost reasons, it would be ineffective to gather the same amount of dependability arguments for a non-critical function as for a critical function. Thus, the `DelayUntil` primitive must be replaced by a stronger mechanism. One such mechanism should allow the critical task to provide correct service even if the non-critical task crashes or enters an infinite loop.

The main requirement for partitioning is to ensure that fault-free partitions are always able to provide correct service, regardless of which software executes in other partitions. However, this requirement may be too strong, since it would be necessary to take into account all possible program behaviors to ensure that a partition remains fault-free in all cases. A thorough argumentation on the informal requirements for partitioning, as well as a comparison between partitioning and computer security, can be found in an excellent report by J. Rushby [21].

In recent years, the relation between dependability and computer security has been clarified. We can view dependability and security as two distinct concepts which share common attributes and are often interdependent [2]. Researchers have realized that many systems are not secure unless they are dependable, and *vice versa*. Examples of such systems are network firewalls, which must be highly available to be secure, and computer systems in power plants, which must be secured against malicious interaction faults to be dependable.

Security research can make a relevant contribution to the partitioning problem, mainly with modeling techniques and with the requirements specification. Security is often concerned with controlling the *information flow* among tasks:

- *Confidentiality* is a system's ability to prevent the flow of sensitive information to unauthorized partitions.

- *Integrity* is the ability to protect sensitive information from being modified by unauthorized partitions.

The dependability field is mostly concerned with integrity rather than confidentiality. Consequently, one of the major goals of partitioning is to assure the integrity of partitions. In this context, fault propagation is the type of information flow that partitioning aims to prevent. Thus, the development of partitioned systems can directly benefit from the research in the field of computer security.

Conventional federated architectures assure the integrity of the different subsystems by using dedicated processing nodes – a basic form of partitioning. When those processing nodes are interconnected and cooperate via message exchange, the network is a potential path for fault propagation. Thus, federated architectures require some mechanisms to provide partitioning among nodes. Examples of such techniques are the electrical isolation of hardware components at the hardware layer; bus guardians at the node layer to prevent untimely network accesses; and redundancy management mechanisms at the system layer to detect and isolate faulty nodes from the system.

The conventional partitioning mechanisms are also necessary when building integrated architectures. A permanent hardware fault in a node, for instance, should not propagate to other processing nodes. Additionally, however, integrated architectures demand finer-grained partitioning mechanisms at the node layer. These mechanisms should ensure the integrity of individual tasks or, possibly, groups of tasks running on the same node.

The partitioning mechanisms should, ideally, provide a level of fault containment among tasks comparable to that of federated architectures. One way to model this is to identify the externally visible behavior of the system when all tasks are running in isolation [22]. When moving the same system to an integrated architecture it is required that no new behaviors are introduced. This notion of *noninterference* [23] was originally introduced by security researchers.

Noninterference is an information flow policy which specifies that the actions of an entity (*e.g.*, a user, a task or a database) should have no observable effects on other entities. Checking that such a property holds throughout the execution of all tasks requires a clear definition of "observable effects" and a clear model of the possible "actions". A formu-

lation of noninterference that can be helpful for the verification process is based on the *determinism* of the observations [24]. Under this formulation the actions of a high-level entity are deemed nondeterministic. If the observations of the lower-level entities are deterministic then they are independent of higher-level entities. The direction of noninterference can be reversed to assure that there is no information flow in neither direction.

A similar line of thought is applied in [25], where task isolation is achieved by ensuring *invariant system performance*. The formulation of invariant performance guarantees that the software components' execution after integration is exactly the same as it was in isolation. A system with invariant performance is required to (i) execute the operations of each task at precisely defined times (unvarying schedule) and (ii) ensure noninterference.

However, for most applications invariant performance is too restrictive to be useful – one must be able to predict which task is executing during each processor cycle. Simple noninterference properties are also too strong and restrictive for real-world applications. This follows from the common notion of task deadline in hard real-time systems: a task should always complete before its deadline. Invariant performance implies that tasks are always completed *exactly at* their deadline; noninterference implies that a task's completion is totally independent of any other tasks. In other words, there would be information flow from a failed task to other tasks if the resulting spare cycles could be reclaimed by those tasks.

For these reasons, well-established scheduling algorithms such as Earliest Deadline First (EDF) and Rate-Monotonic Scheduling (RMS) [26] are not valid options when ensuring invariant performance or plain noninterference. In fact, most real-world approaches to partitioning at the node layer have used time-triggered cyclic schedules. This rules out, for instance, the possibility of integrating low-criticality background tasks which use the spare processor cycles to provide additional features (*e.g.*, monitoring tasks). We therefore require more flexible policies than noninterference to apply event-driven scheduling in partitioned systems.

There are several advantages in using event-driven scheduling instead of time-triggered approaches, even though time-triggered scheduling facilitates the verification process in many ways [27]. Using event-driven

scheduline, sporadic and aperiodic tasks are favoured with more efficient resource utilization; the average response time of such tasks is also improved by avoiding event-polling waiting times. Furthermore, there is usually no reason to prevent tasks from early completion – the real problem is to ensure that they never complete too late. Thus, the models derived from noninterference must be extended with integrity policies that control the information flows instead of ruling them out.

These issues have also been identified in the field of computer security, where there is an ongoing effort to devise less restrictive information flow policies [28]. For practical reasons information is often disclosed intentionally. Web servers, for instance, reveal the family/version of their software without compromising any sensitive information. The notion of *declassification* [29] has been proposed to model those intentional flows. Information is declassified or downgraded by providing intentional leaks. The resulting declassification channels are then expected to be *robust*, *i.e.*, only the intended information should be released.

According to [30], declassification has four dimensions that describe intentional information release: *what* can be released, *when* and *where* can it be released, and *who* can release it. Since we are focusing on the integrity of partitions (not their confidentiality) our concern is that information might change due to faults in other partitions, rather than it being released. Thus, for partitioned systems, the four dimensions describe *what* information can be modified, *when* and *where* can it be modified, and *who* can modify it. These dimensions can be used to characterize the requirements of partitioning mechanisms:

- *Spatial partitioning* mechanisms should ensure the integrity of the information in each partition, *i.e.*, memory address space, storage space, messages on the network, private I/O devices, etc. Pure noninterference is often required for information such as private data structures or code. The communication network, however, exemplifies a structural element that is shared among several partitions. The access to the network is therefore declassified in order to allow several partitions to communicate. In doing so, the system designer must carefully specify *when* may each partition access the network (*e.g.*, using time-triggered scheduling).

- *Temporal partitioning* mechanisms should ensure that the response

time requirements of non-faulty partitions are satisfied. This indicates that the interference among partitions, in the time domain, must be controlled, rather than ruled out. There are numerous issues that may arise when using, for example, memory caches and Direct Memory Access (DMA) for copying memory. Furthermore, recovery procedures consume some time when a partition error is detected. The response time analysis must therefore take into account faulty scenarios.

3.3 Mechanisms for Partitioning

This section identifies existing approaches to fulfill the requirements for partitioning. We examine the topics of spatial and temporal partitioning separately.

3.3.1 Spatial Partitioning

In multitasking environments, preventing the tasks from writing into each other's memory space is fundamental. The concern is that, if the memory spaces are not isolated, a failed task may hinder the correct execution of other tasks. Closing this pathway for fault propagation is an issue for spatial partitioning mechanisms. In computer architecture and operating systems literature [31, 32] this is usually referred to as memory protection. It can employ either software, hardware or a mix of both to allocate memory to different processes and ensure that they cannot access memory outside their own areas.

The most common method for memory protection is *paging*. In the simplest version of paging the memory is divided into fixed-size frames. Each process page is allowed to occupy any such frame. Additionally, it is possible for every process to access its memory through a contiguous virtual address space which aggregates all pages. The page size determines the amount of *internal fragmentation*, *i.e.*, the memory wasted when a process page is smaller than the fixed page size. Small page sizes are often desired in order to reduce internal fragmentation. However, since the operating system must maintain the information of which pages belong to a process in a *page table*, a small page size results in more overhead due to large process page tables. A common page size is 4KB.

However, most memory protection designs allow multiple (simultaneous) page sizes to avoid the drawbacks of fixed-size pages. Depending on the actual design the page sizes can be, for instance, powers of 4KB (4, 16, 64, etc.).

Another common memory protection scheme is *segmentation*. Segmentation allows programs to allocate unequally sized portions of memory in the form of segments. The segments may also be dynamic in order to handle growing/shrinking data structures. Since processes may occupy several segments a memory access must specify the segment number and an offset within that segment. This scheme has the advantage of reducing the internal fragmentation at the expense of increasing the complexity of many aspects of the operating system's design.

Currently, there are numerous techniques to implement memory protection, which can be broadly classified into two categories: software techniques and hardware techniques (which most often require software control).

Hardware Techniques

Hardware-based memory protection by means of a MMU is an established feature of desktop and server computers [32]. The MMU is a gateway between processor and memory with many important features. It provides *memory protection* by restricting programs to memory accesses within specified areas. When a program accesses another program's memory area, an exception is raised and the control is handed over to the operating system, which may then stop the erroneous (or malicious) program. *Address translation* allows all programs to have the same *logical* address space, whilst their code and data can be located in convenient *real* addresses.

A key component in providing efficient address translation is the Translation Look-aside Buffer (TLB) – a small and very fast cache which holds recently used entries. Each entry contains the physical page number, the real page number and the permissions of the currently running process (read, write and execute rights for both supervisor and user mode). Each entry may additionally include a dirty bit to identify pages which have been written to, the caching policy for the page, and other information which depends on the actual hardware. Whenever an effective address is matched against a TLB entry (cache hit), the page number

is found immediately and the real address can be formed. Otherwise, if a TLB miss occurs, the table must be updated with an entry for the missing page. In some architectures this process is done entirely by the hardware whereas in others an exception is generated, requiring the operating system's software to update the TLB. In any case, TLB misses incur severe performance penalties.

Some systems make use of *virtual memory* – the ability to store some pages in memory and others is disk [31]. When a TLB entry describes a virtual page which has no physical memory allocated, a *page fault* will be signaled by the MMU. The operating system must then handle this request by loading the appropriate page from disk (possibly by swapping out an existing page and saving it to disk). A page fault may also indicate a faulty process which should be stopped. Virtual memory allows every process to run as if the entire memory was contiguous and unlimited. In a 32-bit processor, for instance, each process is able to address 4GB of memory.

Furthermore, some MMUs offer *cache control* mechanisms. This feature allows the operating system to decide whether or not a page is cacheable. It may also be possible to specify that a page should always be kept in cache. This feature may prove useful in real-time systems by retaining the pages which belong to critical tasks in the cache. The response times of these tasks will be deterministic as there will be no cache misses. Nevertheless, memory caches are usually small when compared to the size of main memory. Thus, cache entries should only be locked when the cost of cache misses is not acceptable.

Although virtual memory by means of the MMU is the *de facto* method for memory protection in desktop and server computers, it is less frequently used in embedded real-time systems. In order to reduce the cost and the energy consumption of the CPU, most embedded microcontrollers lack the hardware support for advanced memory management. Furthermore, MMUs impose time overhead and make it more difficult to determine the Worst-Case Execution Time (WCET) of programs.

However, Freescale's MPC5554 [33] is an example of a recently introduced embedded microcontroller equipped with an MMU which provides, among other capabilities, memory protection. For real-time applications, it is possible to effectively disable address translation (and virtual memory) by using a one-to-one mapping between virtual and real addresses.

Moreover, one can ensure that the TLB always contains the page entries of the process that is currently running. This approach brings determinism and low-overhead to memory accesses, while ensuring that memory access violations are detected. If all pages of a process have an entry in the TLB and a TLB miss occurs, then the process is accessing memory outside its own area.

Some embedded microcontrollers such as Freescale's MPC565 [34] and ARM's ARM946E-S [35] are equipped with a Memory Protection Unit (MPU). An MPU does not translate virtual addresses but provides basic memory access control in a way similar to an MMU. Depending on the actual processor model, the address space can be partitioned into at most eight segments of data and eight segments of code. Every segment has a minimum size of 4KB and can grow, by a power of 2, up to 4GB. This may lead to internal fragmentation and, consequently, wasted memory.

MPUs can be useful in embedded real-time applications since they only provide simple memory protection. MMUs, on the other hand, provide many other features designed for high average throughput that, when enabled, make worst-case execution time estimations unacceptable. However, the number of segments supported by common MPUs is lower than the number of TLB entries in common MMUs. This makes it possible to use an MMU as an MPU. The converse is not possible, as MMUs provide other useful features such as cache control. In fact, even virtual address translation can be of use for fault tolerance purposes. A viable approach is to store multiple copies of data in memory and switch transparently to another physical address space when an error is detected.

Software Techniques

A number of software techniques to prevent unauthorized memory accesses has been devised in the past. Some involve the use of run-time checks to ensure that every memory access is safe, whereas others aim at proving safety via static code analysis. Generally speaking, software techniques for partitioning are all which do not use specialized hardware and attempt to provide the same level of memory protection.

One such technique is called *intended segment analysis* [36]. This technique provides segment protection by inserting run-time checks before memory accesses to detect segmentation violations. The run-time

checks are inserted at compile-time by an automatic tool which is independent of the programming language.

In order to detect all segmentation violations it would be required to place a run-time check for each memory reference, with a few trivial exceptions (sequential instruction fetches starting in a valid point, constant pointers, etc.). However, the performance of this baseline method, also evaluated in [36], is quite poor. The execution time overhead was found to average 60%, the code size overhead was, on average, 6% and the energy consumption overhead was estimated to an average of 48%.

Consequently, the authors devised a set of optimizations, derived from *compiler theory*, which improved the performance dramatically. The optimizations include, for instance, checking only the reference which *dominates* multiple accesses to the same address (subsequent accesses do not require checking). The optimized solution was found to have an average overhead of 0.72% in execution time, 3.6% in code size and 0.44% in energy consumption. One noteworthy point is that, while the average code size overhead is 3.6%, one of the eight benchmarks yielded an overhead of 25%.

Another method for software memory protection is to use safe programming languages such as Cyclone [37]. Cyclone is a dialect of C which imposes some restrictions to ensure that all operations are safe. The restrictions include ensuring safe type-casts and unions, mandatory pointer initialization, inserting run-time bound checks to prevent segmentation faults, etc. In order to regain the restricted features provided by standard C some extensions are provided by Cyclone.

The Cyclone compiler performs a static code analysis to ensure safety. Under certain conditions the static analysis cannot guarantee that the code is safe but the insertion of run-time checks will ensure the detection of all errors. If neither the static analysis nor the run-time checks can ensure safety, the compiler will reject the program – which may be written in standard C. The programmer then needs to rewrite the program in order for the compiler to verify its safety. The authors estimate that, if the original C code is safe, porting legacy code to Cyclone requires 8% of the code to be modified [38, 37].

The overhead of using the Cyclone compiler depends on the number of run-time checks that are required to ensure safety. This number depends on the performance of the static analysis in avoiding the run-time checks.

When comparing the execution time of the original C code to the Cyclone code, the estimated overhead was on average 30%, with a maximum of 150%. Conceptually speaking, it would be possible to optimize the run-time checks with techniques such as the ones used in the intended segment analysis method. Nonetheless, there is a cost associated to porting legacy code to Cyclone, which is often impractical for the industry to support.

A similar approach is taken by the Control-C programming language [39], which is a restricted subset of C designed to guarantee memory safety without run-time checks. The semantic restrictions required by Control-C (*e.g.*, strong typing, restricted array operations and mandatory pointer initialization) allow the compiler to verify the code entirely by static analysis, thereby avoiding run-time bounds checking and garbage collection. Although Control-C has the same drawback as Cyclone – porting legacy code is expensive and only practical if the original code is written in C – there is no run-time overhead. Furthermore, Control-C may conceptually be used as a tool which checks C programs that are then compiled and linked with standard C compilers.

Safe-C [40] and CCured [41] are program transformation techniques. This type of technique transforms the source code of a program into another program, in the same language, which has run-time checks. Safe-C applies a simple set of transformations to C code in order to provide complete error coverage. The method is not limited to C and can, in theory, be applied to any language. The implementation presented in [40] was benchmarked for pointer-intensive programs. The execution time overhead ranged from 130% to 540% while the code size overhead was estimated to 100%. Nonetheless, the benchmarks were compiled with no compiler optimizations enabled. Thus, by using techniques such as the ones in intended segment analysis [36] the overhead should be reduced significantly.

CCured, on the other hand, attempts to prove memory safety first through static analysis (by enforcing strong types). When the C code does not comply to the CCured type system, run-time checks are used to ensure error detection. The performance of this method is heavily dependent on the amount of run-time checks needed when the static analysis fails. The authors benchmarked CCured with a large set of widely used programs and found run-time overheads ranging from 0 to 87%. This overhead can be improved with compiler optimizations.

Hardware Mechanisms vs. Software Mechanisms

The main advantage of software techniques is their flexibility in providing unlimited memory segments of arbitrary sizes. Moreover, less is required from the hardware, hence microcontroller costs and power consumption are reduced. On the other hand the execution time and code size overheads of run-time checks can be significant. There is also an additional cost associated with changing compilers (which often requires costly certification processes) as well as changing programming languages.

Hardware mechanisms also introduce some overhead. However, this overhead is clearly lower than in software mechanisms and easier to model (*e.g.*, by including context switching overhead in WCET analysis). Furthermore, hardware techniques are systematic in which they can be developed once and used for a long period of time with no additional concerns. Thus, the application programming effort is not directly influenced by the partitioning mechanisms. However, there is an added complexity to the microcontrollers which support hardware memory protection. This results not only in higher cost of acquisition and power consumption but also in higher hardware failure rates as well. Furthermore, the most common hardware mechanisms are designed for desktop and server applications, where some internal fragmentation and a moderate page fault rate are acceptable. In common processors the MMU can hold up to 32 entries in the TLB with a minimum page size of 4KB.

Both hardware and software techniques have the potential to achieve very high or even perfect error detection coverage for software faults (bugs) that cause erroneous memory access attempts. Unless a design fault affects the memory protection mechanisms no process will be able to access outside its own address space.

However, hardware faults can affect the partitioning mechanisms and thereby cause the whole node to fail. This is true whether the mechanisms are implemented in software or in hardware. A transient fault affecting the MMU can result in corrupted memory addresses. A similar fault affecting a software run-time check can have the same effect. Hence, it is not a straightforward issue to determine whether or not software mechanisms are more vulnerable to hardware faults than hardware mechanisms.

Hardware faults must therefore be handled by executing programs on redundant computers. The number of redundant units necessary is intu-

itively lower in integrated architectures than in federated architectures. Thus, the development of memory protection mechanisms facilitates the integration of functions, which in turn facilitates the design of hardware-fault handling mechanisms. This is the case whether memory protection is implemented through software or hardware.

However, hardware memory protection mechanisms can be designed to mask transient hardware faults. TMR and other methods can be applied to the MMU or MPU hardware. This approach is taken in the LEON processors [6], which are able to tolerate SEUs. Consequently, spatial partitioning through hardware can be extended to handle hardware faults.

3.3.2 Temporal Partitioning

For real-time applications it is fundamental for each task to complete before a certain deadline. When multiple processes compete for the same resources (*e.g.*, processor and I/O devices) one must ensure that no process can cause *resource starvation*. Resource starvation occurs when one or more processes are denied access to the shared resources. Such processes may never complete their execution. In general, partitioning requires the software in one partition not to disrupt the timeliness of software in other partitions. This means that, in addition to spatial partitioning mechanisms, one needs to develop temporal partitioning mechanisms as well.

An answer to temporal partitioning is to use well known scheduling algorithms such as RMS and EDF scheduling [26]. In [42] the four main approaches for scheduling are discussed in detail. The approaches are: static table-driven scheduling, static preemptive scheduling, dynamic planning-based scheduling and dynamic best-effort scheduling. These scheduling approaches are discussed in the context of IMA in [21].

However, the existing models of partitioning (discussed in Section 3.2) impose some restrictions on the applicability of the classical scheduling results. An example of this is noninterference. When a task completes its execution earlier than expected it will interfere with other tasks in the temporal domain (they will start executing earlier than expected). This suggests that noninterference in the temporal domain should be relaxed by using the notion of declassification. The main requirement would be to ensure that tasks are unable to hinder other tasks from fulfilling their

response time requirements.

If this policy is accepted, then one can use RMS or EDF to schedule partitions as long as there are mechanisms to ensure that a task cannot execute for more than its assumed WCET. The literature is mostly concerned with analyzing the schedulability of tasks assuming that they release the CPU after having executed for, at most, their WCET. In a partitioned environment this assumption must be implemented in a suitable way. Thus, event-driven scheduling requires several complex mechanisms that are avoided using time-triggered scheduling. Moreover, there are timeliness issues related with concurrent accesses to data items. This issue is solvable through concurrency control techniques [43].

An interesting result obtained using the RMS policy is that there is a non-trivial utilization bound for fault-tolerant scheduling [44]. Re-executing failed tasks, while maintaining the RMS priority assignment, is schedulable for a single fault if the processor utilization does not exceed 0.5. This is an improvement over the trivial bound of $^{\ln(2)}/_{2} \approx 0.346$.

The existing practical approaches to partitioning try to avoid any type of interference, even if benign. A two-level scheduler such as the one presented in [45] is a common paradigm. Under this scheme partitions are executed in a cyclic time-triggered schedule. The individual tasks within each partition are then executed with static (RMS) or dynamic (EDF) priority scheduling.

Time-triggered scheduling of tasks can make the Worst-Case Response Time (WCRT) analysis overly pessimistic. In a scenario where external interrupts are being used to serve a network controller, an interrupt servicing partition A might occur during the execution of partition B. Thus, the execution time of any task in partition B must take into account the frequency at which interrupts for partition A may occur. Nevertheless, time-triggered systems are easier to verify, which is crucial for avionics and automotive systems. In general, the choice between time-triggered and event-driven scheduling depends on each specific application.

3.4 Summary and Discussion

This chapter presented an analysis of robust partitioning methods. It discussed the requirements for partitioning and the existing mechanisms

to implement partitioned systems. Furthermore, it analyzed the development effort necessary to ensure that integrated and federated architectures are equally dependable. The goal of the probabilistic analysis was to determine which factors affect the reliability of integrated architectures, rather than making an accurate estimation of reliability. This allows us to draw some conclusions by triangulating results for the development of integrated systems. If a conclusion is motivated both by probabilistic analysis and by qualitative arguments, it gains more solid support.

To assure the reliability of integrated architectures, a fundamental design decision is whether to use robust partitioning mechanisms or to increase the development effort for all functions. There is a cost associated to both options. Partitioning mechanisms add complexity to the system, thereby increasing the development effort for the entire platform; the other option is to increase the development effort for individual functions, which is costly when the number of functions grows. There is a trade-off between the two choices. However, partitioned systems have the additional advantage of facilitating incremental certification, *i.e.*, to certify a system once and upgrade it with new features without the need for complete re-certification.

When the robust partitioning option is chosen, it is beneficial to segregate all integrated functions. This includes separating functions of the highest criticality from each other. In this chapter we concluded this through probabilistic analysis. Furthermore, the same conclusion is apparently motivated by the perfectionist approach (procedure-based software development). If the emphasis is on using the best available systems engineering practices, then partitioning (or some other type of protection among functions) should always be introduced.

Without careful analysis, one must assume that partitioning mechanisms provide limited or no protection against hardware faults. Thus, structural hardware redundancy is required to protect the system against hardware faults. However, integrated architectures are expected to require less hardware from a functional perspective, leading to a lower overall hardware failure rate. Consequently, integrated architectures are likely to demand less structural redundancy than federated architectures.

Hardware mechanisms for spatial partitioning have a clear advantage over software mechanisms. First, spatial partitioning through software

requires the costly introduction of new tools in the tool-chain (compiler, linker, etc.). Second, the code containing, for instance, runtime checks will be interleaved with the application code. Thus, it may be difficult to persuade certifying authorities that the same object code contains distinct criticality levels for the application and for the spatial partitioning mechanisms.

There is a large set of design choices available for temporal partitioning. In principle, both event-driven and time-triggered execution can fulfill the requirements of partitioning. Time-triggered scheduling of tasks can make the response time analysis overly pessimistic, and is less flexible than event-driven execution. Nevertheless, time-triggered systems are easier to verify, which is crucial for high-integrity systems. This provides a strong motivation for using time-triggered scheduling in systems where verification is crucial. However, there are also advantages in using event-driven scheduling. Thus, the choice between time-triggered and event-driven scheduling depends on each specific application.

CHAPTER 4

Robust Operating Systems

Operating systems are often used for managing critical infrastructures ranging from server rooms to embedded devices, as well as crucial user information on desktop computers. Given that a failure of such computers can have serious consequences, the operating systems must be reliable in the presence of faults. Moreover, they should provide comprehensive error detection and recovery services to hosted applications, so that the system as a whole can be dependable.

This chapter discusses ideas on the design of fault-tolerant operating systems for embedded applications. The principal objectives are to facilitate composability within computer nodes, by preventing undesired interactions among software components that share hardware resources, and to detach recovery mechanisms from applications, so as to promote reusability of fault tolerance services. The ideas are grouped into a concept named SECERN, meaning *to separate* (components from each other and fault tolerance from functionality).

The discussion alternates between the themes of design, implementation and verification; and addresses the detection, isolation and recovery of errant application processes. In the design, the purpose of the operating system is to create a partitioned environment which can be shared by multiple real-time tasks, possibly with distinct levels of criticality

and uneven reliability. Moreover, to ensure sustainable service delivery, the operating system is designed to aid hosted applications with error recovery.

One of the guiding principles is to tolerate both software and hardware faults (affecting application processes) in a comprehensive manner. The avionics industry, on one end of the spectrum, claims to produce software of the highest quality, by applying the best engineering practices, and is mostly concerned with tolerating hardware faults. On the other end, developers of desktop and server applications regard hardware faults as an issue of the past, easily solvable through redundancy, and center their attention on software faults. One can argue that these two mind-sets pose a dependability threat, since they don't take a holistic view of the problem.

Regarding the implementation of SECERN, this chapter describes an extension to μC/OS-II intended for experimentally assessing techniques for building robust operating systems. Reusing an existing code base, instead of creating a new solution, has the advantage of making the results more general and focusing the development effort on fault tolerance mechanisms. However, the trade-off is that many design decisions are inherited and may require adaptation to circumstances differing from the original purpose, thereby requiring some verification effort.

We conducted series of preliminary tests of the implemented mechanisms using fault injection. A new fault injection plug-in was developed for the GOOFI tool [46, 47], aiming to provide robustness testing for partitioned systems. The plug-in targets the Freescale MPC5554 microprocessor, which is the central element of the experimental platform supported by the present version of SECERN. The set of experiments described in this chapter explore the capabilities of the MPC5554 plug-in for testing the robustness of SECERN.

The experiments are conducted according to a methodology of focused fault injection, whose main objective is fault removal, *i.e.*, diagnosis and correction of design faults. It consists of setting up finely controlled experiments in accordance with the system properties that are to be verified. This methodology was applied for verifying that the partitioning mechanisms are able to isolate faulty applications. The experiments exposed two vulnerabilities in the system: one related to configuration management, where some memory pages were marked as writable for all

processes while they should be read-only; and one related to an inherited design decision regarding context switches which is not appropriate for partitioned systems. Although an exhaustive evaluation of SECERN is outside the scope of the thesis, these experiments demonstrate the potential of fault injection as a technique for fault removal in partitioned systems.

In addition to the mechanisms included in the extended real-time kernel, SECERN includes an approach to checkpointing and rollback recovery of real-time tasks named lightweight checkpointing. The lightweight checkpointing scheme allows applications to save snapshots of their state to main memory while providing them with a service for locking the checkpoint area using memory protection. We used the SPIN model checker to verify that the scheme is able to guarantee the integrity of the checkpoints.

4.1 SECERN: An Extension to μC/OS-II

The trend to integrate multiple functions in a single hardware platform has created the need for building strong fault containment around software components. Initiatives such as the standard interface for avionics applications [12] and the AUTOSAR project [13] aim at defining the software infrastructures and, particularly, the operating systems that support this level of fault containment. Since those initiatives target safety-critical systems, a fundamental concern is to ensure that resource sharing can be accomplished in a safe and reliable manner.

We have implemented an experimental prototype of SECERN by extending the μC/OS-II real-time kernel [48]. The kernel is DO-178B certifiable [18] and its source code is well documented and freely available for academic purposes, making it a suitable choice for our implementation. It lacks support for isolating applications from one another and from the operating system, which makes it appropriate for experimentally assessing the SECERN concept.

The extended version of the kernel runs on a computer board featuring a Freescale MPC5554 microprocessor [33], based on the PowerPC architecture. The processor core includes an MMU which provides, among other services, memory protection. The hardware-specific layer of μC/OS-II was implemented by creating a board support package contain-

ing low-level code and macros. The kernel was then extended according to the design principles which are described next.

4.1.1 Design Principles of SECERN

One of the key modifications to μC/OS-II is the distinction between processes and threads, where each process owns a private address space that groups together one or more execution threads. Each process acts as a container which is usually called a partition in IMA terminology. The architecture of SECERN is depicted in Figure 4.1.

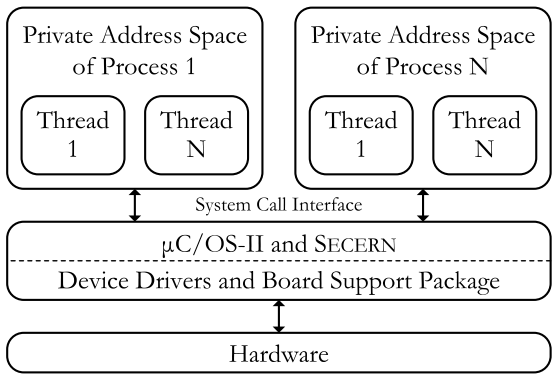

Figure 4.1: μC/OS-II extended with SECERN.

The private address space of each process is protected by the memory management hardware, which lies between the processor core and the memory. Instructions always generate virtual addresses that are translated by the MMU to physical addresses before the memory operation is performed. During this process, the MMU checks that the application process which is executing has the appropriate access rights – read, write or execute permission for user- and kernel-mode instructions. This feature is used to enforce the appropriate access permissions on all memory pages. For simplicity, a direct mapping is set between virtual and physical addresses, *i.e.*, in practice, no use is made of the address translation feature.

Memory protection is a standard feature of desktop and server computers. However, it is seldom used in embedded real-time systems. One reason for this is that microcontrollers are usually not equipped with

the necessary hardware, in order to reduce cost and power consumption. Another reason is the variation in execution time imposed by memory protection and address translation, which is usually optimized for performance rather than predictability.

Typical implementations of memory management hardware make use of a TLB for improving the performance of address translation and memory protection. A TLB is a very fast cache which contains a small number of entries; each entry specifies the virtual and physical addresses where a memory page starts, the size of the page and the access rights. This cache reduces the time overhead of the MMU but there is a large penalty for memory accesses which are not matched by any TLB entry. In this case, which is called a TLB-miss, a processor exception is raised to allow the system software to update the TLB. This may become an issue, since interrupts are generally unwanted in real-time systems and make it more difficult to determine the Worst-Case Response Time (WCRT) of applications.

To deal with this problem, the memory protection routines of SECERN are designed to update the TLB during context switches. The approach is to insert in the TLB the pages that belong to a process before running that process, thereby preventing TLB-misses. This, in turn, simplifies the response time analysis of of hard real-time tasks. Nevertheless, this method adds an overhead to context switches. Measurements on the time necessary to perform a full context switch (from the first instruction of the context switch handler to the first instruction of the next process) are presented in Figure 4.2.

The measurements shown in Figure 4.2 were taken on the MPC5554 processor, which has a 32-entry TLB. Since the kernel requires some pages to be permanently listed in the TLB (to avoid TLB-misses when handling kernel calls and other interrupts) the plot shows the overhead of switching context to a process containing up to 24 pages. The number of instructions executed grows proportionally to the number of pages and, consequently, so does the context switch time.

The time needed for a full context switch without updating any TLB entries is slightly below 10 μs (for saving the numerous PowerPC context registers, updating kernel structures and loading the registers of the next task). Considering a typical embedded application, requiring between 4 and 8 pages of memory, context switching would take between 31 and

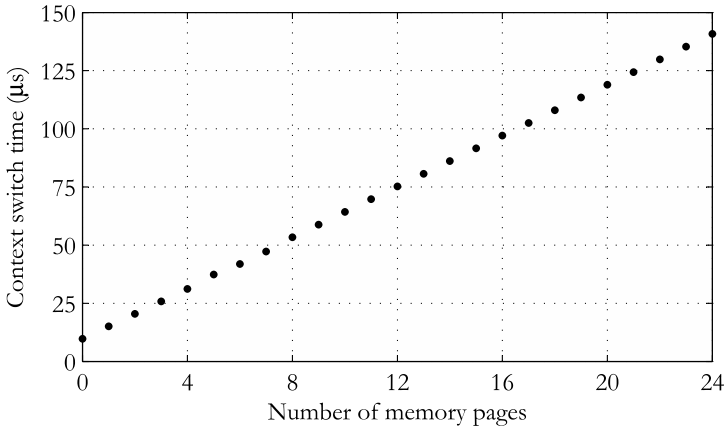

Figure 4.2: Context switching time measurements.

53 μs. This overhead should be carefully examined when considering performance demands, as it is common for real-time operating systems to switch context in less than 10 μs. Nevertheless, when memory protection is used, this increased time is a trade-off rather than a penalty. Without updating the TLB, a process may cause one TLB-miss for each page in the worst case. This is more expensive than doing the update during context switches and generates some execution time jitter.

Introducing memory protection has implications on the design of the system call interface, since it rules out the use of the *branch and link* instruction for calling system services. Instead, service requests are made through the system call interrupt. This process is made transparent to applications by implementing the low-level details in a system library – a common approach in operating system designs.

The system call mechanism is used by applications to request kernel services and to reach device drivers. For this reason, it must be robust in order to prevent application errors from propagating to other parts of the system. This is often a problem, as experimental studies have shown that many operating systems contain vulnerabilities in functions provided by the system call interface [49], *e.g.*, crashing the system when given exceptional input parameters.

Another problem is that the system call mechanism must be able to enforce access policies, in order to control the services that each partition

has the right to access. Some authors propose the usage of sandboxing as the means to protect the system call mechanism [50, 51]. This technique consists of interposing the access to system calls with a filter that enforces a given policy. For real-time kernels, this technique must be implemented as efficiently as possible.

We took a simple approach to implementing system call protection. The kernel provides the partition's ID to the system call handler. The caller ID can be checked by the drivers and by any kernel services to enforce an access policy. It is also possible to check the parameters to the system call interface and report an error of the partition that executed the call. This would act as an additional error detection mechanism.

4.1.2 Error Detection and Fault Handling

In addition to memory protection and checking the system caller ID, our kernel extension makes use of processor exceptions to detect errors and allows application-specific checks to notify the kernel of errors. Many techniques for creating application-specific checks are available in the literature and the kernel provides the means for such checks to report errors. When one of these error detection mechanisms is triggered, the error is handled by one of two central exception handlers:

- *Recoverable condition.* The detected error is confined to a single process (*i.e.*, partition) and it is possible to delete that process and continue executing. In this case, SECERN deletes all threads belonging to the process and resumes execution. Here, it would be possible for the kernel to interact with the system layer by replacing its output with an error code.

- *Unrecoverable condition.* An error is detected and it may be caused by a hardware problem or by a fault in the operating system itself. The currently implemented version enters an infinite loop but there are several other possibilities. It would be possible, for instance, to restart the kernel, check the consistency of the hardware and restart all tasks. In any case, the entire processor node is affected by this error and the kernel should interact with the system layer by sending a failure report.

4.1.3 Scheduler

One of the limitations of our extension to μC/OS-II is that it does not introduce mechanisms for temporal partitioning. μC/OS-II has a priority-based preemptive scheduler that executes always the task with the highest priority which is ready to run. This means that a high priority task may prevent lower priority tasks from executing, if it fails to release the CPU on time. On the other hand, this ensures that the highest priority task is never disturbed by any other task.

A possible way of achieving temporal partitioning would be to introduce time-triggered scheduling in SECERN. This is the option favoured by the ARINC 653 specification. To achieve time-triggered execution of tasks, we would have to add a table to the kernel defining the cyclic schedule. A simple implementation to enforce that schedule could be made by adding the necessary code to a user-definable hook which is called by μC/OS-II at every time tick – the `OSTimeTickHook()` function.

It would be possible to use the `OSTimeTickHook()` function to change the priority of each task according to the predefined cyclic schedule. At every time tick, the function would check which task should be running at that point in the schedule. Then, using the `OSTaskChangePrio()` function provided by μC/OS-II, it would ensure that the task gets the highest priority. This method would effectively implement time-triggered scheduling in SECERN.

We would be trusting the scheduler of μC/OS-II to execute the highest priority task at each point in time. Thus, this approach would allow us to reuse as much code as possible from the original version of μC/OS-II. To verify that this approach works, the setup was configured to run two tasks. Both tasks execute an infinite loop where they read some input, perform a computation, write the output and release the CPU. We then focused on the highest priority task to ensure that it would produce its results regardless of the computations of the low priority task. These tests were made using the fault injection tool that is described next.

4.2 Robustness Testing for Partitioned Systems

We have extended the GOOFI tool [46, 47] with support for injecting faults into the Freescale MPC5554 microprocessor. The new fault injection plug-in is based on an existing plug-in which provides support

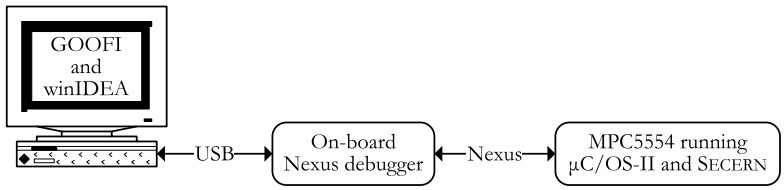

Figure 4.3: Evaluation platform for μC/OS-II and SECERN.

for the MPC565 processor (the plug-in used in Chapter 5). The experimental setup consists of a desktop computer, with GOOFI and the winIDEA development environment, controlling an MPC5554 development board [52]. The development board includes an on-board Nexus debugger. Figure 4.3 depicts the experimental platform.

The MPC5554 fault injection plug-in is capable of injecting bit-flips into processor registers and memory locations. It allows the user to define a range of code addresses where the execution can be stopped for injecting a fault. In each fault injection experiment the tool selects one random address to set a breakpoint. Once this breakpoint is reached, the tool randomly chooses a resource (register or memory location) and one of its bits to inject the bit-flip.

Other fault injection plug-ins for GOOFI, like the one used in Chapter 5, collect the sequence of instructions executed in a fault-free experiment to create a program trace. This is a time-consuming procedure, since the processor needs to be stepped in order to determine the sequence of values of the program counter register. Due to the large number of instructions executed by the kernel, the application processes, the idle task and other system tasks, the stepping process for the reference experiment would take too much time.

To deal with this, the tool allows fault injection experiments to be made without a program trace. This is achieved by choosing a random address from the entire range of user-defined addresses. Since that address might not be reached once the program executes, there are a number of experiments in which a fault is never injected and the outcome is exactly the same as that of a fault-free experiment. Such experiments are simply discarded during analysis and classification.

To provide fault injection for a partitioned environment, the tool is capable of monitoring the execution of the operating system and collect

the output of multiple tasks. The user can define the output address of multiple workloads, so that the results produced by tasks can be collected and classified. Moreover, the tool can set breakpoints for monitoring the activation of the two central exception handlers described in the previous section, in order to monitor the operating system. The activation of breakpoints and the output data are saved to a database for analysis.

One limitation of the tool is that the user may only define one single address as the output address of tasks. This limitation is related to the fact that there are only four hardware breakpoints available for monitoring the execution. We use one for the output of tasks, one for the fault injection breakpoint and two for monitoring the operating system. This limitation can be partially circumvented by setting the output breakpoint to the output address of the task with the highest frequency. This way, we sample all values produced by all tasks, although some samples are repeated.

We used this way of collecting the output of tasks in our experiments. Since there is only one point of the execution when the output is collected, we can only monitor the timing behaviour of one task at a time. We consider two options for a future improvement of the tool. First, we can measure the output time externally, by making the tasks write their results to an output port and reading the values using another development board or a desktop computer. Second, we can use a much greater number of software breakpoints than hardware breakpoints to monitor the execution. This would require an investigation on whether software breakpoints add some intrusiveness to the fault injection process.

Regarding the workloads used as operating system tasks, we use cyclic programs that execute some computations on input data and delay themselves until the next iteration. Figure 4.4 shows the typical structure of the main routine of a workload thread. The output breakpoint can be set to the address before the call to `OSTimeDlyUntil()`.

We used two different workloads in our tests: a wavelet transform and an altimeter function. The first workload applies a wavelet transform to an array of input data and produces an output array containing the result. The altimeter function reads a simulation of pressure values and, for each sample, produces an estimation of the altitude.

```
void thread(void *pdata)
{
  INT32U start_time, period = 20;

  start_time = OSTimeGet();

  while(TRUE)
  {
    getInput();
    computeOutput();
    OSTimeDlyUntil(start_time += period);
  }
}
```

Figure 4.4: Main routine of a workload thread.

4.3 Focused Fault Injection

One may conduct fault injection experiments with the purpose of fault forecasting or fault removal. Fault forecasting experiments aim to estimate diverse measures of dependability and to gain a better understanding of how a system (or one particular component) will behave in the presence of real faults. Such experiments are useful for comparing alternative components with regards to their dependability, for identifying a system's dependability bottlenecks, for characterizing a system's dependability, etc.

The goal of fault removal experiments is to identify flaws in the design or implementation of a component or a system, so that they can be corrected. To achieve this, one places the focus of experimentation on exercising specific parts of the system with suitable types of faults (which the system is designed to tolerate). This form of fault injection is suitable for testing fault tolerance mechanisms and is therefore helpful for the verification of computer systems.

The most frequent objective of fault injection practitioners is fault forecasting. Researchers often adopt this method for experimentally validating new techniques, *e.g.*, by determining the coverage provided by an error-detecting mechanism or the effectiveness of a recovery strategy. Taking a broader perspective, there have been research efforts to pro-

mote the use of measurement theory for estimating dependability [53] and to define methods for benchmarking the dependability of computer systems [54]. Dependability benchmarks aim, among other things, to guide the development effort (*e.g.*, by finding weaknesses in the architecture) and to assist buyers in deciding among competing off-the-shelf components.

Nevertheless, fault removal is also vital for many, if not most, buyers of off-the-shelf software. Consider an example where a system integrator intends to use a COTS operating system for building a given application. The selection process is influenced by numerous factors, including technical findings – such as results of dependability benchmarks – and management decisions – based on each vendor's credentials, guarantees in terms of long term support, cost issues, etc. We can identify two risks here. First, the selected operating system might not be the most dependable among the available choices. Second, regardless of the choice, it may require adaptation to a specific hardware platform and it could contain design or implementation defects. Consequently, system integrators would be interested in coming back to suppliers with problematic test cases that require attention.

In this chapter we adopt fault injection as the means to find such test cases. We are interested in finding and removing vulnerabilities in SECERN – particularly those related to partitioning. To this end, we begin by describing a methodology for fault removal in partitioned systems and then present the results of fault injection experiments targeting our experimental platform.

4.3.1 Methodology

A fault injection experiment with the objective of fault removal has two principal outcomes: either the system fails to cope with the fault that is injected (*e.g.*, the operating system crashes) or the service provided by the system is classified as correct. This classification requires sufficient data to be collected during the experiments, so that we can determine whether or not the system fails to handle any faults. If so, those faults can be regarded as counterexamples, *i.e.*, scenarios where one or more system properties are violated.

Naturally, the faultload must be representative of faults that the system is required to tolerate. On the one hand we wish to test systems ex-

tensively, in order to identify as many existing defects as possible. On the other hand all counterexamples should be meaningful, *i.e.*, they should only locate actual defects rather than calling our attention to situations which the system is not supposed to handle. To achieve this, we adopt a methodology of focusing fault injection experiments in accordance with the system properties that are to be verified.

The concept of focused fault injection has been used in the past for testing distributed systems [55]. We take a conceptually similar approach targeting the verification of node-layer fault tolerance mechanisms. Our goal is to verify that SECERN prevents application errors from propagating to the operating system and to other applications. We are therefore searching for vulnerabilities in the software related to partitioning mechanisms, *e.g.*, the low-level code that controls the hardware. Nevertheless, one should not exclude the possibility of finding hardware design faults such as those reported by Intel [56], affecting the MMU of recent microprocessors. The fault injection experiments were designed by taking the following steps:

- *Configure the workloads in a relevant manner.* We configured the system to execute two processes, each one with a single thread. The two threads executed, in an infinite loop, a data processing routine and released the CPU until the next iteration. The tasks executed with sufficient frequency to force context switches among them at intermediate points of the execution (of the low priority thread).

- *Inject faults that mimic application errors.* The tool injected bit-flips in the context registers (*i.e.*, processor registers that are saved during context switches) of the lowest priority task. Bit-flips are not representative of software faults. Nevertheless, they are representative of faults that the system must handle. The tool was configured to inject faults during the execution of any instruction of the low priority thread.

- *Collect sufficient data to classify experiments.* During each experiment we collected the output of both tasks and monitored the activation of the two central exception handlers described earlier (to infer whether the operating system had crashed).

- *Classify the outcome of the experiments.* We analyzed the data resulting from the experiments in order to check if partitioning had been violated in any way. First, the output of the high priority task was compared to that of a fault-free reference experiment. Any difference in the result would indicate a partitioning violation. Second, the activation of the unrecoverable exception handler would indicate that the operating system had crashed. Third, experiments where the execution ended at a different instruction address than the expected one would be caused by an undetected system crash.

- *Examine experiments that expose counterexamples.* Faults that cause the operating system to crash, the high priority task to produce wrong output or the high priority task to be deleted are classified as partitioning violations. For these experiments one must examine the fault which was injected (the instruction where the bit-flip was injected and the resource affected), since it exemplifies a situation which is not properly handled. Essentially, the question is to understand what led a fault injected in the low priority thread to affect other parts of the system.

- *If necessary, instrument the code and document test cases.* We can manually instrument the code of the threads to mimic as closely as possible a fault that exposes a counterexample. This serves, in our case, as a way of validating the fault injection tool. Moreover, a system integrator verifying a COTS operating system would prefer to send a test case consisting of an example program, instead of sending the fault injection tool and the fault definition to the supplier of the operating system.

4.3.2 Results

We present the results of a campaign consisting of 284 fault injection experiments, where both threads executed the wavelet workload. In our setup it takes 1min 12s to run a reference experiment, to collect the results of a fault-free execution. Each fault injection experiment takes, in average, 1min 25s. Since we do not collect the program trace (*i.e.*, the sequence of instructions executed during the reference experiment), we

must set the fault injection breakpoint without being certain that it will be reached.

Table 4.1 shows that the fault injection breakpoint was reached, in this set of experiments, in 67 occasions. In the remaining 217 experiments the fault injection breakpoint was not reached and this means that no fault was injected.

No. of Experiments	Breakpoint Reached	Breakpoint Not Reached
284	67 (23.6%)	217 (76.4%)

Table 4.1: Activation of the fault injection breakpoint.

We analyzed the 67 experiments where a bit-flip was actually injected to determine whether it was correctly handled. As explained earlier, the classification process takes into account the activation of the centralized exception handlers (recoverable and unrecoverable) and the output of the tasks to determine whether or not the fault was handled. In this case we consider only the output of the high priority task, since we are injecting faults in the low priority task. Table 4.2 shows the classification of the fault injection experiments.

Experiments	Operating System		High Priority Task		
	Operational	Crashed	Correct Output	Wrong Output	Deleted
67	66	1	64	3 (2+1)	0

Table 4.2: Outcome of the fault injection experiments.

As we can see in Table 4.2, the operating system crashed once and the high priority task produced wrong results in three occasions. One of the wrong outputs occurred in the same experiment where the operating system crashed (which made it impossible for the task to continue executing). Thus, we found three experiments where the system failed to handle a fault in the context of the low priority task. One fault led the entire operating system to a crash and two faults caused the high priority task to produce incorrect results. These faults must therefore be examined since they expose flaws in the system.

The Context Switch Flaw

The fault that led the operating system to a crash was injected into processor register R1, which is the stack pointer. At a certain point of the execution of the low priority task, a bit-flip changed the stack pointer from $\mathit{40007F08}_{16}$ to $\mathit{44007F08}_{16}$. In practice, this meant that R1 no longer pointed to the top of the thread's stack and now pointed to an unused memory address.

We used the debugging environment to manually inject a similar fault and observe the sequence of events that then took place. Rather than using the stack pointer, the low priority task was executing a part of the main loop when a context switch occurred. At this point, the μC/OS-II kernel started to save the context of the task to the top of its stack – the approach that it is designed to take. The problem was that the stack pointer no longer pointed to the correct address. Thus, the kernel attempted to write the context of the task to address $\mathit{44007F08}_{16}$. This memory area was unused and therefore not listed in the TLB, thus causing a TLB-miss. In our design, a TLB-miss caused by kernel code is an unrecoverable condition.

The code of the low priority task was manually instrumented to execute correctly for two seconds, corrupt the stack pointer and enter an infinite loop (to wait for a context switch). Figure 4.5 shows the instrumented code.

This fault showed that our extension to μC/OS-II failed to provide perfect partitioning due to an inherited design decision. Since μC/OS-II saves the context of tasks on the top of their own stack, it is possible for a task to corrupt the stack pointer and cause the kernel to write onto an erroneous memory location.

There are numerous possible solutions to remove this partitioning defect. We chose to add a stack pointer check during context switches. The task control block of all tasks (a kernel structure which stores important task information) contains the location and size of each task's stack. We added a check to verify, before saving the context, that R1 points to a memory location in the task's stack and that there is enough space to write all context registers. After modifying the context switching code we executed the test case in Figure 4.5 to verify that the flaw had been removed.

```
void thread(void *pdata)
{
  INT32U start_time, period = 20;

  start_time = OSTimeGet();

  while(TRUE)
  {
    if(start_time > 200) // two seconds after startup
    {
      // set R1 (the stack pointer) to 0x44007F08
      __asm__ __volatile__ (" lis  %R1, 17408 ");
      __asm__ __volatile__ (" addi %R1, %R1, 32520 ");
      while(TRUE){}
    }

    getInput();
    computeOutput();
    OSTimeDlyUntil(start_time += period);
  }
}
```

Figure 4.5: Manual instrumentation of the low priority thread to corrupt the stack pointer and wait for a context switch.

The Configuration Error

The two experiments that caused the high priority task to produce wrong results injected a fault into registers R6 and R29. These faults were injected at a point of the execution where these registers were being used to calculate memory addresses for write operations. The instructions that executed after that attempted to write into a page which was shared by the two tasks, containing code and data belonging to a floating point library.

The issue here was that there were several pages erroneously configured with write permission for all tasks. The initialization sequence inserts into the TLB the pages that are listed permanently (kernel and shared libraries). An inspection of this sequence revealed that the pages were configured with full permissions for all tasks, even though they should be only readable and executable. In this case, a test case would be as simple as instrumenting the code of the low priority thread to write into those addresses. This configuration error was solved by giving only read and execute permissions on the library pages to all tasks.

4.3.3 Limitations

These experiments demonstrate the potential of fault injection as a technique for fault removal in partitioned systems. However, we would have to conduct many more experiments to exhaustively test the mechanisms included in the extended real-time kernel. Moreover, a limitation of these experiments is that we only observed the output of the high priority task in the value domain, *i.e.*, the time when task produced its output was not monitored. Thus, temporal partitioning was only examined indirectly, by monitoring whether the high priority task produced correct results at some point in time (although the exact time was not measured).

We have tested the robustness of the implementation in the presence of bit-flips in the context of one process. Even though this is a type of fault that the system must tolerate, bit-flips in CPU registers and main memory are only representative of transient hardware faults. An exhaustive test of the kernel extension would certainly take into account software faults. These can be injected using software fault emulation operators [57, 58].

4.4 Recovering Errant Applications

As we have argued thus far, operating systems must be resilient to application errors and should prevent those errors from propagating to other applications. This guiding principle assures us that healthy applications are always capable of providing correct service to their users. However, it is also crucial to recover applications that have failed, in order to ensure that service losses do not accumulate over time and lead to redundancy exhaustion. Accordingly, one should judge the dependability of an operating system not only for its resilience to errors but also for the services it provides to hosted applications with regards to error recovery.

To this end, we can make use of a vast multitude of error handling techniques available in the literature. Our goal here is to combine several such techniques into a set of operating system services that support application recovery from both software and hardware errors. The challenge is that the recovery flow usually depends on whether the error was caused by hardware or software. This is generally hard to diagnose. Consider an example where the memory management hardware raises an exception indicating an erroneous memory access; it may have been caused by a missing pointer initialization in the software but also by corruption of a pointer due to a transient hardware fault. In this case it is difficult to choose, for instance, whether to rollback and retry the operation or to transfer control to a user-mode exception handler.

4.4.1 A Comprehensive Recovery Strategy

Let us assume that application errors can be detected by the operating system even though their cause is unknown. Our proposal is to consider *at first* that an error is caused by a transient hardware fault. To cope with these, applications take frequent checkpoints and attempt rollback recovery upon error detection. When the cause is indeed a transient hardware fault, there's a high chance that it will vanish after rolling back. A suitable checkpointing technique is proposed in Section 4.5.

However, if an error is detected again after the rollback – before the next checkpoint is taken –, then we diagnose it as being caused by a software fault. At this point, the operating system should transfer control to an application-specific exception handler, where the designer can decide what should be done. There are several classical software

fault tolerance techniques which may be appropriate for this stage:

- *Design diversity* may be applied by developing alternate versions of a program [59] and switching among them when an error is detected. Since only one version is executed at a time, this approach is similar to the well known recovery blocks [60] technique. The main difference is that we rely on concurrent error detection mechanisms rather than acceptance tests to trigger the version switch. Effective design diversity requires version independence, *i.e.*, uncorrelated errors among versions. Ideal independence is hard to achieve and caution is advised by several experimental studies [61], but in most cases design diversity can increase the reliability of a system.

- *Data diversity* can be effective for some systems [62]. This method consists of re-executing the same program with a slightly different input. It is realistic to assume that production software only fails for a small fraction of the input space. Thus, if an error is detected, one can make a small change to the input (either explicitly or, for example, by reading another sensor value) and execute the program again. This technique is not as costly as design diversity, since it only requires a single version of the software.

- *Restarting* the faulty task may also be sufficient and appropriate for some cases. This is a simple form of software rejuvenation [63] triggered in reaction to error detection. It is the simplest of the three possibilities listed here and, nevertheless, the one which may be applicable in most scenarios.

One must also consider the case when an error is detected a third time, *i.e.*, when both recovery attempts fail. Such cases may be caused by permanent hardware faults which prevent an application from executing correctly. Under such circumstances, the application should be terminated in order to ensure fail-silence and to constrain resource consumption.

An alternative which deserves future examination is to make use of the inherent hardware redundancy available in multi-core processors. In the future we can expect multi-core processors to be used in embedded systems. Thus, we can make use of the available redundancy for improving system dependability. When both recovery attempts fail, it is

beneficial to try to execute the same application on one of the remaining cores. The advantage is that some permanent hardware faults, such as those related to ageing, may be tolerated as part of the recovery strategy.

This approach assumes implicitly that core failures are to some extent independent. At the present moment there is little empirical evidence available to support this assumption. However, it is reasonable to presume that there will be a certain degree of isolation between cores belonging to the same integrated circuit. If both critical and non-critical tasks are being executed on the same multi-core processor, ensuring that the critical ones can execute on a healthy core is likely to increase a system's dependability.

4.5 Lightweight Checkpoints

Checkpointing and rollback recovery is a way of tolerating transient hardware faults at the node layer. Checkpointing involves taking regular snapshots of the system state and storing them in a safe place (sometimes called stable storage). When an error is detected, the system rolls back to a fault-free state by restoring the most recent valid checkpoint. While checkpointing and rollback is widely used in applications such as database systems and transaction processing, it is less frequently used for embedded real-time systems. The main reason for this is the time overhead generated when a rollback recovery is made and when taking the checkpoints.

In real-time systems, the correctness of a computation depends not only on the resulting value but also on the timeliness of the result. Thus, unlike general-purpose applications, the goal of checkpointing in real-time systems is to guarantee that critical deadlines are met even when errors occur. If checkpoints are sufficiently frequent, the amount of recomputation required after an error may be small enough to complete the execution before the deadline expires. However, taking checkpoints increases the execution time in fault-free cases. For this reason, in real-time systems the checkpoint interval can't be arbitrarily small and the checkpointing mechanism must have a low overhead.

One way to reduce the overhead is to store only fundamental data at each checkpoint. The ability to identify these data depends on the checkpoint level, *i.e.*, whether it is done by the kernel, by a user-mode

library or by the application itself [64]. In general, only the application designer can determine which structures are fundamental. Hence, checkpointing can be efficiently implemented at the application-level, with the additional advantage of having no overhead in calls to external code. This approach is therefore attractive for real-time systems.

However, implementing checkpoint and recovery functions increases the development cost of applications. Furthermore, applying this method correctly is often non-trivial, even in uniprocessors. One reason for this is error detection latency, *i.e.*, the amount of time between the occurrence of an error and its detection. During that time interval the application may save a corrupted checkpoint and then recovery is only successful by restoring an older non-corrupted checkpoint. Due to this problem we must introduce a delay between storing a checkpoint and considering it to be reliable. Another reason is the difficulty in establishing reasonable assumptions on the failure modes of applications. A transient hardware fault may, for example, corrupt a pointer and cause an application to overwrite any previous checkpoints. Consequently, the integrity of application-level checkpoints must be assured through careful design.

We propose a lightweight checkpointing technique for real-time embedded systems. It allows applications to checkpoint their state independently but provides the means for them to lock checkpoints using memory protection. The method ensures the integrity of checkpoints for a broad class of application failure modes and takes error detection latency into account. The goal is to provide the level of reliability required by high-integrity applications and meet the needs of real-time systems.

4.5.1 Context and Applicability

Many applications can be made fault-tolerant by checkpointing small amounts of state information (*e.g.*, control algorithms [65, 66]). Likewise, device drivers may recover transparently from failures by retrieving state information lost during a crash [67], thereby requiring a reliable mechanism for guaranteeing the integrity of driver checkpoints. To achieve this, we advocate a clear separation of concerns: each application (or driver) should be responsible for taking checkpoints, while a platform service assures their integrity. In this section we describe the design of one such service and show how it can be implemented on modern microprocessors.

One main concern in designing this operating system service is to en-

sure its real-time performance. The overhead should be as low as possible and each call to the service should be bounded in time, so that it can be used by real-time tasks. By reducing the overhead of checkpointing we can improve fault tolerance, since we are allowed to take checkpoints more frequently. Our approach requires only a very small system call to be made after each checkpoint in order to lock that checkpoint using memory protection.

We assume that main memory is sufficiently reliable to be used as stable storage for an embedded system. Given that main memory is usually protected with error-correcting codes, we can assume that checkpoints are safe when stored in main memory. In general, the DRAM cells used to build memory chips can be considered very reliable [68]. Moreover, even if a fault affects a checkpoint directly in memory, we assume that the probability of another near-coincident fault causing an application to rollback (to the corrupted checkpoint) is negligible.

4.5.2 Failure Modes and Error Detection Latency

Our checkpointing scheme addresses only faults directly affecting the applications. Errors affecting the operating system or the checkpointing service may be detected but we provide no means to recover from them. Typically, a real-time kernel executes less than 5% of the time [48]. During the remaining time the processor is either idle or running applications. It is therefore likely for a transient hardware fault to affect only the context of a single application. We assume that errors affecting the entire computing platform can be handled by other fault tolerance mechanisms (possibly those implemented at the system layer).

We assume that application errors can be detected within a bounded amount of time. Most error detection mechanisms take some time to discover and flag application errors [69]. There is, nonetheless, strong empirical evidence showing that the vast majority of detectable errors is detected within a bounded time interval. An example is the high success ratio of recoveries achieved through low-level checkpointing and rollback in high-end mainframe microprocessors [70, 71]. Implementing effective error detection is fundamental to ensure the success of the recovery process. Thus, if some errors remain latent for more time than expected, we can only give probabilistic guarantees that a successful recovery will eventually happen.

Error detection latency introduces problems in any scheme for checkpointing and rollback recovery. During the time between an error and its detection an application may save an incorrect state. To recover from this type of failure one must restore an older checkpoint. Due to this problem we must maintain, at any time, several past checkpoints and consider each one to be unreliable until the maximum error detection latency has passed after its creation. There is, naturally, a limit on the number of checkpoints that can be maintained in an embedded system.

The integrity of application-level checkpoints is strongly dependent on the failure modes of applications, *i.e.*, their behaviour in faulty circumstances. Since we assume that applications write their own snapshots to main memory, there is a concern that an errant application may overwrite all previous checkpoints before the error is detected. We make no direct assumptions on the failure modes of applications. We assume only that the memory area where a checkpoint is stored can be locked from the application by using memory protection.

4.5.3 Assuring the Integrity of Checkpoints

The fundamental requirement on the integrity of checkpoints is that an application should always rollback to a correct checkpoint upon error detection. Since, under our assumptions, an errant application may overwrite the checkpoint area before the error is detected, we introduce a lock() system call that prevents any further writing to that area. Once a checkpoint is locked, it may be considered reliable after an amount of time equal to the maximum error detection latency has passed. Until then, we have to assume that an error may have occurred before the lock() call was made.

At some point, the application should take another checkpoint without overwriting the previous one. The concern here is that an error may occur precisely when a checkpoint is being taken. For this reason, it is common practice to have at least two checkpoint areas and switch between them [64]. We can make the switch transparent to applications by mapping the logical checkpoint area (in the virtual address space) to a different set of physical addresses. Each application keeps a single pointer to the logical checkpoint area and the lock() function makes the switch by replacing one physical checkpoint area with another.

In our case, however, two checkpoints are not sufficient to ensure that

at least one of them is correct. An error may cause the application to checkpoint an incorrect state, call the lock() function and overwrite the second checkpoint area. Note that this sequence of events may occur for any arbitrarily small error detection latency.

To deal with this problem, we opted for having three checkpoints and imposing a minimum time between calls to the lock() function. The three checkpoints are used in a round-robin manner, where a lock() call always locks the most recent checkpoint and unlocks the oldest checkpoint. By having a minimum locking interval greater than the error detection latency, we can ensure that an error can affect at most the two most recent checkpoints. We can formulate this property as the following theorem.

Theorem 4.1. *If the minimum locking interval is greater than the maximum error detection latency, then, when an error is detected (and rollback is triggered) the oldest of three checkpoints is correct.*

Proof. Let δ denote the minimum locking interval and ϵ denote the maximum error detection latency. By definition of ϵ, an error detected at time t occurs within $[t-\epsilon, t]$. Clearly, in this time interval of length ϵ, at most one lock operation can be executed, since $\delta > \epsilon$. So we have two cases: in $[t-\epsilon, t]$, either the lock operation was not executed or it was executed exactly once.

If one lock operation was executed, then at most two checkpoints may have been affected – the most recent and the previous one. If no lock operations were executed, then the error may have affected the most recent checkpoint, but none of the other two. In either case the oldest of three checkpoints is correct. □

The theorem makes two implicit assumptions. The first is that all checkpoints contain a correct state when the first error occurs. All checkpointing schemes make this or a similar assumption which can be implemented, in our case, by taking three checkpoints at start-up. This means that an error occurring early in a program's execution will bring the computation back to its start. The second assumption is that error detection causes the execution of the errant application to stop immediately. Since error detection is handled by the operating system, this can be implemented by transferring the execution to the checkpointing service, which will in turn call the application's exception handler.

This theorem shows that using three checkpoints is sufficient, under our assumptions, to ensure the integrity of checkpoints. However, it abstracts away most of the details involved in creating a practical implementation of our lightweight checkpointing scheme. The following sections elaborate on the necessary implementation details to create an operating system service and describe the usage of model checking to verify its correctness, thereby increasing our confidence in that all details are taken into account.

4.5.4 Implementation Aspects

The checkpoint service must allow applications to allocate memory for the checkpoint areas, so that they can save state snapshots. In real-time embedded systems this operation is usually done statically by the linker. However, it is also feasible to introduce a system call for allocating checkpoint areas. In this chapter we adopt the static approach, which is the one used in the implementation of SECERN.

Another issue related to the configuration of the checkpointing service is that each application must define an exception handler. The checkpoint service transfers control to an application's exception handler when an error affecting that application is detected. The handler must be defined by the application designer in order to implement a lightweight rollback, *i.e.*, to restore the application's state from a stored checkpoint. After restoring the checkpoint, the exception handler should resume the normal execution of the application.

One important implementation detail is to ensure that the time between successive calls to the lock() function is greater than the minimum error detection latency. There are two possibilities: one may count the elapsed time or the number of instructions executed by an application between two calls. Modern microprocessors provide a wide range of performance counters that can be used to monitor diverse parameters of the execution. These can be used to count the number of instructions executed. Counting time is simpler, since we only need access to a timer, and may provide also accurate results.

Assuming that we are counting the number of instructions, the lock() function begins by checking whether the counter has incremented by a programmable amount since the application made the previous call. This implies that we must have a very accurate estimation of the maximum

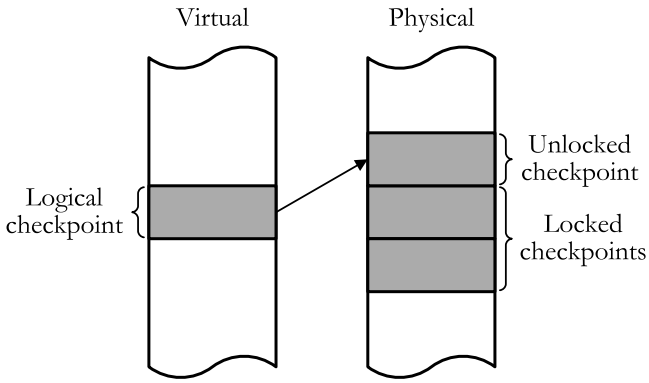

Figure 4.6: Logical checkpoint area (visible to the application) mapped to one of three physical checkpoints.

error detection latency. A possibility to make this estimation is to use fault injection testing. This is, however, not in the scope of this thesis. Consequently, we assume only that this parameter can be estimated with sufficient accuracy.

In addition to ensuring a minimum time between locks, the lock() system call replaces the checkpoint areas in a round-robin fashion. Using the address translation features provided by an MMU, it replaces the mapping between the virtual addresses seen by the application with one of three distinct sets of real addresses (containing the three physical checkpoint areas). This way, the existence of three physical checkpoints is invisible to applications. Each application has only a single pointer to a logical checkpoint area which is transparently remapped to another physical area when lock() is called. This is illustrated in Figure 4.6.

The lock operation must be atomic in order to ensure that no checkpointing is taking place when the checkpoint areas are switched. This issue is simple to solve since a system call is typically implemented using a CPU interrupt. In this case we know that no other instructions will execute before the interrupt is handled, *i.e.*, there is no concurrency between the lock() system call and the calling application.

The worst-case execution time of the lock() system call – the only run-time overhead introduced by our scheme – is also an important implementation detail. As mentioned above, this system call begins by checking that the maximum error detection latency has elapsed since

the last call (either in units of time or number of instructions) and then rotates the checkpoints. These operations are simple enough to find a bound on the execution time. Moreover, there are no complex operations involved, meaning that the total overhead to execute one lock() system call can be kept as low as few microseconds. For these reasons, we expect no difficulties in meeting the timing requirements of real-time embedded systems.

One final issue that needs to be considered is the content of checkpoints. This can be left to the application designers, since they have the best understanding of which program structures are essential. However, this may also be achieved by using a compiler-assisted technique [72]. In this chapter we address only the issue of ensuring the integrity of checkpoints. However, a future extension may consider making the entire process automatic by means of tools.

4.5.5 Verification using Model Checking

We used SPIN [73, 74] to verify the correctness of the design of lightweight checkpoints. This section describes in detail the formal model of the system, written in the PROMELA language. This language is formal enough to be verified by SPIN but maintains the typical constructs existing in a programming language. Thus, the code provided in the following sections shows our programming model and helps anyone attempting an implementation of the service to understand our scheme in detail.

Modeling the Application

In our programming model a designer stores the application state on the checkpoint area and issues a lock() system call. This is done in the app process, which is defined in Figure 4.7. The app process loops forever storing the application state on the checkpoint variable, which represents the pointer to the checkpoint area visible to the application. The application state is abstracted as a single bit variable, named app_state, which represents either a correct state (when its value is 1) or an incorrect state (when its value is 0). After saving its state, thereby making a lightweight checkpoint, the app process calls the lock() function by sending a message to the lock channel. This channel is read by the checkpointing service in Figure 4.9, as we describe later in this chapter.

```
bit  app_state  = 1;
bit  checkpoint = 1;
bool exception  = false;
chan lock       = [0] of {bit};

active proctype app() provided (!exception)
{
  do
  :: checkpoint = app_state;
     lock!0
  od
}

active proctype app_exception_handler() provided (exception)
{
  do
  :: app_state = checkpoint;
Lrollback:
     exception = false
  od
}
```

Figure 4.7: Application and exception handler models.

Note that the progress of the app process is non-deterministic, *i.e.*, it may or may not execute any instructions. However, it will only execute provided that no exception has occurred (*i.e.*, when the variable exception is false). The provided clauses force the execution of the two processes to alternate. By setting exception to true the app process becomes blocked and the app_exception_handler process starts executing. The exception is raised by the checkpointing service upon error detection.

The exception handler is a very simple routine that executes only a lightweight rollback, by setting the application state to whatever is contained in the checkpoint, and returns the execution to the main body of the application. This last part is done by setting the variable exception to false, which blocks the exception handler and resumes the execution of the app process. In summary, Figure 4.7 contains the PROMELA code corresponding to what should be implemented by the application designer.

The application's errant behaviour is modeled through a process that can only execute provided that the application state is incorrect (*i.e.*, the value of app_state is 0) and no exception has been raised. The code implementing this errant behaviour is shown in Figure 4.8.

```
active proctype app_errant()
provided (app_state == 0 && !exception)
{
  do
  :: checkpoint = app_state
  :: lock!0
  od
}
```

Figure 4.8: Model of the application's errant behaviour.

According to our assumptions, the errant behaviour of an application is non-deterministic. When an error occurs the application may overwrite the checkpoint area and make calls to the lock() function. It may also not execute at all, since we make no assumptions on the progress of a faulty applications. The app_errant process executes until an exception is raised or the application state becomes correct again. Note that the subtle change in behaviour is modeled by making it possible for the

app_errant process to call lock() without saving any checkpoints and *vice versa*.

Modeling the Checkpoint Service

The checkpointing service provides the lock() function called by the application. As shown in Figure 4.9, the checkpointing_service process waits for any message to be inserted in the lock channel and implements the functionality described earlier in this chapter. Whenever a lock() call is made, the checkpointing service will lock the most recent checkpoint and make the oldest one available to the application.

```
chan unlock = [0] of {bit};

active proctype checkpointing_service()
{
  bit cp1 = 1, cp2 = 1, tmp;

  do
  :: atomic {
       lock?_ ->
       tmp = cp1;
       cp1 = cp2;
       cp2 = checkpoint;
       checkpoint = tmp
     };
Llock:
     skip
  :: atomic {
       unlock?_ ->
       checkpoint = cp1;
       cp2 = cp1;
       exception = true
     }
  od
}
```

Figure 4.9: Model of the checkpointing service.

In Figure 4.9, the variables cp1 and cp2 represent the two checkpoint

areas which are invisible to the application. When the lock() system call is made the checkpointing service swaps the areas in a round-robin manner. Note that this is done atomically in accordance with the assumptions in the previous section.

The other functionality provided by the checkpointing service is to unlock the oldest checkpoint. To implement this, it waits for messages arriving at the unlock channel. Such messages may only be sent by the error detector. The response to an unlocking event is to make the oldest checkpoint available to the application and to copy the contents of that checkpoint (which must be correct) to all other checkpoint areas. When the unlock call terminates, the checkpointing service raises an exception (by setting exception to true), thereby triggering a lightweight rollback.

Modeling Error Injection and Error Detection

We defined an error_injector process that sets the app_state variable to 0 (representing an incorrect state) at any non-deterministic point in time of the execution, provided that an error is not already active. This process is shown in Figure 4.10.

```
active proctype error_injector() provided (app_state == 1)
{
  do
  :: app_state = 0;
Lerror:
     skip
  od
}

active proctype error_detector() provided (!exception)
{
  do
  :: (app_state == 0) -> unlock!0
  od
}
```

Figure 4.10: Error injector and error detector processes.

The error_detector process, also shown in Figure 4.10, implements the

error detection functionality. Whenever an erroneous state is found, the process may place a message on the unlock channel, thereby notifying the checkpointing process that an error has been found. This process executes provided that no exception is being handled at the moment, as the end result of a call to unlock is only to make the app_exception_handler executable.

Formal Specification and Verification

SPIN accepts correctness properties specified in Linear Temporal Logic (LTL) [74]. To verify a given LTL formula SPIN creates a *never* claim which consists of the negation of the LTL formula. The verification process consists of checking that there is no possible execution matching the negated formula.

We wish to verify that, when an error is detected, the application is able to rollback to a correct checkpoint. This property is only required to hold if the error detection latency does not exceed the locking interval. As observed in the proof of Theorem 4.1, this means that at most one lock operation can be executed between an error and its detection. Thus, we want to verify that

$$(\mathit{rollback} \rightarrow \mathit{correct\_checkpoint})\ \mathcal{W}\ \neg(\mathit{error\_injected} \rightarrow (\neg \mathit{lock}\ \mathcal{U}\ (\mathit{lock}\ \mathcal{U}\ (\neg \mathit{lock}\ \mathcal{U}\ \mathit{rollback})))).$$

The property should be read as: *rollback* implies *correct_checkpoint*, unless more than one *lock* occurs between *error_injected* and *rollback*.

The first part of the formula states that *correct_checkpoint* is implied by *rollback*. This is the fundamental property that we wish to verify. However, it is only required to hold *unless* the lock() function is called more than once before an error is detected. The symbols used in the LTL formula were defined in SPIN's LTL manager as follows:

rollback app_exception_handler@Lrollback

correct_checkpoint checkpoint == 1

error_injected error_injector@Lerror

lock checkpointing_service@Llock

In addition to the usual logic connectives, the above formula uses the temporal modal operators *until* ($\mathcal{U}$) and *unless* ($\mathcal{W}$), also known as *weak until*. Note that the weak until operator is not supported by SPIN but one can use the equivalence $p\,\mathcal{W}\,q \equiv p\,\mathcal{U}\,q \vee \Box p$, which uses the operator *always* ($\Box$), to circumvent this limitation.

We began by finding a counterexample that shows that two checkpoints are insufficient to ensure, under our assumptions, that at least one of them is correct. This was achieved by removing the cp2 variable from Figure 4.9. This effectively means that the checkpointing service would toggle between two checkpoint areas. SPIN takes a very short amount of time to find a counterexample with the following sequence of events: an error is injected, the application makes an errant checkpoint, locks that checkpoint and the error detector triggers a rollback recovery. In this case there is only one lock call and the application will rollback to the oldest checkpoint – which is incorrect.

The goal of the modeling effort was, nevertheless, to verify that our scheme is correct when using three checkpoints. Using the code provided in Figures 4.7 to 4.10, SPIN is able to search the state space exhaustively and confirm that the model is valid. Thus, we can have a very high confidence in that our scheme works as intended.

4.6 Related Research

A great deal of work has been dedicated to ensuring that operating systems are resilient to internal failures. In this context, kernel extensions such as device drivers are usually identified as a major source of problems. The *microkernel* approach attempts to solve this issue elegantly by isolating kernel extensions in user-mode, where fault containment can be more easily achieved. This design principle is used in the Minix operating system [67]. There is a price to pay for the increased reliability: obtaining an operating system service often involves full context switching and additional data copying. This performance penalty is worth the trade-off in systems where reliability is the main concern [75].

The approach implemented in Nooks [76] uses the more common *monolithic* kernel structure where extensions run in kernel-mode. It should be emphasized that kernel-mode instructions access main memory through the memory management hardware – just like user-mode

instructions. The difference is that user-mode execution has restricted access to privileged registers and instructions. If we abstract from malicious faults which replace instructions, device drivers can be isolated by marking unnecessary pages as read-only during their execution. The authors make use of this feature to implement lightweight protection domains. Additionally, they propose the usage of wrappers to monitor control-flow between the drivers and the kernel.

This chapter addresses the problem of checkpointing for real-time uniprocessor systems. A central problem when using checkpointing and rollback recovery in such systems is to ensure task schedulability. Given a failure hypothesis and a set of real-time tasks, one must determine if all tasks will meet their deadlines (both when errors occur and in fault-free cases). The work presented in [77] and [78], among others, studies the effect of checkpointing on the schedulability of fault-tolerant task sets.

An issue closely related to scheduling is the optimal checkpoint interval. If checkpoints are too frequent, their combined overhead is too high; if they are too sparse in time, recoveries may require too much re-computation. In real-time systems, the optimal checkpoint interval should maximize the probability of meeting deadlines when errors occur, while ensuring that deadlines can always be met in the error-free case [79]. Most mathematical models, both for general-purpose and real-time computing, assume not only the integrity of checkpoints but also that errors are detected instantly. Hence, our work can be applied to handle those assumptions.

Several existing implementations of platform services provide checkpointing for uniprocessors. These may be offered in the form of libraries supporting both transparent and non-transparent checkpointing, *e.g.*, by allowing programmers to specify which memory addresses should be excluded from the snapshots [80, 81]. Kernel- and user-level checkpointing techniques have the advantage of making it simple to protect checkpoints from faulty applications. However, these provide no automatic means for dealing with error detection latency and impose the additional overhead of calling external code, which is undesirable for real-time embedded systems.

Error recovery in communication systems has motivated extensive research on distributed checkpointing [82, 83]. Ensuring system-wide consistency when determining the recovery line may lead to the domino

effect, where a sequence of rollbacks brings all nodes to the beginning of the computation. This problem can be solved by creating globally coordinated checkpoints [84]. Some of the proposed schemes deal with error detection latency [85, 86]. These assume either that at most one checkpoint can be affected by an error or that an arbitrary number of past checkpoints can be stored. Coordinated checkpointing complements our scheme, since we consider uniprocessor systems and focus on protecting the integrity of checkpoints carried out independently by applications.

4.7 Summary and Discussion

This chapter presented SECERN – an approach for providing partitioning and fault tolerance to real-time kernels. SECERN includes several mechanisms to confine errors to the applications where they originate. These mechanisms are necessary for creating a partitioned environment which can be shared by multiple real-time tasks, possibly with distinct criticality.

We implemented several of these mechanisms as extensions to the μC/OS-II real-time kernel. The extension uses memory protection, processor exceptions, system call policies and application-specific checks to detect errors. These techniques were implemented taking into account that they must respect the requirements of real-time tasks, *i.e.*, they must introduce low overhead and, if possible, no execution time jitter.

A new fault injection plug-in was developed for the GOOFI tool, targeting the Freescale MPC5554 microprocessor. We conducted a series of fault injection experiments using the tool for testing the kernel extension. These experiments were conducted according to a methodology of focused fault injection, with the goal of diagnosing and removing design faults. They exposed two vulnerabilities in the extended kernel. Even though the tests are not exhaustive, they show the importance and benefits of using fault injection for the assessment of partitioned systems.

We identified several sources of uncertainty in our fault injection experiments. The first is related to the number of experiments. More experiments are required to remove any remaining faults and progressively reach confidence in that the kernel extension is fault free. Another source of uncertainty is the representativeness of faults. A bit-flip in the context of a single task is an example of a fault which must be tolerated

by the kernel extension. However, it would also be necessary to emulate software faults to verify the implementation. Lastly, the fault injection tool might have defects. We instrumented the code of the applications to ensure that the diagnosed flaws really existed. However, the analysis process may have failed to classify experiments correctly. These problems are shared with any fault injection tool and usually impose that the tool itself must be verified.

In addition to the mechanisms included in the extended real-time kernel, this chapter proposed a technique named lightweight checkpointing. It allows the application designer to decide the content and timing of checkpoints while providing a service for locking the checkpoint area using memory protection. The locking makes it possible to deal with failure modes where an application attempts to overwrite any previous checkpoints. To deal with error detection latency, the scheme uses three checkpoints, transparently to applications, and enforces a minimum time between calls to the locking mechanism.

We used the Spin model checker to verify the correctness of the checkpointing mechanism. One of the advantages of Spin is that it accepts the Promela language, which has a syntax similar to the C programming language. For this reason, the code provided in this chapter simplifies the work of understanding and implementing the lightweight checkpointing scheme.

The checkpoints are primarily intended for correcting errors caused by transient hardware faults. This chapter discussed how a recovery strategy can use the checkpointing mechanism to distinguish between hardware and software faults. If an error is detected again after a rollback, it assumes that the cause is a software fault. If this happens, the operating system transfers control to an application-specific exception handler, which the application designer can use to implement recovery for software faults.

CHAPTER 5

On the Efficiency of Fault Injection

Computer systems are increasingly being used in safety-critical applications such as aerospace or vehicular systems. To achieve the high safety levels required by these applications, systems are designed with fault tolerance mechanisms in order to deliver correct service even in the presence of faults. Faults may, for instance, occur when processors are disturbed by high energy particles such as neutrons or heavy ions. Such particles may sometimes interfere with the processor and cause an SEU – an error that typically changes the state of a single bit in the system.

In order to validate the correctness and efficiency of their fault tolerance features, safety-critical systems must be thoroughly tested. Fault injection has become an effective technique for the experimental dependability validation of computer systems. The objective of fault injection is to test fault tolerance mechanisms and measure system dependability by introducing artificial faults and errors.

A problem commonly observed during fault injection campaigns is that not all faults fulfill the purpose of disturbing the system. Often 80–90% of randomly injected faults are not activated [87, 88]. A fault placed in a register just before the register is written or faults that are injected into unused memory locations are examples of faults with no possibility of activation. In most tools the location and the time for fault injection

are chosen randomly from the complete fault-space, which is typically extremely large. The statistical implication of this is that the cost of obtaining appropriate confidence levels of the dependability measures becomes unnecessarily high.

To deal with this and other similar problems and to reduce the cost of validation through fault injection, two main classes of analysis techniques have been proposed: pre-injection and post-injection analysis [89]. Post-injection analysis aims at predicting dependability measures using the results of completed fault injection experiments. Pre-injection analysis instead uses knowledge of program flow and resource usage to choose the location and time where faults should be injected, before any experiment is performed.

This chapter presents a pre-injection analysis technique that is applicable to the injection of transient bit-flips into CPU user registers and memory locations. The bit-flip fault model is often used in fault injection experiments to emulate the effects of single event upsets and other transient disturbances.

The objective of the pre-injection analysis is to optimize[1] the fault-space from which the injected faults are sampled. The analysis uses program execution information to (i) eliminate faults that have no possibility of activation and (ii) find equivalence classes among faults and insert only one of these into the optimized fault-space. This is achieved by applying the following rule: *faults should only be placed in resources immediately before these are read by each instruction.* A bit-flip in any resource[2] will only manifest itself once this resource is read to perform an operation. Delaying the injection of the fault until the moment just before the targeted resource is read accomplishes the two objectives stated above. It should be noted that collapsing all faults in a given class into a single fault in the optimized fault-space may cause a bias in the estimated dependability measures (*e.g.*, error detection coverage). One of the objectives of this research is therefore to investigate the magnitude of this bias.

The pre-injection analysis technique was implemented in the GOOFI

[1]The word *optimize* should not suggest that the optimal fault-space is found but rather an improvement on the usual random approach. Further optimization is therefore achievable.

[2]In this chapter we use the word *resource* as a common term for CPU register, main memory locations and other state-elements where bit-flips may occur.

tool [46, 47], for Nexus-based fault injection [90, 88, 91], and is also suitable for implementation in other platforms. The effectiveness of the technique was assessed by comparing fault injection results with results obtained by non-optimized fault injection on the same target system. The system is based on the Freescale MPC565 [34] – a microcontroller aimed at the automotive and other control-intensive applications based on the PowerPC architecture. By applying assembly-level knowledge of this architecture we identify which resources are read by each executed instruction. This information, along with the time of the fault injections, is used to define the optimized fault-space, which is stored in a database. The fault injection experiments are then conducted by random sampling of faults from the optimized fault-space.

5.1 Related Research

The resources available in computers are usually greater than the needs of the applications executed. This fact motivates a first optimization by injecting faults only in used resources. Yuste *et al.* [88] take, in their experiments, special care to avoid placing faults in empty (*i.e.*, not used) memory regions. They obtained 12% effective faults and pointed out that random sampling from an unrestricted fault-space consisting of all possible fault locations (bits) and all time points is not a time-effective approach.

Avoiding unused memory regions might be done manually by analyzing the memory map of the application and choosing the segments (stack, heap, etc.) as valid locations for fault injection. This approach is quite simple but does not consider the dynamic usage of resources along the time dimension.

Studies conducted in the past have shown that error manifestation (rate and effects) is affected by workload [92, 93, 15]. In [94] the concept of failure acceleration was introduced by Chillarege and Bowen. They achieve fault acceleration by injecting faults only on pages that are currently in use and by using a workload pushing toward the limits in CPU and I/O capacity.

Güthoff and Sieh presented the operational-profile-based fault injection in [95]. They propose that the number of fault injections into a specific system component should be proportional to its utilization. Register

utilization is defined as the measure of the probability that an injected fault manifests itself as an error. Additionally, the times for fault injection are selected based on the data life-cycles. A data life-cycle starts with the initialization of a register (write access) and ends with the last read access before the next write access. Under the single bit-flip fault model, faults need to be injected only within the data life-cycles, just before each read access.

Benso *et al.* proposed a set of rules with the purpose of collapsing fault-lists in [96]. The rules reduce the fault-list without affecting the accuracy of the results of fault injection campaigns, by avoiding the injection of faults for which the behavior can be foreseen.

In [97], Tsai *et al.* introduced a technique denominated path-based injection. With this technique a fault is injected into a resource that will be used by the test program, given a particular input set. After manual derivation of the input sets, the path of execution is described in terms of a list of executed basic blocks. For each path, faults are only injected in the utilized resources.

Working in fault injection for testing fault-tolerant circuits, using VHDL models, a set of techniques for speeding up campaigns is described by Berrojo *et al.* in [98]. One of these techniques is named workload dependent fault collapsing. During the reference run (a fault-free execution to monitor and store a program's normal behavior) all read and write operations on memory elements are tracked with bit granularity. Having this log of read and write operations on each bit of each signal, at the circuit level, all possible bit-flips are then collapsed by (i) marking as silent all bit-flips between an operation (either read or write) and a write operation, and (ii) marking as equivalent all bit-flips between an operation (either read or write) and the subsequent read operation.

Arlat *et al.* [99] increased the efficiency of their fault injection experiments targeting the code segment by logging the control flow activated by the workload processes. If a randomly selected address for fault injection is not part of the log (an instruction trace), then the corresponding experiment can simply be skipped as the outcome is already known.

5.2 Fault-space Optimization Method

For single bit-flip fault injection, we define a fault-space to be a set of time-location pairs that determines where and when the bit-flip is injected. The time is selected from an interval during the execution of the workload selected for the experiment. The time granularity is based on the execution of machine instructions, *i.e.*, bit-flips can only be injected between the execution of two machine instructions. The complete (non-optimized) fault-space consists of all possible time-location pairs.

The fault-space optimization method presented in this chapter states that faults should only be placed in a resource immediately before the resource is read by an instruction. The following sections describe the input needed for the analysis, the output created and the optimization procedure.

5.2.1 Optimization Input

In order to determine the optimized fault-space it is necessary to gather information about the code of the application and the computer system executing it:

- Assembly code of the application;
- The Program Counter (PC) trace over time;
- The effective address of each memory read access;
- The definition of which resources are read by each assembly instruction.

In our experimental setup, the assembly code is textual information obtained by disassembling the executable binaries of the application, processed automatically by the optimization program. The Program Counter trace and the values of the General Purpose Registers are stored during the execution of the reference run. The effective address of each memory read access is calculated with these values. The definitions of which resources are read by each assembly instruction are built into the optimization program. These were obtained from Motorola's RISC CPU Reference Manual [100] and are available in [101].

5.2.2 Optimization Output

The resulting output (the optimized fault-space) consists of a list of possible locations and times for fault injection. The optimization procedure has been adapted to both one-shot applications and control applications executing in loops. Each element on the optimized fault-space contains the following information:

- Control loop index;
- Breakpoint address;
- Number of breakpoint invocations within the control loop;
- The fault injection location.

The **control loop index** is specific for control applications which execute in cycles. It defines the cycle during which a fault should be injected. For applications that do not execute in loops, the control loop index is always set to one. The **breakpoint address** specifies the breakpoint position inside the control loop and the **number of breakpoint invocations** specifies the number of times this breakpoint should be reached before fault injection.

5.2.3 Performing the Optimization

Using the Program Counter trace over time, the disassembled code of the application is parsed to obtain the sequence of assembly instructions executed. Each of the instructions is then analyzed in order to determine which resources the instruction reads. The pseudo-code for this procedure is presented in Algorithm 5.1.

The most important stage (line number 6 in the pseudo-code) is the identification of the resources read by each instruction. To accomplish this, the first step is to find the definition on the list matching the given instruction. This is done by matching the *opcode* and the *operands*. Then, by examining the possible assembly constructs, the symbols available in the *read list* of the definition are replaced by the resources actually read by the given instruction. Figure 5.1 illustrates this process.

In Figure 5.1, the instruction at address $39DE8_{16}$ adds *R10* to *R11* and stores the result in *R5*. The definition for this instruction is found

```
1  programTrace: Array holding the Program Counter trace over time.

2  foreach programCounter in programTrace do
3  |  controlLoopIndex := currentControlLoop();
4  |  breakpointInvocation := countInvocations(programCounter);
5  |  instruction := instructionAtCodeAddress(programCounter);
6  |  instructionReadList :=
   |  resourcesReadByInstruction(instruction);
7  |  foreach resource in instructionReadList do
8  |  |  usefulFault := ⟨ controlLoopIndex, programCounter,
   |  |  breakpointInvocation, resource ⟩;
9  |  |  storeIntoDatabase(usefulFault);
10 |  end
11 end
```

Algorithm 5.1. Pseudo-code for the optimization procedure.

in the table where the read list contains *rA* and *rB* – *R10* and *R11*, respectively. Since these are the two resources read by this instruction, two new time-location pairs are added to the optimized fault-space for code address $39DE8_{16}$ (the control loop index and the internal loop count are assumed to hold the specified values).

The second instruction, at address $39DEC_{16}$, fetches the memory word at the effective address *(R6)*+24 and stores it in *R7*. Its definition in the table specifies *rA* and *Mem32(d+rA)* – *R6* and the 32-bit word at 1000+24 – as being read. The value of *R6* (1000 in this example) is collected during the reference run. The two resources, along with the timings, are then added to the fault-space.

5.3 Experimental Setup

Figure 5.2 describes the evaluation platform used to evaluate the effectiveness of the optimization technique for experiments performed on the jet engine control software, which is one of two workloads investigated in this chapter. The GOOFI fault injection tool controls the experiments by using the winIDEA debugging environment in conjunction with the iSystem's iC3000 debugger. Faults are injected into the MPC565 microcontroller running the control software. In the case of the jet engine

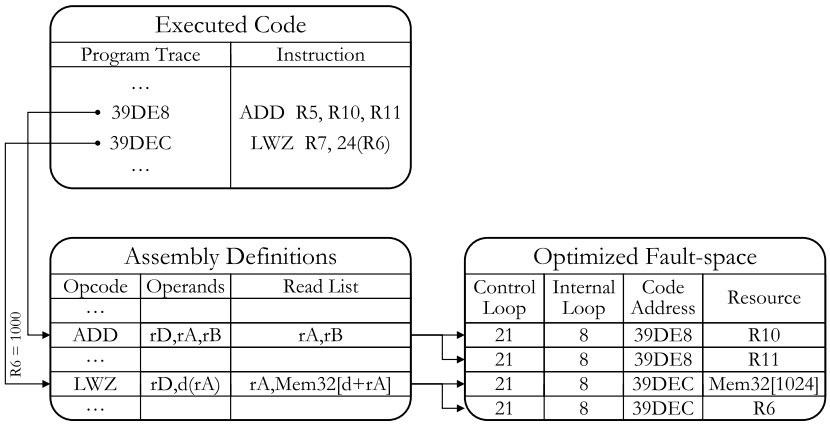

Figure 5.1: Example of the optimization procedure.

controller one computer board was used to run the jet engine control software and one board to execute the model of the jet engine. The experimental setup used for the other workload (an implementation of the quicksort algorithm) used only one computer board.

5.3.1 Fault Injection Tool

GOOFI is a fault injection tool developed at Chalmers University of Technology. It provides the ability to define and conduct fault injection campaigns on a variety of microprocessors. During each campaign GOOFI is responsible for controlling all the necessary software and hardware, and storing the acquired data into a database.

A plug-in [91] has recently been developed in GOOFI which uses the Nexus port [90] to inject faults on Freescale's MPC565. Nexus is an attempt to create a standard on-chip debug interface for embedded applications. This standard is suitable to be used for fault injection [88] since it provides read/write access to the processor's resources and code execution trace capture.

The pre-injection analysis technique was implemented to enhance the existing Nexus fault injection plug-in. The target platform for the current implementation is therefore the MPC565 microcontroller. The technique may however be implemented for any microprocessor.

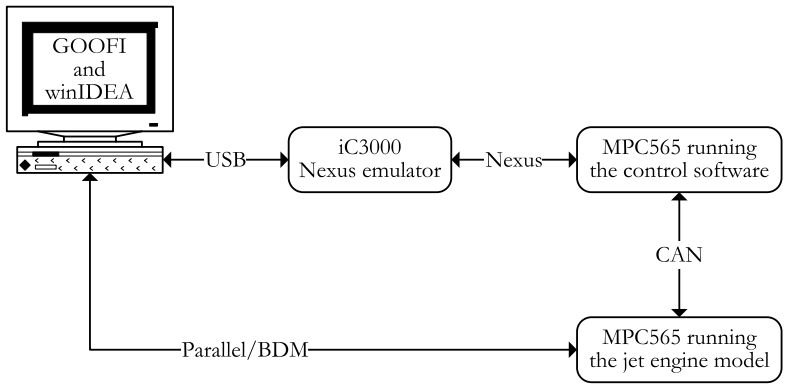

Figure 5.2: Evaluation platform for the jet engine application.

5.3.2 MPC565 Microcontroller

The MPC565 is a microcontroller originally developed by Motorola (who left the semiconductor market to its spin-off Freescale in 2004) that implements the PowerPC instruction standard architecture. It targets the automotive market as well as other control-intensive applications. The complete computer system was based on the phyCORE-MPC565 [102] development board. It includes Freescale's MPC565 processor, which offers a Nexus debug port, enabling real-time trace of program and data flow.

To establish a connection through this port the iSystem iC3000 Active Emulator was used to access the Nexus working environment. The iC3000 emulator was, in its turn, controlled by GOOFI via winIDEA – an integrated development environment offered by iSystem AG. GOOFI and winIDEA are executing on the same host PC.

5.3.3 Workloads

Fault injection campaigns were conducted to evaluate the optimization technique using two different workloads: a sort program using the quicksort algorithm and a jet engine controller. Different campaigns targeting registers and data memory, using both optimized and non-optimized fault selection, were carried out. The technique is fully implemented in the sense that all the assembly instructions executed by the workloads are

analysed and all registers and data memory locations where optimization is achievable with this method are considered. The outcome of each fault injection experiment was classified into one of the following categories:

- *Detected error.* All effective errors that are signaled by hardware error detection mechanisms included in the processor.
- *Wrong output.* All effective errors that are not detected by the processor but lead to the production of wrong results.
- *Non-effective error.* Errors that do not affect the system execution during the experiment time-frame.

Quicksort

The quicksort workload is a recursive implementation of the well-known sorting algorithm. It sorts an array containing seven double-precision floats.

The reference run execution takes two minutes during which the processor is being stepped and all the required data is obtained. The optimization procedure takes 20 seconds to complete. Each fault injection experiment takes less than half a minute to perform. During the execution of the reference run for this application, the MPC565 processor executed 34 distinct assembly instructions (opcodes) and a total of 815 instructions.

Jet Engine Controller

This workload is a control application that executes in loops in order to control a jet engine. At the end of each loop the controller has to produce results and exchange information with the engine (sensor values from the engine and actuator commands from the controller). It is significantly more complex than the quicksort program, allowing the fault-space optimization technique to be evaluated using a real-world application.

The execution of the reference run takes almost 12 hours. The optimization procedure takes 10 minutes to complete. Each fault injection experiment is then performed in less than two minutes for the selected configuration (number of control loops and memory locations to be logged).

Forty control loops of execution were logged during each experiment. From these, ten loops (21 to 30) were chosen as possible temporal locations for fault injection (corresponding to 50ms of real-time execution of the controller). During these ten control loops, in the reference run, the MPC565 processor executed 231.097 instructions. A total of 88 different assembly instructions (opcodes) were executed.

5.3.4 Fault Model and Fault Selection

The fault model applied is the single bit-flip model of the effects of transient faults. The technique assumes this model as the basis for optimization.

The faults in the non-optimized campaigns were chosen using a uniform distribution. In the case of the optimized campaigns the faults are selected randomly from the optimized fault-space itself (the list of temporal and spatial locations for fault injection described in Section 3.2). This implies that the distribution of faults in resources is proportional to the representation of each resource in the optimized fault-space.

Microprocessor registers were selected as spatial locations for fault injection both in the quicksort and in the jet-engine controller campaigns. Memory locations were only targeted using the jet-engine controller. The registers targeted in the non-optimized campaigns are the ones considered by the optimization method:

- General Purpose Registers (32 registers of 32 bits);
- Floating Point Registers (32 registers of 64 bits);
- Link Register (32 bits);
- Condition Register (32 bits);
- Integer Exception Register (32 bits);
- Count Register (32 bits).

These constitute the User Instruction Set Architecture (UISA) register set. User-level instructions are limited to this register set while supervisor-level instructions have also access to the Special Purpose Registers (SPRs).

Two limitations of winIDEA (the debugging environment) are important to mention. The floating point registers are only allowed to be injected with faults in the least significant 32 bits. These are the least significant bits of the 52-bit mantissa. The Floating Point Status And Control Register (FPSCR), targeted by the optimization, is also not available for fault injection.

The fault injection campaigns in memory targeted the stack, heap and all other read/write and read-only data segments of the controller. A total of 100 kB of memory were targeted as spatial locations.

The analysis of faults in the code segment was still not implemented and was therefore not studied. The optimization is easily extendable to support faults in the code segment by targeting, in each instruction, the 32-bit memory contents addressed by the Program Counter. This would be equivalent to the analysis performed in [99] by using the instruction trace.

5.4 Experimental Results

This section compares the results of random fault selection with those obtained using the pre-injection analysis. We describe the results of faults injected into microprocessor registers first, followed by the results of faults injected into memory locations.

5.4.1 Fault Injection in Registers

Table 5.1 shows the distribution of the outcomes of faults in the fault injection campaigns targeting microprocessor registers for both the quicksort and the jet engine controller workloads. The quicksort campaigns include approximately the same number of experiments. For the non-optimized jet engine controller campaign, a much higher number of experiments had to be performed in order to increase the confidence in the results.

The percentage of effective faults (either detected or producing wrong output) increases from 5.0% using non-optimized fault selection to 47.7% choosing faults from the optimized fault-space when targeting the quicksort workload. In the jet engine controller this increase is from 4.4% to 38.2%. The improvement in the effectiveness of faults is, therefore, one

Campaign		No. Exp.	Non-effective	Detected	Wrong Output
Quicksort	Non-optimized	2739	2603 (95.0%)	83 (3.0%)	53 (2.0%)
	Optimized	2791	1461 (52.3%)	744 (26.7%)	586 (21.0%)
Jet Engine Controller	Non-optimized	5708	5457 (95.6%)	200 (3.5%)	51 (0.9%)
	Optimized	1559	964 (61.8%)	466 (29.9%)	129 (8.3%)

Table 5.1: Distribution of outcomes of fault injection in registers.

Campaign		Error detection coverage (95% confidence)
Quicksort	Non-optimized	$61.0 \pm 8.2\%$
	Optimized	$55.9 \pm 2.7\%$
Jet Engine Controller	Non-optimized	$79.7 \pm 5.0\%$
	Optimized	$78.3 \pm 3.3\%$

Table 5.2: Error detection coverage estimations (faults injected in registers).

order of magnitude.

Table 5.2 shows the estimated error detection coverage obtained in each campaign. We here define error detection coverage as the quotient between the number of detected and the number of effective faults.

The values of the error detection coverage estimations are quite similar whether applying non-optimized or optimized fault selection. In the optimized campaigns the faults are only injected in the location that will activate them (at the time that the register is read). Since no weights are applied to reflect the length of the data life-cycle on the outcomes of faults, it could be expected that the error detection coverage would be skewed.

The detected errors were signaled by the exceptions provided in the MPC565 processor. The distribution among these exceptions is presented in Figures 5.3 and 5.4 for the quicksort campaigns, and in Figures 5.5 and 5.6 for the jet engine controller campaigns.

It is possible to observe that the detection mechanisms are activated in a similar but not identical way for the non-optimized and the optimized campaigns. Figures 5.3 to 5.6 provide an insight on the magnitude of the differences between non-optimized and optimized fault selection. A brief description follows of the most frequently activated exceptions.

- Checkstop (CHSTP) – The processor was configured to enter the

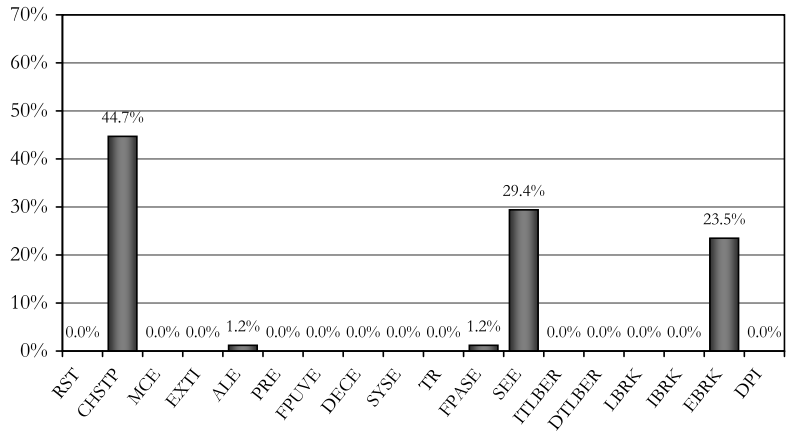

Figure 5.3: Exception distribution in the non-optimized quicksort campaign (83 faults in registers).

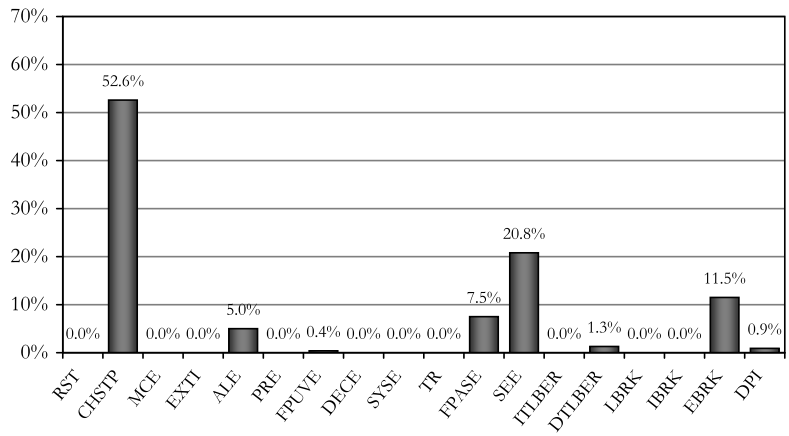

Figure 5.4: Exception distribution in the optimized quicksort campaign (744 faults in registers).

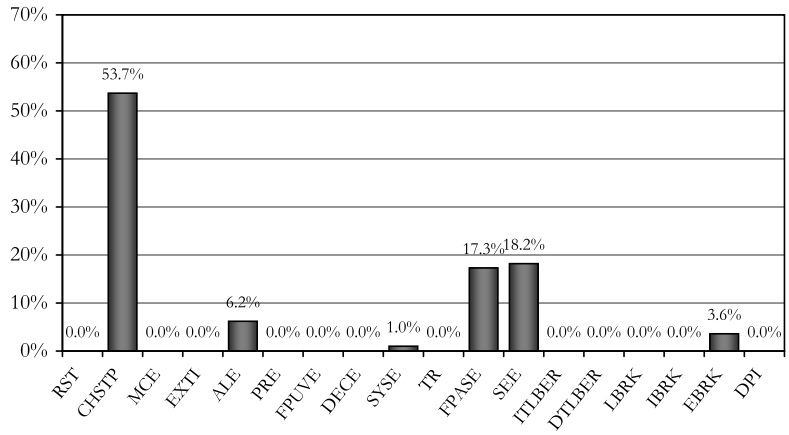

Figure 5.5: Exception distribution in the non-optimized jet engine controller campaign (200 faults in registers).

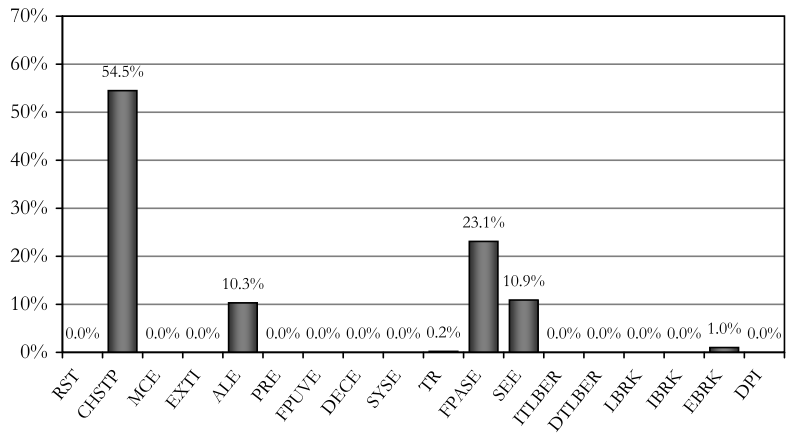

Figure 5.6: Exception distribution in the optimized jet engine controller campaign (466 faults in registers).

checkstop state instead of taking the Machine Check Exception (MCE) itself when the MCE occurs. CHSTP does not represent an actual exception, but rather a state of the processor. The processor may also be configured to take the MCE handling routine or enter debug mode. The MCE, which, in this case, leads to the checkstop state, is caused, for instance, when the accessed memory address does not exist.

- Alignment Exception (ALE) – The alignment exception is triggered under the following conditions:
  - The operand of a floating point load or store instruction is not word-aligned;
  - The operand of a load or store multiple instruction is not word-aligned;
  - The operand of lwarx or stwcx. is not word-aligned;
  - The operand of a load or store instruction is not naturally aligned;
  - The processor attempts to execute a multiple or string instruction.
- Floating-Point Assist Exception (FPASE) – This exception occurs in the following cases:
  - A floating-point enabled exception condition is detected, the corresponding floating-point enable bit in the Floating Point Status And Control Register (FPSCR) is set (exception enabled);
  - A tiny result is detected and the floating point underflow exception is disabled;
  - In some cases when at least one of the source operands is denormalized.
- Software Emulation Exception (SEE) – An implementation dependent software emulation exception occurs in the following cases:
  - An attempt is made to execute an instruction that is not implemented;

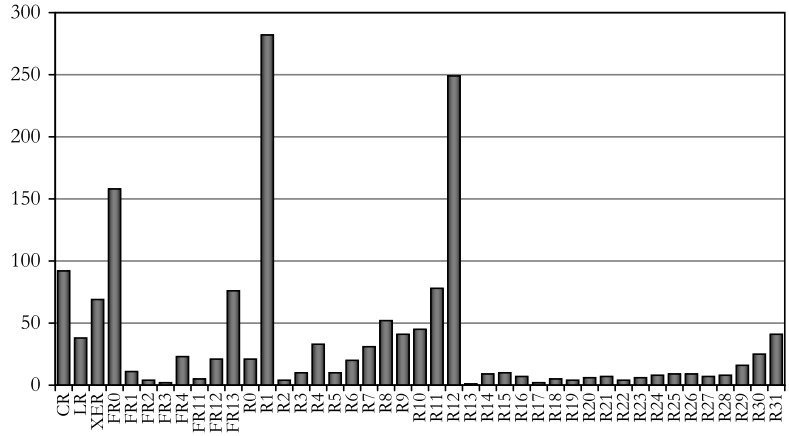

Figure 5.7: Number of faults injected in each register (1559 faults in the optimized jet engine controller campaign).

  - An attempt is made to execute an mtspr or mfspr instruction that specifies an unimplemented Special Purpose Register (SPR).

- External Breakpoint Exception (EBRK) – This exception occurs when an external breakpoint is asserted.

Figure 5.7 shows the distribution of faults over the processor registers in the optimized jet engine controller campaign (*cf.* Table 5.1). By using the optimization method, the number of faults injected in a given register is directly proportional to the number of times the register is read. The figure clearly demonstrates the non-uniform distribution caused by the optimization.

The stack pointer (R1 in the conventional usage of PowerPC processors) is targeted the most, followed by R12, which is used by the compiler very often to calculate effective addresses for memory operations. Register FR0 is also read very often in floating point calculations and the condition register (CR) is read by all conditional jumps.

Campaign		No. Exp.	Non-effective	Detected	Wrong Output
Jet Engine Controller	Non-optimized	6666	6532 (98.0%)	40 (0.6%)	94 (1.4%)
	Optimized	2658	2150 (80.9%)	166 (6.3%)	342 (12.8%)

Table 5.3: Distribution of outcomes of fault injection in memory.

Campaign		Error detection coverage (95% confidence)
Jet Engine Controller	Non-optimized	$29.9 \pm 7.7\%$
	Optimized	$32.7 \pm 4.1\%$

Table 5.4: Error detection coverage estimations (faults injected in memory).

5.4.2 Fault Injection in Memory

Fault injection in memory locations was performed only for the jet engine controller. Table 5.3 shows the distribution of the outcomes of faults for both non-optimized and optimized fault selection.

The effectiveness of faults increases from 2.0% using non-optimized fault selection to 19.1% choosing faults from the optimized fault-space. The improvement in the effectiveness of faults is one order of magnitude, similar to that obtained for faults affecting microprocessor registers. Table 5.4 shows the error detection coverage estimations obtained with non-optimized and optimized fault selection.

We here observe a similar pattern to that observed for microprocessor registers, where the error detection coverage estimation using non-optimized or optimized fault selection is quite similar. In this case the estimation from the non-optimized campaign is not very accurate since the 95% confidence interval is still wide due to the small number of effective faults (only 2% of the total).

Figures 5.8 and 5.9 show the distribution of detected errors among the exception mechanisms for the two campaigns. Again, it is possible to observe that the error detection mechanisms are activated in a similar but not identical way for non-optimized and optimized campaigns.

5.4.3 Fault-space Considerations

Applying the optimization method to the fault-space of registers for the jet engine controller resulted in the determination of 7.7×10^6 distinct

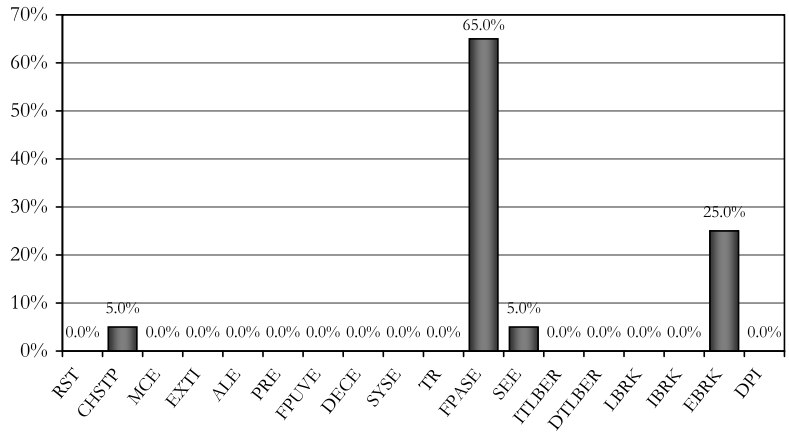

Figure 5.8: Exception distribution in the non-optimized jet engine controller campaign (40 faults in memory).

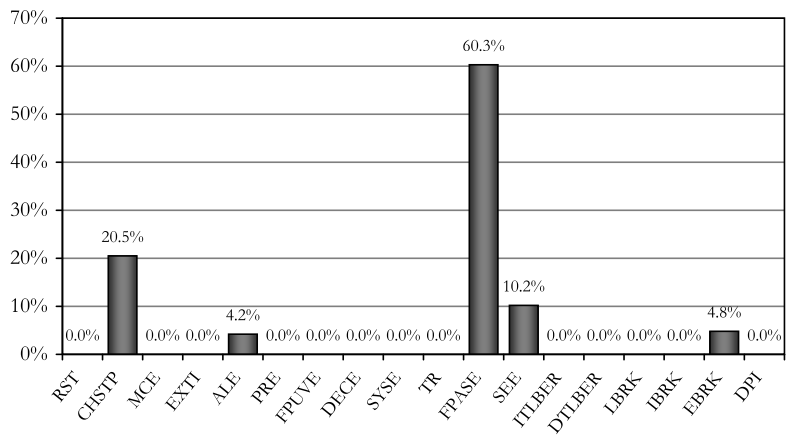

Figure 5.9: Exception distribution in the optimized jet engine controller campaign (166 faults in memory).

Campaign		Size of the fault-space (time-location pairs for bit-flips)
Jet Engine Controller	Non-optimized	5.0×10^8
	Optimized	7.7×10^6
Ratio		1.5%

Table 5.5: Comparison of fault-space sizes (registers).

Campaign		Size of the fault-space (time-location pairs for bit-flips)
Jet Engine Controller	Non-optimized	1.9×10^{11}
	Optimized	3.3×10^6
Ratio		0.0017%

Table 5.6: Comparison of fault-space sizes (memory).

time-location pairs for bit-flips. All the targeted registers are 32-bit registers[3]. The complete non-optimized fault-space of these registers is obtained by flipping each bit of each register, for each instruction executed. This results in a set containing over 500 million bit-flips. Table 5.5 summarizes these results.

In the case of the memory fault-space 3.3×10^6 possible time-location pairs for bit-flips were determined using optimized fault selection. The complete fault-space of memory is obtained by flipping each bit of each memory location used by the program, for each instruction executed. Considering a memory usage of 100 kB for data by the jet engine controller, the size of the complete fault-space is near 200 billion bit-flips.

5.5 Summary and Discussion

The study presented in this chapter shows the efficiency of eliminating faults with no possibility of activation and determining equivalence classes among faults. A comparison with traditional non-optimized fault selection (from the complete fault-space) shows an order of magnitude increase in the effectiveness of faults. The fault-space itself is reduced

[3]Floating Point Registers are 64-bits long, limited by the used version of winIDEA to the least significant 32-bits.

two orders of magnitude for the registers and four to five orders of magnitude for the memory. Even though these fault-spaces are still quite large when targeting the complete execution of programs, the exhaustive evaluation of small enough subroutines against all possible bit-flips becomes possible.

All faults targeting the same bit of a given resource, before this resource is read, are considered equivalent. This way, only one representative of these faults is injected. To obtain an accurate estimation of the error detection coverage (or any other dependability measure) it would be necessary to apply a weight corresponding to the number of faults in each equivalence class. However, the error detection coverage estimated by the optimized fault selection is found to be quite similar to the coverage estimated by non-optimized fault selection.

Even though activation of faults is ensured by the optimization technique (activation in the sense that the faulty resources are always utilized) not all faults result in effective errors. Even though the optimization increases the percentage of effective errors, a majority of the activated faults (both in registers and memory) is still non-effective. This occurs either when the data is used in a non-sensitive way by the code, or when the error remains latent within the time frame of the experiment.

There are several advantages in injecting faults in real-time, *i.e.*, without stopping the target processor. Furthermore, it would be interesting to inject faults into a set of tasks instead of isolated applications (*e.g.*, to test partitioning mechanisms). For such an experimental setup it would be interesting to use the optimization tool in real-time, without executing a fault-free experiment (golden run). The valid locations for fault injection would be chosen on-demand once the time point for fault injection had been defined.

The outcome of a fault is highly dependent on the targeted resource. Faults in some registers were observed to have a greater tendency to cause wrong output while faults in other registers cause detected errors more frequently. This motivates a possible evolution in fault selection by using the results of previous fault injection experiments to select the faults that should be injected next (a combination of pre-injection and post-injection analysis). It would be possible to achieve a faster evaluation of specific error detection mechanisms by injecting faults in the resources that are more likely to activate them.

Even though activation of faults is ensured by the optimization technique (activation in the sense that the faulty resources are always utilized) a majority of faults results in non-effective errors. An interesting topic for further studies would be to investigate which activated faults are non-effective and to find the reasons for this.

CHAPTER 6

Distributed Redundancy Management

A fault-tolerant system must be equipped with the means to detect and recover from faults, so that it can be dependable even under faulty circumstances. To achieve this, a key factor is the ability to diagnose faults and activate the appropriate isolation, reconfiguration and reinitialization mechanisms. In distributed systems, two primary goals of the recovery process are to isolate any faulty nodes and to reconfigure the system according to the remaining nodes in operation. Thus, working nodes must maintain a consensus on the nodes that should, and those that should not, participate in service delivery. The algorithms designed to provide this consensus are usually known as processor-group membership agreement protocols or, for short, membership protocols, where the word *membership* refers to the set of working nodes.

This chapter proposes a membership protocol intended to serve as a building block for distributed redundancy management. The protocol is suitable for synchronous systems, where it is executed in a sequence of rounds. It is especially designed for systems using time-triggered communication, where nodes broadcast periodically according to a predefined round-robin order, *i.e.*, the message schedule progresses in rounds. This method is adopted by communication standards such as FlexRay [103], TTCAN [104] or TTP [105] for scheduling static real-time traffic. Among

other factors, the design of protocols for such systems is constrained by the limited amount of available bandwidth, the failure assumptions and non-functional requirements such as reliability and availability.

We assume a generalized omission failure model where send/receive omissions can be either transient or permanent. The goal is to model systems where nodes communicate, in the presence of accidental faults, through a broadcast channel. The proposed membership protocol relies on nodes observing the periodic transmissions of other nodes to detect failures. Independent observations are unreliable and consensus on the membership (consistent observation of failures and repairs) is achieved by exchanging a configurable number of acknowledgements for each node's message.

Each sending node piggybacks k Boolean flags to its message so as to confirm or refute having received the messages from its predecessors, in the order of broadcast, that are in the membership. Increasing k makes the protocol resilient to a greater number of simultaneous or near-coincident failures but imposes a higher tax on the communication bandwidth. For this reason, the balance between protocol resilience and overhead can be adjusted, at design time, for each particular system. We expect this feature to be useful in improving the cost-effectiveness of real-time embedded systems.

To prevent redundancy exhaustion, it is essential to repair faulty nodes and allow them to join the group again. After handling an error locally (by employing backward or forward recovery) the node begins by retrieving the global state, which includes, particularly, the membership state. An important problem here is that the membership state is constantly changing – at least potentially – as failures occur. Furthermore, the nodes that are operating correctly must observe the recovery in a consistent manner. For these reasons, inclusion of nodes in the membership is designed to guarantee that all nodes include the repaired node or none of them does, and that the repaired node is only reintegrated if it agrees with the membership state.

In this thesis we consider group membership for systems relying on synchronous communication, where messages are transmitted within a known amount of time and nodes have a global notion of time. The membership problem in such systems was first described with detail in [106] and [107], addressing specific synchrony premises. Group membership

agreement is one of many consensus problems, which are at the core of fault tolerance for distributed systems [108].

6.1 System Model and Assumptions

We consider a distributed system composed of a set of processing nodes linked by a synchronous broadcast channel. We assume that the network has either a bus or a star topology. Processor nodes have their clocks tightly synchronized and execute a deterministic round-based schedule. In each communication round, nodes transmit a fixed amount of traffic in their pre-allocated transmission slots. For the membership protocol, it is sufficient to count time in terms of transmission slots.

We assume the existence of a reliable start-up mechanism and accurate clock synchronization mechanisms [109, 110] to maintain the system's synchrony. Nodes can identify the "current" slot number and, consequently, the sender of each message. This can be implemented, for example, by introducing unique message IDs to identify the sender or by using unique message lengths that act as implicit message IDs.

Each node has a single dedicated transmission slot in every communication round, which it uses to broadcast its messages. Processing nodes are assumed to be fail-silent, *i.e.*, either correct results or no results are produced, or fail-reporting, *i.e.*, either the correct result or a failure report, specifying the causes of failure, is produced. (The term fail-signaling is sometimes used instead of fail-reporting.)

Under fault-free conditions, a node will always send a message in its transmission slot. The physical link ensures that the message is delivered to all other nodes (*i.e.*, the receiving nodes). Under these circumstances, a failure occurs when a node does not receive an expected message. Such an event may be caused by a failure of the *sending node*, a failure of the *receiving node*, a *network failure* or a combination of these.

We assume that failures can occur in the nodes, their incoming and outgoing links (protocol processors which provide the interface to the network), and the network itself. To simplify the discussion about the kind of failures our protocol can handle, we map these failure types into four different failure modes according to their persistence – *permanent* or *transient* – and whether they affect the *sending* side or the *receiving* side.

6.1.1 Failure Modes

In our system model, a transient failure is assumed to affect a single message. If several consecutive messages are lost, for instance, due to electromagnetic interference on the network, then we consider this as a case of multiple transient failures. Permanent failures remain in the system until it is repaired, and may affect one node, its outgoing or incoming link, or a point-to-point connection between a node and the hub if the network has a star topology. (A permanent failure of a non-redundant bus network will lead to a failure of the entire system, and is thus not relevant for our membership protocol.) For the protocol, any failure with a duration greater than two communication rounds is considered permanent.

Regarding the impact on the system, we assume that faults lead to sending/receiving omission failures. Thus, one failure prevents one node either from sending or from receiving messages. A situation where some nodes receive a message correctly and two or more nodes receive the message incorrectly is assumed to occur only in the presence of multiple failures. Such cases can be dealt with by configuring the protocol appropriately. Table 6.1 shows how the different types of component failures are mapped to the four failure modes.

Component failures	Permanent	Transient
Sending node	Permanent sending omission	Transient sending omission
Outgoing link		
Network (outgoing)		
Network (incoming)	Permanent receiving omission	Transient receiving omission
Incoming link		
Receiving node		

Table 6.1: Mapping of component failures to failure modes.

The protocol allows all nodes to diagnose such failures in a consistent manner. The first three failure modes in Table 6.1 lead to exclusion of the faulty node, while transient receiving omissions do not. This feature is intended for systems where each node executes multiple tasks. In the event of a transient receiving omission, only a subset of the tasks is likely to be affected. Thus, excluding the entire node from service

delivery would disable correctly functioning parts of the system.

Other authors have also recognized the disadvantages of isolating nodes that suffer transient failures. In [111], a count-and-threshold strategy is adapted to a diagnostic protocol in order that faulty nodes are only isolated after exceeding a certain threshold on the number of transient failures. Their work, as well as ours, requires applications to be designed to tolerate omissions (or outages).

6.1.2 Rationale

The rationale for this failure model is to have a clear definition of what we mean by a failure, as we express the fault tolerance capabilities of our protocol in terms of the number of simultaneous or near-coincident failures the protocol can cope with. As previously explained, the number of simultaneous or near-coincident failures under which the protocol maintains agreement on the membership depends on the number of acknowledgement bits used.

From the viewpoint of healthy nodes, a failure of the sending node means missing at least one message from this node or receiving at least one failure report. In any of these cases the failure will be consistently detected by all healthy nodes. The same can be assumed when an outgoing link failure of the sending node occurs. These two failure types can therefore be classified as sending failures.

On the other hand, when a receiving node suffers a transient failure it will miss a single message. A transient incoming link failure will also have the same consequence. These failures are classified as transient receiving omissions. When the incoming link of a single node becomes permanently faulty we classify it as a permanent receiving omission.

To model communication failures we must consider the topology of the network [112]. We assume that the network is based either on a bus topology or a star topology. Our protocol can be used with both redundant and non-redundant networks. Common examples of redundant networks are duplicated buses or duplicated stars. The protocol may be used with other topologies as well but we restrict our analysis to these two, which are very common, since the first step is to clearly define the failure model. On a different network topology it may be necessary to model failures in a different way, thereby requiring changes to the way we express the fault tolerance capabilities of the protocol.

The network failure model is supported by the following analysis: We assume that applying structural redundancy will allow single transient faults to be masked by the physical layer. When the network uses a bus topology it is reasonable to assume that the probability of an error causing some nodes to receive the correct message while other nodes receive a corrupted version, is negligible. If this assumption does not hold (*e.g.*, slightly-off-failures are a concern), then the number of acknowledgement bits must be increased to the maximum value, to guarantee that each node acknowledges the messages from all other nodes.

When the star topology is used, network failures in the connection between the sending node and the star hub will be detected by all receiving nodes. On the other hand, failures occurring in a receiving node's connection to the star hub will only be perceived by this node. We assume that the hub itself will not introduce changes to this failure model. When all nodes miss a single message due to a transient network failure we have a transient sending omission. On the other hand, if only one receiving node misses a single message, a transient receiving failure has occurred. When a network failure is permanent, in a star topology, then either one node is unable to send messages (permanent sending omission) or one node is unable to receive messages (permanent receiving omission).

6.1.3 Node Restarts

The membership protocol provides the means for a restarted node to be included in the membership again. In fault-tolerant systems, the available redundancy decreases as permanent failures occur. Thus, restarting previously failed nodes and including them in the set of working nodes is key to ensuring sustainable service delivery.

When a failed node is able to restart, after a downtime period, we assume that fundamental data such as the communication schedule is undamaged. Furthermore, we assume that the node is able to synchronize itself with the active nodes, attempt to send messages and execute the protocol's reintegration routines.

6.2 The Membership Protocol

This section makes a detailed description of the membership protocol. For simplicity, we divide the explanation into three sub-protocols which, combined, achieve consensus on membership changes, *i.e.*, exclusion of failed nodes and inclusion of restarted nodes. The three sub-protocols are:

- *Agreement on exclusion*, which handles departure from the membership of nodes that have failed.
- *Inclusion ordering*, which supplies the number of the ongoing communication round to restarting nodes, so that they can establish an order of reintegration.
- *Agreement on inclusion*, that specifies how nodes attempt reintegration and how the remaining nodes achieve agreement on a successful inclusion.

We begin by introducing the notation and definitions used in the remainder of this chapter.

6.2.1 Notation and Definitions

Let $\mathcal{N}$ denote the set of processing nodes $\{N_1, N_2, \ldots, N_n\}$, ordered by the round-based schedule, where n is the number of nodes. Each node N_i maintains a local view $\nu_i(s)$ of the membership set, where $s \in \mathbb{N}$ and $\nu_i(s) \subseteq \mathcal{N}$. Intuitively, $\nu_i(s)$ is the view of the membership that node N_i has at the synchronous time-point s (at the end of transmission slot s).

The membership protocol relies on the periodic messages sent by each node to piggyback a sequence of acknowledgements. Each node will append k acknowledgement flags to its message, confirming (or refuting) the reception of each of the previous k messages *from the nodes in the membership*. An inclusion flag (*i-flag*) is also appended to each message to allow restarted nodes to be included in the membership. The periodic messages therefore respect the following format:

$$message = \langle data,\ ack_1,\ \cdots,\ ack_k,\ \textit{i-flag} \rangle.$$

The *data* field contains the payload of the message, which we ignore in this protocol specification. The *ack* flags, as well as the *i-flag*, are

Booleans and can thus be represented by a single bit. The three sub-protocols describe how the *ack* flags and the *i-flag* are set in response to certain events. The protocol responds to message *reception*, *loss* and *sending* events. These three events are mutually exclusive, *i.e.*, in any given transmission slot a node will either receive, lose or send a message.

A message loss may occur due to corruption of one or more bits during transmission. In other words, it is desirable to detect corrupted messages and discard them. To deal with this, it is common practice to protect physical frames with checksums – redundant information added to each message so that errors can be detected and, in some cases, corrected. Error detection is fundamental to ensure that the membership protocol utilizes uncorrupted information. To this end, we assume that protocol-specific data (*ack* flags and *i-flag*) are included in the message payload and protected with an effective checksum technique [113].

In our protocol a node is said to be *sponsoring* node N_j if it acknowledges, using one of its *ack* flags, the last message from N_j. Under normal conditions each node will have k sponsors (and will be sponsoring k nodes). If, in a given slot s, the membership set contains n_s nodes and $n_s \leq k$, *a node should not sponsor itself.* In this special case, each node will be sponsoring its $k_s = n_s - 1$ membership predecessors in order of transmission; otherwise, $k_s = k$.

We define the predicate $\mathit{lastSponsor}(N_i, N_j)$ as *true* if and only if node N_i is sponsoring N_j but the immediate successor of N_i in the membership is not sponsoring N_j. Intuitively, this states whether N_i is the last node to acknowledge the previous message from N_j.

We define a *failure report* as a message that has all flags (*ack* flags and *i-flag*) set to *false*. This special message is sent by nodes when they exclude themselves from the membership, *i.e.*, when they wish to inform other nodes that they have failed. We define an *inclusion request* as a message that has all *ack* flags set to *false* and the *i-flag* set to *true*. This message is sent by nodes attempting inclusion in the membership.

We define the predicate $\mathit{failure}(N_i, s)$ as *true* (in slot s) if and only if node N_i suffers a failure, of any kind, in slot s. When a specific failure mode is to be addressed, we use the predicates $\mathit{failure}_{ps}(N_i, s)$, $\mathit{failure}_{pr}(N_i, s)$, $\mathit{failure}_{ts}(N_i, s)$ and $\mathit{failure}_{tr}(N_i, s)$. We define the predicate $\mathit{restart}(N_i, s)$ as *true* if and only if node N_i has restarted in transmission slot s.

6.2.2 Agreement on Exclusion

Each node holds a `membershipView` set, representing its view of the membership. At the end of slot s, $\nu_i(s)$ equals the `membershipView` of node N_i. This set can be conveniently represented as a Boolean array containing n elements. We assume that the start-up mechanism supplies the set of initially active nodes, *i.e.*, $\nu_i(0)$, to the membership service of each node. The goal of the membership protocol is to ensure consensus on membership changes occurring after start-up.

The two events that trigger the *reactive* part of the membership protocol are message receptions and message losses. The *active* part of the protocol is triggered by a message sending event. We use a "pull" convention to model message sending, *i.e.*, the lower layers request a message from the membership service at a node's sending slot.

- In line 5 of Algorithm 6.1 a message is received and stored in the `msg` variable. The message sender is the owner of the current slot, represented by the `sid` variable.

- In line 18 a message loss event is reported (the slot time elapses and no message is received). The `sid` variable identifies the node which failed to send.

- A message sending event is reported in line 24, where the `msg` variable, representing the message about to be transmitted, is built (by setting the *ack* flags and *i-flag*) and sent.

Agreement on exclusion requires nodes to keep track of the received messages and their acknowledgements. A convenient way to do this is for each node to have a `presentNodes` set. This set is used to gather evidence that either a message or one of its acknowledgements has been received, from the sending node or from its sponsors, respectively. At start-up, the `presentNodes` set is initialized with the same contents as the `membershipView`.

In Algorithm 6.1, the `presentNodes` set is updated at four different locations. One location is line 21, when a message from N_{sid}, a node in the membership, is lost. That node is removed from the `presentNodes` set. Though an expected message from that node was lost, an acknowledgement might be received from one of its sponsors. Thus, the node

```
1  membershipView: Local view of the membership set;
2  presentNodes: Local view of the set of present nodes;
3  currentRound: Cyclic round counter (from 1 to 3n+4);
4  nextIFlag: Status of this node's i-flag on the next sent message;

5  On Message Reception:
6  msg: The received message;
7  sid: Sending node ID (the current slot number);
8  if N_sid ∈ membershipView then
9      if msg.i-flag = true and currentRound > 3 then
10         nextIFlag := true;
11     if msg = failure-report then
12         Remove N_sid from presentNodes;
13     Add acknowledged nodes to presentNodes;
14     exclusionDecision(sid);
15 else if sameView(msg) and currentRound = (sid × 3 + 2) then
16     nextIFlag := true;
17 inclusionDecision(sid, currentRound);

18 On Message Loss:
19 sid: Sending node ID (the current slot number);
20 if N_sid ∈ membershipView then
21     Remove N_sid from presentNodes;
22     exclusionDecision(sid);
23 inclusionDecision(sid, currentRound);

24 On Message Sending:
25 sid: This node's ID (the current slot number);
26 if N_sid ∈ membershipView then
27     Build msg acknowledging the sponsored nodes;
28     if 1 ≤ currentRound ≤ 3 or nextIFlag = true then
29         msg.i-flag := true;
30     else
31         msg.i-flag := false;
32     send (msg);
33     Remove N_sid from presentNodes;
34     exclusionDecision(sid);
35     inclusionDecision(sid, currentRound);
36 else
37     send (failure-report);
```

Algorithm 6.1. Pseudo-code of the membership protocol.

is kept in the `membershipView` temporarily until its last sponsor broadcasts. A second location is line 13. When a message is received from a membership node, the nodes that are positively acknowledged by that message are added to the `presentNodes` set. A third location is line 12, when a failure report is received from a node in the membership – this node will be excluded once its last sponsor broadcasts.

Last, a node removes *itself* from the `presentNodes` set upon message sending (in line 33). This is done to ensure that each node receives at least one acknowledgement for its own message; if this does not happen, the node suffered either a sending failure or a permanent receiving failure and must exclude itself from the membership. Note that once a node removes itself from the `presentNodes` set, it will only add itself again in line 13 if some other node acknowledges its message.

A given node N_j will be removed from the membership view of N_i if and only if N_i does not receive a message from N_j nor any positive acknowledgement for that message from any sponsor of N_j. Node N_i removes N_j from the membership immediately after the sending slot of the last sponsor of N_j. This is achieved by calling the `exclusionDecision` procedure at several locations in Algorithm 6.1. The pseudo-code for this procedure is shown in Algorithm 6.2.

The `exclusionDecision` procedure (line 1 of Algorithm 6.2) has two main functions. First, it excludes the nodes that are not in the `presentNodes` set by the time their last sponsor has broadcasted (line 4). This may be a self-exclusion of a node that does not receive any positive acknowledgement for its own message.

Second, it handles self-exclusion of nodes that have suffered permanent receiving failures. In line 6 a node removes itself from its membership view when the $k_s - 1$ messages from the preceding nodes in the membership have been lost. As we will describe later in this chapter, the protocol is resilient to $f < k_s - 1$ failures in any two consecutive rounds of communication; if a node loses $k_s - 1$ expected messages, then it concludes that it cannot receive any messages.

6.2.3 Inclusion Ordering

The protocol establishes a cyclic order that nodes must follow to attempt inclusion in the membership. The goal is to ensure that there are never two inclusions being executed at the same time. Ensuring inclusion or-

```
1  On Exclusion Decision:
2  sid: The current slot number;
3  if ∃N_j : N_j ∉ presentNodes and lastSponsor (N_sid, N_j) then
4      Remove N_j from membershipView;
5  if the last k_s−1 membership messages were lost then
6      Remove N_self from membershipView;

7  On Inclusion Decision:
8  sid: The current slot number;
9  currentRound: The current round number;
10 if nextIFlag = true then
11     local nextSlot := (sid mod n) + 1;
12     local nextRound := currentRound;
13     if nextSlot = 1 then
14         nextRound++;
15     if nextRound = (nextSlot × 3 + 3) then
16         Add N_nextSlot to membershipView;
17         Add N_nextSlot to presentNodes;
18         nextIFlag := false;
```

Algorithm 6.2. Decision procedures.

dering only requires nodes to agree upon the value of a cyclic counter of rounds. This cyclic round counter determines which node can join the membership in a given round.

For this purpose we define an *inclusion cycle* as a sequence of rounds where every node has three dedicated inclusion rounds. The length of every such inclusion cycle is $3n+4$ rounds, where n is the number of nodes. The round counter is therefore incremented by 1 each time a new round begins; if the value of the counter is $3n+4$, the next value is 1 (a new inclusion cycle begins).

Agreement on the round number is kept by the membership nodes as the communication schedule progresses, by updating the `currentRound` variable. A failed node is, however, unable to determine the round number unless active nodes explicitly signal it. The reason for this is that one cannot expect nodes to execute the protocol or maintain correct state information after a crash. To deal with this problem, the protocol supplies the round number to restarting nodes through a simple algorithm which uses the *i-flag* of working nodes in the membership. This service does not impose any additional overhead, since the *i-flag* is required to signal successful reintegrations (as described in the next section).

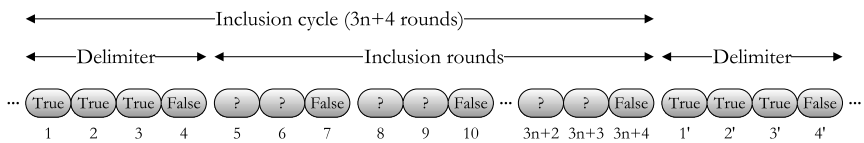

Figure 6.1: Round number signaling by a node in the membership, using the *i-flag* of its messages (one message per round).

During the first 3 rounds of an inclusion cycle, all sending nodes set their *i-flag* to *true*; on the fourth round their *i-flag* is set to *false*. This is done in lines 28 to 31 of Algorithm 6.1. The following $3n$ rounds of each inclusion cycle constitute the *inclusion rounds*, where nodes can send inclusion requests and attempt to join the membership. On every third inclusion round, nodes set their *i-flag* to *false* (done in line 31 of Algorithm 6.1). This method guarantees that the *i-flag* is set to *false* during, at least, one out of any three consecutive rounds. The only exception occurs intentionally during the first 3 rounds, where the *i-flag* is always set to *true*. Figure 6.1 depicts the inclusion cycle by showing

the state of a node's *i-flag* on the messages sent during an inclusion cycle.

Any restarting node synchronizes its round counter with the membership nodes by listening to their messages on the network. When the *i-flags* are observed to be *true* in three consecutive rounds, a restarting node sets its `currentRound` variable to 3. We note that receiving one message where the *i-flag* is *true* in each of those three rounds is enough to detect the start of an inclusion cycle.

6.2.4 Agreement on Inclusion

The procedure for agreement on node inclusion starts when a given node N_r synchronizes its round counter with the membership nodes. During the inclusion cycle, described in the previous section, node N_r has one dedicated round to *send its inclusion request:* round $3r+2$. No other node will send an inclusion request in this round since node IDs are unique.

An inclusion request is a special type of message which does not include the regular data payload sent by membership nodes. Instead, the message should include the membership view of the restarted node, acquired by listening to the ongoing messages, so that all other nodes are able to confirm that a successful inclusion is taking place. The concern here is that failures during restart would lead to a node being included in the membership without agreeing on the membership state.

There is no need for explicit broadcast of the membership state by active nodes. A restarting node listens to incoming messages and detects which other nodes are communicating (and therefore in the membership). It is sufficient for such a node to do this two rounds prior to sending its inclusion request, *i.e.*, during round $3r$ for node N_r. After that, the node should start executing the agreement on exclusion sub-protocol. This process is fault-intolerant, as a restarting node may obtain an incorrect membership view if failures occur. However, that node will be denied reintegration once its inclusion request is validated by the remaining nodes.

A given node N_r will be included in the membership if it sends an inclusion request in round $3r+2$ with a correct view of the membership. Since N_r is not in the membership, all receiving nodes perceive the message as an inclusion request (line 15 of Algorithm 6.1). Normal messages can therefore be distinguished from inclusion requests without any

additional message fields. Any nodes that receive an inclusion request compare their view to the restarting node's view (also in line 15). If the views are equal then the inclusion request is correct and the inclusion will be acknowledged by setting the *i-flag* to *true* in the next message to be sent (line 16 of Algorithm 6.1). When a correct inclusion request or its acknowledgement (through the *i-flag*) is received, the restarted node is included in the membership in round $3r+3$. The inclusion is completed in lines 15 to 18 of Algorithm 6.2.

Failures during inclusion attempts may prevent a restarted node from joining the membership. The restarting node may obtain an incorrect view of the membership; a sending failure may prevent the inclusion request from reaching the membership nodes. In these cases the inclusion will be unsuccessful and the restarted node must detect this condition and attempt inclusion in the next inclusion cycle. To achieve this, the node must verify if at least one received message contains the *i-flag* set to *true*, acknowledging its successful inclusion. If not, the restarting node must attempt inclusion at a later point in time.

6.2.5 Integration with Node-Layer Fault Tolerance

It is worth emphasizing an important feature of the protocol: it can be integrated with node-layer fault tolerance mechanisms, *i.e.*, error detection and recovery mechanisms executed locally at each node. First, it allows node-layer error detection mechanisms to notify the membership service that an error prevents a node from producing correct results. To achieve this, the node must exclude itself from its membership view upon internal error detection. In Algorithm 6.1, when the node is about to send a message, it checks whether or not it belongs to the membership. If it does not, a failure report is sent (line 37). This ensures that nodes exhibit a fail-reporting behaviour.

Second, the protocol is capable of providing accurate self-exclusion information to node-layer recovery mechanisms. When such mechanisms exist, they can access a node's view of the membership (locally available for each node) to check whether that node has been excluded from service delivery by the remaining working nodes. This feature allows nodes to rapidly trigger local recovery procedures upon faults that affect their ability to provide service at the system layer.

6.2.6 Tuning the Protocol

The membership protocol ensures membership consensus if no more than $f < k_s - 1$ failures occur in any two consecutive communication rounds. Under normal conditions, as we discussed earlier, $k_s = k$, where k is the number of acknowledgement flags on each message. The value k can be set to any number between 3 and $n - 1$.

For the protocol to work there must be, at any time, at least 3 membership nodes not subject to failures. This is the minimum required number of nodes; the nodes that are fault-free can vary with time. Periods of the execution when this assumption does not hold must be properly handled by blackout mechanisms, such as the ones used in TTP. During temporary blackouts the nodes attempt to maintain themselves in a safe state while monitoring the network. When other nodes start to recover it is possible to return to a normal operating mode.

Choosing the number k of sponsors per node defines the balance between resilience to failures and available resources – a classical trade-off in dependable computing. The choice of k is therefore application-dependent in the same way as choosing the strength of checksums for embedded real-time networks [113].

The first factor that one must weigh is the expected error probability for each message (counting only errors that are uncorrectable by the checksums). This factor is difficult to estimate. There is empirical evidence, combined with probabilistic analysis, substantiating that several consecutive transmission slots can be affected by electromagnetic interference [114]. The error probability in such extreme conditions is determined by the duration of the external events and by the length of each TDMA slot (shorter slots lead to lower error rates).

Nonetheless, in normal circumstances (assuming uncorrelated events) the error rates can be low enough to disregard simultaneous and near-coincident failures. The TTP/C protocol, for example, assumes at most a single failure in two consecutive rounds. This reasoning indicates the protocol can be safely configured with small values of k (close to 3).

If the possibility of correlated failures is to be addressed, then the value k must increase accordingly. However, there are fundamental limits on the achievable fault tolerance. An effective clique avoidance technique is to shut down nodes that view themselves as part of a sub-group containing less than half of the total number of nodes [115]. This pre-

vents uncoordinated and potentially hazardous actions from being taken by minority groups that can communicate only among themselves, by allowing only the majority clique to continue functioning. Consequently, regardless of the number of sponsors per node, a system cannot tolerate the failure of more than half of its nodes, meaning that reasonable values of k should still be low even considering correlated failures.

6.3 Prototype Implementation

We have implemented the membership protocol in a prototype of a distributed system that uses time-triggered communication. The network is based on COTS Ethernet hardware, programmed to schedule messages according to the Time Division Multiple Access (TDMA) method. This prototype implementation allowed us to test the feasibility and the performance of the protocol. Figure 6.2 depicts our experimental setup, which includes 6 processing nodes.

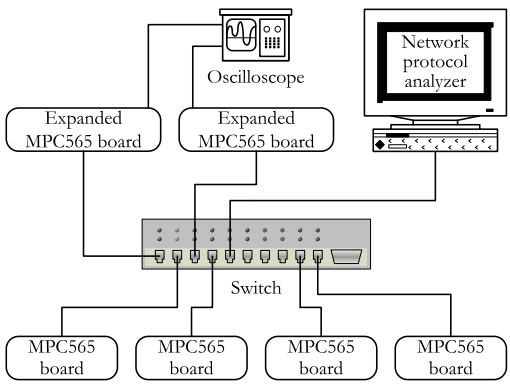

Figure 6.2: The experimental real-time Ethernet network.

The computer nodes in Figure 6.2 are Phytec's phyCORE-MPC565 development boards [102]. Each contains a Freescale MPC565 microcontroller, based on the PowerPC architecture. The boards include an RJ45 socket and an Ethernet controller. Additionally, the boards include controllers for CAN and serial communication which are unused in our setup.

The two boards shown on the upper-left corner of Figure 6.2 are

expanded with a custom board. We developed these expansion boards in order to output the internal clock of the nodes and to have a 7-segment display (for showing the number of active nodes in the membership). We connect the clock outputs to an oscilloscope in order to measure their synchronization. The expansion boards can be used with any processor board (to test for slight differences among nodes). Moreover, we connect a regular PC running Wireshark – a protocol analyzer – to the network, in order to verify the execution of the protocol. Each board executes a small software module that allows failure scenarios to be configured and tested.

The experimental network is based on a star topology with a central switch – HP's ProCurve Switch 2324. The Ethernet controller included in the boards runs at 10 Mbit/s (10Base-T standard). To maintain the TDMA schedule we implemented the daisy-chain clock synchronization algorithm [116]. This algorithm adjusts the clock of each node every time a new message is received. The adjustment is a fraction of the difference between the *expected* and the *actual* arrival time of a message.

6.3.1 Network Configuration

The length of Ethernet frames can vary between 64 and 1518 bytes. We used 64-byte packets in our experiments – 46 bytes of payload data, 4 bytes for the CRC checksum and 14 bytes for the MAC header. The MAC header identifies the source address (*i.e.*, the message sender) and the destination address, which is set to *broadcast*. With this configuration the estimated propagation delay for the Ethernet frames was 215 μs, with occasional variations of a few μs.

The duration of a transmission slot was configured to 400 μs for most tests (the lower bound for this parameter is $\sim$250 μs in our setup), resulting in 2.4 ms communication rounds that aim to be representative of real-time systems. Under these conditions the daisy-chain algorithm maintained the processor nodes synchronized within 3 μs. Table 6.2 summarizes the most important network parameters.

The membership protocol was configured to 4 sponsors per node. Each 64-byte packet therefore included 5 bits of membership information (4 acknowledgements and 1 *i-flag*).

Parameter	*Value*
Number of nodes	6
Transmission slot	400 μs
Communication round	2.4 ms
Reintegration cycle	52.8 ms
Packet size	64 bytes
Clock skew (measured)	< 3 μs

Table 6.2: Configuration of the real-time Ethernet network and resulting clock skew.

6.3.2 Network and Membership Performance

The nominal bandwidth of the network is 10 Mbit/s. However, real-time communication using TDMA must take into account propagation delays and clock skews to ensure that there are never two messages being transmitted at the same time. This is achieved by inserting guard times between consecutive messages. Due to these guard times, we estimate that our experimental network can achieve a maximum bandwidth of 3.3 Mbit/s using 1518-byte packets.

In our experiments, we used 64-byte packets and transmission slots of 400 μs, which results in a network bandwidth of 1.3 Mbit/s. This way, we can calculate the resource usage when the protocol executes at nearly the highest possible frequency for our setup. Since each frame reserves 18 bytes for the header and the CRC checksum, we have, for this configuration, 920 Kbit/s of *effective bandwidth* available for payload data (which includes the membership information).

In our experiments, each message had 5 bits of piggybacked membership information and messages were sent once every 400 μs. The bandwidth required by the membership service is therefore 12.5 Kbit/s. Since we have 920 Kbit/s of effective bandwidth available, the membership service imposes a 1.4% communication overhead. If we consider the network's nominal bandwidth of 10 Mbit/s, the membership's overhead is less than 0.2%. We emphasize that these values were obtained for 64-byte packet sizes, which provide the lowest effective bandwidth. Increasing the packet size would reduce the membership's communication overhead significantly.

A departure is detected by the group when the node's last sponsor

transmits its message. In the worst case, this may occur $n-1$ slots after the message is lost. Since a node may fail immediately after broadcasting a message, it may take n slots until a message is missed by the other nodes. The latency for agreement on exclusion is therefore $(6+6-1) \times 400\ \mu s = 4400\ \mu s$. This and other important latency values are shown in Table 6.3. It should be noted that these are calculated (not measured) values.

The worst case latency for reintegration occurs when node 6 (the last node) wishes to be reintegrated and starts listening on round 2 of the inclusion cycle; the node has to wait $3 \times 6 + 7 = 25$ rounds for the next complete delimiter pattern and then $3 \times 6 - 1 = 17$ rounds to be included in the membership.

Activity	*Latency*
Agreement on exclusion of a crashed node	4.4 ms
Fault-free inclusion from restart	100.8 ms
Recovery of the round number (included in the 100.8 ms)	57.6 ms

Table 6.3: Node departure and node reintegration latencies (worst case).

A direct implementation of our protocol requires nodes to acknowledge their immediate predecessors. An important concern is therefore to ensure that nodes have enough time to react to received/lost messages. In our experimental setup, we have verified through extensive testing that the nodes were able to send their acknowledgements on time. However, for systems where nodes have a long reaction time, the order of the acknowledgements can be set in a way such that node N_i sponsors the nodes starting at N_{i-2}, instead of sponsoring its immediate predecessor.

Another important aspect of the implementation of membership protocols is that the processing capacity of nodes may be very limited. For our experimental setup, we estimate that the size of the code related to the membership service is less than 4KB; the data structures occupy 42 bytes in memory. We measured the CPU usage with and without the membership service enabled and observed that the CPU overhead of the membership service is negligible.

6.4 Related Research

This chapter addresses the group membership problem for synchronous systems. The seminal efforts described in [106] and [107] were followed by many solutions for systems relying on synchronous communication [117, 118, 119, 120]. More recently, the membership problem has been clarified [121] and the design of membership services has been improved with respect to modularity [122] and configurability [111].

Group membership has also been widely studied in the context of asynchronous systems [123, 124, 125, 126, 127]. In such systems, the challenge is in finding the best way of dealing with the well-known result that consensus is impossible under complete asynchrony [128]. One noteworthy approach is to make systems partly synchronous by constructing "wormholes" [129].

Closely related to our line of research – where systems are characterized by their reduced bandwidth and strict dependability requirements – is the TTP communication protocol [105]. It includes a membership service that provides agreement under the assumption that there is, at most, a single failure in any two consecutive rounds. Our protocol, in contrast, is able to cope with multiple simultaneous or near-coincident failures. Furthermore, TTP requires the membership state to be periodically broadcasted to support node inclusion. In our approach, nodes recover the membership state by listening on the network.

A solution that isolates TTP's membership protocol from the CRC mechanisms was presented in [118]. Their protocol uses a single acknowledgement flag to ensure that faulty nodes are promptly removed from the membership under the single failure assumption, whereas our approach imposes a minimum overhead of three bits to implement a similar functionality. However, their scheme does not provide inclusion capabilities and, in fact, the sub-protocols that we propose for inclusion ordering and agreement on inclusion can be used with their solution, as well as TTP's, to guarantee a reliable restart process. This would require adding only the *i-flag* to their message format.

The protocols proposed in [130] and [117] require nodes to send the complete membership vector along with all periodic broadcasts. The drawback of this approach is that the overhead grows quadratically with the number of nodes. An approach to minimize the effect of this problem is to send the membership vector only when there are membership

changes [131, 132]. This method is viable in networks that provide event-driven scheduling in addition to the static schedule. In comparison to these protocols, our protocol can be configured with the maximum value $k = n - 1$ to achieve a similar degree of fault tolerance, thereby requiring the same bandwidth. However, the value k can be decreased, providing a trade-off between resilience and communication overhead.

In [133] a solution based on a variable number of sponsors is presented, but in a context unrelated to hard real-time systems. Ref. [134] briefly and informally presents a protocol based on a variable number of sponsors as well. In that scheme, permanent node failures lead to a rapid decrease in the reliability of the protocol and prevent nodes from reintegrating the membership (when none of their sponsors are working). On the other hand, reintegrating nodes is simple and does not incur additional overhead.

6.5 Summary and Discussion

This chapter proposed a group membership protocol for guaranteeing consistent views of failures and restarts among nodes in a distributed system – a building block for system-layer fault tolerance. The protocol is especially designed for systems using time-triggered communication. From the design perspective, it tolerates a configurable number of simultaneous or near-coincident failures. This provides the system designer with the ability to adjust the reliability of the protocol to the available resources.

Moreover, the protocol supports inclusion of restarted nodes under the same failure assumptions as exclusion. One problem in achieving this is that, before joining the group, a restarted node must recover the correct membership state (which may change as failures and other restarts occur). This issue is solved by establishing a cyclic order that nodes follow to send inclusion requests. The sub-protocols that provide inclusion ordering and agreement on inclusion can be used to extend other solutions existing in the literature which provide no means for including nodes in the group.

An important feature of the protocol is that it can be integrated with node-layer fault tolerance mechanisms. First, it is capable of providing accurate self-exclusion information to node-layer recovery mechanisms.

This feature allows nodes to rapidly trigger local recovery procedures when they are excluded from service delivery by the remaining nodes. In this situation, a node can recover independently while the remaining working nodes continue providing service. Once the local recovery is complete, the node may send an inclusion request and join the group.

Second, it allows node-layer error detection mechanisms, executed locally at each node, to notify the group membership service that an error prevents a node from producing correct results. In this case, the usual approach is to ensure fail-silence, *i.e.*, the node sends no more messages. In contrast, our protocol can send a failure report upon error detection. The practical outcome of using fail-report semantics is that node failures are not interpreted by other nodes as communication failures. As we show later, this has a positive impact on the protocol's reliability.

CHAPTER 7

Formal Verification of Consistent Diagnosis

A challenge in the development of distributed algorithms is to ensure that they are free from design faults. This is especially relevant when designing fault tolerance mechanisms, which are introduced with the exclusive goal of improving system dependability but have the potential to generate severe failure modes when poorly designed [27]. With this in mind, we chose to examine the correctness of our membership protocol using model checking.

Over the past years, automated formal methods have become an attractive way to increase the confidence in that a design is fault-free. We used SPIN [73, 74] – a well established model checker for distributed software systems – to formally verify the correctness of our protocol. Model checking tools work on models of the system, which can be built before the actual implementation takes place. Thus, one of the advantages of model checking is the ability to detect design faults at early development stages.

Model checking is a process for verifying whether a model fulfills a given specification. A model is an abstract description of a system, written in a formal modeling language. A system's specification is a set of properties, or logical formulæ, which the system is expected to satisfy. Model checking tools accept a model and its specification as input. Their

output is either "valid", when the model is correct, or a counterexample, *i.e.*, a case where the correctness properties are violated.

SPIN is an explicit-state model checker. As such, it builds a graph of the reachable system states; each vertex explicitly represents a global system state and each edge represents a possible state transition. Verifying a property consists of checking that it holds in all vertices reachable from the initial system state. Explicit model checkers are affected by the well known problem of state-space explosion. As a model grows, so does the number of possible global states. Visiting all reachable states often becomes a computationally expensive problem.

The rapid evolution of computers gave researchers access to enough memory and processing power for dealing with fairly complex problems. Paper proofs are common in the literature but may overlook special cases, as it was the case in [118], which contained flaws subsequently discovered using automated tools. This has led researchers to advocating the general use of automated formal methods for verifying sensitive algorithms [135]. Due to their criticality, membership protocols have been the object of formal verification using techniques such as model checking [136] and theorem proving [137].

This chapter describes how we modeled the protocol and presents the results of the exhaustively verified model instances. We begin by specifying the correctness properties and continue by detailing the formal models of the protocol and the time-triggered communication channel. Finally, we discuss the results of the verification process and summarize the main conclusions of the chapter.

7.1 Formal Specification of the Protocol

We begin by specifying the set of correctness properties which should hold throughout the execution of the protocol. These are first expressed using predicate logic and then translated into LTL or assertions – the two methods for specifying properties in SPIN. We consider four *safety* properties, that ensure nothing wrong happens throughout the execution, and two *liveness* properties, that ensure something useful will eventually happen during the execution.

- *Agreement.* Any two non-faulty nodes have the same view of the membership: $\forall s, \forall N_i, N_j \in \mathcal{N} : \neg\exists s' : s' < s \wedge (\text{failure}(N_i, s') \vee$

$$\text{failure}(N_j, s')) \implies \nu_i(s) = \nu_j(s).$$

This property specifies that the membership state should be consensual among nodes that have never failed. Consequently, it does not constrain the behaviour of nodes that have just failed (and are temporarily still part of the group) nor that of nodes that have failed in the past but recovered successfully. We therefore introduce the *integrity* property.

- *Integrity.* Any two nodes – faulty or non-faulty – that include themselves in their own view of the membership have the same view of the membership: $\forall s, \forall N_i, N_j \in \mathcal{N} : N_i \in \nu_i(s) \wedge N_j \in \nu_j(s) \implies \nu_i(s) = \nu_j(s)$.

This property pertains to all nodes, including those that have failed. Note that crashed nodes may stop executing the protocol and their notion of membership view becomes irrelevant. Crashed nodes are assumed to be fail-silent or fail-signaling, and we can therefore map a node's crash to a permanent sending omission. If a failed node is still executing the protocol, then it will agree with the membership state until it excludes itself from its own view (it becomes self-excluded). Otherwise, it becomes silent in order not to cause any damage to the system.

- *Accuracy.* Fault-free nodes only exclude faulty ones from the membership: $\forall s, \forall N_i, N_j \in \mathcal{N} : (\neg\exists s' : s' < s \wedge \text{failure}(N_j, s')) \wedge N_i \notin \nu_j(s) \implies \exists s'' : s'' < s \wedge \text{failure}(N_i, s'')$.

Naturally, it is necessary to prevent situations where a healthy node is excluded from the group of operational nodes. This is specified by the *accuracy* property. Furthermore, as we discussed in the previous chapter, nodes that are excluded from service delivery at the system layer should rapidly initiate node-layer recovery. To achieve this, we introduce the *self-exclusion* property.

- *Self-exclusion.* A node excluded by fault-free nodes also excludes itself from its view of the membership: $\forall s, \forall N_i, N_j \in \mathcal{N} : (\neg\exists s' : s' < s \wedge \text{failure}(N_j, s')) \implies (N_i \notin \nu_j(s) \implies N_i \notin \nu_i(s))$.

The above four safety properties define the membership view of each node with respect to all other nodes – fundamentally, nodes either agree on the membership state or self-diagnose as faulty. However, these safety

properties alone do not rule out trivial solutions. If, for example, all nodes keep all other nodes in their membership view, then these safety properties would be trivially fulfilled. We therefore specify two liveness properties which guarantee that the system does react to node failures and restarts in the appropriate way.

- *Exclusion liveness.* A node that suffers a sending failure (transient or permanent) or a permanent receiving failure is eventually excluded from the views of fault-free nodes: $\forall N_i \in \mathcal{N}, \exists s : \text{failure}_{\text{ts}}(N_i, s) \vee \text{failure}_{\text{ps}}(N_i, s) \vee \text{failure}_{\text{pr}}(N_i, s) \implies \exists s' : s' > s \wedge (\forall N_j \in \mathcal{N} : (\exists s'' : s'' < s' \wedge \text{failure}(N_j, s'') \vee N_i \notin \nu_j(s'))$.

- *Inclusion liveness.* A restarted node is eventually included in the membership if no failures occur: $\forall N_r \in \mathcal{N}, \exists s : \text{restart}(N_r, s) \wedge (\forall N_i \in \mathcal{N}, \neg\exists s' : s' \geqslant s \wedge \text{failure}(N_i, s')) \implies \exists s'' : s'' > s \wedge (\forall N_j \in \mathcal{N} : (\exists s''' : s''' < s'' \wedge \text{failure}(N_j, s''')) \vee N_r \in \nu_j(s''))$.

A limitation of these liveness properties is that they are unbounded in time, *i.e.*, they make no restrictions on the amount of time it should take for excluding/including nodes. Bounded liveness is important for real-time systems which require well-known limits on the time required for handling errors. As we will see in the following sections, SPIN provides convenient ways of specifying properties that should eventually hold. However, placing bounds on liveness usually requires changes to the system model in order to count the passage of time.

Another possibility is to determine the liveness bounds through testing, as a complement to formal verification. One of the parameters which is known in an experimental setup is the duration of each transmission slot. This makes it possible to measure the actual time it takes to handle errors and repairs.

7.2 System and Protocol Models

The formal modeling language accepted by SPIN is called PROMELA. The PROMELA language is appropriate for defining finite-state transition systems. Concurrent processes can be specified using inter-process communication via global variables (to model shared memory) or via message channels that can be synchronous or asynchronous. We used synchronous channels in order to model a synchronous system.

7.2.1 The Broadcast Channel

The PROMELA language does not provide broadcast channels. There are, nevertheless, many simple ways to model broadcast channels using the existing point-to-point channels. We defined a *broadcast process* that has one incoming channel and n outgoing channels – one to each processor node. The broadcast process notifies events to nodes by sending messages to their individual channels. The channels and the data structures are defined in Figure 7.1.

```
mtype = {MSG_RECEPTION, MSG_LOSS, MSG_SENDING};

typedef message
{
  bool ack[K];
  bool iFlag
};

chan toNetwork = [0] of {mtype, message};
chan toNode[N] = [0] of {mtype, message};
```

Figure 7.1: Data structures for the broadcast channel.

The broadcast process consists of a simple `do` loop that (i) reports a message sending event to the owner of the current slot, (ii) waits for the node to send its message and (iii) distributes that message to all other nodes. The PROMELA code for the broadcast process is shown in Figure 7.2 (note that the `for` macro is replaced by a `do` loop during pre-processing).

7.2.2 The Processor Nodes

Each processor node has its local view of the membership state, which can be represented as a Boolean array containing n elements. There are n local views in the system, totaling $n \times n$ Booleans. The necessary data structures are shown in Figure 7.3. The views of all nodes are global variables of the model, as this is the most convenient way for specifying the properties through assertions and LTL formulas. However, each node accesses only its own local view.

```
do
:: failureInjector();
   toNode[currentSlot] ! MSG_SENDING(DUMMY_MSG);
   toNetwork ? msgType(msg);
   for(i,0,N)                          /* broadcast the message */
      if
      :: i != currentSlot ->
         if
         :: !failureTS && !failurePS[currentSlot]
            && !failureTR[i] && !failurePR[i] ->
            toNode[i] ! msgType(msg)      /* deliver message */
         :: else ->
            toNode[i] ! MSG_LOSS(DUMMY_MSG)       /* failure */
         fi;
         timeout   /* wait for the receiving node to execute */
      :: else -> skip
      fi
   rof(i,0,N);

   currentSlot = (currentSlot + 1) % N;         /* next slot */
   if
   :: currentSlot == 0 ->                       /* new round */
      currentRound = (currentRound + 1) % (3*N+4);

      failuresLastRound = failuresThisRound;
      failuresThisRound = 0       /* update failure counters */

   :: else -> skip
   fi
od;
```

Figure 7.2: The broadcast process.

```
typedef membershipView
{
  bool view[N]
};

membershipView localView[N];
```

Figure 7.3: The membership views of all processor nodes.

Modeling the protocol from the viewpoint of membership nodes respects the structure of Algorithm 6.1. Each node is a PROMELA process that responds to the events notified by the broadcast process, as shown in Figure 7.4. Algorithms 6.1 and 6.2 are, in essence, a compact version of our PROMELA code.

```
do
:: toNode[nodeID] ? nMsgType(nMsg) ->
   if
   :: nMsgType == MSG_RECEPTION ->
      /* On Message Reception */
   :: nMsgType == MSG_LOSS ->
      /* On Message Loss */
   :: nMsgType == MSG_SENDING ->
      /* On Message Sending */
   fi
od;
```

Figure 7.4: Structure of the processor nodes, where the comments represent the code in Algorithms 6.1 and 6.2.

We prevent any interleaving of instructions among node processes. In PROMELA, we specify this by using the `timeout` statement, which blocks the *broadcast process* until each receiving node completes processing the message reception/loss event, before distributing the message to the next node. The more commonly used `atomic` statements accomplish a similar effect. The reason for modeling broadcasts in this way is that the state-space becomes significantly smaller. However, this is not the general way of modeling synchronous broadcast channels and is only suitable for time-triggered systems.

The difference between synchronous systems and time-triggered systems is essentially one of scheduling. A system is said to be synchronous when it meets two conditions:

- There is an upper bound on message transmission delays.
- Processes make progress and take actions within known amounts of time.

These two properties make it possible to synchronize processes and im-

plement time-triggered schedules where nodes follow a predetermined round-based order for sending their messages. Time-triggered systems are therefore a special case of synchronous systems. The fact that nodes make tightly synchronized progress makes it possible to verify the system without considering arbitrary orders in the execution of instructions. This facilitates the verification process significantly.

7.2.3 Modeling Failures

Failures are modeled by having the broadcast process call a failure injection routine at the beginning of each new transmission slot. We abstract away failures occurring at intermediate steps of the execution. This abstraction is possible since the impact of failures occurring during a transmission slot is the same of failures that occur at the start of that slot. We can use this abstraction to limit the possible interleaving of failure injection instructions with protocol instructions.

We use three Boolean arrays and one Boolean variable to keep track of which type of failure affects each node. The failure injection routine consists of a non-deterministic set of actions that update these arrays, according to the failure model described in the previous chapter. Each of the four failure modes has a specific impact on the system:

- *Permanent sending omission.* None of the messages sent by a node will reach any other node.
- *Permanent receiving omission.* A node will not receive messages any longer.
- *Transient sending omission.* A single message from a node will not reach any of its intended receivers.
- *Transient receiving omission.* A single message will not reach one of its intended receivers.

The Boolean variable and arrays are used by the broadcast process to determine the nodes that receive messages and those that fail to receive them. The code in Figure 7.5 begins by clearing transient failures that affected any nodes in the previous transmission slot. Then, unless the maximum number of failures has been reached (a parameter of the model), failures may be injected at non-deterministic points in time. In

Figure 7.5 there are two fallible nodes: N_1 and N_3 which are the nodes with indices i=0 and i=2). As we will see later, the list of fallible nodes is one of the parameters of the model and we have to verify one instance of the model for each combination of fallible nodes.

```
for(i,0,N)       /* clear previous slot's transient failures */
   failureTR[i] = false
rof(i,0,N);
failureTS = false;

do
:: failureCounter < MAX_FAILURES &&     /* inject a failure? */
   failuresThisRound + failuresLastRound < F ->

   if                    /* choose one of the fallible nodes */
   :: i = 0
   :: i = 2
   fi;

   if              /* activate one of the four failure modes */
   :: i == currentSlot -> failureTS = true
   :: failureTR[i] = true
   :: failurePS[i] = true
   :: failurePR[i] = true
   fi;

   failuresThisRound++;    /* increment the failure counters */
   failureCounter++

:: break      /* stop injecting failures (non-deterministic) */
od;
```

Figure 7.5: Failure injection routine (inline), called by the broadcast process.

7.2.4 Modeling Restarts

We abstracted away some of the independent restart process of nodes in order to provide SPIN with a model verifiable within reasonable time and memory constraints. The restart is non-deterministic, *i.e.*, a node

may or may not be restarted. However, nodes are only restarted on the round before they may attempt inclusion (in the inclusion cycle). This limits the amount of possible restarts to a minimum which allows safety properties to be verified, while retaining most of the information concerning liveness.

The model was restricted to restarting nodes that are failed from start-up, *i.e.*, in the initial system state the working nodes already excluded the restarting nodes from the membership. Furthermore, we reduced the possible ways in which a node obtains a wrong membership state. Our criterion was to allow line 15 of Algorithm 6.1 to be executed with the two possible outcomes: either the message contains the correct membership view or not. This way we abstract away the numerous wrong membership views. Our main concern with these restrictions was to ensure that the safety properties, as well as exclusion liveness, maintained their complete meaning.

7.2.5 Specifying the Correctness Properties

One way of checking properties in SPIN is to use assertions. This method is appropriate for specifying invariant properties. We placed assertions at the end of each slot to verify the safety properties, which should hold at all synchronous time-points. The *agreement* property was defined using the code in Figure 7.6 (the other three safety properties are specified in a similar manner, also at the end of each slot).

Regarding the liveness properties, we used SPIN's LTL manager to specify the appropriate LTL formulas. We verified that a faulty node is eventually excluded by fault-free nodes, *i.e.*, exclusion liveness:

$$\Box(\text{node\_failure} \rightarrow \Diamond \text{node\_exclusion}).$$

Moreover, we verified that a restarted node is eventually included in the membership if no failures occur, *i.e.*, inclusion liveness:

$$\Box(\text{node\_restart} \rightarrow \Diamond(\text{node\_inclusion} \mid\mid \text{restart\_failure})).$$

The two liveness properties were verified simultaneously, by providing the LTL manager with their conjunction. SPIN creates a *never claim* which consists of the negation of the LTL formula. The verification process consists of checking that there is no possible execution matching the negated formula.

```
for(i,0,N)                          /* find a non-faulty node Nj */
   if
   :: !faulty[i] -> j = i; break
   :: else -> skip
   fi
rof(i,0,N);

for(i,0,N)                  /* non-faulty nodes agree with Nj */
   if
   :: !faulty[i] ->
      for(p,0,N)
         assert(localView[i].view[p] == localView[j].view[p])
      rof(p,0,N)
   :: else -> skip
   fi
rof(i,0,N);
```

Figure 7.6: Assertion for verifying the *agreement* property.

7.2.6 Parametrization of the Model

Due to the well known problem of state-space explosion, the size of the model was limited in diverse ways. Explicit-state model checkers generate the graph of all system states reachable from the initial state. Consequently, it is computationally expensive to verify very large models. To deal with this problem we introduced a set of parameters that limit the complexity of the verification process by restricting the following values:

- the total number of nodes n,
- the number of sponsors per node k and the associated maximum value $f = k - 2$,
- the total number of failures that may occur during the execution,
- the list nodes that are subject to failures and
- the nodes that are restartable.

Thus, it becomes necessary to verify many different instances of the model, *i.e.*, verify the model for many different combinations of parame-

ters. This does not provide a complete proof of correctness but increases our confidence in that the protocol is free from design defects.

7.3 Verification Results

The correctness of various model instances was checked by executing a large set of verifications. These were done using SPIN version 4.3.0 running on a 3.20 GHz Pentium 4 CPU with 1 GB of RAM. Two advanced reduction algorithms provided by SPIN were extensively used: state-vector compression and minimized automaton encoding. These two techniques have the potential to reduce the memory required for storing the state-space of large models, while the runtime of the verification process can be expected to increase. Combined, these techniques reduced the state-space of the largest model instances to less than 1% of their uncompressed size, making it possible to verify models requiring more than 30 GB using about 200 MB.

Table 7.1 summarizes the results of the verified protocol configurations regarding the safety properties. Table 7.2 provides the equivalent results for liveness properties. Each line in the tables shows the average number of states and verification time of several model instances. The reason for this is that the fallible nodes and the restartable nodes are parametrized as a list for each model instance. Thus, given n nodes where t are affected by faults and r can restart, we verified all combinations of fallible/restartable nodes by generating $\mathrm{C}_t^n \times \mathrm{C}_r^n$ combinations of model parameters.

We created a small tool that generates model instances automatically, verifies them using SPIN and summarizes the results of the verification process. The fifth row of Table 7.1, for example, gathers the results of verifying safety properties on a system with 7 nodes where 2 nodes are fallible; we verified $\mathrm{C}_2^7 = 21$ model instances, where each instance took an average of 2.1 hours to be exhaustively verified and reached on average 1.11×10^8 states.

In total, 181 instances of the model were exhaustively verified during 8 days of continuous computation. The protocol configurations shown in Tables 7.1 and 7.2 were chosen to cover distinct values of parameters n and k for which the model would fit in the available memory. We attempted to verify larger models (*e.g.*, 6 nodes where 3 of them may

n	k	Failures	Fallible Nodes	Restartable Nodes	No. Instances	Avg. States	Avg. Time
4	3	4	Any single node	–	4	4.97×10^5	17 s
5	4	2	Any single node	Any single node	$5 \times 5 = 25$	3.99×10^7	35 min
6	3	2	Any single node	Any single node	$6 \times 6 = 36$	3.54×10^7	41 min
6	5	3	Any two nodes	–	$C_2^6 = 15$	1.08×10^8	2.0 h
7	4	3	Any two nodes	–	$C_2^7 = 21$	1.11×10^8	2.1 h

Table 7.1: Exhaustively verified protocol configurations with respect to safety properties.

n	k	Failures	Fallible Nodes	Restartable Nodes	No. Instances	Avg. States	Avg. Time
4	3	4	Any single node	–	4	3.60×10^5	13 s
5	4	2	Any single node	Any single node	$5 \times 5 = 25$	2.85×10^7	28 min
6	3	2	Any single node	Any single node	$6 \times 6 = 36$	2.48×10^7	30 min
6	5	3	Any two nodes	–	$C_2^6 = 15$	1.46×10^8	3.8 h

Table 7.2: Exhaustively verified protocol configurations with respect to liveness properties.

fail) which eventually consumed all the memory. No errors were found during those partial verifications.

We only verified systems with a single restartable node. This is not expected to limit the validity of the analysis since there are never two inclusions being executed at the same time – there is no concurrency among nodes attempting inclusion in the membership. The protocol is designed for each repaired node to wait for its turn in the inclusion cycle before sending the inclusion request. This was checked by asserting that an inclusion request from a given node N_r in round $3r+2$ leads to an inclusion decision being completed during round $3r+3$ (a sanity check).

7.3.1 Further Considerations

In general, verifying more model instances can only contribute to an increased confidence in that a design is defect-free. Similarly to software testing, one should choose relevant cases that can be verified within reasonable limitations of time and computational resources. Another important aspect is to ensure that the system model itself is a correct representation of reality. Random simulation (available in SPIN) is an effective way to test the model before attempting exhaustive verification.

Furthermore, we believe it is good practice to attempt to verify properties which are not expected to hold. If SPIN finds a counterexample as expected, then one may inspect it to determine whether the sequence of events had been anticipated. If no counterexamples are found, then it most likely means that the model is inaccurate. This methodology therefore contributes to *validating* the formal system model.

The SPIN tool is very efficient in finding counterexamples when they do exist, owing much to the strategy of depth-first searching the state-space. We attempted to verify, for instance, a liveness property which specified that if N_0 fails, then N_1 will eventually be excluded from the membership. This was checked for a system with 6 nodes, with $k = 4$ and fallible nodes N_0 and N_1. Clearly, the property does not hold, as node N_1 may remain non-faulty throughout the entire computation even though it is fallible. In this case, it took SPIN 12.3 seconds to find an appropriate counterexample. Checking the same system with respect to the actual correctness properties takes about 7 minutes.

This efficiency in finding counterexamples for "evident" design faults can be used to our advantage. Several protocol configurations were verified where SPIN returned without completing the verification, having ran out of memory. These executions do, however, contribute to increasing the confidence in that the design is correct. Additionally, SPIN provides alternative search methods, such as bit-state hashing, for fast partial exploration of the state-space. We opted for presenting only the results of exhaustive verification runs, since partial verifications may fail to discover counterexamples even when they do exist.

Ultimately, the most desirable outcome would be to check the correctness of protocol configurations that are actually used in real systems. Since model checking only proves that the correctness properties hold on the verified models, it may be advantageous to verify instances of the model that match the final configuration of a system, *e.g.*, to verify the model for $n = 10$ when a real system is composed of 10 processor nodes. If verifying such configurations exhaustively is computationally infeasible, then at least partially verifying them would still be valuable.

To some extent, this approach is related to just-in-time certification [138], which advocates the usage of automated formal methods at runtime or load time. The premise is that the verification process is more effective by deferring parts of it to the moment when the final configu-

ration is known. Likewise, in our case it would be sensible to focus the verification effort on a particular protocol configuration once the details of a given architecture are settled.

7.4 Summary and Discussion

An important step in developing mechanisms for distributed redundancy management is to ensure that they are free from design faults. Such mechanisms are introduced in systems with the goal of improving dependability but may cause severe system failures if designed incorrectly. With this in mind, we chose to examine the correctness of our membership protocol using the SPIN model checker.

We formalized the correctness properties and the protocol in the PROMELA language, in order to build a model which could be verified by SPIN. The exhaustively verified protocol configurations contribute substantially to our confidence in that the protocol obeys the specified properties. As a model checker, SPIN has the advantage of being able to pinpoint design flaws at early development stages. On the other hand, explicit-state model checkers face the well known problem of state-space explosion. In our case, this effect is partly caused by the highly combinatorial nature of failures. For this reason, a major effort was put into creating an efficient model which was successfully verified for configurations of up to seven processor nodes.

One of the main strategies for reducing the state-space was to prevent instructions of nodes from being interleaved. This way of modeling broadcast channels is applicable to time-triggered communication systems, where processor nodes operate in close synchrony – in each transmission slot there is only one sender and all receivers are able to interpret the message before the next slot. This property of time-triggered communication systems can be used for simplifying the verification process, particularly when using model checking. It is therefore not surprising that this paradigm is used in numerous communication standards intended for safety-critical applications.

One should expect any verification process to have sources of uncertainty. We identified three aspects of our verification effort that create uncertainty and therefore required consideration. First, it is important to validate the formal model and make sure it is an accurate represen-

tation of reality. We tested the model, as it was being built, by using SPIN's random simulation features. Moreover, once the model was complete, we ran the verification for properties that were not expected to hold and checked that the counterexamples had been foreseen.

A second aspect is that the correctness properties must be meaningful and precise in defining the behaviour of the system. This point is related to the previous one, since writing a good specification is just as important as building an accurate model. One limitation of our liveness properties is that they leave the timing facet unbounded. Even though we observed that the system reacts to failures in a timely manner – both in the experimental setup described in the previous chapter and through random simulation using SPIN –, the liveness properties are not as strict as the safety properties. Liveness could be effectively bounded either by extending the formal model or through extensive testing.

Lastly, model checking proves that the protocol is correct for the configurations that were exhaustively verified. This also increases the confidence in that the protocol is correct for the general case but there is some uncertainty regarding unverified configurations. Thus, in addition to the protocol configurations which were successfully verified in this chapter, a safe approach would be to verify model instances matching configurations that are actually used.

CHAPTER 8

Interoperability between Layers

The issue of interoperability between layers was addressed in part by the previous chapters. One form of interoperability is the notification of self-exclusion provided by the system layer to the node layer. When a node is excluded by the remaining nodes, the node also excludes itself from its view of the membership (if it is still able to execute the protocol). By passing this information to the node layer, a faulty node is able to initiate local recovery and return to providing service after a downtime period.

Another form of interoperability is fail-reporting, which consists in sending a message that receiving nodes perceive as a signal that the node has failed. Using fail-reporting, a node can notify the system layer that an error was detected locally by node-layer mechanisms but transparent recovery was not possible. In Chapter 6 we described that the protocol gives the option to report failures, even though it is assumed that fail-silent semantics is the normal case that the protocol must handle. In this chapter we analyze whether using fail-report semantics is beneficial to the protocol's reliability.

To implement distributed redundancy management, nodes maintain a consensus on which nodes are operational a which ones are faulty. This service is provided by the processor-group membership protocol.

In distributed systems where the processing nodes offer effective fault-containment between different application processes executed on a node, it is desirable for the protocol to provide information also on which tasks are operational. This chapter extends the protocol to allow each node to send multiple messages in each communication round. Moreover, the protocol is extended to keep track of application-process failures.

8.1 Advantages of Fail-Report Semantics

In the system model described in Chapter 6 nodes are assumed to fail silently, *i.e.*, by sending no more messages when an error is detected. This class of failure is modeled by a permanent sending omission. If a node fails silently, the remaining nodes cannot discover whether it was a sending omission or a receiving omission without executing the protocol. This would not be the case of all internal failures were signaled by sending a failure report. In this section we investigate whether fail-report semantics has a positive impact on the reliability of the protocol.

To this end, we analyze a scenario where the group membership protocol executes only in presence of node failures which are detected by node-layer mechanisms. Using fail-silent semantics, faulty nodes send no more messages; using fail-report semantics, faulty nodes send a message which all other nodes interpret as a signaled failure rather than a communication failure.

Consider that the membership protocol is configured with $k = 3$ and is therefore capable of tolerating a single failure in any two consecutive rounds. If two nodes fail silently at the same time, and the second node broadcasts immediately after the first, then there will be two expected messages missing from the network. The remaining nodes fail to receive two consecutive messages and diagnose themselves as faulty. This is done in line 6 of Algorithm 6.2 to ensure that nodes exclude themselves from the membership when they are unable to receive any more messages. Thus, due to the failure of two nodes, all remaining nodes self-exclude from the membership and initiate local recovery.

Since the protocol can tolerate up to $f = k - 2$ failures in any two consecutive rounds of communication, any node that fails to receive $k-1$ consecutive messages diagnoses itself as faulty. TTP's group membership protocol has the same property. It is designed to tolerate a single failure

and any node failing to receive two consecutive messages diagnoses itself as faulty.

The alternative is for a node to report failures instead of failing silently. If all nodes send a failure report upon error detection, then it is possible to prevent self-exclusion of working nodes. We modified the failure injector model of the previous chapter, shown in Figure 7.5, to inject only node failures which are detected (the other possible failure modes are not considered). The PROMELA code in Figure 8.1 injects a simultaneous failure (by self-exclusion) of up to $k-1$ nodes. This models a situation where up to $k-1$ nodes detect an internal error at the same time.

```
for(j,1,K)                              /* any K-1 nodes may fail */

   if                                            /* choose one node */
   :: i=0 :: i=1 :: i=2 :: i=3 :: i=4 :: i=5 :: i=6
   fi;
   localView[i].view[i] = false  /* the node excludes itself */

rof(j,1,K);
```

Figure 8.1: Failure injection model, modified to inject only node errors that are detected (for a system with seven nodes).

By excluding a node from its own view of the membership, that node sends a failure report according to line 37 of Algorithm 6.1. Thus, the code in Figure 8.1 causes up to $k-1$ nodes to send a failure report instead of failing silently (the situation which would lead all nodes to diagnosing themselves as faulty).

The modified PROMELA model was verified using SPIN for systems with 6 and 7 nodes configured with $k = 3$ and $k = 4$ (four combinations in total). SPIN verified that the correctness properties (specified in the previous chapter) hold in these configurations. This shows that the protocol is capable of handling certain failure modes by reporting failures which are not handled if nodes fail silently.

We can therefore draw the conclusion that using fail-report instead of fail-silent semantics has a positive effect on the protocol's reliability. However, the effect is difficult to quantify. Furthermore, this result is

difficult to generalize to other systems, even though it is reasonable to presume that fail-report semantics may improve the reliability of other protocols for synchronous systems.

8.2 Multiple Transmission Slots

We assumed earlier in the thesis that nodes only transmit one message per communication round. However, when a node contains multiple tasks, the network schedule should accommodate multiple messages from that node in each round (possibly one message per task). A trivial solution would be to concatenate the messages of all tasks into a single physical message, but this solution is not general. Without changing the protocol, we describe how it can maintain consensus on the working nodes while allowing them to broadcast more than once in each round.

To build a service implementing our protocol, the lower network layers notify the reception/loss of *physical messages* to the membership service. The service reacts to those events and updates the membership set accordingly. To support multiple transmissions by each node, we propose that the events notified to the membership service refer to *logical messages*, instead of physical ones. We define a node's logical message as the concatenation of all physical messages sent by that node during one round. When a physical message is lost by a given node, that node will consider the corresponding logical message to be lost.

Rather than imposing physical concatenation of messages (as in the trivial solution), this method concatenates messages logically. By doing so, it allows nodes to have multiple transmission slots in each round. The reception/loss of a node's logical message should be notified to the membership service only after the last transmission slot of that node. Accordingly, only the last physical message transmitted by each node should carry membership information (acknowledgements and *i-flag* referring to logical messages).

This scheme does not change the membership protocol. It only changes the way in which membership events are reported to the membership service. In a system with n nodes there will be n logical messages transmitted every round, regardless of the number of physical messages. Thus, the overhead of the membership protocol depends only on the total number of nodes.

8.3 Application-Process Membership

So far, we have described a processor-group membership protocol that guarantees consensus on the status of all processor nodes. In the previous section the protocol was extended to allow each node to broadcast multiple messages in each communication round. In this section we describe, without giving a full specification, how the protocol can be extended to keep track of both node failures and application-process (or task) failures.

To achieve this for our protocol, we can add one extra bit to each physical message sent by a task. This fail-report bit indicates that a message is carrying a failure report for the task, rather than a regular message. When multiple tasks, running on the same node, share the same transmission slot to send their messages, one bit is added for each application (with the same indication).

A task is removed from the task-group by other nodes when they receive a failure report for that task. Task failures are this way *reported* to all nodes in the system. When nodes receive a regular message, instead of the failure report, from a task which had previously failed, the task is included in the task-group again. When a complete node fails (and is removed from the processor-group membership) all its tasks are removed from the task-group by the other nodes.

This type of membership agreement is weaker than the processor-group membership agreement described in the previous chapters. When all nodes receive the failure report, there will be agreement on inclusion/exclusion of the corresponding task. However, if any other failures occur simultaneously, the nodes will disagree on the task membership until a fault-free period of the execution allows the nodes to refresh the status of all tasks. Such cases can only occur when a task fails and there is a near-coincident network failure affecting the transmission of the report. It should be noted that the node must send a failure report for a failed task at all transmission slots dedicated to that task.

With respect to the processor-group membership protocol, there is no difference between messages that carry failure reports and those that carry regular membership information. That is, the acknowledgement bits and the *i-flag* will have the same function for both message types. Similarly, a message containing a failure report will be handled by the node membership service exactly in the same way as a regular message.

8.4 Summary and Discussion

This chapter unified the building blocks that were presented in the thesis. In the previous chapters we began by investigating techniques for ensuring containment of faults within application processes. When an application error is detected, the operating system attempts recovery using the lightweight checkpointing technique. If the recovery fails or the task fails to send a message, the operating system sends a failure report on behalf of the faulty application.

However, an application error may remain undetected and cause the entire node to fail. The same may happen if the operating system itself is faulty. Moreover, network failures may prevent a node from sending or receiving messages. These cases are handled by the group-membership protocol by excluding the complete node from service delivery. In this case, all the tasks executing on the excluded node are also considered to be faulty.

In some cases the operating system is able to detect errors locally that prevent the entire node from delivering service. In such cases we show that it is beneficial to signal failures by sending a failure report. It was shown in [139] that signaling failures improves the performance of protocols for asynchronous systems. For our protocol, which is appropriate for synchronous systems, we show that fail-report semantics improves the protocol's reliability. This piece of evidence suggests that it might also be the case in other protocols for synchronous systems, even though the result is difficult to generalize.

CHAPTER 9

Conclusions

This thesis deals with principles and techniques for achieving fault tolerance in distributed embedded systems. More specifically, it addresses the problem of how to implement fault tolerance in a cost-effective way in systems where the processor nodes execute many applications and system services, so called integrated systems. As a starting point, we propose a design philosophy called layered fault tolerance, which identifies three layers – the system layer, the node layer and the hardware layer – where mechanisms for fault tolerance can be implemented. We argue that one should make a careful trade-off between the cost and the complexity of mechanisms that are employed at each layer to minimize the overall cost of a system. While all three layers are important for achieving high dependability, the contributions of the thesis focus on the node layer and the system layer.

A key issue in the design of integrated embedded systems is how to achieve temporal and spatial partitioning of programs. This issue is addressed in the context of SECERN – an approach for implementing support for partitioning and fault tolerance in real-time kernels. SECERN includes several mechanisms that aim to confine errors to the applications where they originate. Several of these mechanisms were implemented as extensions to the μC/OS-II real-time kernel. These mecha-

nisms were memory protection, processor exceptions, system call protection and application-specific checks. The extended kernel was developed for Freescale's MPC5554 microcontroller.

Memory protection was achieved by using the microcontroller's memory management unit. One disadvantage of using MMUs in real-time systems is that they can increase the jitter (variability) in the execution time of real-time tasks. The jitter problem arises when an application accesses a page which is not listed in the cache holding page entries – the processor's TLB. In our design, TLB-misses are avoided by updating the TLB during context switches. The approach is to insert in the TLB the pages that belong to a process before switching context to that process. This adds some overhead to the context switches, which was measured and found to be acceptable for many applications. Since this method prevents TLB misses to occur during the execution of tasks, it avoids execution time jitter and thereby simplifies the response time analysis for hard real-time tasks.

Unfortunately, it was not possible to conduct an extensive experimental assessment of the mechanisms included in the extended real-time kernel within the time frame of this thesis project. Nevertheless, we developed a fault injection tool and conducted a series of preliminary tests of these mechanisms. The tests were conducted according to a methodology of focused fault injection, whose main objective is fault removal, *i.e.*, identification and removal of design faults. It consists of setting up finely controlled experiments in accordance with the system properties that are to be verified. Since our goal was to verify the partitioning mechanisms, we configured the system with two processes, injected faults into the context of one of them and observed the outcome. Given that the extended kernel is supposed to handle this type of fault, any experiment where the fault-free task or the operating system are affected would indicate the existence of a design flaw in the partitioning mechanisms.

The experiments exposed two vulnerabilities in the extended kernel: one related to configuration management, where some memory pages were marked as writable for all processes while they should be read-only; and one related to an inherited design decision regarding context switches which is not appropriate for partitioned systems. Even though these experiments did not provide an exhaustive assessment of the extended real-time kernel, they demonstrated the importance and potential

benefits of using fault injection for fault removal in partitioned systems.

In addition to the mechanisms included in the extended real-time kernel, SECERN includes an approach to checkpointing and rollback recovery of real-time tasks named lightweight checkpointing. The goal of this approach is to provide detection, isolation and recovery of errant application processes. The checkpoints are primarily intended for recovery from errors caused by transient hardware faults. However, a key feature is that the recovery strategy can distinguish between transient hardware faults and software faults. It relies on the checkpointing mechanism to diagnose the actual cause of an error. If an error reappears after a rollback, it assumes that the cause is a software fault. If this happens, the operating system transfers control to an application-specific exception handler, which the application designer can use to implement a recovery strategy for software faults. This strategy could be based on design diversity, data diversity, or simply do a restart of the task.

The lightweight checkpointing scheme allows applications to save snapshots of their state to main memory while providing them with a service for locking the checkpoint area using memory protection. The content of the snapshots is determined by the application designer. The locking makes it possible to deal with failure modes where an application attempts to overwrite any previous checkpoints. To deal with error detection latency, the scheme uses three checkpoints, transparently to applications, and enforces a minimum time between calls to the locking mechanism. We show that this method ensures the integrity of application-level checkpoints while introducing only a small and fixed overhead to each checkpoint for locking the memory.

In addition to applying fault injection as a means of verification, the thesis addressed the problem of making fault injection campaigns more efficient. A problem commonly observed during fault injection campaigns is that most faults are not activated when chosen randomly. Thus, since each experiment is a time-consuming procedure, it is important to reduce the number of experiments that have no impact on the system.

To this end, a pre-injection analysis was proposed and experimentally evaluated. We compared the results of selecting faults randomly with injecting faults in registers and memory locations only when they are read. This increased the effectiveness of the fault injections by one order of magnitude. The pre-injection analysis is suitable for emulating the

effects of faults that hit registers and memory locations directly, since it selects the points in time where resources are read, rather than written. Nevertheless, we observed that the error detection coverage estimation was, in our experiments, similar when selecting faults randomly or using the pre-injection analysis.

Even though it is possible to develop node-layer mechanisms for making each node highly dependable, nodes may still fail. To deal with node failures and errors occurring in the communication network, a system must be equipped with appropriate system-layer mechanisms. To this end, the thesis addressed the problem of redundancy management in distributed embedded systems.

We propose a group membership protocol for guaranteeing consistent views of failures and restarts among all working nodes. The protocol is especially designed for systems using time-triggered communication. We assume that errors lead to send/receive omissions that can be either transient or permanent. The protocol tolerates a configurable number of simultaneous or near-coincident failures. This provides the system designer with the ability to adjust the reliability of the protocol to the available resources.

Moreover, the protocol supports inclusion of restarted nodes under the same failure assumptions as exclusion. To achieve this, we address the problem of ensuring that a restarted node recovers the correct membership state – which may change at any point in time – before joining the group. The concern is that a node must remain excluded from the group if any failures prevent that node from agreeing on the membership state. We found that node inclusion is safe by allowing only a single node to be reintegrated in any given round. This is achieved by establishing a cyclic order that nodes follow to send inclusion requests. The part of the protocol that provides agreement on inclusion can be combined with other solutions existing in the literature that have no such functionality.

We formalized the correctness properties and the protocol in the PROMELA language, in order to build a model which could be verified by SPIN. The exhaustively verified protocol configurations contribute substantially to our confidence in that the protocol obeys the specified properties. As a model checker, SPIN has the advantage of being able to pinpoint design flaws at early development stages. On the other hand, explicit-state model checkers face the well known problem of state-space

explosion. In our case, this effect is partly caused by the highly combinatorial nature of failures. For this reason, a major effort was put into creating an efficient model which was successfully verified for configurations of up to seven processor nodes.

One of the main strategies for reducing the state-space was to prevent instructions of nodes from being interleaved. This way of modeling broadcast channels is applicable to time-triggered communication systems, where processor nodes operate in close synchrony – in each transmission slot there is only one sender and all receivers are able to interpret the message before the next slot. This property of time-triggered communication systems can be used for simplifying the verification process, particularly when using model checking. It is therefore not surprising that this paradigm is used in numerous communication standards intended for safety-critical applications.

The thesis also considers the problem of interoperability between node and system layers. An important feature of the group membership protocol is that it can be integrated with node-layer fault tolerance mechanisms. First, the protocol is capable of providing accurate self-exclusion information to node-layer recovery mechanisms. This feature allows nodes to rapidly trigger local recovery procedures when they are excluded from service delivery by the remaining nodes. Second, it allows node-layer error detection mechanisms, executed locally at each node, to notify the group membership service that an error prevents a node from producing correct results. In this case, the usual approach is to ensure fail-silence, *i.e.*, the node sends no more messages. In contrast, our protocol can send a failure report upon error detection. The practical outcome of using fail-report semantics is that node failures are not interpreted by other nodes as communication failures.

This leads us to the final contribution of the thesis. We show that using fail-reporting instead of fail-silent semantics has a positive impact on the protocol's reliability. We modified the formal model of the protocol to show, using SPIN, that the protocol is capable of handling certain failure modes by reporting failures which are not handled if nodes fail silently. This shows that the reliability of the protocol improves by using fail-report semantics. This piece of evidence suggests that it might also be the case in other protocols for synchronous systems.

To summarize, the thesis proposes several ideas on how to imple-

ment fault tolerance in distributed embedded systems where the processor nodes are shared by many system functions. These ideas have been assessed and validated by implementation studies, fault injection, probabilistic modeling and model checking. Nevertheless, there are uncertainties associated with these efforts. This thesis was written in the hope that, despite these uncertainties, designers of distributed embedded systems will find the proposed ideas useful.

References

[1] Neil Storey, *Safety-Critical Computer Systems.* Prentice Hall, 1996.

[2] Algirdas Avižienis, Jean-Claude Laprie, Brian Randell, and Carl E. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11–33, Jan.-Mar. 2004.

[3] Leslie Lamport, Robert Shostak, and Marshall Pease, "The Byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4, no. 3, pp. 382–401, Jul. 1982.

[4] Robin A. Sahner, Kishor S. Trivedi, and Antonio Puliafito, *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package.* Kluwer Academic Publishers, 1995.

[5] Andrew S. Tanenbaum, *Modern Operating Systems*, 2nd ed. Pearson Education, 2001.

[6] Jiri Gaisler, "A portable and fault-tolerant microprocessor based on the SPARC V8 architecture," in *Proceedings of the 2002 International Conference on Dependable Systems and Networks (DSN 2002)*, Jun. 2002, pp. 409–415.

[7] Joakim Aidemark, "Node-level fault tolerance for embedded real-time systems," Ph.D. dissertation, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 2004.

[8] Daniel P. Siewiorek, "Architecture of fault-tolerant computers: an historical perspective," *Proceedings of the IEEE*, vol. 79, no. 12, pp. 1710–1734, Dec. 1991.

[9] Babak Rostamzadeh, Henrik Lönn, Rolf Snedsbøl, and Jan Torin, "DACAPO: A distributed computer architecture for safety-critical control applications," *Proceedings of the Intelligent Vehicles '95 Symposium*, pp. 376–381, Sep. 1995.

[10] Joakim Aidemark, Peter Folkesson, and Johan Karlsson, "A framework for node-level fault tolerance in distributed real-time systems," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN 2005)*, Jun./Jul. 2005, pp. 656–665.

[11] Aeronautical Radio, Inc., "ARINC specification 651-1: Design guidance for integrated modular avionics," Nov. 1997.

[12] Aeronautical Radio, Inc., "ARINC specification 653-1: Avionics application software standard interface," Oct. 2003.

[13] Harald Heinecke, Klaus-Peter Schnelle, Helmut Fennel, Jürgen Bortolazzi, Lennart Lundh, Jean Leflour, Jean-Luc Maté, Kenji Nishikawa, and Thomas Scharnhorst, "AUTomotive Open System ARchitecture - an industry-wide initiative to manage the complexity of emerging automotive E/E architectures," in *Proceedings of the 2004 International Congress on Transportation Electronics (Convergence 2004)*, Oct. 2004, pp. 325–332.

[14] Shekhar Borkar, "Designing reliable systems from unreliable components: The challenges of transistor variability and degradation," *IEEE Micro*, vol. 25, no. 6, pp. 10–16, Nov./Dec. 2005.

[15] Edward W. Czeck and Daniel P. Siewiorek, "Observations on the effects of fault manifestation as a function of workload," *IEEE Transactions on Computers*, vol. 41, no. 5, pp. 559–566, May 1992.

[16] Swapna S. Gokhale, Peter N. Marinos, and Kishor S. Trivedi, "Important milestones in software reliability modeling," in *Proceedings of the 8th International Conference on Software Engineering and Knowledge Engineering (SEKE'96)*, Jun. 1996, pp. 345–352.

[17] Xuemei Zhang and Hoang Pham, "Software field failure rate prediction before software deployment," *The Journal of Systems and Software*, vol. 79, no. 3, pp. 291–300, Mar. 2006.

[18] RTCA, Inc., "DO-178B: Software considerations in airborne systems and equipment certification," Dec. 1991.

[19] Antonia Bertolino and Lorenzo Strigini, "Assessing the risk due to software faults: Estimates of failure rate versus evidence of perfection," *Software Testing, Verification and Reliability*, vol. 8, no. 3, pp. 155–166, Sep. 1998.

[20] International Electrotechnical Commission (IEC), "IEC 61508: Functional safety of electrical/electronic/programmable electronic safety-related systems (parts 1 to 7)," 1998 and 2000.

[21] John Rushby, "Partitioning in avionics architectures: Requirements, mechanisms, and assurance," NASA Langley Research Center, Tech. Rep. NASA/CR-1999-209347, Jun. 1999.

[22] Ben L. Di Vito, "A model of cooperative noninterference for integrated modular avionics," in *Proceedings of the 7th International IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-7)*, Jan. 1999, pp. 269–286.

[23] Joseph A. Goguen and José Meseguer, "Security policies and security models," in *Proceedings of the 1982 IEEE Symposium on Security and Privacy*, Apr. 1982, pp. 11–20.

[24] A. W. Roscoe, J. C. P. Woodcock, and L. Wulf, "Non-interference through determinism," in *Proceedings of the Third European Symposium on Research in Computer Security (ESORICS 94)*, ser. Lecture Notes in Computer Science, Nov. 1994, vol. 875, no. 1, pp. 31–53.

[25] Matthew M. Wilding, David S. Hardin, and David A. Greve, "Invariant performance: A statement of task isolation useful for embedded application integration," in *Proceedings of the 7th International IFIP Working Conference on Dependable Computing for Critical Applications (DCCA-7)*, Jan. 1999, pp. 287–300.

[26] C. L. Liu and James W. Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of the ACM*, vol. 1, no. 20, pp. 44–61, Jan. 1973.

[27] Hermann Kopetz, *Real-Time Systems: Design Principles for Distributed Embedded Applications.* Kluwer Academic Publishers, 1997.

[28] Steve Zdancewic and Andrew C. Myers, "Robust declassification," in *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW-14)*, Jun. 2001, pp. 15–26.

[29] Steve Zdancewic, "Challenges for information-flow security," in *Proceedings of the First International Workshop on Programming Language Interference and Dependence (PLID'04)*, Aug. 2004.

[30] Andrei Sabelfeld and David Sands, "Dimensions and principles of declassification," in *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW-18)*, Jun. 2005, pp. 255–269.

[31] William Stallings, *Operating Systems: Internals and Design Principles*, 4th ed. Prentice Hall, 2001.

[32] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, 3rd ed. Morgan Kaufman, 2004.

[33] Freescale Semiconductor, Inc., *MPC5553/MPC5554 Microcontroller Reference Manual (Rev 4.0)*, Apr. 2007.

[34] Freescale Semiconductor, Inc., *MPC565 Reference Manual (Rev 2.2)*, Nov. 2005.

[35] ARM Ltd., *ARM946E-S Technical Reference Manual (Rev. r1p1)*, Apr. 2007.

[36] Matthew Simpson, Bhuvan Middha, and Rajeev Barua, "Segment protection for embedded systems using run-time checks," in *Proceedings of the 2005 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2005)*, Sep. 2005, pp. 66–77.

[37] Trevor Jim, Greg Morrisett, Dan Grossman, Michael Hicks, James Cheney, and Yanling Wang, "Cyclone: A safe dialect of C," in *Proceedings of the 2002 USENIX Annual Technical Conference, General Track*, Jun. 2002, pp. 275–288.

[38] Dan Grossman, Greg Morrisett, Trevor Jim, Michael Hicks, Yanling Wang, and James Cheney, "Region-based memory management in Cyclone," in *Proceedings of the ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI)*, Jun. 2002, pp. 282–293.

[39] Sumant Kowshik, Dinakar Dhurjati, and Vikram S. Adve, "Ensuring code safety without runtime checks for real-time control systems," in *Proceedings of the 2002 International Conference on Compilers, Architecture, and Synthesis for Embedded Systems (CASES 2002)*, Oct. 2002, pp. 288–297.

[40] Todd M. Austin, Scott E. Breach, and Gurindar S. Sohi, "Efficient detection of all pointer and array access errors," in *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation (PLDI)*, Jun. 1994, pp. 290–301.

[41] Jeremy Condit, Matthew Harren, Scott McPeak, George C. Necula, and Westley Weimer, "CCured in the real world," in *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation (PLDI)*, May 2003, pp. 232–244.

[42] Krithi Ramamritham and John A. Stankovic, "Scheduling algorithms and operating systems support for real-time systems," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 55–67, Jan. 1994.

[43] Min-Ih Chen and Kwei-Jay Lin, "Dynamic priority ceilings: A concurrency control protocol for real-time systems," *Real-Time Systems*, vol. 2, no. 4, pp. 325–346, Nov. 1990.

[44] Mihir Pandya and Miroslaw Malek, "Minimum achievable utilization for fault-tolerant processing of periodic tasks," *IEEE Transactions on Computers*, vol. 47, no. 10, pp. 1102–1112, Oct. 1998.

[45] Yann-Hang Lee, Daeyoung Kim, Mohamed F. Younis, Jeffrey X. Zhou, and James McElroy, "Resource scheduling in dependable integrated modular avionics," in *Proceedings of the 2000 International Conference on Dependable Systems and Networks (DSN 2000)*, Jun. 2000, pp. 14–23.

[46] Joakim Aidemark, Jonny Vinter, Peter Folkesson, and Johan Karlsson, "GOOFI: Generic object-oriented fault injection tool," in *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN 2001)*, Jul. 2001, pp. 83–88.

[47] Jonny Vinter, Joakim Aidemark, Daniel Skarin, Raul Barbosa, Peter Folkesson, and Johan Karlsson, "An overview of GOOFI – a generic object-oriented fault injection framework," Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, Tech. Rep. 05-07, 2005.

[48] Jean J. Labrosse, *MicroC/OS-II: The Real-Time Kernel*, 2nd ed. CMP Books, 2002.

[49] Philip Koopman, Kobey DeVale, and John DeVale, *Dependability Benchmarking for Computer Systems.* Wiley, 2008, ch. Interface Robustness Testing: Experience and Lessons Learned from the Ballista Project, pp. 201–226.

[50] David S. Peterson, Matt Bishop, and Raju Pandey, "A flexible containment mechanism for executing untrusted code," in *Proceedings of the 11th USENIX Security Symposium*, Aug. 2002, pp. 207–225.

[51] Niels Provos, "Improving host security with system call policies," in *Proceedings of the 12th USENIX Security Symposium*, Aug. 2003, pp. 257–272.

[52] iSYSTEM AG, *EVB-5554 Evaluation and Development Kit for Freescale PowerPC MPC5554 Microcontroller (User's Manual)*, Jul. 2007.

[53] Andrea Bondavalli, Andrea Ceccarelli, Lorenzo Falai, and Michele Vadursi, "Foundations of measurement theory applied to the evaluation of dependability attributes," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007)*, Jun. 2007, pp. 522–533.

[54] Karama Kanoun and Lisa Spainhower, Eds., *Dependability Benchmarking for Computer Systems.* Wiley, 2008.

[55] Sha Tao, Paul D. Ezhilchelvan, and Santosh K. Shrivastava, "Focused fault injection testing of software implemented fault tolerance mechanisms of Voltan TMR nodes," *Distributed Systems Engineering*, vol. 2, no. 1, pp. 39–49, Mar. 1995.

[56] Intel Corporation, *Intel® Core™2 Extreme Processor X6800 and Intel® Core™2 Duo Desktop Processor E6000 and E4000 Sequence: Specification Update*, Document No. 313279-026, May 2008.

[57] João Durães and Henrique Madeira, "Definition of software fault emulation operators: A field data study," in *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, Jun. 2003, pp. 105–114.

[58] João A. Durães and Henrique S. Madeira, "Emulation of software faults: A field data study and a practical approach," *IEEE Transactions on Software Engineering*, vol. 32, no. 11, pp. 849–867, Nov. 2006.

[59] Algirdas Avižienis and John P. J. Kelly, "Fault tolerance by design diversity: Concepts and experiments," *IEEE Computer*, vol. 17, no. 8, pp. 67–80, Aug. 1984.

[60] James J. Horning, Hugh C. Lauer, P. M. Melliar-Smith, and Brian Randell, "A program structure for error detection and recovery," in *Proceedings of the International Symposium on Operating Systems*, ser. Lecture Notes in Computer Science, Apr. 1974, vol. 16, pp. 171–187.

[61] Susan S. Brilliant, John C. Knight, and Nancy G. Leveson, "Analysis of faults in an N-version software experiment," *IEEE Transactions on Software Engineering*, vol. 16, no. 2, pp. 238–247, Feb. 1990.

[62] Paul E. Ammann and John C. Knight, "Data diversity: an approach to software fault tolerance," *IEEE Transactions on Computers*, vol. 37, no. 4, pp. 418–425, Apr. 1988.

[63] Yennun Huang, Chandra Kintala, Nick Kolettis, and N. Dudley Fulton, "Software rejuvenation: analysis, module and applications," in *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*, Jun. 1995, pp. 381–390.

[64] Israel Koren and C. Mani Krishna, *Fault-Tolerant Systems*. Morgan Kaufmann, 2007.

[65] Daniel Skarin and Johan Karlsson, "Software implemented detection and recovery of soft errors in a brake-by-wire system," in *Proceedings of the 7th European Dependable Computing Conference (EDCC-7)*, May 2008, pp. 145–154.

[66] Jonny Vinter, Andreas Johansson, Peter Folkesson, and Johan Karlsson, "On the design of robust integrators for fail-bounded control systems," in *Proceedings of the 2003 International Conference on Dependable Systems and Networks (DSN 2003)*, Jun. 2003, pp. 415–424.

[67] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum, "Failure resilience for device drivers," in *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN 2007)*, Jun. 2007, pp. 41–50.

[68] Robert Baumann, "Soft errors in advanced computer systems," *IEEE Design and Test of Computers*, vol. 22, no. 3, pp. 258–266, May/Jun. 2005.

[69] Subhachandra Chandra and Peter M. Chen, "How fail-stop are faulty programs?" in *Proceedings of the 28th Annual International*

Symposium on Fault-Tolerant Computing (FTCS-28), Jun. 1998, pp. 240–249.

[70] Kevin Reick, Pia N. Sanda, Scott Swaney, Jeffrey W. Kellington, Michael Mack, Michael Floyd, and Daniel Henderson, "Fault-tolerant design of the IBM Power6 microprocessor," *IEEE Micro*, vol. 28, no. 2, pp. 30–38, Mar./Apr. 2008.

[71] Patrick J. Meaney, Scott B. Swaney, Pia N. Sanda, and Lisa Spainhower, "IBM z990 soft error detection and recovery," *IEEE Transactions on Device and Materials Reliability*, vol. 5, no. 3, pp. 419–427, Sep. 2005.

[72] Chung-Chi Jim Li and W. Kent Fuchs, "CATCH – compiler assisted techniques for checkpointing," in *Proceedings of the 20th International Symposium on Fault-Tolerant Computing (FTCS-20)*, Jun. 1990, pp. 74–81.

[73] Gerard J. Holzmann, "The model checker SPIN," *IEEE Transactions on Software Engineering*, vol. 23, no. 5, pp. 279–295, May 1997.

[74] Gerard J. Holzmann, *The SPIN Model Checker: Primer and Reference Manual*. Addison-Wesley, 2003.

[75] Andrew S. Tanenbaum, Jorrit N. Herder, and Herbert Bos, "Can we make operating systems reliable and secure?" *IEEE Computer*, vol. 39, no. 5, pp. 44–51, May 2006.

[76] Michael M. Swift, Brian N. Bershad, and Henry M. Levy, "Improving the reliability of commodity operating systems," *ACM Transactions on Computer Systems*, vol. 23, no. 1, pp. 77–110, Feb. 2005.

[77] Sasikumar Punnekkat, Alan Burns, and Robert Davis, "Analysis of checkpointing for real-time systems," *Real-Time Systems*, vol. 20, no. 1, pp. 83–102, Jan. 2001.

[78] Ying Zhang and Krishnendu Chakrabarty, "Fault recovery based on checkpointing for hard real-time embedded systems," in *Proceedings of the 18th IEEE International Symposium on Defect and Fault Tolerance in VLSI Systems*, Nov. 2003, pp. 320–327.

[79] C. M. Krishna, Yann-Hang Lee, and Kang G. Shin, "Optimization criteria for checkpoint placement," *Communications of the ACM*, vol. 27, no. 10, pp. 1008–1012, Oct. 1984.

[80] James S. Plank, Micah Beck, Gerry Kingsley, and Kai Li, "Libckpt: Transparent checkpointing under Unix," in *Usenix Winter Technical Conference*, Jan. 1995, pp. 213–223.

[81] Yennun Huang, Chandra Kintala, and Yi-Min Wang, "Software tools and libraries for fault tolerance," *IEEE Technical Committee on Operating Systems and Application Environments*, vol. 7, no. 4, pp. 5–9, 1995.

[82] E. N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson, "A survey of rollback-recovery protocols in message-passing systems," *ACM Computing Surveys*, vol. 34, no. 3, pp. 375–408, Sep. 2002.

[83] Ozalp Babaoglu and Keith Marzullo, "Consistent global states of distributed systems: Fundamental concepts and mechanisms," in *Distributed Systems*, 2nd ed., S. Mullender, Ed. Addison-Wesley, 1993, pp. 55–96.

[84] Guohong Cao and Mukesh Singhal, "On coordinated checkpointing in distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 9, no. 12, pp. 1213–1225, Dec. 1998.

[85] Luís M. Silva and João G. Silva, "Global checkpointing for distributed programs," in *Proceedings of the 11th Symposium on Reliable Distributed Systems*, Oct. 1992, pp. 155–162.

[86] P. Krishna, Nitin H. Vaidya, and Dhiraj K. Pradhan, "Recovery in multicomputers with finite error detection latency," in *Proceedings of the 23rd International Conference on Parallel Processing (ICPP'94)*, Aug. 1994, pp. II:206–210.

[87] Henrique Madeira and João G. Silva, "Experimental evaluation of the fail-silent behaviour in computers without error masking," in *Proceedings of the 24th International Symposium on Fault-Tolerant Computing (FTCS-24)*, Jun. 1994, pp. 350–359.

[88] Pedro Yuste, Juan-Carlos Ruiz-Garcia, Lenin Lemus, and Pedro J. Gil, "Non-intrusive software-implemented fault injection in embedded systems," in *Proceedings of the First Latin-American Symposium on Dependable Computing (LADC 2003)*, ser. Lecture Notes in Computer Science, Oct. 2003, vol. 2847, pp. 23–38.

[89] Joakim Aidemark, Peter Folkesson, and Johan Karlsson, "Path-based error coverage prediction," in *Proceedings of the 7th International On-Line Testing Workshop (IOLTW 2001)*, Jul. 2001, pp. 14–20.

[90] IEEE Industry Standards and Technology Organization (IEEE-ISTO), "The nexus 5001 forum™ standard for a global embedded processor debug interface (version 2.0)," Dec. 2003.

[91] Daniel Skarin, Jonny Vinter, Peter Folkesson, and Johan Karlsson, "Implementation and usage of the GOOFI MPC565 nexus fault injection plug-in," Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, Tech. Rep. 04-08, 2004.

[92] Xavier Castillo and Daniel P. Siewiorek, "Workload, performance and reliability of digital computing systems," in *Proceedings of the 11th International Symposium on Fault-Tolerant Computing (FTCS-11)*, Jun. 1981, pp. 84–89.

[93] Ram Chillarege and Ravishankar K. Iyer, "The effect of system workload on error latency: An experimental study," in *Proceedings of the 1985 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems (SIGMETRICS '85)*, Aug. 1985, pp. 69–77.

[94] Ram Chillarege and Nicholas S. Bowen, "Understanding large system failures - a fault injection experiment," in *Proceedings of the 19th International Symposium on Fault-Tolerant Computing (FTCS-19)*, Jun. 1989, pp. 356–365.

[95] Jens Güthoff and Volkmar Sieh, "Combining software-implemented and simulation-based fault injection into a single fault injection

method," in *Proceedings of the 25th International Symposium on Fault-Tolerant Computing (FTCS-25)*, Jun. 1995, pp. 196–206.

[96] Alfredo Benso, Maurizio Rebaudengo, Leonardo Impagliazzo, and Pietro Marmo, "Fault-list collapsing for fault injection experiments," in *Proceedings of the 1998 Annual Reliability and Maintainability Symposium (RAMS 1998)*, Jan. 1998, pp. 383–388.

[97] Timothy K. Tsai, Mei-Chen Hsueh, Hong Zhao, Zbigniew Kalbarczyk, and Ravishankar K. Iyer, "Stress-based and path-based fault injection," *IEEE Transactions on Computers*, vol. 48, no. 11, pp. 1183–1201, Nov. 1999.

[98] Luis Berrojo, I. González, Fulvio Corno, Matteo Sonza Reorda, Giovanni Squillero, Luis Entrena, and Celia Lopez, "New techniques for speeding-up fault-injection campaigns," in *Proceedings of the 2002 Design, Automation and Test in Europe Conference and Exhibition (DATE 2002)*, Mar. 2002, pp. 847–852.

[99] Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frédéric Salles, "Dependability of COTS microkernel-based systems," *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 138–163, Feb. 2002.

[100] Freescale Semiconductor, Inc., *RISC Central Processing Unit Reference Manual (Revision 1)*, Feb. 1999.

[101] Raul Barbosa, "Fault injection optimization through assembly-level pre-injection analysis," Master's thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 2004.

[102] PHYTEC Meßtechnik GmbH, *phyCORE-MPC565 Hardware Manual*, 6th ed., Apr. 2004.

[103] Josef Berwanger, Christian Ebner, Anton Schedl, Ralf Belschner, Sven Fluhrer, Peter Lohrmann, Emmerich Fuchs, Dietmar Millinger, Michael Sprachmann, Florian Bogenberger, Gary Hay, Andreas Krüger, Mathias Rausch, Wolfgang Budde, Peter Fuhrmann, and Robert Mores, "FlexRay: The communication system for advanced automotive control systems," *SAE Transactions*, vol. 110, no. 7, pp. 303–314, 2001.

[104] Thomas Führer, Bernd Müller, Werner Dieterle, Florian Hartwich, Robert Hugel, and Michael Walther, "Time triggered communication on CAN (Time Triggered CAN - TTCAN)," Robert Bosch GmbH, Tech. Rep., 2000.

[105] Hermann Kopetz and Günther Bauer, "The time-triggered architecture," *Proceedings of the IEEE*, vol. 91, no. 1, pp. 112–126, Jan. 2003.

[106] Kenneth P. Birman and Thomas A. Joseph, "Reliable communication in the presence of failures," *ACM Transactions on Computer Systems*, vol. 5, no. 1, pp. 47–76, Feb. 1987.

[107] Flaviu Cristian, "Agreeing on who is present and who is absent in a synchronous distributed system," in *Proceedings of the 18th International Symposium on Fault-Tolerant Computing (FTCS-18)*, Jun. 1988, pp. 206–211.

[108] Michael Barborak, Anton Dahbura, and Miroslaw Malek, "The consensus problem in fault-tolerant computing," *ACM Computing Surveys*, vol. 25, no. 2, pp. 171–220, Jun. 1993.

[109] Vilgot Claesson, Henrik Lönn, and Neeraj Suri, "An efficient TDMA start-up and restart synchronization approach for distributed embedded systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 8, pp. 725–739, Aug. 2004.

[110] Wilfried Steiner and Hermann Kopetz, "The startup problem in fault-tolerant time-triggered communication," in *Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN 2006)*, Jun. 2006, pp. 35–44.

[111] Marco Serafini, Andrea Bondavalli, and Neeraj Suri, "Online diagnosis and recovery: On the choice and impact of tuning parameters," *IEEE Transactions on Dependable and Secure Computing*, vol. 4, no. 4, pp. 295–312, Oct.-Dec. 2007.

[112] John Rushby, "Bus architectures for safety-critical embedded systems," in *Proceedings of the First International Workshop on Embedded Software (EMSOFT 2001)*, ser. Lecture Notes in Computer Science, Oct. 2001, vol. 2211, pp. 306–323.

[113] Theresa C. Maxino and Philip J. Koopman, "The effectiveness of checksums for embedded control networks," *IEEE Transactions on Dependable and Secure Computing*, to appear.

[114] Cédric Wilwert, Françoise Simonot-Lion, Ye-Qiong Song, and François Simonot, "Quantitative evaluation of the safety of X-by-wire architecture subject to EMI perturbations," in *Proceedings of the 10th IEEE Conference on Emerging Technologies and Factory Automation (ETFA 2005)*, vol. 1, Sep. 2005, pp. 755–762.

[115] Günther Bauer and Michael Paulitsch, "An investigation of membership and clique avoidance in TTP/C," in *Proceedings of the 19th IEEE Symposium on Reliable Distributed Systems (SRDS-2000)*, Oct. 2000, pp. 118–124.

[116] Henrik Lönn, "A fault tolerant clock synchronization algorithm for systems with low-precision oscillators," in *Proceedings of the 3rd European Dependable Computing Conference (EDCC-3)*, ser. Lecture Notes in Computer Science, Sep. 1999, vol. 1667, pp. 88–105.

[117] K. H. Kim, Hermann Kopetz, Kinji Mori, Eltefaat H. Shokri, and Günter Grünsteidl, "An efficient decentralized approach to processor-group membership maintenance in real-time LAN systems: The PRHB/ED scheme," in *Proceedings of the 11th Symposium on Reliable Distributed Systems*, Oct. 1992, pp. 74–83.

[118] Shmuel Katz, Pat Lincoln, and John Rushby, "Low-overhead time-triggered group membership," in *Proceedings of the 11th International Workshop on Distributed Algorithms (WDAG'97)*, ser. Lecture Notes in Computer Science, Sep. 1997, vol. 1320, pp. 155–169.

[119] Matthew Clegg and Keith Marzullo, "A low-cost processor group membership protocol for a hard real-time distributed system," in *Proceedings of the 18th IEEE Real-Time Systems Symposium (RTSS'97)*, Dec. 1997, pp. 90–98.

[120] Luís Rodrigues, Paulo Veríssimo, and José Rufino, "A low-level processor group membership protocol for LANS," in *Proceedings the 13th International Conference on Distributed Computing Systems (ICDCS'93)*, May 1993, pp. 541–550.

[121] André Schiper and Sam Toueg, "From set membership to group membership: A separation of concerns," *IEEE Transactions on Dependable and Secure Computing*, vol. 3, no. 1, pp. 2–12, Jan.-Mar. 2006.

[122] Matti A. Hiltunen and Richard D. Schlichting, "A configurable membership service," *IEEE Transactions on Computers*, vol. 47, no. 5, pp. 573–586, May 1998.

[123] Aleta Ricciardi and Kenneth P. Birman, "Using process groups to implement failure detection in asynchronous environments," in *Proceedings of the Tenth Annual ACM Symposium on Principles of Distributed Computing*, Aug. 1991, pp. 341–353.

[124] Christof Fetzer and Flaviu Christian, "A fail-aware membership service," in *Proceedings of the 16th Symposium on Reliable Distributed Systems (SRDS '97)*, Oct. 1997, pp. 157–164.

[125] Louise E. Moser, P. M. Melliar-Smith, and Vivek Agrawala, "Processor membership in asynchronous distributed systems," *IEEE Transactions on Parallel and Distributed Systems*, vol. 5, no. 5, pp. 459–473, May 1994.

[126] Paul D. Ezhilchelvan, Raimundo A. Macêdo, and Santosh K. Shrivastava, "Newtop: A fault-tolerant group communication protocol," in *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, May/Jun. 1995, pp. 296–306.

[127] Massimo Franceschetti and Jehoshua Bruck, "A group membership algorithm with a practical specification," *IEEE Transactions on Parallel and Distributed Systems*, vol. 12, no. 11, pp. 1190–1200, Nov. 2001.

[128] Michael J. Fischer, Nancy A. Lynch, and Michael S. Paterson, "Impossibility of distributed consensus with one faulty process," *Journal of the Association for Computing Machinery*, vol. 32, no. 2, pp. 374–382, Apr. 1985.

[129] Nuno F. Neves, Miguel Correia, and Paulo Veríssimo, "Solving vector consensus with a wormhole," *IEEE Transactions on Parallel and Distributed Systems*, vol. 16, no. 12, pp. 1120–1131, Dec. 2005.

[130] Paul D. Ezhilchelvan and Rogério de Lemos, "A robust group membership algorithm for distributed real-time systems," in *Proceedings of the 11th Real-Time Systems Symposium (RTSS'90)*, Dec. 1990, pp. 173–179.

[131] Valério Rosset, Pedro F. Souto, and Francisco Vasques, "A group membership protocol for communication systems with both static and dynamic scheduling," in *Proceedings of the 6th IEEE International Workshop on Factory Communication Systems (WFCS 2006)*, Jun. 2006, pp. 22–31.

[132] Carl Bergenhem and Johan Karlsson, "A process group membership service for active safety systems using TT/ET communication scheduling," in *Proceedings of the 13th Pacific Rim International Symposium on Dependable Computing (PRDC 2007)*, Dec. 2007, pp. 282–289.

[133] Richard Golding, "Weak-consistency group communication and membership," Ph.D. dissertation, University of California, Santa Cruz, USA, 1992.

[134] Henrik Lönn, "Synchronization and communication results in safety-critical real-time systems," Ph.D. dissertation, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1999.

[135] John Rushby, "Systematic formal verification for fault-tolerant time-triggered algorithms," *IEEE Transactions on Software Engineering*, vol. 25, no. 5, pp. 651–660, Sep./Oct. 1999.

[136] Valério Rosset, Pedro F. Souto, and Francisco Vasques, "Formal verification of a group membership protocol using model checking," in *On the Move to Meaningful Internet Systems 2007: CoopIS, DOA, ODBASE, GADA, and IS*, ser. Lecture Notes in Computer Science, Nov. 2007, vol. 4803, pp. 471–488.

[137] Holger Pfeifer, "Formal verification of the TTP group membership algorithm," in *Proceedings of the IFIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX)*, Oct. 2000, pp. 3–18.

[138] John Rushby, "Just-in-time certification," in *Proceedings of the 12th International Conference on Engineering of Complex Computer Systems (ICECCS 2007)*, Jul. 2007, pp. 15–24.

[139] Qurat ul Ain Inayat and Paul D. Ezhilchelvan, "A performance study on the signal-on-fail approach to imposing total order in the streets of Byzantium," in *Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN 2006)*, Jun. 2006, pp. 578–590.