Thesis for the Degree of Licentiate of Engineering

Multi-Layer Fault Tolerance for Distributed Real-Time Systems

RAUL BARBOSA

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2007

Multi-Layer Fault Tolerance for Distributed Real-Time Systems

Raul Barbosa

Technical Report No. 39L
ISSN 1652-876X
Department of Computer Science and Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden
Phone: +46 (0)31–772 1000

Contact information:

Department of Computer Science and Engineering
Chalmers University of Technology
SE–412 96 Göteborg, Sweden

Phone: +46 (0)31–772 1664
Fax: +46 (0)31–772 3663
Email: rbarbosa@ce.chalmers.se
URL: http://www.ce.chalmers.se/∼rbarbosa

Chalmers Reproservice
Göteborg, Sweden, 2007

Abstract

This thesis addresses issues in building fault-tolerant distributed real-time systems. Such systems are increasingly deployed in automotive and avionics applications. We focus on the design and validation of fault tolerance mechanisms.

From the design viewpoint, we develop the notion of multi-layer fault tolerance. A fault-tolerant distributed system contains 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. Circuit layer mechanisms provide the basic fault tolerance implemented in hardware; node layer mechanisms are executed locally in computer nodes; and system layer techniques involve multiple computer nodes to prevent faults from disturbing the system.

We make a probabilistic modeling analysis to compare federated to integrated architectures. Federated architectures have few or no fault tolerance mechanisms at the node layer and a node is the elementary unit of failure; integrated architectures provide robust partitioning mechanisms at the node layer in order to ensure that individual tasks are the unit of failure. We compare the reliability of the two architectures and propose a set of guidelines for building integrated architectures.

The thesis also addresses the problem of distributed redundancy management. We propose a group membership protocol to achieve consensus on the operational state of all nodes. The protocol is based on the principle that each message sent by a node in the membership is acknowledged by k other nodes, in a system with n nodes. Agreement on node departure is guaranteed if no more than $f = k - 1$ failures occur during n consecutive transmission slots. Additionally, we provide a solution for the reintegration of restarted nodes in the membership. This protocol is part of the system layer of fault tolerance mechanisms.

We address the validation of fault tolerance mechanisms by fault injection. This thesis describes an automated analysis technique to reduce the cost of fault injection campaigns. The analysis uses knowledge of program flow and resource usage to eliminate faults that have no possibility of activation. Our experimental results show that the fault-spaces are reduced by several orders of magnitude, when compared with the usual random approach.

List of Publications

This thesis is partly based on the following publications:

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

II 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, April 2005.

Acknowledgements

I would like to express my deepest gratitude to Professor Johan Karlsson for the knowledge shared with me and for his valuable advice as my supervisor.

I owe very special thanks to Professor emeritus Jan Torin, Professor Simin Nadjm-Tehrani, Sam Nicander and Dr. Kristina Forsberg for the insightful discussions on the partitioning problem and for the substantial comments on the third chapter of this thesis.

I would like to show my appreciation to Professor Mário Rela for the introduction to the field of dependability. His enthusiasm had a strong influence in my decision to pursue studies in this area. Many thanks go also to Professor João Cunha for sharing his knowledge with me, in discussions both at the University of Coimbra and at Chalmers University of Technology.

Special thanks are due to Daniel Skarin, with whom I have 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 the final stretch of their M.Sc. studies. I would also like to thank Carl Bergenhem for the valuable discussions and for his joyful attitude.

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

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

Last and most importantly, I am grateful to my parents Ângelo and Marta for everything they have done for me, for their sacrifices in my upbringing and for the everlasting support in pursuing my goals in life. Obrigado!

This work has been 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 **3**
2.1 Terminology . . . 3
2.1.1 Faults, Errors and Failures . . . 3
2.1.2 Dependability Attributes . . . 5
2.1.3 The Means to Dependability . . . 6
2.2 System Model . . . 7
2.3 Multi-Layer Fault Tolerance . . . 7
2.4 Goals . . . 9
2.5 Thesis Contributions . . . 10

3 Partitioning for Integrated Architectures **13**
3.1 Introduction . . . 13
3.2 Theoretical Motivation . . . 14
3.2.1 Software Failures . . . 16
3.2.2 Hardware Failures . . . 18
3.2.3 Derivation of Formulas . . . 20
3.3 Requirements for Partitioning . . . 25
3.4 Mechanisms for Partitioning . . . 29
3.4.1 Spatial Partitioning . . . 29
3.4.2 Temporal Partitioning . . . 34
3.5 Guidelines for Integrated Architectures . . . 35
3.6 Discussion and Conclusion . . . 36

4 Distributed Redundancy Management **39**
4.1 Introduction . . . 39
4.2 System Model . . . 41

4.3 Protocol Specification 43
4.3.1 Definitions 44
4.3.2 Agreement on Departure 45
4.3.3 Agreement on Reintegration Ordering 48
4.3.4 Recovery of the Membership State 50
4.3.5 Agreement on Reintegration 50
4.4 Discussion 54
4.4.1 Further Considerations 56
4.5 Prototype Implementation 56
4.5.1 Network Configuration 57
4.5.2 Network and Membership Performance 58
4.6 Task-Group Membership Agreement 59
4.6.1 Multiple Sending Slots 59
4.6.2 Task-Group Membership through Fail-Reporting 60
4.7 Related Research 61
4.8 Discussion and Conclusion 61

5 Fault Injection Testing **63**
5.1 Introduction 63
5.2 Related Research 65
5.3 Fault-Space Optimization Method 66
5.3.1 Optimization Input 66
5.3.2 Optimization Output 67
5.3.3 Performing the Optimization 67
5.4 Experimental Setup 68
5.4.1 Fault Injection Tool 68
5.4.2 MPC565 Microcontroller 69
5.4.3 Workloads 70
5.4.4 Fault Model and Fault Selection 71
5.5 Experimental Results 72
5.5.1 Fault Injection in Registers 72
5.5.2 Fault Injection in Memory 75
5.5.3 Fault-Space Considerations 77
5.6 Discussion and Conclusion 78

6 Conclusion **79**
6.1 Future Work 80

References **82**

Acronyms

CAN	Controller Area Network
COTS	Commercial Off-The-Shelf
CPU	Central Processing Unit
ECC	Error-Correcting Code
EDF	Earliest Deadline First
FMEA	Failure Modes and Effects Analysis
IMA	Integrated Modular Avionics
MMU	Memory Management Unit
MPU	Memory Protection Unit
MTTF	Mean Time To Failure
RM	Rate-Monotonic Scheduling
SEU	Single Event Upset
SIL	Safety Integrity Level
TDMA	Time Division Multiple Access
TLB	Translation Lookaside Buffer
TMR	Triple Modular Redundancy
WCET	Worst-Case Execution Time

List of Figures

2.1 The dependability tree. 4
2.2 Structural elements of the architectural framework. 7
2.3 Multi-layer fault tolerance in distributed real-time systems. 9

3.1 Relative development effort for software as a function of partitioning effort for a constant MTTF. 17
3.2 Relative development effort for hardware as a function of the number of tasks for a constant MTTF. 19
3.3 Task assignment in the federated architecture. 21
3.4 Task assignment in the integrated architecture. 21
3.5 State transition diagram for the federated architecture (software faults). 22
3.6 State transition diagram for the integrated architecture (software faults). 22
3.7 State transition diagram for the federated architecture (hardware faults). 25
3.8 State transition diagram for the integrated architecture (hardware faults). 25

4.1 Round number signaling by a node in the membership, using the *r-flag*. 49
4.2 Sponsor-graph of a 6-node system ($k = 2$). 55
4.3 Sponsor-graph immediately after the failure of sending node 1. . . . 56
4.4 Sponsor-graph 2 slots after the failure of node 1. 56
4.5 The real-time Ethernet prototype. 57

5.1 Example of the optimization procedure. 69
5.2 Evaluation platform for the jet engine application. 70
5.3 Exceptions in the quicksort non-optimized campaign (83 faults in registers). 73

5.4 Exceptions in the quicksort optimized campaign (744 faults in registers). 74
5.5 Exceptions in the jet engine controller non-optimized campaign (200 faults in registers). 74
5.6 Exceptions in the jet engine controller optimized campaign (466 faults in registers). 75
5.7 Number of faults injected per register. 76
5.8 Exceptions in the jet engine controller non-optimized campaign (40 faults in memory). 77
5.9 Exceptions in the jet engine controller optimized campaign (166 faults in memory). 77

List of Tables

3.1 Reliability of 1-resilient systems with a hardware failure rate of 10^{-6} failures/h per processor. 19

4.1 Mapping of component failures to failure classes. 42
4.2 Parameters of the real-time Ethernet prototype. 58
4.3 Node departure and node reintegration latencies. 59

5.1 Distribution of outcomes of fault injection in registers. 72
5.2 Error detection coverage estimations (registers). 73
5.3 Distribution of outcomes of fault injection in memory. 76
5.4 Error detection coverage estimations (memory). 76
5.5 Comparison between fault-space sizes (registers). 78
5.6 Comparison between fault-space sizes (memory). 78

CHAPTER 1

Introduction

We often depend on a computer system without being aware of its existence. Whether it is our mobile phone or the plane 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 system failure against the cost of reducing the risk of such an event. Thus, we are willing to pay for safety along with the functional benefits of a system.

From the designer's viewpoint, safety and functional features impose conflicting requirements. Adding new and enhancing existing features increases both the hardware and the software complexity – a major obstacle to creating dependable systems. On the other hand, society craves for new products that contribute to its well-being. The increased dependence placed on computers – a steady trend in most economic sectors (e.g. transport, energy, medical, etc.) – demands strict attention to their reliability, availability, safety and other attributes of dependability.

There is no simple procedure for building dependable computer systems. One failure every million hours is a common failure rate for electronic components. This failure rate is, however, too high for most safety-critical systems, where a failure is potentially catastrophic. Examples of this are avionics systems, where many lives are at stake, and automotive systems, where the number cars in use is very large. For these reasons, systems must be more reliable than their constituting parts.

To achieve this, many critical systems use several redundant computers and activate a spare when a failure is detected. This form of redundancy is an effective approach to fault tolerance, i.e. to provide an acceptable level of service despite the occurrence of faults. However, redundancy leads to greater cost and higher fault rate, due to a larger number of system parts. The objective of the designer must therefore be to minimize the redundancy. Moreover, hardware redundancy is insufficient to tolerate software defects. Multiple copies of the same faulty program are

likely to produce the same erroneous result, even if running on different computers. A highly dependable system must therefore be equipped with the means to tolerate faults of diverse origins. Furthermore, the implementation of such means should be cost-effective, in order to ensure that they can be applied in a systematic way.

In critical applications computers are usually embedded into the devices they control. Users seldom perceive the presence of those computers and their operation is limited to the scope of the application. Though many 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, the embedded computer systems are usually distributed as well. Thus, the concerns with fault tolerance go beyond a single computer. 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 for the cost-effective implementation of fault tolerance in distributed real-time 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 safety-critical and non-critical functions to be executed on the same processing node. The framework aims to provide fault tolerance using a multi-layer approach which combines circuit layer, node layer and system layer mechanisms. The core idea is to handle a majority of the errors at the node layer without any involvement of the other nodes in the system. Thus, nodes should be able to both detect and recover from errors autonomously. System layer mechanisms are provided to deal with errors that cannot be corrected at the node layer, e.g. a permanent hardware fault in a node.

The supporting services provide node layer detection and recovery of transient faults which ensures fail-silent or fail-signaled failure semantics, consistency among replicated programs and distributed redundancy management under fail-silent or fail-signaled failure semantics. These services are characterized by the support for both time-triggered and event-driven execution of application tasks and by the use of a real-time communication network which supports both on-line and off-line scheduling of messages. We address the validation of fault tolerance mechanisms by fault injection.

The thesis is organized as follows: Chapter 2 describes the dependable computing background and introduces the architectural framework for multi-layer fault tolerance. Chapter 3 looks into fault-containment at the node level and provides a set of guidelines for building distributed systems with partitioning techniques. Chapter 4 discusses redundancy management and describes a membership agreement protocol (a basic service for redundancy management). Chapter 5 focuses on the evaluation of fault tolerance mechanisms through fault injection.

CHAPTER 2

The Architectural Framework

This chapter introduces the architectural framework for multi-layer fault tolerance in distributed real-time systems. First, some background to the field of dependable computing is given, followed by a detailed description of the framework.

2.1 Terminology

Safety can be defined as "a property of a system that it will not endanger human life or the environment" [73]. According to the taxonomy of dependable and secure computing [12], 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 [12] by including only the attributes of interest for dependability. The following sections describe the *threats*, *attributes* and *means* to attain dependability.

2.1.1 Faults, Errors and Failures

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

- A *fault* is a defect in the system which, when activated, leads to an error.

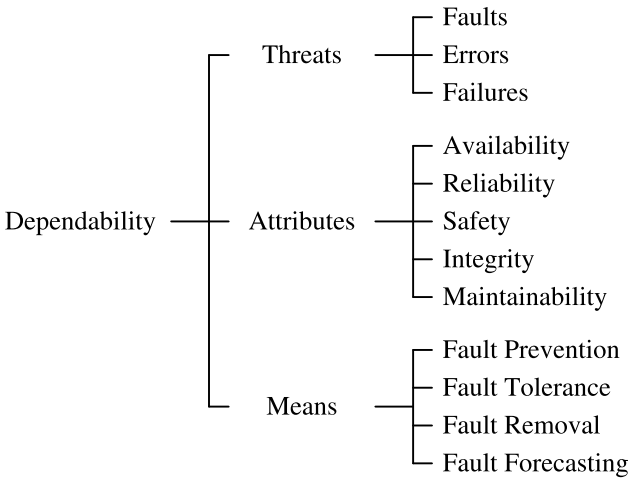

Figure 2.1: The dependability tree.

- An *error* is an incorrect system state that may affect the external behaviour, thereby causing a failure.
- A *failure* occurs when the delivered service deviates from what is considered correct.

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 receivers.
- 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 measure 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 [50] 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 [68]. 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 the 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. However, a probabilistic approach allows us to characterize and compare alternative designs with respect to their dependability.

2.1.3 The Means to Dependability

The *means* to attain dependability consist of methods and techniques to assure 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. The most common approaches use either hardware redundancy, software redundancy, time redundancy or information redundancy to identify erroneous conditions. The subsequent recovery process relies on the remaining fault-free parts of the system to correct the errors and/or prevent them from reappearing.

- *Fault removal* is applied during the use phase of a system, either by corrective or preventive maintenance. It usually requires human intervention to replace faulty units. Though faults can be removed during the development phase, we view this activity as fault prevention[1]. Obviously, product developers should take into account previous system generations when creating new products. Thus, a fault removed in one product should most certainly be prevented in the future.

- *Fault forecasting* methods provide the assurance regarding 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.

[1] This view differs from the definition in [12].

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, actuators, etc.). 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 [74]. 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.

A service's tasks 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.

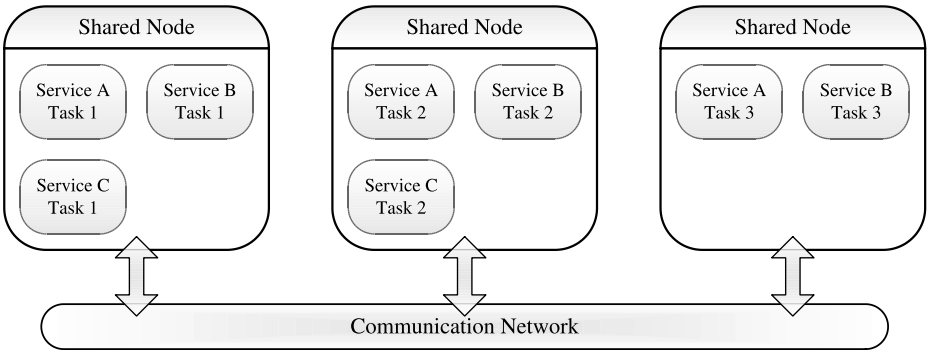

Figure 2.2: Structural elements of the architectural framework.

2.3 Multi-Layer 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:

- *Circuit layer* mechanisms provide the basic fault tolerance implemented in hardware. Most hardware units include some forms of fault tolerance. Exam-

ples are the ability of most microprocessors to detect exceptional conditions (e.g. invalid instructions, erroneous memory accesses, etc.), cache protection with parity checks and main memory protection with error-correcting codes (ECCs). Triple modular redundant (TMR) logic at the transistors [34] is an example of a more advanced circuit 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 are those that involve multiple processing nodes. A commonly used fault-tolerant system is the TMR system, in which the results of three nodes are voted to obtain a single output. TMR systems are an example of static redundancy. A cold standby spare, activated when the primary node fails, is an example of dynamic redundancy.

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 *circuit 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 once that task produces no results. 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.

The combination of fault tolerance mechanisms at different layers is fundamental to ensure their cost-efficiency, 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 guideline 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 ear-

lier 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 circuit and node layers in order to guarantee consistent failure modes. Fault tolerance mechanisms thus form a hierarchy. Figure 2.3, adapted from [1], illustrates the three-layer hierarchy of fault tolerance mechanisms.

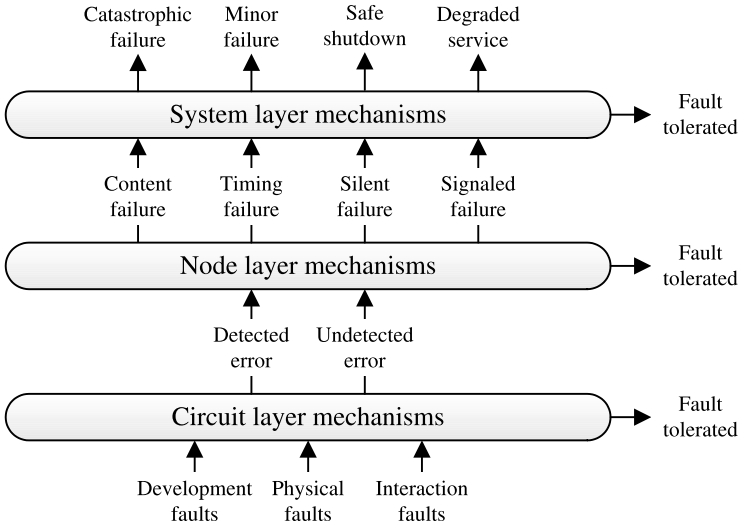

Figure 2.3: Multi-layer fault tolerance in distributed real-time systems.

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 circuit 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.

2.4 Goals

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 [63] or with software and time redundancy [3].

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 multi-layer approach, which combines circuit 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 safety-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 safety-critical tasks and event-driven execution for non-critical tasks and recovery mechanisms.
- Use a real-time communication network which supports both on-line and off-line scheduling of messages.

2.5 Thesis Contributions

This thesis builds on the concept of multi-layer fault tolerance for the design of distributed real-time systems. The main contributions can be grouped into three areas of study: analysis of robust partitioning methods, distributed redundancy management and validation of fault tolerance mechanisms through fault injection.

- *Analysis of robust partitioning methods* – The requirements of partitioned systems are examined in the light of declassification – a computer security notion that we found useful for specifying partitioning requirements. Subsequently, we survey the existing mechanisms to fulfill those requirements. Moreover, we present a probabilistic analysis of the reliability of federated and integrated architectures. Based on this analysis we propose a set of guidelines for building integrated architectures.

- *Distributed redundancy management* – We propose a processor-group membership protocol that achieves consensus on the operational state of all nodes. It is intended for systems that use time-triggered communication. The protocol builds on the idea that each message sent by a node in the membership is acknowledged by a configurable number of sponsor nodes. Incrementing the number of sponsors increases the reliability, while more resources must be allocated to the protocol. We have built a real-time Ethernet prototype to serve as an experimental platform. Though not regarded as a scientific contribution, the experimental platform allowed the protocol to be tested in a realistic environment. The protocol description is based on Paper I.

- *Fault injection testing* – We describe 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 technique was presented in Paper II.

CHAPTER 3

Partitioning for Integrated Architectures

One of the goals of this thesis is to study methods that allow a task to be the elementary unit of failure. Node layer mechanisms should therefore detect and recover from faulty tasks without involving other processing nodes. To achieve this, robust partitioning mechanisms must assure fault containment within nodes. Robust partitioning allows both safety-critical and non-critical functions to be executed on the same node. Furthermore, it allows nodes to provide degraded service in the event of faults. This chapter identifies the state-of-the-art in robust partitioning techniques and proposes a set of guidelines for building integrated architectures, where the nodes are shared among several tasks. The guidelines are derived through probabilistic analysis.

3.1 Introduction

Embedded systems have traditionally been implemented by dedicating a computer system 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 [7] and the ARINC 653 standard [8] for the aerospace industry; and the AUTOSAR project [41] 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 favor 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. The next section provides a probabilistic analysis to understand the impact of integrated architectures on the system's reliability. Section 3.3 identifies the requirements for partitioning and Section 3.4 discusses the existing mechanisms to fulfill those requirements. Section 3.5 provides a set of qualitative guidelines for building integrated architectures, based on the argumentation presented throughout the chapter. Section 3.6 summarizes the main conclusions of the chapter.

3.2 Theoretical Motivation

In this section we analyze the development 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 that can be reduced by increasing the development effort. Furthermore, we assume that the

development process follows a standard that assigns criticality levels to components; higher criticality levels imply 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 the 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 which provide 100% error detection coverage.

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 tradeoff 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. Thus, we define the error detection coverage of the partitioning mechanisms (denoted by c) as a 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 as well as the coverage of the partitioning mechanisms to be as low as possible. These are, however, two separate issues.

3.2.1 Software Failures

This section focuses the reliability assessment regarding software failures. We apply continuous-time Markov modeling [68] 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 [36] 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 [81].

However, the statistical approach to software reliability assessment is not always used in practice. The DO-178B [64] 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 [16] 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 failure rate for software in order to reason about the dependability attributes of the two architectures. Furthermore, we assume that the failure rate decreases when the software is developed to a higher criticality level. The IEC 61508 [44] international standard for functional safety defines four Safety Integrity Levels (SILs) for safety-related functions. To each integrity level corresponds a range of failure probabilities. Higher integrity levels impose lower probabilities of failure (either specified in terms of probability of failure per hour *or* probability of failure on demand). 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.

Consider the scenario where two tasks (software components), which were previously granted their own microcontroller, are to be deployed on a single one. Assume,

for instance, that the two tasks are equally critical, with the same failure rate, and that the system can cope with the failure of any of them, but not both. In this scenario we can adjust two different parameters: (i) the development effort to decrease the software failure rate (in failures/h) from λ_F in the federated architecture to λ_I in the integrated architecture and (ii) the error detection coverage of the partitioning mechanisms (denoted by c). Equation (3.1) shows the relative development effort ($^{\lambda_F}/_{\lambda_I}$) as a function the partitioning coverage, given the requirement that the two architectures should have the same Mean Time To Failure (MTTF).

$$\frac{\lambda_F}{\lambda_I} = \frac{3}{2c+1} \text{ with } c \in [0,1] \tag{3.1}$$

Equation (3.1) applies to the failure of software in systems which can tolerate the failure of a single task. We call these systems *1-out-of-n-resilient* systems, or simply *1-resilient* systems. Equation (3.1) applies to the special case where the system is composed of two tasks (a 1-out-of-2-resilient system). Equation (3.2) generalizes the relation for 1-resilient systems with any number of tasks. These equations are derived in Section 3.2.3.

$$\frac{\lambda_F}{\lambda_I} = \frac{2n-1}{(c+1)n-1} \text{ with } n \in \{2,3,\ldots\} \wedge c \in [0,1] \tag{3.2}$$

The plots in Figure 3.1 illustrate the relative software development effort (to decrease the failure rate of the different tasks) as a function of the partitioning effort (to increase the error detection coverage of the partitioning mechanisms).

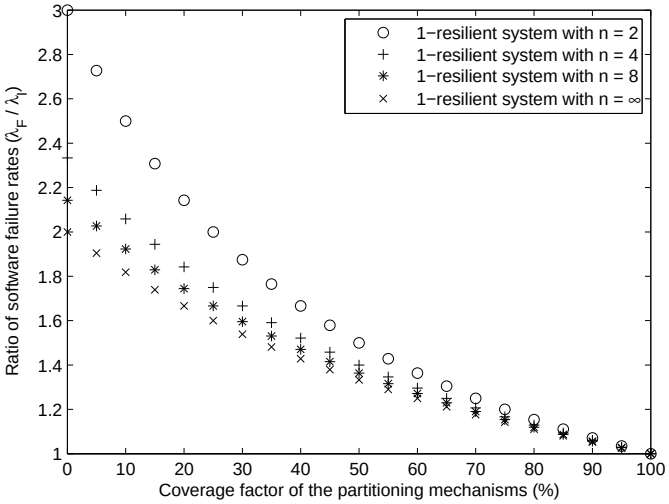

Figure 3.1: Relative development effort for software as a function of partitioning effort for a constant MTTF.

One can conclude by analysing Figure 3.1 that, without partitioning (when $c = 0$), all tasks must be guaranteed to fail with, at most, half of their original failure rate (one third in the worst case) in order for the integrated architecture to maintain the same MTTF as the federated architecture. On the other hand, if partitioning is 100% effective there is no need for software development at higher criticality levels. In fact, with perfect partitioning (when $c = 100\%$), the reliability (and not only the MTTF) of the two architectures will be the same over any period of time.

3.2.2 Hardware Failures

So far our analysis has focused on the failure of software components. It is when dealing with hardware failures that integrated architectures have a great potential to improve the system's reliability. Such architectures allow complex systems to be built using a smaller number of microcontrollers. Hence, the overall hardware failure rate will be determined by a smaller number of hardware components.

Consider again the above-described case of 1-resilient systems, but when facing hardware failures. Again, we wish to share the same processor among n different tasks. In a federated architecture each task would have a dedicated processor with a known failure rate λ'_F. In an integrated architecture all tasks share a single hardware platform which has a λ'_I failure rate. We have to assume that the partitioning mechanisms are not effective against hardware failures. All partitions, as well as the partitioning mechanisms, depend on the correct operation of their underlying hardware. Some transient hardware faults will eventually be confined to a single partition but this is not a requirement posed on partitioning mechanisms. We therefore assume that a hardware failure in the integrated architecture will result in the failure of all tasks sharing the faulty processor. Under these assumptions the relative hardware development effort (λ'_F/λ'_I), for the two architectures to present the same MTTF, is a function of the number of tasks. This relation, derived in Section 3.2.3, is shown in Equation (3.3). The equation is plotted in Figure 3.2.

$$\frac{\lambda'_F}{\lambda'_I} = \frac{2n-1}{n^2-n} \text{ with } n \in \{2, 3, \ldots\} \tag{3.3}$$

By analysing Figure 3.2 it is possible to come to the conclusion that the development effort for hardware is lower in integrated architectures, even with no partitioning at all, if three or more tasks are integrated. This is true in cases where three or more tasks are merged into the same processing node, whenever the MTTF is the most significant measure of dependability.

However, the MTTF is not as meaningful as a reliability value with an associated time to compare the dependability of the two architectures. Most automotive and avionics computer systems are mission-critical, designed to have very high reliability during a 5 to 10 year life cycle. For such systems the MTTF is not always a good measure for comparison since it integrates the reliability function over an

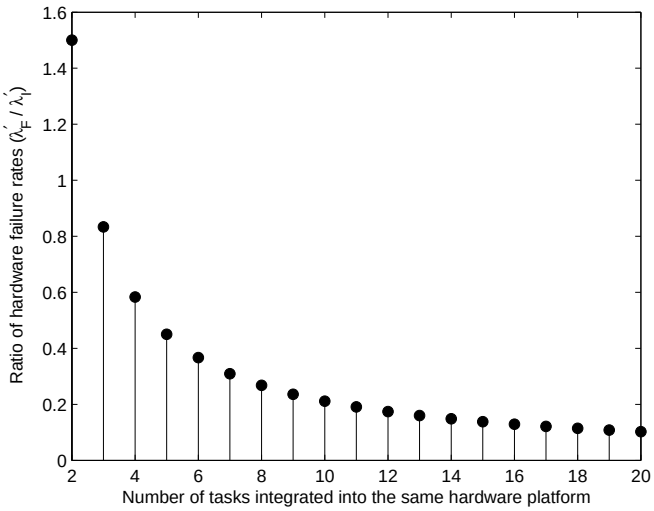

Figure 3.2: Relative development effort for hardware as a function of the number of tasks for a constant MTTF.

infinite amount of time. Table 3.1 provides the values of the reliability of the two architectures (for 1-resilient systems) after 5 and 10 years. The reliability of the integrated architecture is not affected by the number of tasks since the hardware failure rate depends on the number of hardware units rather than the number of tasks. Regarding the federated architecture Table 3.1 shows the reliability for systems with 5, 10 and 15 tasks – one task in each processing unit. We assume that the hardware failure rate per processor is the same in both architectures ($\lambda'_F = \lambda'_I = 10^{-6}$ failures/h). The values in Table 3.1 were obtained using Equations (3.12) and (3.13), derived in Section 3.2.3.

	Number of processors	**Reliability after 5 years**	**Reliability after 10 years**
Federated Architecture	*5*	*0.983*	*0.941*
	10	*0.934*	*0.798*
	15	*0.867*	*0.638*
Integrated Architecture	*1*	*0.957*	*0.916*

Table 3.1: Reliability of 1-resilient systems with a hardware failure rate of 10^{-6} failures/h per processor.

The first conclusion one can draw from Table 3.1 is that the reliability of the federated architecture decreases with the number of processors. Hence, integrated architectures are a promising alternative by reducing the number of processors.

The second and perhaps most important conclusion (also from Table 3.1) is that integrated architectures are not beneficial in all situations. It is only when the number of processors exceeds a certain threshold (around 7 processors for 1-

resilient systems) that we can benefit from integrated architectures. This number has already been surpassed by the industry as there can be as many as 70 processors in a high-end road vehicle and 50 in a modern airplane, with the consequent penalty in safety and reliability.

The third conclusion is that, if we only consider hardware failures, it is simpler to integrate mixed levels of criticality (e.g. one critical function per hardware unit coexisting with several non-critical functions) than to integrate highly critical functions. In the integrated architecture hardware failures can potentially disrupt more than one critical function. Thus, when the number of functions is small (e.g. 5 tasks in Table 3.1), the federated architecture offers better reliability. Usually, even if the system has hundreds of functions, only a small fraction of those is critical. Thus, deploying all critical functions in a single shared processor is not beneficial unless the system suffers a catastrophic failure when *any of them* fails. In this case, which is a 0-resilient system, the hardware reliability increases by using fewer hardware components. A larger number of hardware components can only increase reliability when used redundantly.

3.2.3 Derivation of Formulas

This section contains the demonstrations of the theoretical results described textually in the previous section. We use continuous-time Markov processes [68] to model the system. We assume that the failure of one component is statistically independent of the failures of other components. Furthermore, we assume that component failures are exponentially distributed with a constant failure rate.

Since we assume independent failures, we do not consider correlated component failures. This does not mean, however, that the failure of one component will not propagate to other system components. Under certain conditions, a failure may have consequences outside the component where it originated. Nevertheless, the occurrence of one such failure is assumed to be statistically independent of any other component failure.

We consider the scenario of 1-resilient systems, described earlier in this chapter.

Definition 1. *A system is said to be 1-out-of-n-resilient if it can can tolerate the failure of any single component. For short, we call these systems 1-resilient.*

We begin by modeling the federated and the integrated architectures regarding software failures. The problem described in the beginning of Section 3.2.1 involved merging two or more tasks (software components), which were previously granted their own microcontroller, into a single one. These tasks compose a 1-resilient system and are equally critical, with the same failure rate. Consider the following notations:

- λ_F – Failure rate of each task in the federated architecture.

- λ_I – Failure rate of each task in the integrated architecture.
- c – Coverage factor of the partitioning mechanisms, i.e. the conditional probability that a partitioning violation does not occur, given that a task has failed.

Definition 2. *We define the relative development effort as the quotient* $^{\lambda_F}/_{\lambda_I}$. *The relative development effort is a measure of the difference in the development effort for a component moved from a federated architecture to an integrated architecture. This definition is based on our assumption that the development effort of a component is inversely related to its failure rate.*

In the typical federated architecture each of the n software components would have a dedicated microcontroller, as shown in Figure 3.3. On the other hand, with an integrated architecture, the software components would share a single microcontroller, as shown in Figure 3.4.

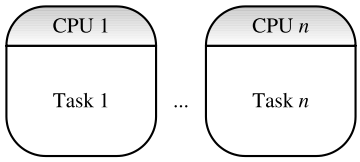

Figure 3.3: Task assignment in the federated architecture.

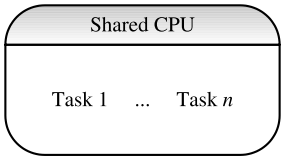

Figure 3.4: Task assignment in the integrated architecture.

The first problem is to determine the development effort necessary to ensure that the integrated architecture is as dependable as the federated architecture. Considering the MTTF as the measure to compare the dependability of the two architectures, Equation (3.2) can be derived through continuous Markov modeling.

Figures 3.5 and 3.6 show the state transition diagrams for the federated architecture and the integrated architecture, respectively. There is a direct mapping between the states of the two models:

- A and A' – All n tasks are functioning correctly.
- B and B' – One task has failed and the remaining $n-1$ tasks are functioning correctly.
- F and F' – A second task has failed. Since the system can only tolerate a single failure, this state represents a system failure.

The reliability of the system is the probability that the system functions correctly over a specified period of time [68]. Let $R(t)$ denote the probability that the system is functional in $[0, t]$. For the two architectures in question, $R_F(t) = P_{A'}(t) + P_{B'}(t)$ and $R_I(t) = P_A(t) + P_B(t)$ represent the reliability of the federated and the

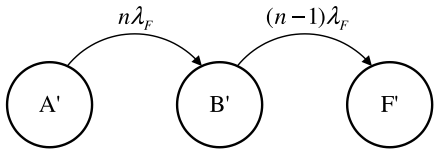

Figure 3.5: State transition diagram for the federated architecture (software faults).

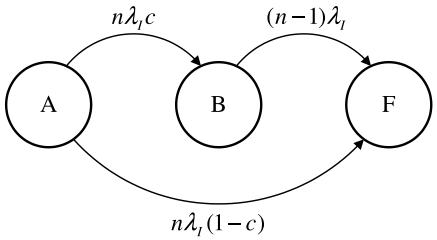

Figure 3.6: State transition diagram for the integrated architecture (software faults).

integrated design, respectively. $P_A(t)$ is the probability of being in state A at time t.

An immediate conclusion one can draw is that, when $c = 1$ (perfect partitioning coverage), $R_F(t) = R_I(t)$ if and only if $\lambda_F = \lambda_I$. When $c = 1$ the Markov models are exactly the same, with the exception of the failure rates. Thus, $P_A(t) = P_{A'}(t)$ and $P_B(t) = P_{B'}(t)$ as long as $\lambda_F = \lambda_I$. This means that, with perfect partitioning, we obtain the same reliability for the two architectures, regarding software failures, with no additional development effort ($\lambda_F = \lambda_I \Leftrightarrow {}^{\lambda_F}/_{\lambda_I} = 1$).

We now wish to obtain the expression which relates the development effort to any given value of c. From the Markov model in Figure 3.6 we obtain the transition rate matrix Q:

$$Q = \begin{bmatrix} -n\lambda_I & n\lambda_I c & n\lambda_I(1-c) \\ 0 & -(n-1)\lambda_I & (n-1)\lambda_I \\ 0 & 0 & 0 \end{bmatrix}$$

Since we know that

$$P'(t) = P(t) \cdot Q$$
$$P(0) = \begin{bmatrix} 1 & 0 & 0 \end{bmatrix}$$

we obtain the system of equations

$$P'_A(t) = -n\lambda_I P_A(t), \tag{3.4}$$
$$P'_B(t) = n\lambda_I c P_A(t) - (n-1)\lambda_I P_B(t), \tag{3.5}$$
$$P'_F(t) = n\lambda_I(1-c)P_A(t) + (n-1)\lambda_I P_B(t) \tag{3.6}$$

to which we can apply the Laplace transform

$$\mathscr{L}\{P'(t)\} = s\tilde{P}(s) - P(0).$$

So we have

$$s\tilde{P}_A(s) - 1 = -n\lambda_I \tilde{P}_A(s), \tag{3.7}$$
$$s\tilde{P}_B(s) = n\lambda_I c\tilde{P}_A(s) - (n-1)\lambda_I \tilde{P}_B(s), \tag{3.8}$$
$$s\tilde{P}_F(s) = n\lambda_I(1-c)\tilde{P}_A(s) + (n-1)\lambda_I \tilde{P}_B(s). \tag{3.9}$$

From (3.7) we get

$$\tilde{P}_A(s) = \frac{1}{s + n\lambda_I} \tag{3.10}$$

From (3.7) and (3.8) we get

$$\begin{aligned} \tilde{P}_B(s) &= \frac{1}{s + (n-1)\lambda_I} n\lambda_I c\, \tilde{P}_A(s) \\ &= \frac{n\lambda_I c}{(s + (n-1)\lambda_I)(s + n\lambda_I)} \\ &= \frac{nc}{s + (n-1)\lambda_I} - \frac{nc}{s + n\lambda_I} \end{aligned} \tag{3.11}$$

Applying the inverse Laplace transform to both sides of (3.10) and (3.11) we get

$$\begin{aligned} \mathscr{L}^{-1}\{\tilde{P}_A(s)\} &= \mathscr{L}^{-1}\Big\{\frac{1}{s + n\lambda_I}\Big\} \quad \Rightarrow \\ \Rightarrow \quad P_A(t) &= e^{-n\lambda_I t} \end{aligned}$$

and

$$\begin{aligned} \mathscr{L}^{-1}\{\tilde{P}_B(s)\} &= \mathscr{L}^{-1}\Big\{\frac{nc}{s + (n-1)\lambda_I} - \frac{nc}{s + n\lambda_I}\Big\} \quad \Rightarrow \\ \Rightarrow \quad P_B(t) &= nc(e^{-(n-1)\lambda_I t} - e^{-n\lambda_I t}). \end{aligned}$$

Thus

$$R_I(t) = P_A(t) + P_B(t) = nce^{-(n-1)\lambda_I t} - (nc-1)e^{-n\lambda_I t}.$$

By replacing c with 1 we obtain $R_F(t)$ (the reliability of the federated architecture):

$$R_F(t) = ne^{-(n-1)\lambda_F t} - (n-1)e^{-n\lambda_F t}.$$

We now obtain the MTTF of both architectures

$$\begin{aligned} MTTF_I &= \int_0^\infty R_I(t)\, dt \\ &= \left[-\frac{nc}{(n-1)\lambda_I} e^{-(n-1)\lambda_I t} + \frac{nc-1}{n\lambda_I} e^{-n\lambda_I t} \right]_0^\infty \\ &= \frac{nc + n - 1}{(n^2 - n)\,\lambda_I} \\ MTTF_F &= \int_0^\infty R_F(t)\, dt \\ &= \left[-\frac{n}{(n-1)\lambda_F} e^{-(n-1)\lambda_F t} + \frac{n-1}{n\lambda_F} e^{-n\lambda_F t} \right]_0^\infty \\ &= \frac{2n - 1}{(n^2 - n)\,\lambda_F} \end{aligned}$$

We equate the MTTF of the two architectures to obtain the relative development effort:

$$MTTF_I = MTTF_F \quad \Leftrightarrow \quad \frac{nc+n-1}{(n^2-n)\,\lambda_I} = \frac{2n-1}{(n^2-n)\,\lambda_F}$$
$$\Leftrightarrow \quad \frac{\lambda_F}{\lambda_I} = \frac{2n-1}{(c+1)n-1}$$

Which completes the demonstration of Equations (3.1) and (3.2) presented in Section 3.2.1. Equation (3.1) describes the special case when $n = 2$. We now turn to the problem of hardware failures and demonstrate Equation (3.3).

Equation (3.3) defines the relative development effort necessary to ensure that the integrated and the federated architecture are equally dependable. The MTTF is the measure taken into account to compare their dependability.

The task assignment is the same as in Figures 3.3 and 3.4. We assume that a hardware failure affecting a processing node will hinder all of the processor's tasks from executing correctly, i.e. partitioning is not effective against hardware failures. Consider the following notation:

- λ'_F – Hardware failure rate of each processing node in the federated architecture.
- λ'_I – Hardware failure rate of the shared processing node in the integrated architecture.

Figures 3.7 and 3.8 show the state transition diagrams for the two architectures, regarding hardware failures. The models consist of the following states:

- A' – All n processing nodes are functioning correctly.
- B' – One processing node has failed but the remaining $n-1$ are functioning correctly.
- F' – A second processor has failed. Since the system can only tolerate a single failure, this state represents a system failure.
- A – The shared processing node is functioning correctly.
- B – The shared processing node has failed. Since all tasks share this node, this is a failed state.

The Markov model of the federated architecture is the same as the one in Figure 3.5 (for software failures). Thus, we can reuse the previous set of calculations to obtain the reliability of the federated architecture concerning hardware failures:

$$R_F(t) = ne^{-(n-1)\lambda'_F t} - (n-1)e^{-n\lambda'_F t}. \tag{3.12}$$

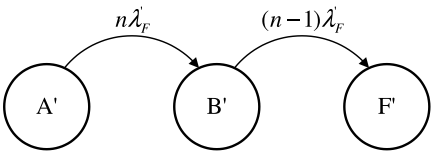

Figure 3.7: State transition diagram for the federated architecture (hardware faults).

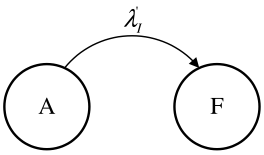

Figure 3.8: State transition diagram for the integrated architecture (hardware faults).

Hardware failures are assumed to be exponentially distributed, so

$$R_I(t) = e^{-\lambda_I' t}. \tag{3.13}$$

We now obtain the MTTF, regarding hardware, of both architectures

$$\begin{aligned} MTTF_F &= \int_0^\infty R_F(t)\,dt = \frac{2n-1}{(n^2-n)\lambda_F'} \\ MTTF_I &= \int_0^\infty R_I(t)\,dt = \frac{1}{\lambda_I'} \end{aligned}$$

We equate the MTTF of the two architectures to obtain the relative development effort:

$$\begin{aligned} MTTF_I = MTTF_F \quad &\Leftrightarrow \quad \frac{1}{\lambda_I'} = \frac{2n-1}{(n^2-n)\lambda_F'} \\ &\Leftrightarrow \quad \frac{\lambda_F'}{\lambda_I'} = \frac{2n-1}{n^2-n} \end{aligned}$$

We have now found the relative development effort for hardware, which completes the demonstration of Equation (3.3), used in Section 3.2.1.

3.3 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 *sleep-until* 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 sleep-until 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 sleep-until primitive on time. For cost reasons, it would not be effective to gather the same amount of dependability arguments for a non-critical function and for a critical function. Thus, the sleep-until primitive must be replaced by a stronger mechanism. One such mechanism should allow the safety-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 [65].

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 [12]. Researchers have realized that many systems are not secure unless they are dependable, and *vice versa*. Examples of such systems are are network firewalls, which must be highly available to be secure; computer systems in power plants, which must be secured against malicious interaction faults to be dependable; and so on.

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 system nodes. Examples of such techniques are the electrical isolation of hardware components at the circuit 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 [28]. When moving the same system to an integrated architecture it should be required that no new behaviors are introduced. This notion of *noninterference* [35] was originally introduced by security researchers.

Noninterference is an information flow policy which specifies that the actions of an entity[1] 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 formulation of noninterference that can be helpful for the verification process is based on the *determinism* of the observations [62]. 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 any direction.

A similar line of thought is applied in [77], 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 (RM) [52] 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

[1] An entity can can be a user, a task, a database, etc.

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. 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 [80]. 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* [79] 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 [67], 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 belonging to 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. 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. The response time analysis must therefore take into account faulty scenarios.

3.4 Mechanisms for Partitioning

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

3.4.1 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 [72, 42] spatial partitioning 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 unequal-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 some software to be managed).

Hardware Techniques

Hardware-based memory protection by means of a MMU is an established feature of desktop and server computers [42]. 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 Lookaside 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 [72]. 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 [31] 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 [5, 6]. 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 [32] and ARM's 940T [10] 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 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 a fault 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* [70]. 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 language-independent.

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 (e.g. sequential instruction fetches starting in a valid point, constant pointers, etc.). However, the

performance of this baseline method, also evaluated in [70], 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 [45]. 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, 45].

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 [49], 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, mandatory pointer initialization, etc.) 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 con-

ceptually be used as a tool which checks C programs that are then compiled and linked with standard C compilers.

Safe-C [11] and CCured [24] 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 [11] 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 [70] 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%. Similarly to Safe-C, 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 erro-

neous 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 intuitively 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 [34], which are able to tolerate transient SEUs. Consequently, spatial partitioning through hardware can be extended to handle hardware faults. Nonetheless, this approach implies moving away from COTS hardware.

3.4.2 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, I/O devices, etc.) 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 that the software in one partition does not 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.

The simplest answer to temporal partitioning is to use well known scheduling algorithms such as RM and EDF scheduling [52]. In [59] 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 [65].

However, the existing models of partitioning (discussed in Section 3.3) 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 RM or EDF to schedule partitions as long as there are mechanisms to ensure that a task cannot execute past its deadline (e.g. watchdog timers). When the deadlines are *enforced* the only timeliness issues arise with concurrent accesses to data items. This issue is also solvable through *concurrency control* techniques [19].

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 [51] 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 (RM) or dynamic (EDF) priority scheduling.

The drawbacks of this approach are twofold. First, the WCET analysis of tasks can be overly pessimistic. In a scenario where external interrupts are being used to serve, for instance, 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 are occurring. Second, the maximum frequency of execution of any task is determined by the frequency of execution of the partition. Clearly, context switching times between partitions make it impractical to give partitions the illusion of a dedicated (slower) processor by using, for instance a cycle frequency of 1KHz. This creates a limitation in managing tasks with disparate frequencies.

An approach which can be useful for temporal partitioning under faulty conditions is described in [69]. Along with the usage of an MMU for spatial partitioning, an imprecise computation model is proposed for temporal partitioning. Whenever the execution of a task exceeds a certain time limit (before the actual deadline), a backup routine is started. The backup routine will produce an imprecise result during the remaining time before the deadline. This approach may present some advantages if imprecise results can be acceptable. Nonetheless, this approach is an extension to partitioning methods rather than a separate temporal partitioning technique.

3.5 Guidelines for Integrated Architectures

This section provides a set of qualitative guidelines for building integrated architectures, based on the analysis provided in this chapter. These guidelines are derived from the quantitative assessment discussed in Section 3.2 and the available partitioning mechanisms surveyed in Section 3.4.

- 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 conclude 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.

- Partitioning mechanisms provide limited or no protection against hardware failures. Thus, structural hardware redundancy is required to protect the system against hardware failures.

- 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 toolchain (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 the certifying authorities that the same object code contains distinct criticality levels for the application and for the spatial partitioning mechanism.

3.6 Discussion and Conclusion

This chapter presented an analysis of robust partitioning methods. We discussed the requirements for partitioning and the existing mechanisms to implement partitioned systems. Furthermore, we analyzed the development effort necessary to ensure that integrated and federated architectures are equally dependable.

We conclude that different functions can be integrated into the same hardware platform without robust partitioning. However, without partitioning, the development effort for each individual function must be increased. Furthermore, whenever the system is upgraded there is a need for re-certification. On the other hand, with robust partitioning (100% effective), software can be developed and certified without taking into account that other tasks are sharing the same processing node.

Regarding hardware failures, the reliability of federated architectures decreases with the number of processors. However, integrated architectures are not beneficial in all scenarios. It is only when the number of processors used in a federated architecture exceeds a certain threshold that an integrated architecture would provide better reliability. Our analysis shows that this threshold is around 7 processors for 1-resilient systems.

We discuss the requirements for partitioning in the light of computer security concepts. When specifying the integrity requirements of partitions we find the notion of declassification to be less restrictive and therefore more useful than pure noninterference policies.

We survey the existing spatial partitioning mechanisms and come to the conclusion that hardware mechanisms have clear advantages over software mechanisms. Furthermore, the hardware support for advanced memory management is gradually being introduced in embedded microcontrollers.

On the other hand, software mechanisms have some advantages (e.g. unlimited memory segments of arbitrary sizes) that may be useful for some applications. It should be noted that the overhead of some software mechanisms can be reduced by eliminating the runtime checks for instructions that read data from memory. Our primary concern is the integrity of partitions. Thus, we only need to insert runtime checks for write instructions.

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. This is therefore an architecture-dependent choice. However, there are several advantages in using event-driven scheduling. This type of scheduling would therefore be beneficial in assuring the response time requirements of partitioned systems.

CHAPTER 4

Distributed Redundancy Management

This chapter addresses the problem of redundancy management in distributed systems. We propose a group membership protocol that achieves consensus on the operational state of all nodes in a system using time-triggered communication. The protocol ensures agreement on node departure under faulty conditions; the number of near-coincident failures that can be tolerated is configurable at design time, offering a trade-off between reliability and communication overhead. Additionally, the protocol provides agreement on reintegration of restarted nodes.

4.1 Introduction

Recovery from processing node failures requires distributed mechanisms that support *diagnosis*, *isolation*, *reconfiguration* and *reinitialization* of nodes. These four activities are implemented at the system layer by distributed redundancy management protocols. Such protocols ensure that redundancy management is coordinated, i.e. working nodes have a consensus on the location of the fault – diagnosis –, on the components that should be excluded – isolation – and on the reconfiguration that must take place to handle faults correctly.

Group membership is often a cornerstone of the design of redundancy management mechanisms for fault-tolerant distributed systems. The entities, or nodes, which constitute such systems, must cooperate in providing a correct service to its user(s), even in the presence of faults. A correct behavior of the system requires that the non-faulty nodes have access to timely and consistent information about the operational state of all nodes. The algorithms devised to maintain this information, preferably in a decentralized manner, are usually known as processor-group membership protocols.

This chapter proposes a processor-group membership protocol appropriate for hard real-time systems which utilize network standards such as TTCAN [33] or FlexRay [17]. Such systems impose hard deadlines for achieving consensus on membership changes, while they offer limited bandwidth for the implementation of the membership services.

The numerous solutions proposed in the past reflect the diversity of situations where membership agreement services are of use. In this thesis we consider group membership for systems relying on synchronous communication, where messages are transmitted within a known amount of time and processing nodes have a global notion of time. The membership problem in such systems was first described in [25]. Specially tailored solutions, for different applications, were subsequently devised and proposed in the literature [47], [46], [23] and [61]. The problem has also been widely studied in the context of asynchronous systems, which do not observe the synchrony hypothesis (e.g. [60], [30], [56] and [29]). The timed asynchronous model [26] combines these two classes of systems by assuming that all services, including membership, are timed although interprocess communication time is unbounded.

The protocol described in this chapter assumes that each node in the system periodically sends messages that reach all other nodes under fault-free circumstances. It piggybacks all protocol-specific information on such periodic messages. The protocol allows fault-free nodes to reach agreement on the operational state of all nodes in the presence of node failures as well as network failures (lost or corrupted messages).

It is based on the principle that each message sent by a node which is present in the membership should be acknowledged by k other nodes in a system of n nodes. The value k can be set to any number between 2 and $n-1$. Each node must acknowledge k messages, and thus each message must contain k acknowledgement bits. The number of acknowledgement bits determines the number of simultaneous failures that can be tolerated during n consecutive transmission slots. Therefore, our protocol contrasts to most solutions intended for hard real-time systems in its flexibility to configure the trade-off between reliability and communication overhead, at design time.

Agreement on departure is guaranteed if no more than $f=k-1$ failures occur during n consecutive transmission slots, while at most one node can attempt reintegration into the group of operational nodes (the membership) per communication round. In our scheme, listening on the network is enough for a restarted node to recover the state of the membership, before attempting reintegration. We introduce an additional service for agreement on round number to ensure that nodes send their reintegration requests in a known order.

In addition to the protocol for processor-group membership agreement, this chapter presents the outline of an extension which keeps track of application process failures. This feature is intended for distributed real-time systems where the processing nodes offer effective fault-containment between different application pro-

cesses executed on the node. In such systems the membership service should preferably handle both node failures and application process failures.

4.2 System Model

We consider a set of processing nodes linked by a time-triggered broadcast network. We assume that the network has either as bus or star topology. Processors (nodes) have their clocks tightly synchronized and send messages at predefined time-slots according to a time-triggered cyclic schedule. Each cycle in the schedule corresponds to a *communication round*.

Each node has a single dedicated transmission slot on the network which it uses to broadcast its messages in every communication round. Processing nodes are assumed to be *fail-silent*, i.e. either correct results or no results are broadcasted, 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). If a node does not receive a message during a slot, then a failure has occurred. It could be a *failure of the sending node*, a *failure of the receiving node*, a *network failure* or a combination of these.

In our system model, 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 classes according their persistence, *permanent* or *transient*, and whether their impact on the system is *consistent* or *inconsistent*. We assume that a *transient failure* only affects one 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. A *permanent failure* will remain in the system until it is repaired, and may affect one node, or 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 group membership protocol.)

We classify a failure as consistent if all receiving nodes fail to receive a message correctly, since all the receiving nodes in this case will have a consistent view of the system state. On the other hand, an inconsistent failure occurs if one node fails to receive a message correctly, since that node then has a different view of the system compared to all other nodes.

We assume that a situation where some nodes receive a message correctly, and two or more nodes receive the message incorrectly, can only occur in presence of

multiple failures. We deal with multiple failures by configuring the membership protocol appropriately. Table 4.1 shows how the different types of component failures are mapped into the different failure classes.

Component Failures	Transient	Permanent
Sending Node	1. Transient Consistent	3. Permanent Consistent
Outgoing Link		
Network (Consistent)		
Network (Inconsistent)	2. Transient Inconsistent	4. Permanent Inconsistent
Incoming Link		
Receiving Node		

Table 4.1: Mapping of component failures to failure classes.

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[1] failures the protocol can cope with. As previously explained, the number of simultaneous failures under which the protocol reaches agreement on the membership depends on the number of acknowledgement bits used. An example of a similar failure model, used in conjunction with processor-group membership protocols for time-triggered systems, can be found in [47].

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 consistent failures, i.e. class 1 or class 3, depending on their persistence.

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 inconsistent failures (class 2). When the incoming link of a single node becomes permanently faulty we classify it as a permanent inconsistent failure (class 4).

Modeling communication failures must consider the topology of the network [66]. 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 network failure model is supported by the following analysis. We assume that applying structural redundancy will allow single transient failures 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

[1]By simultaneous failures we mean failures that occur during n consecutive transmission slots, where n is the number of nodes in the system.

message while other nodes receive a corrupted version, is negligible. If this assumption does not hold (e.g. slightly-off-specification failures are a concern), then the number of acknowledgement bits must be increased to a suitable configuration.

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 failure of the transient consistent class. On the other hand, if only one receiving node misses a single message, a failure of the transient inconsistent class has occurred. When the network failure is permanent then either one node is not able to send messages (permanent consistent failure) or one node is not able to receive messages (permanent inconsistent failure).

4.3 Protocol Specification

This section provides the specification of the membership agreement protocol. For simplicity, we divide the specification into four mini-protocols which, combined, provide agreement on departure of failed nodes and reintegration of recovered nodes. The four mini-protocols are:

- *Agreement on departure*, which handles the departure of failed nodes from the membership set.
- *Agreement on reintegration*, that specifies how nodes are reintegrated into the membership set.
- *Agreement on reintegration ordering*, that supplies the "current" round number to restarting nodes, which must follow an order of reintegration.
- *Recovery of the membership state*, which provides the means for a restarting node to retrieve the "current" membership state.

The membership protocol relies on the periodic messages sent by each node to piggyback a *sequence of acknowledgements*. Each node will append k acknowledgement bits to its message, confirming (or refuting) the reception of each of the previous k messages *from the nodes in the membership*. A *reintegration flag* is also appended to each message to allow reinitialized nodes to reintegrate the membership. The periodic messages therefore respect the following format:

$$message = \langle data,\ ack_1,\ \cdots,\ ack_k,\ r\text{-}flag \rangle$$

The *data* field contains the payload of the message, i.e. the normal information sent by nodes on the membership. The *ack* fields, as well as the *r-flag*, are

Booleans that can be represented by a single bit. The four mini-protocols describe how the *ack* flags and the *r-flag* are set in response to certain events. The events that are reported to a node are *message receptions*, *message losses* and *new round* notifications.

4.3.1 Definitions

Let N denote the totally ordered set of processing nodes N_1, N_2, ..., N_n. The totality condition is intuitively satisfied by the order in which nodes broadcast in every round, imposed by the network schedule. We use n to indicate the number of nodes in the system. We exclude systems with $n \leq 2$ processing nodes, where membership agreement is irrelevant.

Each node has access to a global view of time, supplied by a clock synchronization mechanism. In what concerns the membership problem, the required granularity of time is a *transmission slot*. Let s denote the slot progression, starting at 1 and incrementing to infinity. Processing nodes are not required to know the value of s (the absolute slot number since start-up). The membership protocol only requires nodes to hold the slot number *within the communication round*. Hence, each node holds a local variable *currentSlot* which is synchronized among all nodes. Additionally, nodes have a *currentRound* variable to count communication rounds (sequences of n slots starting with the slot of node N_1). Each node holds also a local variable *membershipArray*.

- The integer variable *currentSlot* represents the progression of time (in slot units). This variable is a cyclic counter of slots, starting at 1 and incrementing to n. It indicates which of the processing nodes N_1, N_2, ..., N_n is the sending node in each transmission slot.

- The integer variable *currentRound* holds the number of the current round within the reintegration cycle. This variable is used for agreement on reintegration, where it counts the progression of rounds in a cyclic manner, starting at 1 and incrementing to $2n+3$, as will be described in Section 4.3.3.

- The variable *membershipArray* represents the local view of the membership set. It is a Boolean array with n elements – element i specifies whether or not N_i is in the membership.

We assume the existence, at any time, of at least two working nodes, which agree on the membership and are not subject to failures which would break the agreement. Booting into such a configuration relies on the existence of a reliable start-up mechanism [22]. Periods of the execution where 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.

In our protocol a node is said to be *sponsoring* node N_j if the acknowledgement of the last message from N_j is attached to its message. Only nodes in the membership are involved in sponsoring relations. Under normal conditions each node will have k sponsors (and will be sponsoring k nodes). The value k stipulates how resilient the protocol will be to near-coincident failures, at the expense of communication overhead. If, in a given slot s, the membership set contains n_s nodes and $n_s \leq k$, *a node is not expected to sponsor itself.* Hence, when $n_s \leq k$, each node will be sponsoring its $k_s = n_s - 1$ membership predecessors; otherwise, $k_s = k$.

We define the predicate *sponsor(i, j)* as *true* if and only if N_i is sponsoring N_j. According to the definition of sponsor, N_i is sponsoring its k_s predecessors (in the order of broadcast) which are contained in the membership. Additionally, we define the predicate *lastSponsor(i, 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. This predicate essentially states that N_i is the last node to acknowledge the previous message from N_j.

4.3.2 Agreement on Departure

Initially, the membership set contains all active processing nodes. We assume that the start-up mechanism, responsible for establishing a sufficient level of network-wide synchronization, also supplies the set of initially active nodes to the membership service.

A given node N_j will be removed from the *local membership view* of N_i if and only if N_i does not receive a message from N_j and no positive acknowledgement for that message is received from any of the *sponsors* of N_j. Node N_i will remove N_j from the membership immediately after the sending slot of the last sponsor of N_j. The pseudo-code for this mini-protocol is shown in Algorithm 1.

The two events that trigger the mini-protocol for agreement on departure are *message receptions* and *message losses*. Logically, these two events are mutually exclusive in any given slot, i.e. a node will either receive or lose a message in a communication slot.

- Line 4 – A message is received and stored in the *msg* variable. The message sender is the owner of the current slot. Thus, the *currentSlot* variable identifies the sending node as well.

- Line 13 – The slot time elapses with no message being received. A message loss event is reported and the owner of the current slot is specified by the *currentSlot* variable.

Each node holds a Boolean array *presentArray* to store the temporary information on the presence of other nodes (line 2 of the pseudo-code). Initially, all the elements of *presentArray* are set to *true*, even if some nodes are not in the membership. This simplification is allowed since the only elements of *presentArray* which

```
1  membershipArray: Local view of the membership.
2  presentArray: Temporary information on the presence of other nodes.
3  ackStack: Circular stack to store this node's acknowledgements.

4  on Message Received:
5  msg: The received message.
6  currentSlot: The current slot number (also the sending node ID).
7  if membershipArray[currentSlot] = true then /* Node in membership */
8  |  presentArray[currentSlot] := true;
9  |  processIncomingAcks(msg, currentSlot);
10 |  push(ackStack, true);
11 |  decisionStep(currentSlot);
12 end

13 on Message Lost:
14 currentSlot: The current slot number (also the sending node ID).
15 if membershipArray[currentSlot] = true then /* Node in membership */
16 |  presentArray[currentSlot] := false;
17 |  push(ackStack, false);
18 |  decisionStep(currentSlot);
19 end
```

Algorithm 1: Pseudo-code for the agreement on departure.

are used are those that correspond to nodes in the membership; such nodes must be considered present at start-up. The *presentArray* is updated at three different locations in the pseudo-code:

- Line 8 – A message from a node in the membership is received. That node is marked as present by setting its flag to *true* in the *presentArray*.
- Line 16 – A message from a node in the membership is lost. That node is temporarily marked as absent by setting its flag to false in the *presentArray*.
- Line 9 – The `processIncomingAcks` method updates the *presentArray* according to the acknowledgement flags contained in the received message. Each message contains k flags that confirm or refute the reception of the messages from the sponsored nodes. The `processIncomingAcks` sets to *true* the *presentArray* elements that are positively acknowledged by the message (by processing the acknowledgement flags).

At each slot the sending node (N_i such that $i = currentSlot$) sends a message that contains k acknowledgement flags. When a node is about to send its message, it will pop k flags from the *ackStack*. These flags are the message's acknowledgement flags. At a node's sending slot, the *ackStack* contains the k flags that refer to the predecessor nodes in the membership. A node therefore builds the acknowledgement field of its message simply by copying the contents of *ackStack*.

- Line 3 – The *ackStack* is a circular stack which can hold up to k Booleans. A circular stack is a LIFO structure (last in, first out) that discards surplus elements – instead of overflowing. Thus, it contains the k "freshest" elements that were pushed to the stack. The *ackStack* is initialized with k elements set to *true*.
- Line 10 – A message from a node in the membership is received. That node's presence will be acknowledged later in the communication schedule (if it is a sponsored node).
- Line 17 – A message from a node in the membership is lost. That node will be reported as absent later in the communication schedule (if it is a sponsored node).

Synchronous Decision Step

When the slot of a node N_i, present in the membership, comes to an end, all nodes execute the synchronous decision-making step. They do so by calling the `decisionStep` method. When the `decisionStep(currentSlot)` call is made, node N_j, such that *lastSponsor(currentSlot, j) = true*, will be removed from the local view of the membership if and only if *presentArray[j] = false*. Node N_j is removed by executing *membershipArray[j] := false*.

We emphasize that the `decisionStep` method must be called at the end of each slot owned by a node in the membership. Pseudo-code lines 11 and 18 show the call when a message is received or lost, respectively. Additionally, the node which has sent a message must also execute the decision step – that node is also the last sponsor for some other node.

Self Diagnosis of Faulty Nodes

During the synchronous decision step, nodes verify if at least one message was received from their k predecessors in the membership. If not, the only conclusion is that the node has suffered a permanent failure which does not allow the reception of messages. We assume that no more than f failures occur during n consecutive slots, where $k{=}f{+}1$. Thus, if a node misses k messages in a row, then it must self diagnose a permanent inconsistent failure. With respect to the pseudo-code in Algorithm 1, the first condition for self diagnosis is:

- When the `decisionStep(currentSlot)` method is called, each node checks if the *ackStack* contains at least one element set to *true*. If it doesn't, then it diagnoses itself as faulty.

A second condition is provided for self diagnosis of nodes that have failed to send their message. These nodes will be removed from the membership by all other membership nodes. Thus, the second condition for self diagnosis is:

- When the `decisionStep(currentSlot)` method is called, each node checks if it has received at least one positive acknowledgement for *its own message* by one of its sponsors. If it hasn't, then it diagnoses itself as faulty.

Nodes that have diagnosed themselves as faulty should send no more messages. The remaining fault-free nodes will remove the faulty ones (once their message is lost and their last sponsor broadcasts). The sensible procedure would be to reinitialize that node and attempt reintegration later in time.

4.3.3 Agreement on Reintegration Ordering

In our protocol only one node can reintegrate the membership per communication round. We therefore introduce a service for agreement on round number to ensure that nodes send their reintegration requests in a known order. The main goal is to prevent nodes from attempting simultaneous reintegration in the membership.

Agreement on the round number is kept by the nodes in the membership as the communication schedule progresses. A failed node is, however, unable to determine the current round number unless fault-free nodes explicitly signal it.

We assume that the synchronization mechanisms are able to detect the current slot number. Examples of such mechanisms are the introduction of message IDs which are unique, to identify the message sender; using unique message lengths that

act as implicit message IDs; and having the network (or some specific node) signal new rounds with some special type of message or delimiter.

The protocol supplies the round number to restarting nodes through a simple algorithm which uses the reintegration flag (*r-flag*) of nodes in the membership. This service does not impose any additional overhead since the *r-flag* is required anyhow to signal successful reintegrations (see Section 4.3.5).

Ensuring an order of reintegration only requires nodes to agree on the value of a cyclic counter of rounds. Thus, the absolute round number since startup is not required. The cyclic round counter determines which node can reintegrate in a given round. For this purpose we define the *reintegration cycle* as a sequence of rounds where every node has one dedicated reintegration round. The length of every such reintegration cycle is $2n+3$ rounds, where n is the number of nodes. The round counter is therefore incremented by 1 when a new round begins; if the counter reaches $2n+3$, then the next value will be 1 (a new cycle begins). Figure 4.1 shows the state of the *r-flag* (sent by a node in the membership) during a reintegration cycle.

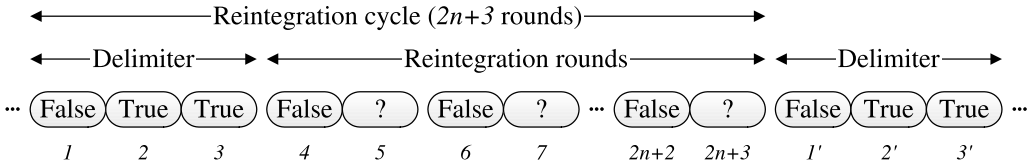

Figure 4.1: Round number signaling by a node in the membership, using the *r-flag*.

During the first 3 rounds of a reintegration cycle all nodes in the membership use their *r-flag* to signal the *delimiter pattern*. When the counter equals 1, nodes set their *r-flag* to *false*; when the counter equals 2 or 3, nodes set their *r-flag* to *true*. Two such delimiters are shown in Figure 4.1. The first delimiter (rounds 1, 2 and 3) signals the start of a reintegration cycle. The second delimiter (rounds $1'$, $2'$ and $3'$) comes after $2n+3$ rounds to signal the start of the subsequent reintegration cycle.

The $2n$ rounds between two consecutive delimiters constitute the *reintegration rounds*, where nodes can reintegrate the membership. On even reintegration rounds the *r-flag* is always set to *false*; on odd reintegration rounds the state of *r-flag* (illustrated with question marks) is set to either *true* or *false* according to the specification in Section 4.3.5.

This method guarantees that the *r-flag* is set to *false* during, at least, one out of any two consecutive rounds. The only exception to this occurs intentionally during the delimiter pattern, where the *r-flag* is set to *true* in rounds 2 and 3. Any restarting node synchronizes its round counter with the nodes in the membership by listening to their messages on the network. When the *r-flags* are observed to be *true* in two consecutive rounds, a restarting node sets its round counter to 3 (the

end of a delimiter pattern). We note that receiving one message where the *r-flag* is *true* in each of those two rounds is enough to detect a delimiter pattern. The subsequent round will therefore be round number 4 (the first reintegration round).

According to our failure model, the number of failures in a communication round is assumed to be less than the number of active nodes in the membership. This implies that a restarting node (free from permanent incoming link failures) receives at least one message per round. Thus, such a node is guaranteed to obtain the round number after listening for, at most, $2n+4$ rounds. The worst case occurs when the node starts listening on round 3; in this case the node has to wait until the reintegration cycle is complete to observe a complete delimiter pattern.

4.3.4 Recovery of the Membership State

When a previously failed node is able to restart, after a downtime period, it will need to recover the state of the membership before broadcasting any messages. We assume that fundamental data such as the communication schedule will survive the crash. The node only needs to recover the state of the membership, which is dynamic.

There is no need for explicit broadcast of the membership state by active nodes. A restarting node listens to the incoming messages and detects which nodes are in the membership. We propose a simple scheme which is fault-intolerant, i.e. a restarting node may obtain an incorrect membership state if failures occur. However, such a node will be denied reintegration once its reintegration request is validated by membership nodes.

Consider a restarting node that intends to send its reintegration request on round r. To recover the membership state, the node should start listening to the incoming messages on round $r-2$. Starting with its membership set empty, the node adds other nodes to the local view when their message is received. If no failures occur, the restarting node has the correct membership view once round $r-2$ is completed.

At the beginning of round $r-1$ the node starts executing the mini-protocol for agreement on departure. Once that communication round is completed, the restarting node may attempt reintegration during round r, at its transmission slot. This method to recover the membership state is clearly not fault-tolerant. On the other hand, the benefit of this approach is its simplicity. We emphasize that membership agreement is not compromised by this approach – a node will not be accepted in the membership unless its view is confirmed by the nodes in the membership.

4.3.5 Agreement on Reintegration

Agreement on reintegration is, *per se*, a simple procedure in which a node sends a reintegration request and membership nodes acknowledge its reception, adding

the node to their local membership view. However, before attempting reintegration, a restarting node must (i) obtain the current round number and (ii) obtain the set of nodes in the membership. The first requirement is satisfied by the algorithm in Section 4.3.3 and the second requirement is satisfied by the algorithm in Section 4.3.4.

Reintegration Requests

The procedure for agreement on reintegration starts when node N_r obtains both the current round number and the membership state. During the reintegration cycle, described in Section 4.3.3, node N_r has two dedicated reintegration rounds:

- Round $2r+2$ – Node N_r sends the reintegration request in its communication slot. We note that no other node will send a reintegration request during this round since node IDs are unique.

- Round $2r+3$ – All nodes in the membership acknowledge the reintegration request with their *r-flag*. Node N_r is reintegrated when this round is completed.

The first reintegration round comes immediately after the delimiter pattern, as shown in Figure 4.1. If node N_1 were to attempt reintegration, it should send its reintegration request in round 4 ($2\times1+2$), node N_2 in round 6 ($2\times2+2$), etc. We note that node N_r should not send any messages during round $2r+3$. This node will only be integrated in the membership once round $2r+3$ is completed. Thus, allowing a node to send "normal" messages before it is included in the membership would only increase the complexity of the protocol.

The reintegration request of node N_r is a special type of message. Reintegration requests do not include the *normal* information sent by nodes in the membership. Instead, the message should include the membership view of node N_r, so that all other nodes are able to *confirm that a successful reintegration is taking place*. The concern here is that failures during restart would lead to a node being reintegrated *without agreeing on the membership state*. Such an event would break the membership agreement. We therefore require nodes to send their complete membership view (instead of the normal payload) in reintegration requests:

$$message_r = \langle viewOfNode_1, viewOfNode_2, \cdots, viewOfNode_n \rangle$$

Reintegration Acknowledgements

A given node N_r will be reintegrated into the membership if it sends a correct reintegration request in round $2r+2$. Node N_r is the only node allowed to attempt reintegration during that round. Since node N_r is not in the membership, all receiving nodes perceive its message as a reintegration request. Thus, nodes in the

membership can distinguish normal messages from reintegration requests, without any additional message fields.

The mini-protocol for agreement on reintegration ensures that node reintegration is fault-tolerant, i.e. membership agreement is ensured under the failure hypothesis. To achieve this, all receiving nodes compare the membership view supplied in the reintegration request with their own membership view. If the membership views are equal, nodes acknowledge the reintegration by setting their *r-flag* to *true* in the round that follows; if they differ, the *r-flag* remains *false*. The pseudo code for this mini-protocol is provided in Algorithm 2.

The two events that trigger the mini-protocol for agreement on reintegration are *message receptions* and *new round* notifications:

- Line 6 – A message is received and stored in the *msg* variable. The message sender is the owner of the current slot. Thus, the *currentSlot* variable identifies the sending node as well.

- Line 15 – This event signals that the current round was completed and a new round will start.

The variable *currentRound* maintains the value of the round number within the reintegration cycle (pseudo-code line number 2). It should be initialized to $2n+3$ and a *new round* event should be reported at start-up, to begin the execution of the mini-protocol for agreement on reintegration.

Each node holds three Boolean variables (declared in the pseudo-code lines 3 through 5) which the node uses to respond to the membership events. The *reintegrateOnNewRound* flag indicates whether a node should be reintegrated upon completion of the current round; the *nextRoundRFlag* holds the value of the node's *r-flag* for the next round; the *currentRoundRFlag* holds the value of the node's *r-flag* for the current round's message (see pseudo-code line 24).

Upon the reception of a reintegration request, membership nodes verify if the membership view of the reintegrating node is correct. If so, the reintegration is acknowledged during the following round. When a node is about to send a message, it sets its *r-flag* to the value of the *currentRoundRFlag* variable.

- Line 12 – A reintegration request is received. The `reintegrationRequestIsCorrect()` method verifies whether or not the reintegrating node has the correct view of the membership. If so, its reintegration will be acknowledged during the following round.

It should be noted that the reintegration request can be missed by some nodes in the membership. Such nodes will be aware that a reintegration is taking place once they receive the *r-flags* transmitted by other nodes in the membership:

- Line 10 – A message from a node in the membership is received. If the piggybacked *r-flag* is *true* then it can be either a *reintegration acknowledgement*

```
1  membershipArray: Local view of the membership.
2  currentRound: Current round number (cyclic counter).
3  reintegrateOnNewRound: Whether a node will be reintegrated next round.
4  nextRoundRFlag: Status of this node's r-flag during the next round.
5  currentRoundRFlag: Status of this node's r-flag during the current round.

6  on Message Received:
7  msg: The received message.
8  currentSlot: The current slot number (also the sending node ID).
9  if membershipArray[currentSlot] = true then /* Node in membership */
10 |  if msg.r-flag = true and currentRound > 3 then
11 |  |  reintegrateOnNewRound := true;
12 else if reintegrationRequestIsCorrect(msg) then
13 |  nextRoundRFlag := true;    /* Acknowledge during next round */
14 end

15 on New Round:
16 if reintegrateOnNewRound = true then
17 |  reintegratingNode := (currentRound–3)/2;
18 |  membershipArray[reintegratingNode] := true;      /* Reintegrate */
19 |  updateAckStack(reintegratingNode);
20 |  reintegrateOnNewRound := false;
21 end
22 if nextRoundRFlag = true and currentRound > 3 then
23 |  reintegrateOnNewRound := true;
24 currentRoundRFlag := nextRoundRFlag;
25 currentRound := currentRound + 1;  /* Update the round counter */
26 if currentRound > 2n+3 then currentRound := 1;
27 if currentRound = 1 or currentRound = 2 then  /* Round signaling */
28 |  nextRoundRFlag := true;          /* r-flag in rounds 2 and 3 */
29 else
30 |  nextRoundRFlag := false;
```

Algorithm 2: Pseudo-code for the agreement on reintegration.

or a *delimiter pattern*; if the current round number is greater than 3 then it is a reintegration acknowledgement and a node will be reintegrated on new round.

There are two different ways in which a node becomes aware of a reintegration: (i) when it receives a reintegration request, acknowledges the reintegration and sets its *reintegrateOnNewRound* variable to *true* (later, in pseudo-code line 23) and (ii) when a reintegration acknowledgement is received (pseudo-code line 11). In both cases the node will be reintegrated in the membership:

- Line 17 – A new round event was reported and the *reintegrateOnNewRound* variable is set to *true*. The synchronous decision-making step about the reintegration of a node is executed. The ID of the node which is reintegrated is obtained from the *currentRound* variable – the order of reintegration specifies, with no ambiguity, which node reintegrates in each round.

The reintegration is processed by adding the node to the *membershipArray* (line 18) and by updating the *ackStack*. After a node is reintegrated into the membership the nodes must refresh their *ackStack* to insert the acknowledgement flag for the reintegrated node (in the correct position). This is done in pseudo-code line 19 by calling the `updateAckStack` method. The nodes which become sponsors for the reintegrated node will send a positive acknowledgement bit for that node in their following messages.

- Lines 25 to 30 – The round counter is updated to start a new round. In rounds 1, 2 and 3 the *r-flag* signals the delimiter pattern, according to the specification in Section 4.3.3.

4.4 Discussion

The mini-protocol for agreement on departure requires k=2 sponsors per node in its minimum configuration. Setting k=1 does not allow the agreement to be maintained after a transient receiving failure of a sponsor, since the sponsor itself will erroneously remove its sponsored node from the local membership view.

Choosing the number k of sponsors per node defines the balance between fault tolerance and communication overhead. It is possible to identify the failure conditions which maintain the agreement by analyzing the directed graphs of sponsoring relations (sponsor-graphs). Figure 4.2 depicts such a sponsor-graph for an example of a 6-node system with $k = 2$.

A sponsor-graph is initialized with each vertex representing a node in the system. Initially, each directed edge from a given node N_i to N_j denotes that $sponsor(i, j) = true$. This graph evolves at every synchronous step by removing an edge whenever the corresponding acknowledgement is missed by at least one node. When a

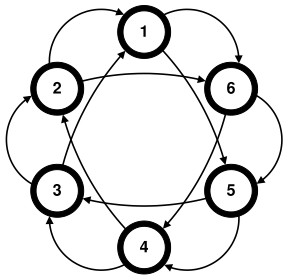

Figure 4.2: Sponsor-graph of a 6-node system ($k = 2$).

decision-making slot comes and N_i is found to be faulty by all other nodes, then the sponsor-graph will be updated by removing the edges directed to vertex i and adding edges directed to the nodes which then obtained new sponsors (the previous sponsors of N_i are now sponsoring some other nodes).

Agreement will be maintained when all fault-free nodes receive at least one acknowledgement of any given message. Therefore the system is able to maintain the agreement if there is always at least one edge directed to each vertex.

Any failure which is not a permanent inconsistent failure will remove at most one edge directed to any given node. Therefore, if f such failures are to be tolerated in each communication round, the number of initial edges (i.e. the number of sponsors), should be $k = f + 1$. A permanent inconsistent failure will become apparent as a permanent consistent failure when the failed node diagnoses itself as failed and stops sending messages.

Agreement on the "real" state of the membership is reached k slots after the most recent failure. Consequently, at this point the system has redirected the sponsoring edges into a safe configuration again.

For any value of k the protocol maintains consistent views of the membership even when multiple permanent node failures occur, if this is the only type of failure to occur in the period between the first node failure and k slots after the last node failure. This type of failure is consistently perceived by all fault-free nodes. Therefore, all fault-free nodes will miss the messages from the permanently failed nodes and no positive acknowledgement on these messages will be sent. Agreement on the departure of the last node to suffer such a failure is reached k slots later.

The graph in Figure 4.2 exemplifies a sponsor-graph where no faults have occurred. When a failure of sending node 1 occurs (either transient or permanent) the graph will change into the one illustrated in Figure 4.3. Two slots later, after node 3 (the last sponsor of node 1) broadcasts, the system is "reconfigured" into the graph in Figure 4.4. In the critical moment, before reconfiguration, there are edges directed to each node, therefore agreement is ensured.

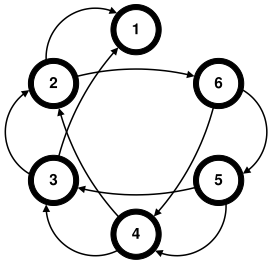

Figure 4.3: Sponsor-graph immediately after the failure of sending node 1.

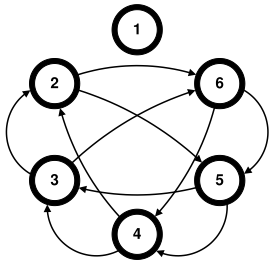

Figure 4.4: Sponsor-graph 2 slots after the failure of node 1.

4.4.1 Further Considerations

A node is expected to broadcast not only its message but also the information necessary for membership agreement. For practical purposes the total order in the membership set can be adapted in order to permit a "reaction time" from processing nodes. If a node is to start the broadcast immediately after the previous slot ends, there may be insufficient time to evaluate the correctness of the previous message and to execute the membership protocol. At the expense of detection latency, each node N_i can be set to be sponsoring the nodes starting from N_{i-2} instead of sponsoring its immediate predecessor.

4.5 Prototype Implementation

We have implemented the membership protocol in a prototype of a distributed real-time system. The real-time network prototype [40] is based on COTS Ethernet hardware, programmed to schedule messages according to the Time Division Multiple Access (TDMA) method. Figure 4.5 shows the 6 processing nodes included in our experimental setup.

The computer nodes shown in Figure 4.5 are Phytec's phyCORE-MPC565 [58] development boards. Each contains a Motorola[2] 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.

The two boards on the upper-left corner of Figure 4.5 are expanded with a custom board. We developed these expansion boards in order to have a 7-segment display (for showing the number of active nodes in the membership) and to output the internal clock of the nodes. We connect these clock outputs to an oscilloscope in order to measure their synchronization.

The prototype 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

[2]Since 2004, Motorola's semiconductor products are developed under the name Freescale Semiconductor, Inc.

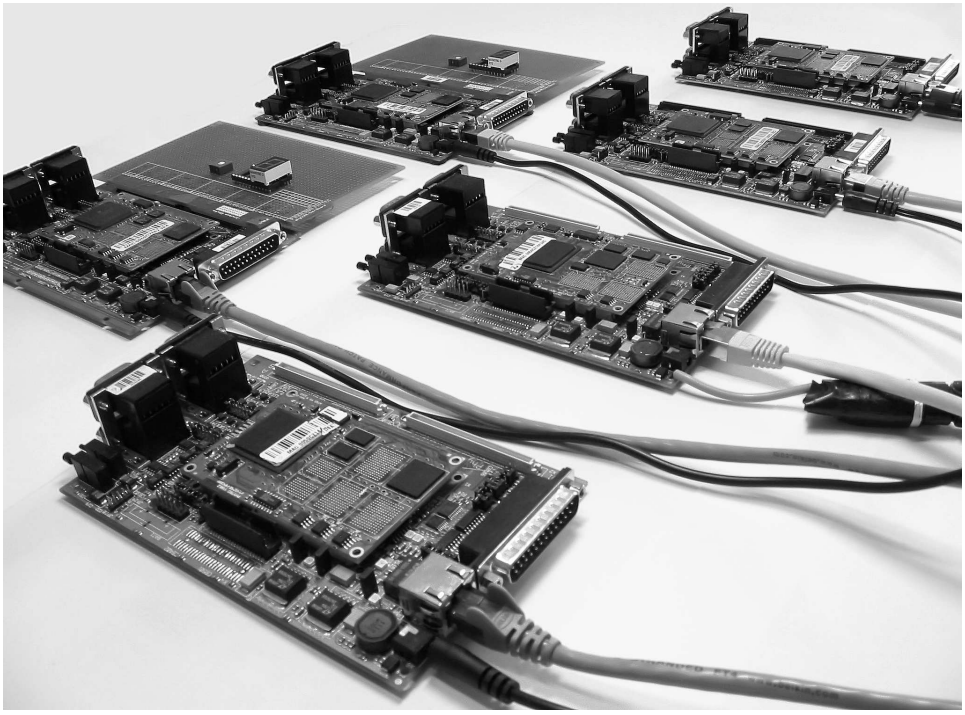

Figure 4.5: The real-time Ethernet prototype.

Mbit/s (standard 10BASE-T). To maintain the TDMA schedule we implemented the daisy-chain clock synchronization algorithm [53].

4.5.1 Network Configuration

The length of the Ethernet frames can vary between 64 and 1518 bytes. We used 64-byte packets for this set of 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.

The duration of a transmission slot was set to 400 μs (the lower bound for this parameter is $\sim$250 μs in our setup). Under these conditions the daisy-chain algorithm maintained the processor nodes synchronized within 3 μs. Table 4.2 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 *r-flag*).

Parameter	*Value*
Number of nodes	6
Packet size	64 bytes
Transmission slot	400 μs
Clock skew	< 3 μs
Reintegration cycle	36 ms
Communication round	2.4 ms

Table 4.2: Parameters of the real-time Ethernet prototype.

4.5.2 Network and Membership Performance

The nominal bandwidth of the network is 10 Mbit/s. However, real-time communication using TDMA must take into account the propagation delays and the clock skews in order to ensure that there are never two messages being transmitted at the same time. This is achieved by inserting guard times in between messages. Due to these guard times, we estimate that our network prototype 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. Since each frame reserves 18 bytes for the header and the CRC checksum, we have 920 Kbit/s of *useful bandwidth* available for payload data (which includes the membership information).

In our experiments, each message had 5 bits of piggybacked membership information. Messages were sent once every 400 μs. The bandwidth required by the membership service was therefore 12.5 Kbit/s. Since there were 920 Kbit/s of useful 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 useful bandwidth. Increasing the packet size would therefore reduce the membership's communication overhead significantly.

Using 4 sponsors in the membership protocol means that a departure will be detected by membership nodes 4 transmission slots after the message is lost. The latency for agreement on departure, counting the slot where the failure occurs, is therefore $(4 + 1) \times 400 = 2000$ μs. This and other important latency values are shown in Table 4.3. It should be noted that these are calculated (not measured) values. The worst case latencies for reintegration occur when node 6 wishes to be reintegrated and starts listening on round 3; the node has to wait $2 \times 6 + 4 = 16$ rounds for the next delimiter pattern and then 12 (2×6) rounds to be reintegrated.

A practical implementation of our protocol requires nodes to acknowledge their 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

Activity	*Latency*
Agreement on node departure	2 ms
Recovery of the round number (worst case)	38.4 ms
Fault-free reintegration from restart (worst case)	67.2 ms

Table 4.3: Node departure and node reintegration latencies.

on time.

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 an early estimation shows that the CPU overhead of the membership service is negligible.

4.6 Task-Group Membership Agreement

Chapter 3 of this thesis studies integrated architectures, where many tasks share a common hardware platform with relatively few but powerful processing elements. Such designs have a great potential to reduce both product and maintenance costs, and to improve system reliability, since they require fewer hardware components than designs where every function has their own dedicated microcontroller.

From a fault tolerance perspective there are clear differences between systems that do not provide partitioning at the node layer and those that do. In the former, the smallest unit of failure typically corresponds to an entire node, while in the latter the smallest unit of failure is an application process. Thus, for systems provided with partitioning and other mechanisms for achieving fault containment at the nodes, the membership service should preferably handle both node failures and application process failures.

So far, we have described our protocol in terms of a processor-group membership protocol. However, it can easily be extended to keep track of both node failures and task failures. Without giving a full protocol description, we will here outline how such an extension could be implemented. This extension is not included in the prototype implementation described in the previous section.

4.6.1 Multiple Sending Slots

We have assumed before that a node only broadcasts 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 – one message per task.

It would be possible to extend the processor-group membership protocol directly

to individual tasks. Each task could acknowledge the predecessing task in the network schedule. The creation of network schedules would only be restricted to prevent tasks on the same node from sponsoring each other (i.e. to prevent a node from sponsoring itself). However, this approach would consume a very large amount of bandwidth.

Instead we propose that, in each communication round, only the last message sent by each node should carry membership information. We define a *logical message* as the set of messages sent by a node during one round. Our proposal is that the acknowledgements sent with a logical message should refer to the other nodes' logical messages.

When a physical message is lost by a given node, that node will immediately consider that the corresponding logical message was lost. Nodes must therefore receive the entire set of physical messages in order to acknowledge the successful reception of the logical message.

4.6.2 Task-Group Membership through Fail-Reporting

We assume that a node equipped with node layer fault tolerance mechanisms can exhibit two different failure modes: *fail-silent* and *fail-reporting*. A fail-silent failure implies that the node does not produce any results at all. We assume that fail-silent failures only occur when the entire node has failed, for example, as a result of an operating system crash or a permanent hardware failure.

On the other hand, a fail-reporting failure occurs when a node cannot send a result because one of its tasks has failed. In that case, the operating system is still operational and can produce a *failure report message* that the node sends instead of the regular message. Such failure reports could of course be generated without the involvement of the processor-group membership protocol. However, we believe that there are many advantages in combining processor-group and task-group membership into a single service.

To achieve this for our protocol, we can simply add one extra bit to each message sent by a task. This fail-report bit would indicate 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-membership set of other nodes when they receive a failure report for that task. Task failures are this way *reported* to all nodes in the system. A task is reintegrated into the task-membership when nodes receive a regular message from that task, instead of the failure report. When a complete node fails (and is removed from the processor-group membership) all its tasks are removed from the task-membership set of other nodes.

This type of membership agreement is weaker than the processor-group membership agreement described earlier. When all nodes receive the failure report, there will be agreement on the departure 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 task-membership. 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 acknowledgment bits and the *r-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. It should be noted that this issue only concerns the messages that include membership information, i.e. the last message sent by a node in every round.

4.7 Related Research

The protocol specification allows a comparison with existing membership protocols. The TTP specification [48] includes a membership service which imposes no direct overhead on the network. This is achieved instead by placing the load of the failure detection on the CRC mechanisms and by requiring the membership state to be explicitly broadcasted in order to allow the reintegration of previously failed nodes. In our approach a node needs only to listen on the bus for the period of one round on the bus to recover the membership state.

Ref. [46] presents a solution which isolates TTP's membership protocol from the other elements in the protocol. By using a single acknowledgement bit it ensures membership agreement, under the single failure assumption, by prompt removal of failed nodes. Since the proposed scheme does not provide reintegration capabilities, prompt removal is a costly approach to tolerating transient failures. Nevertheless, this scheme requires only one overhead bit per message, whereas our approach imposes an overhead of at least two acknowledgement bits to tolerate any random failure.

In [37] a solution based on a variable number of sponsors is presented, but in a context unrelated to hard real-time systems. Ref. [54] briefly and informally presents a protocol based on a variable number of sponsors as well. In the proposed scheme, permanent node failures will lead to a rapid decrease in the reliability of the protocol. On the other hand, reintegrating nodes is easy and does not incur additional overhead.

4.8 Discussion and Conclusion

This chapter provided the precise specification of a protocol for membership agreement in synchronous systems. This protocol is flexible in the sense that the trade-off

between reliability and communication overhead can be defined at design time.

The existence of structural redundancy (e.g. duplicate buses) supports our assumptions on the failure model. However, the protocol is designed to provide a fair level of reliability even when there is no redundancy. In networks based on a bus topology or a star topology, a transient failure is likely to affect either all receiving nodes or a single node. When designing mission-critical systems it is possible to increase the fault tolerance of the protocol by specifying the failure rate that is to be endured and setting the necessary overhead accordingly.

Reaching agreement is based on the principle that a node should not be removed from the membership in the event of a transient failure. Following this principle implies that the layers built on top of the membership, in particular the applications, are able to handle omission failures. This arises from the fact that single transient receiving failures result in lost messages without membership departures.

We have implemented the processor-group membership protocol in a real-time Ethernet network prototype. In our experimental setup, we have verified through extensive testing that the nodes are able to acknowledge their immediate predecessor in the membership. The prototype implementation allowed the protocol to be tested in a realistic environment. It also allowed us to calculate the departure/reintegration latencies and the bandwidth consumption in a realistic configuration. In our setup, we estimate that the size of the code related to the membership service is less than 4KB, while the data structures occupy 42 bytes in memory.

In addition to node membership, this chapter specifies how task-group membership may be provided. Such a service is intended for systems which benefit from the knowledge about the operational state of individual applications. We base task-membership on the fail-reporting behavior – an extension of the fail-silent behavior by producing either the correct result or a report on the cause of failure. Additionally, we provide a method for nodes to broadcast more than once per communication round.

CHAPTER 5

Fault Injection Testing

This chapter describes a fully automated pre-injection analysis technique aimed at reducing the cost of fault injection campaigns. The technique optimizes the fault-space by utilizing assembly-level knowledge of the target system in order to place single bit-flips in registers and memory locations only immediately before these are read by the executed instructions. This way, faults (time-location pairs) that are overwritten or have identical impact on program execution are removed.

5.1 Introduction

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 [55, 78]. 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 [2]. 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 tool [4, 76], for Nexus-based fault injection [43, 78, 71], 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 Motorola MPC565 [31] – 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

[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.

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.

5.2 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. P. Yuste et al. [78] take, in their experiments, special care to avoid placing faults in empty (i.e. not used) memory regions. They obtained 12% of effective faults and pointed out that a 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 dynamical usage of resources along the time dimension.

Studies conducted in the past have shown that error manifestation (rate and effects) is affected by workload [18, 21, 27]. In [20] the concept of failure acceleration was introduced by R. Chillarege and N. Bowen. They achieve fault acceleration by injecting faults only on pages that are currently in use and by using a workload pushing towards the limits in CPU and I/O capacity.

J. Güthoff and V. Sieh presented in [39] the operational-profile-based fault injection. They state 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.

A. Benso et al. presented in [14] a set of rules with the purpose of collapsing faultlists. 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 [75], T. Tsai et al. introduced a technique named 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 the 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 the test of fault-tolerant circuits, using VHDL models, a set of techniques for speeding up campaigns is described by L. Berrojo et al. in [15]. One of these techniques is workload dependent fault collapsing. During

the reference run (a fault-free execution in order to store the 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.

J. Arlat et al. [9] increased the efficiency of their fault injection experiments targeting the code segment by logging the control flow activated by the workload processes. If the randomly selected address for fault injection is not part of the log (instruction trace), then the corresponding experiment can simply be skipped (as the outcome is already known).

5.3 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.

5.3.1 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 [57] and are available in [13].

5.3.2 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.

5.3.3 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 3.

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 in the table and the read list contains *rA* and *rB*, respectively, *R10* and *R11*. Since these are the two resources read by this instruction, two new lines are inserted into the fault locations for code address $39DE8_{16}$ (the control loop index and the breakpoint invocation are assumed to hold the specified values).

```
1  programTrace: Array containing 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 3: Pseudo-code for the optimization procedure.

The second instruction, at address $39DEC_{16}$, fetches the memory word addressed by the effective address *(R6)*+24 and stores it in *R7*. Its definition in the table specifies *rA* and *MEM32(d+rA)*, respectively, *R6* and the 32-bit word at 1000+24, as being read. The value 1000 of *R6* is obtained during the reference run. The two resources along with the timings are then inserted into the fault-space.

5.4 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 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.

5.4.1 Fault Injection Tool

GOOFI is a fault injection tool developed at the Department of Computer Engineering, 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.

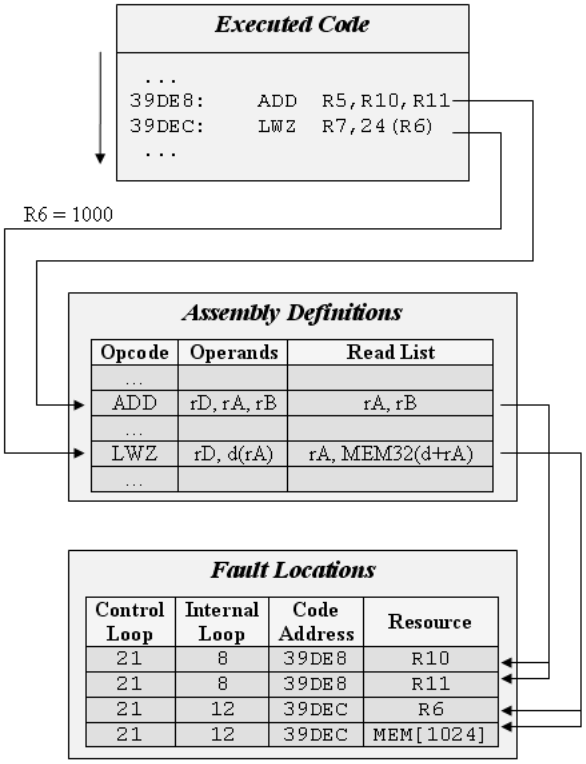

Figure 5.1: Example of the optimization procedure.

A plug-in [71] has recently been developed in GOOFI which uses the Nexus port [43] to inject faults in Motorola'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 [78] 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.

5.4.2 MPC565 Microcontroller

The MPC565 is a microcontroller developed by Motorola that implements the PowerPC instruction standard architecture. It is aimed at the high performance automotive market as well as other control-intensive applications. The complete computer system was based on the phyCORE-MPC565 [58] development board. It includes Motorola'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

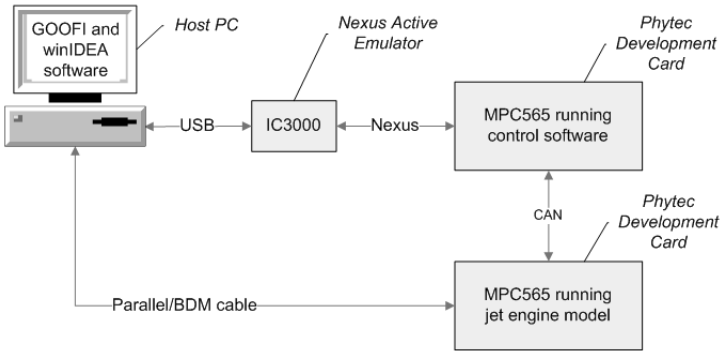

Figure 5.2: Evaluation platform for the jet engine application.

turn, controlled by GOOFI via winIDEA – an integrated development environment offered by iSystem AG. GOOFI and winIDEA are executing on the same host PC.

5.4.3 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 signalled 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 chosen 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.

5.4.4 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 registers constitute the User Instruction Set Architecture (UISA) Register Set. User-level instructions are limited to this register set while supervisor-level instructions have access to other, 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 100KB 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 [9] by using the instruction trace.

5.5 Experimental Results

5.5.1 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.

Campaign		# 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.

The percentage of effective faults (detected or 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 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.

Campaign		Estimated error detection coverage (95% confidence)
Quicksort	*Non-optimized*	61.0 ± 8.2%
	Optimized	55.9 ± 2.7%
Jet Engine Controller	*Non-optimized*	79.7 ± 5.0%
	Optimized	78.3 ± 3.3%

Table 5.2: Error detection coverage estimations (registers).

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 signalled 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.

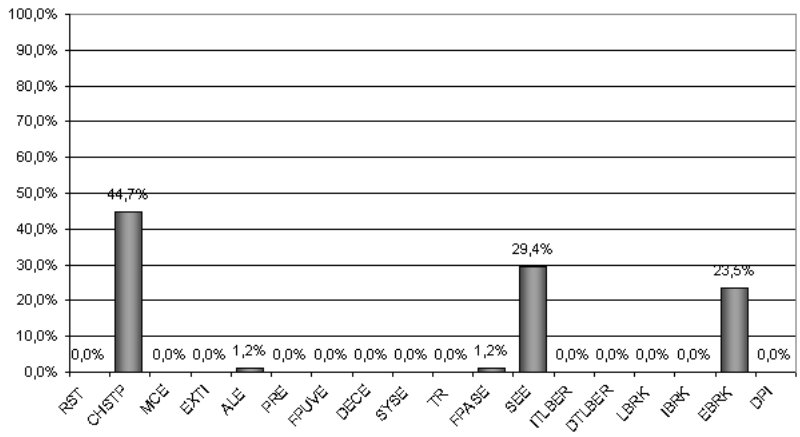

Figure 5.3: Exceptions in the quicksort non-optimized campaign (83 faults in registers).

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 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

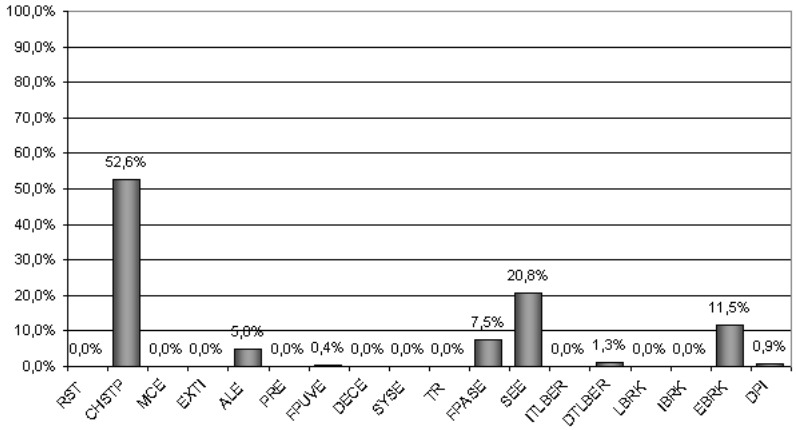

Figure 5.4: Exceptions in the quicksort optimized campaign (744 faults in registers).

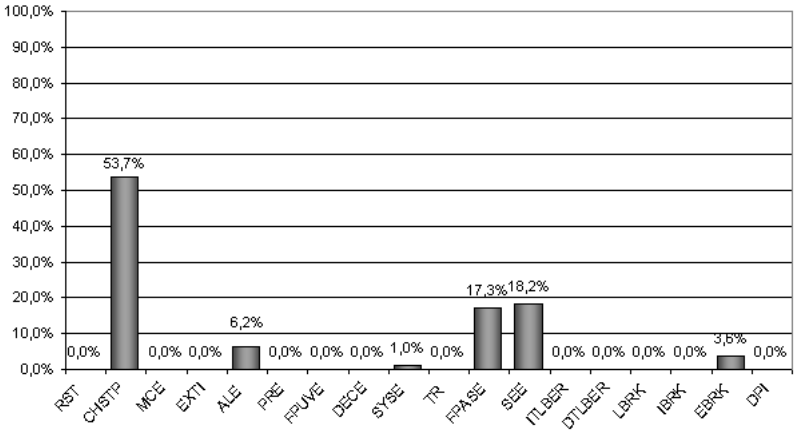

Figure 5.5: Exceptions in the jet engine controller non-optimized campaign (200 faults in registers).

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.

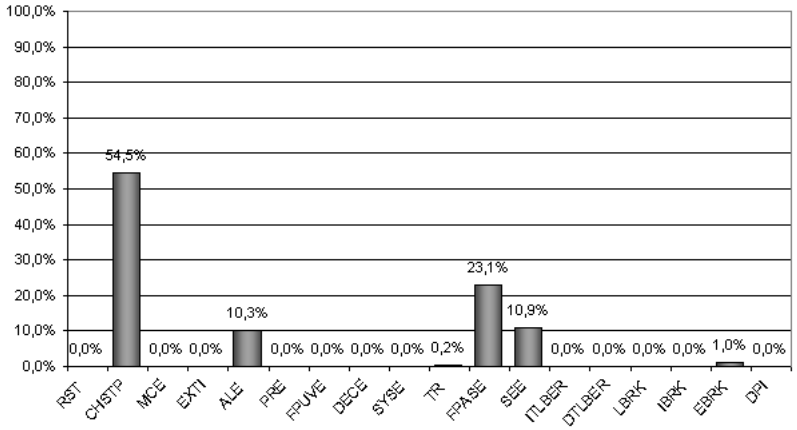

Figure 5.6: Exceptions in the jet engine controller optimized campaign (466 faults in registers).

- 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;
  - An attempt is made to execute an mtspr or mfspr instruction that specifies an unimplemented Special Puspose Register (SPR).
- External Breakpoint Exception (EBRK) – This exception occurs when an external breakpoint is asserted.

Figure 5.7 shows the distribution of faults per register for the optimized campaign. The figure clearly demonstrates the non-uniform distribution caused by the optimization. The number of faults per register is directly proportional to the number of times the register is read.

5.5.2 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

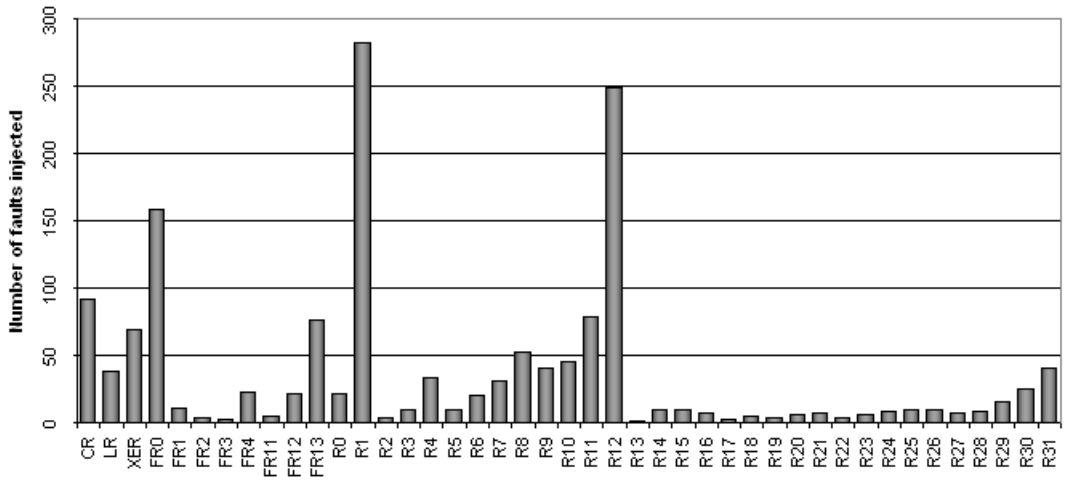

Figure 5.7: Number of faults injected per register.

and optimized fault selection.

Campaign		# 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.

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 one obtained for faults in microprocessor registers.

Table 5.4 shows the error detection coverage estimations obtained with non-optimized and optimized fault selection.

Campaign		Estimated error detection coverage (95% confidence)
Jet Engine Controller	*Non-optimized*	29.9 ± 7.7%
	Optimized	32.7 ± 4.1%

Table 5.4: Error detection coverage estimations (memory).

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 (2%).

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 detection mechanisms are activated in a similar but not identical way for the non-optimized and the optimized campaigns.

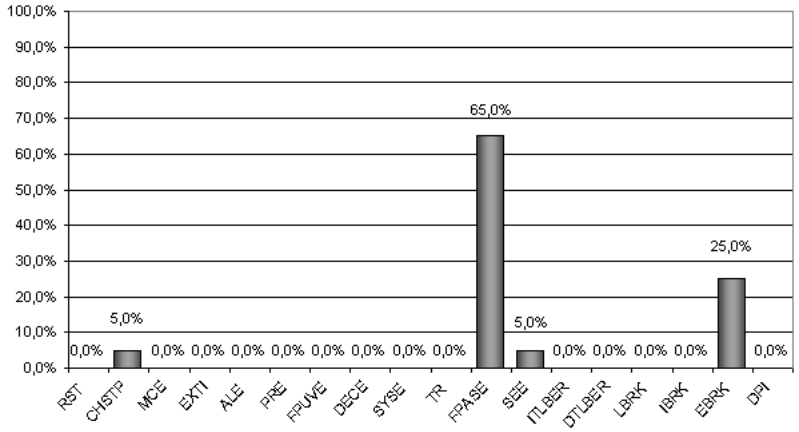

Figure 5.8: Exceptions in the jet engine controller non-optimized campaign (40 faults in memory).

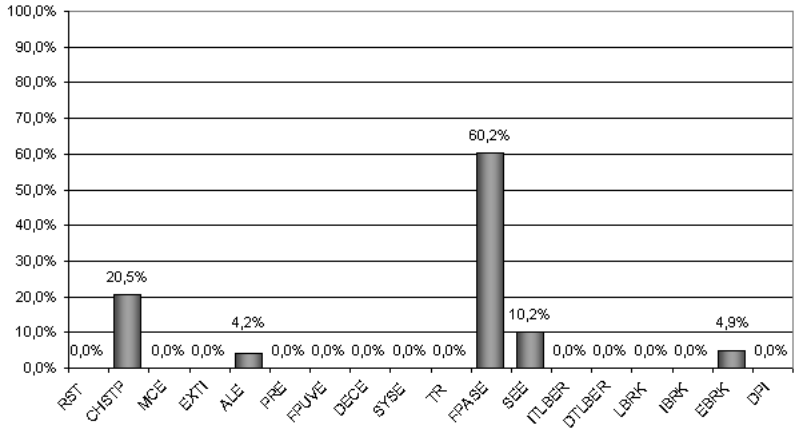

Figure 5.9: Exceptions in the jet engine controller optimized campaign (166 faults in memory).

5.5.3 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 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 100KB for

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

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 between fault-space sizes (registers).

data by the jet engine controller, the size of the complete fault-space is near 200 billion bit-flips.

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 between fault-space sizes (memory).

5.6 Discussion and Conclusion

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 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.

CHAPTER 6

Conclusion

This thesis develops the idea of multi-layer fault tolerance for distributed real-time systems. A fault-tolerant distributed system contains a set of error detection and recovery mechanisms that can be structured into three layers – circuit layer, node layer and system layer –, based on where they are implemented and what parts of the system they involve.

Those fault tolerance layers are not working in isolation from each other. Cooperation among fault tolerance mechanisms that belong to different layers is fundamental to ensure their cost-efficiency. This cooperation can be explicit, e.g. when a fault is detected at one layer and signaled to the upper layers, so that it can be handled; it can also be implicit, when lower fault tolerance layers restrict the failure modes exhibited to the upper layers.

One of the concerns of this research is to understand the best ways in which layers can cooperate. For this reason, we address design issues in building integrated architectures and in providing system-wide redundancy management. In the future, our ambition is to enhance the interaction between node layer and system layer mechanisms.

To achieve this, we are currently building a prototype of a distributed real-time system, which we regard as an instance of our architectural framework. We intend to test our ideas in a realistic setup and to be able to address the validation of fault tolerance mechanisms by fault injection, targeting the nodes of the distributed system. There is therefore much work to be done in the future regarding all the problem areas that we address.

6.1 Future Work

There are several limitations in our theoretical analysis of partitioned systems that we wish to address in the future. It would be interesting to introduce the cost of the partitioning mechanisms in the reliability models, in order to obtain a more realistic comparison between the two architectures. Though we take this cost into consideration in the set of guidelines for building integrated architectures, we have not included it directly in the reliability models.

Furthermore, we have assumed that the failure rate of the partitioning mechanisms is zero. To overcome this limitation, a possibility would be to count the partitioning mechanisms as an additional task, which contributes to the software failure rate. Another possibility would be to assume a higher hardware failure rate for integrated architectures, to reflect the increased complexity of a processor that has support for partitioning.

The hardware failure rate will probably be higher in integrated architectures than in federated architectures, due to the use of more powerful microcontrollers and to the additional hardware necessary to provide partitioning. It would therefore be interesting to conduct a sensitivity analysis to understand the impact of small increases in the hardware failure rate of integrated architectures.

We have assumed in our probabilistic analysis that partitioning at the node layer provides no coverage against hardware faults. However, some faults (e.g. faults that affect only the context of one task) will be covered by the partitioning mechanisms. A sensitivity analysis with values greater than zero for the coverage of hardware faults would therefore be useful.

This thesis provides the precise specification of a protocol for membership agreement. Any such protocol must be formally verified before it can make part of a fault-tolerant distributed system. One of the next steps is therefore to formally verify the correctness of the protocol by using, for instance, a model-checking tool.

We have implemented and tested the protocol in a real-time Ethernet prototype. The protocol extension which provides task-group membership agreement is not included in the current implementation. We would therefore like to implement and test this extension in the future. This issue is very interesting in the context of the thesis – in integrated architectures the membership service should handle both node failures and task failures.

Another important issue that we would like to address in the future is to adapt the protocol, in an efficient way, to networks that provide event-driven scheduling. Time-triggered network standards such as FlexRay and TTCAN provide event-driven communication slots in addition to the static schedule, for which nodes can compete in response to events. It would therefore be interesting to adapt our protocol and make an efficient use of this feature.

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.

References

[1] J. Aidemark. *Node-Level Fault Tolerance for Embedded Real-Time Systems*. PhD thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 2004.

[2] J. Aidemark, P. Folkesson, and J. Karlsson. Path-based error coverage prediction. In *Proceedings of the 7th IEEE International On-Line Testing Workshop (IOLTW-7)*, pages 14–20. IEEE Computer Society, July 2001.

[3] J. Aidemark, P. Folkesson, and J. 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)*, pages 656–665, Yokohama, Japan, June-July 2005. IEEE Computer Society.

[4] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. GOOFI: Generic object-oriented fault injection tool. In *Proceedings of the 2001 International Conference on Dependable Systems and Networks (DSN'01)*, pages 83–88, Washington - Brussels - Tokyo, July 2001. IEEE.

[5] J. Alçada. Extension of a real-time kernel with memory management unit support. Master's thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2006.

[6] J. Alçada, R. Barbosa, and J. Karlsson. Memory protection in a real-time kernel (fast abstract). In *EDCC-6, 6th European Dependable Computing Conference, Coimbra, Portugal*, October 2006.

[7] ARINC Incorporated. ARINC specification 651: Design guidance for integrated modular avionics, November 1997.

[8] ARINC Incorporated. ARINC specification 653: Avionics application software standard interface, 2006.

[9] J. Arlat, J.-C. Fabre, M. Rodríguez, and F. Salles. Dependability of COTS microkernel-based systems. *IEEE Transactions on Computers*, 51(2):138–163, Feb. 2002.

[10] ARM Ltd. *ARM40T Technical Reference Manual (Rev 2)*, November 2000.

[11] T. M. Austin, S. E. Breach, and G. 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*, pages 290–301, Orlando, Florida, June 20–24, 1994.

[12] A. Avižienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[13] R. Barbosa. Fault injection optimization through assembly-level pre-injection analysis. Master's thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 2004.

[14] A. Benso, M. Rebaudengo, L. Impagliazzo, and P. Marmo. Fault-list collapsing for fault injection experiments. In *Proceedings of the Annual Reliability and Maintainability Symposium (RAMS'98)*, pages 383–388, Jan. 1998.

[15] L. Berrojo, I. Gónzález, F. Corno, M. S. Reorda, G. Squillero, L. Entrena, and C. López. New techniques for speeding-up fault-injection campaigns. In *DATE*, pages 847–853. IEEE Computer Society, Mar. 2002.

[16] A. Bertolino and L. Strigini. Assessing the risk due to software faults: Estimates of failure rate versus evidence of perfection. *Software Testing, Verification and Reliability*, 8(3):155–166, 1998.

[17] J. Berwanger, C. Ebner, A. Schedl, R. Belschner, S. Fluhrer, P. Lohrmann, E. Fuchs, D. Millinger, M. Sprachmann, F. Bogenberger, G. Hay, A. Krüger, M. Rausch, W. Budde, P. Fuhrmann, and R. Mores. FlexRay: The communication system for advanced automotive control systems. *SAE transactions*, 110(7):303–314, 2001.

[18] X. Castillo and D. P. Siewiorek. Workload, performance and reliability of digital computing systems. In *Proceedings of the 11th Intl. Symp. on Fault Tolerant Computing (FTCS-11)*, pages 84–89, Portland, Maine, USA, June 1981. IEEE Computer Society Press.

[19] M.-I. Chen and K.-J. Lin. Dynamic priority ceilings: A concurrency control protocol for real-time systems. *Real-Time Systems*, 2(4):325–346, 1990.

[20] R. Chillarege and N. S. Bowen. Understanding large file system failures : A fault injection experiment. In *Proceedings of the 19th Fault-Tolerant Computing Symposium (FTCS'89)*, pages 356–365, Washington, D.C., USA, June 1989. IEEE Computer Society Press.

[21] R. Chillarege and R. K. Iyer. The effect of system workload on error latency: An experimental study. In *SIGMETRICS*, pages 69–77, Aug. 1985.

[22] V. Claesson, H. Lönn, and N. Suri. An efficient TDMA start-up and restart synchronization approach for distributed embedded systems. *IEEE Trans. Parallel Distrib. Syst.*, 15(8):725–739, 2004.

[23] M. Clegg and K. Marzullo. A low-cost processor group membership protocol for a hard real-time distributed system. In *IEEE Real-Time Systems Symposium*, pages 90–98. IEEE Computer Society, 1997.

[24] J. Condit, M. Harren, S. McPeak, G. C. Necula, and W. Weimer. CCured in the real world. *ACM SIGPLAN Notices*, 38(5):232–244, May 2003.

[25] F. Cristian. Reaching agreement on processor-group membership in synchronous distributed systems. *Distributed Computing*, 4:175–187, 1991.

[26] F. Cristian and C. Fetzer. The timed asynchronous distributed system model. *IEEE Trans. Parallel and Distrib. Systems*, 10(6):642–657, 1999.

[27] E. W. Czeck and D. P. Siewiorek. Observations on the effects of fault manifestation as a function of workload. *IEEE Transactions on Computers*, 41(5):559–566, 1992.

[28] B. L. Di Vito. A model of cooperative noninterference for integrated modular avionics. In *Proceedings of the Conference on Dependable Computing for Critical Applications (DCCA-99)*, pages 269–286, 1999.

[29] P. D. Ezhilchelvan, R. A. Macêdo, and S. K. Shrivastava. Newtop: A fault-tolerant group communication protocol. In *Proceedings of the 15th International Conference on Distributed Computing Systems (ICDCS'95)*, pages 296–306, Los Alamitos, CA, USA, May 30–June 2 1995.

[30] C. Fetzer and F. Christian. A fail-aware membership service. In *Proceedings of The 16th Symposium on Reliable Distributed Systems (SRDS '97)*, pages 157–164, Oct. 1997.

[31] Freescale Semiconductor, Inc. *MPC5553/MPC5554 Microcontroller Reference Manual (Rev 3.1)*, October 2005.

[32] Freescale Semiconductor, Inc. *MPC565 Reference Manual (Rev 2.2)*, November 2005.

[33] T. Führer, B. Müller, W. Dieterle, F. Hartwich, R. Hugel, and M. Walther. Time triggered communication on CAN (Time Triggered CAN - TTCAN). Technical report, Robert Bosch GmbH, 2000.

[34] J. 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), Bethesda, MD, USA*, pages 409–415. IEEE Computer Society, June 2002.

[35] J. A. Goguen and J. Meseguer. Security policies and security models. In *Proc. IEEE Symposium on Security and Privacy*, pages 11–20, 1982.

[36] S. S. Gokhale, P. N. Marinos, and K. S. Trivedi. Important milestones in software reliability modeling. In *Proc. of Software Engineering and Knowledge Engineering (SEKE) '96, Lake Tahoe, NV, USA*, pages 345–352, 1996.

[37] R. Golding. *Weak-Consistency Group Communication and Membership*. PhD thesis, University of California, Santa Cruz, USA, 1992.

[38] D. Grossman, G. Morrisett, T. Jim, M. Hicks, Y. Wang, and J. Cheney. Region-based memory management in Cyclone. In *ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI) , Berlin, Germany*, pages 282–293, 2002.

[39] J. Güthoff and V. 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'95)*, pages 196–206, Los Alamitos, June 1995. IEEE Computer Society Press.

[40] M. Hedén. A deterministic ethernet based real-time communication prototype. Master's thesis, Department of Computer Science and Engineering, Chalmers University of Technology, Göteborg, Sweden, 2007.

[41] H. Heinecke, K.-P. Schnelle, H. Fennel, J. Bortolazzi, L. Lundh, J. Leflour, J.-L. Maté, K. Nishikawa, and T. Scharnhorst. Automotive open system architecture - an industry-wide initiative to manage the complexity of emerging automotive e/e architectures. In *Proc. Convergence*, 2004.

[42] J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufman, second edition, 1996.

[43] IEEE - Industry Standards and Technology Organization. The nexus 5001 forum™ standard for a global embedded processor debug interface (available from IEEE-ISTO - www.nexus5001.org), 2003.

[44] International Electrotechnical Commission. IEC 61508 - Functional safety of electrical/electronic/programmable electronic safety-related systems (7 parts available from IEC Geneva, Switzerland - www.iec.ch).

[45] T. Jim, J. G. Morrisett, D. Grossman, M. W. Hicks, J. Cheney, and Y. Wang. Cyclone: A safe dialect of C. In C. S. Ellis, editor, *USENIX Annual Technical Conference, General Track*, pages 275–288. USENIX, 2002.

[46] S. Katz, P. Lincoln, and J. Rushby. Low-overhead time-triggered group membership. *Lecture Notes in Computer Science*, 1320:155–169, 1997.

[47] K. H. Kim, H. Kopetz, K. Mori, E. Shokri, and G. Grünsteidl. An efficient decentralized approach to processor-group membership maintenance in real-time LAN systems: The PRHB/ED scheme. In *Symposium on Reliable Distributed Systems*, pages 74–83, 1992.

[48] H. Kopetz and G. Bauer. The time-triggered architecture. *Proceedings of the IEEE*, 91(1):112–126, 2003.

[49] S. Kowshik, D. Dhurjati, and V. S. Adve. Ensuring code safety without runtime checks for real-time control systems. In S. S. Bhattacharyya, T. N. Mudge, W. Wolf, and A. A. Jerraya, editors, *CASES*, pages 288–297. ACM, 2002.

[50] L. Lamport, R. Shostak, and M. Pease. The Byzantine generals problem. *ACM Transactions on Programming Languages and Systems*, 4(3):382–401, July 1982.

[51] Y.-H. Lee, D. Kim, M. F. Younis, J. X. Zhou, and J. McElroy. Resource scheduling in dependable integrated modular avionics. In *Proceedings of the 2000 International Conference on Dependable Systems and Networks (DSN 2000)*, pages 14–23. IEEE Computer Society, 2000.

[52] C. L. Liu and J. W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 1(20):44–61, Jan. 1973.

[53] H. Lönn. A fault tolerant clock synchronization algorithm for systems with low-precision oscillators. In *Proceedings of the 3rd European Dependable Computing Conference (EDCC-3), Prague, Czech Republic*, volume 1667 of *Lecture Notes in Computer Science*, pages 88–105. Springer, Sept. 1999.

[54] H. Lönn. *Synchronization and Communication Results in Safety-Critical Real-Time Systems*. PhD thesis, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, 1999.

[55] H. Madeira and J. G. Silva. Experimental evaluation of the fail-silent behaviour in computers without error masking. In *Proceedings of the 24th Fault-Tolerant*

Computing Symposium (FTCS'94), pages 350–359, Austin, Texas, June 1994. IEEE Computer Society.

[56] L. E. Moser, P. M. Melliar-Smith, and V. Agrawala. Processor membership in asynchronous distributed systems. *IEEE Trans. Parallel Distrib. Syst.*, 5(5):459–473, 1994.

[57] Motorola, Inc. *RISC Central Processing Unit Reference Manual (Rev 1)*, Feb. 1999.

[58] PHYTEC Meßtechnik GmbH. *phyCORE-MPC565 Hardware Manual (available from www.phytec.de)*, April 2004.

[59] K. Ramamritham and J. Stankovic. Scheduling algorithms and operating systems support for real-time systems, Jan. 1994.

[60] A. Ricciardi and K. 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*, pages 341–353, Montreal, Quebec, Canada, 19–21 Aug. 1991.

[61] L. Rodrigues, P. Veríssimo, and J. Rufino. A low-level processor group membership protocol for LANS. In *ICDCS*, pages 541–550, 1993.

[62] A. Roscoe, J. Woodcock, and L. Wulf. Non-interference through determinism. *Journal of Computer Security*, 4(1):27–54, 1996.

[63] B. Rostamzadeh, H. Lönn, R. Snedsbøl, and J. Torin. DACAPO: A distributed computer architecture for safety-critical control applications. *Proc. IEEE International Symposium on Intelligent Vehicles, Detroit, MI, USA*, pages 376–381, 1995.

[64] RTCA Inc. DO-178B/ED-12B - Software considerations in airborne systems and equipment certification, December 1991.

[65] J. Rushby. Partitioning in avionics architectures: Requirements, mechanisms, and assurance, SRI International, Menlo Park, California, USA. Technical report, June 1999.

[66] J. Rushby. Bus architectures for safety-critical embedded systems. In T. A. Henzinger and C. M. Kirsch, editors, *Embedded Software, First International Workshop, EMSOFT 2001, Tahoe City, CA, USA, October, 8-10, 2001, Proceedings*, volume 2211 of *Lecture Notes in Computer Science*, pages 306–323. Springer, 2001.

[67] A. Sabelfeld and D. Sands. Dimensions and principles of declassification. In *Proceedings of the 18th IEEE Computer Security Foundations Workshop (CSFW'05)*, pages 255–269. IEEE Computer Society Press, June 2005.

[68] R. A. Sahner, K. S. Trivedi, and A. Puliafito. *Performance and Reliability Analysis of Computer Systems: An Example-Based Approach Using the SHARPE Software Package*. Kluwer Academic Publishers, Boston, Nov. 1995.

[69] R. Seyer, C. Siemers, R. Falsett, K. H. Ecker, and H. Richter. Robust partitioning for reliable real-time systems. In *Workshop on Parallel and Distributed Real-Time Systems – WPDRTS, 18th International Parallel and Distributed Processing Symposium (IPDPS 2004)*, Apr. 2004.

[70] M. Simpson, B. Middha, and R. Barua. Segment protection for embedded systems using run-time checks. In T. M. Conte, P. Faraboschi, W. H. Mangione-Smith, and W. A. Najjar, editors, *CASES*, pages 66–77. ACM, 2005.

[71] D. Skarin, J. Vinter, P. Folkesson, and J. Karlsson. Implementation and usage of the GOOFI MPC565 nexus fault injection plug-in. Technical Report 04-08, Chalmers University of Technology, Department of Computer Engineering, Göteborg, Sweden, 2004.

[72] W. Stallings. *Operating Systems: Internals and Design Principles*. Prentice-Hall, fourth edition, 2001.

[73] N. Storey. *Safety-Critical Computer Systems*. Addison-Wesley, 1996.

[74] A. S. Tanenbaum. *Modern Operating Systems*. Prentice-Hall, Upper Saddle River, NJ, USA, second edition, 2001.

[75] T. K. Tsai, M.-C. Hsueh, H. Zhao, Z. Kalbarczyk, and R. K. Iyer. Stress-based and path-based fault injection. *IEEE Transactions on Computers*, 48(11):1183–1201, Nov. 1999.

[76] J. Vinter, J. Aidemark, D. Skarin, R. Barbosa, P. Folkesson, and J. Karlsson. An overview of GOOFI – a generic object-oriented fault injection framework. Technical Report 05-07, Chalmers University of Technology, Department of Computer Science and Engineering, Göteborg, Sweden, 2005.

[77] M. M. Wilding, D. S. Hardin, and D. A. Greve. Invariant performance: A statement of task isolation useful for embedded application integration. In *Proceedings of the Conference on Dependable Computing for Critical Applications (DCCA-99)*, pages 287–300, 1999.

[78] P. Yuste, J.-C. Ruiz-Garcia, L. Lemus, and P. J. Gil. Non-intrusive software-implemented fault injection in embedded systems. In *LADC*, volume 2847 of *Lecture Notes in Computer Science*, pages 23–38. Springer, 2003.

[79] S. Zdancewic. Challenges for information-flow security. In *Proceedings of the 1st International Workshop on the Programming Language Interference and Dependence (PLID'04)*, Aug. 2004.

[80] S. Zdancewic and A. C. Myers. Robust declassification. In *Proceedings of the 14th IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 15–26. IEEE Computer Society Press, June 2001.

[81] X. Zhang and H. Pham. Software field failure rate prediction before software deployment. *Journal of Systems and Software*, 79(3):291–300, 2006.