Igor Schagaev · Eugene Zouev · Kaegi Thomas

Software Design for Resilient Computer Systems

Second Edition

Springer

Software Design for Resilient Computer Systems

Igor Schagaev · Eugene Zouev ·
Kaegi Thomas

Software Design for Resilient Computer Systems

Second Edition

Springer

Igor Schagaev
IT-ACS Ltd
Stevenage, UK

Kaegi Thomas
IT-ACS Ltd
Stevenage, UK

Eugene Zouev
Department of Informatics
Technopolis
Innopolis, Kazan, Russia

ISBN 978-3-030-21243-8 ISBN 978-3-030-21244-5 (eBook)
https://doi.org/10.1007/978-3-030-21244-5

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

What I consider to be the strongest point of this work, indeed its main advantage is the extension of the winning strategy of military pilots to a multimodal complex: “If your action leads to the unexpected, step back and play anew.” To recognize that there is a hierarchy of response options and then choose the least obvious sequence, yet the one leading to survival is a human miracle. To apply this consistently to an array of systems run by different algorithms is new engineering. Contrary to our experiences with autonomous and semi-autonomous multimodular systems, we can avoid creating irreconcilable paradoxes (shutdowns or tonal failures.) In fact, they can be resisted if our causational design logic is augmented (or replaced) with an interactive-transformation-interactive approach. This changes our thinking from compensating for some top-down hierarchy of (event) causes to enabling response “negotiation” across and between all system modules. One way to view the Resilient System Theory is to recognize that it can resist unacceptable outcomes by negotiating multiple options to resolve multiple conflicts at multiple levels. The introduction of “system resilience” requires nonlinear logic and redistributed capacity for flexible coordination and re-coordination of internal regime conditions and parameters. From this perspective, the survival of a multimodal complex is achieved not by insisting on the maximum recovery from losses in or of its key component(s) but on achieving a total system response and behavior with at least minimally optimal integration and recovery under a variety of disorienting, disabling, or dysfunctional conditions. The resilience theory of Prof. Schagaev and his colleagues promises to integrate this conceptual framework into a radically purposeful engineering-design framework. Boris Gorbis Los Angeles.

Stevenage, UK
Kazan, Russia

Igor Schagaev
Eugene Zouev
Kaegi Thomas

Introduction for Second Edition

When in 1989 an anonymous reviewer commented on my short paper that “this classification should be extended to description of distributed systems,” (Yet another approach to classification of redundancy, CIM IMEKO Symposium 1990, Helsinki, pp. 117–124) I was really excited, because people in the research community were thinking much deeper and wider than myself (- I had just defended my Ph.D.).

Further, fault tolerance was migrating to dependability (Jean Claude Laprie was an indisputable authority and expert in this domain, see more www.springer.com/gb/book/9783709191729, which later emerged as the concept of resilience.

In principle, all these new properties had concrete reasoning and meaning behind them: when something erroneous happens, any system of our design should be able to cope with the problem. Options vary, as well as circumstances and area of application, thus:

– If it stops the error propagating and freezes in a *safe state,* it is *fail-stop,* or *fail-safe*;
– If it can cope with *permanent faults* inside the system, it is a *fault-tolerant* system;
– When it continues with reduced functionality, it is *graceful degradation*;
– If it is designed with attention having been paid to reliability, availability, and maintenance or serviceability, it is *dependable* system;
– If it is capable of tolerating obstacles caused by internal and external factors and can spring back, recover, and continue, then a system can be considered as *resilient.*

There are two major ways to achieve any of the properties mentioned above: at system level or at local level (technological). Obviously, any reasonable combination of both levels is also welcome. We do not want to repeat our papers and books (https://www.springer.com/gb/book/9783319150680, https://www.springer.com/gb/book/9783319468129) but to incorporate into the second edition any significant progress that has emerged.

Speaking about ICT systems, especially safety-critical and real-time ones, we might think about the implementation of resilience from the system level down through to hardware and systems software. In addition, we need to consider that each of the parts will both interact with and support each other.

Non-Functional Requirements (NFRs) of each part of the system were considered, such as:

– Performance;
– Reliability;
– Efficiency (mostly energy efficient).

Therefore, the systems that we design should be PRE-smart and provide these properties throughout the life cycle.

Neither our books to date—have appeared as complete. These books have been used in China, Switzerland, Russia, and USA (mostly Masters and Ph.D. students), and we have received substantial feedback, such as:

- While reliability of hardware and availability at the system level are explained and fine, there are no sections, or chapters about performance, especially where parallel and distributed systems are concerned;
- How to apply (as mentioned in the above review) the classification and properties of resilience for and within distributed systems;
- How real-time and safety-critical applications should be treated considering the system resilience: rules for system and for packages—have they changed?

It was especially satisfying when we discovered that these segments are being updated by researchers around the globe, providing excellent contributions to the content.

Thus, our book became an evolving system in itself, aggregating our further efforts with the efforts and results of our colleagues from China, Switzerland, UK, and Russia. Our book has therefore become itself resilient, benefiting from the contributions from the following:

Performance chapter (including element-level performance and parallel design) was prepared and included using materials and having contributions from:

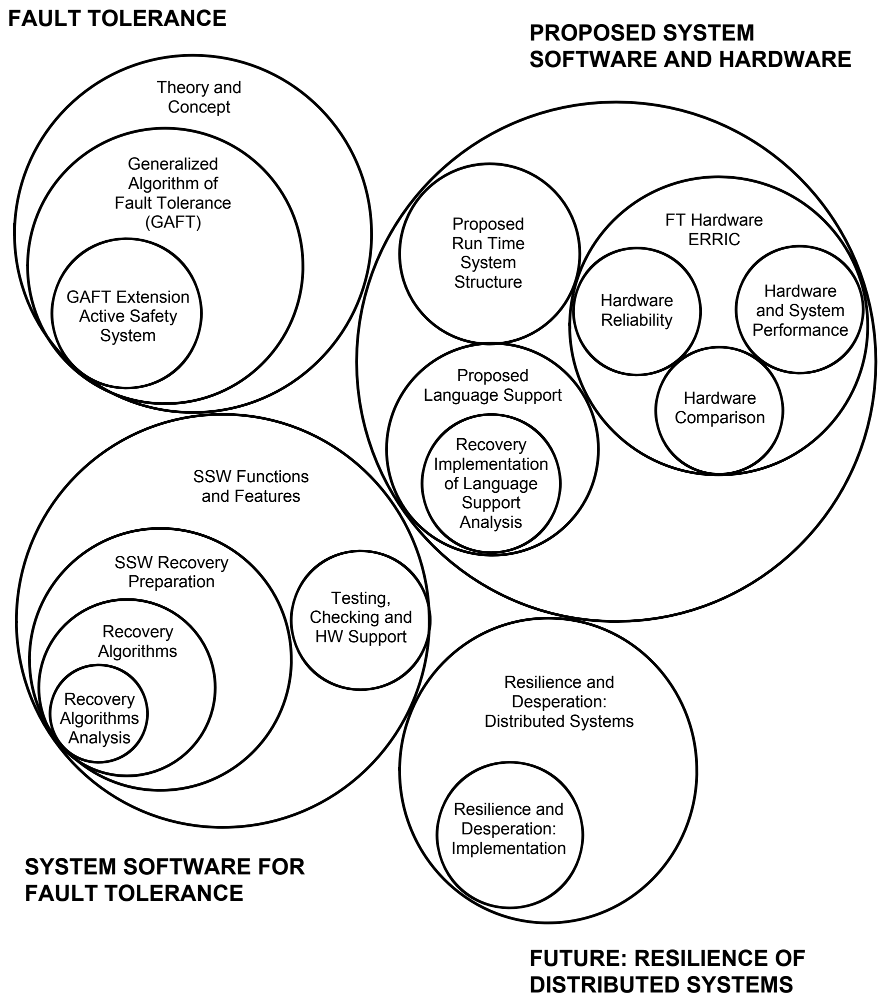
– Professor Hao Kai, Shantou University, China;
– Simon Monkman, IT-ACS Ltd researcher.

System software chapters were part of substantial efforts from:

– Professor Eugeny Zuev and his team in Technopolis, Kazan, Russia.

In turn, requested in 1989 consideration of system level of resilience for distributed systems were developed as two chapters: system level and algorithmic implementation prepared by me and Stephen Farrell. In these chapters, we have introduced a concept of desperation (for transactions within distributed systems) and show that our existing and new results, even patented: https://www.ipo.gov.uk/p-ipsum/Case/PublicationNumber/GB2448351 can be extremely useful making the whole network really resilient and achieving by far better service for applications, especially when critical level of their use was assumed.

The structure of the book now looks like figure below illustrates:


FAULT TOLERANCE
Theory and Concept
Generalized Algorithm of Fault Tolerance (GAFT)
GAFT Extension Active Safety System
PROPOSED SYSTEM SOFTWARE AND HARDWARE
Proposed Run Time System Structure
FT Hardware ERRIC
Hardware Reliability
Hardware and System Performance
Hardware Comparison
Proposed Language Support
Recovery Implementation of Language Support Analysis
SSW Functions and Features
SSW Recovery Preparation
Recovery Algorithms
Recovery Algorithms Analysis
Testing, Checking and HW Support
Resilience and Desperation: Distributed Systems
Resilience and Desperation: Implementation
SYSTEM SOFTWARE FOR FAULT TOLERANCE
FUTURE: RESILIENCE OF DISTRIBUTED SYSTEMS

Contents

About the Authors

Dr. Igor Schagaev is a Professor and Director of IT-ACS Ltd (UK). He is a Fellow of the Institute of Analyst and Programmers (UK), Fellow of British Computer Society (UK). His career has started as an Electromechanical Engineer at the Smolensk Aviation Factory, USSR, a Senior Programmer and Design Engineer at the Institute of Advanced Computations, Central Bureau, Smolensk Branch, and a Senior Design Engineer and System Programmer for Avionics. Completed Ph.D. in Russian Academy of Science (Institute of Control Science) and involvement in projects of hardware and software for submarines, satellites, and aircrafts enabled Igor to absorb an experience which he shared with British Aerosystems in 1994 and Boeing in 98–99. In 1994, Igor has established ATLAB Ltd. Bristol that in 2006 was now transformed into IT-ACS Ltd. He has published seven books in three languages, over 70 papers, since 2006 holds international patent on New Active System Control and supportive mathematical models. He has been honored with several industry awards, achievements, and grants.

Prof. Eugene Zouev is currently a Professor in Innopolis University, Russia. Eugene has graduated and defended his Ph.D. in Moscow State University (1976 and 1999, respectively). He was involved in many research and industrial projects in system software, programming languages and their compilers.

Among his achievements were full ISO-compliant C++ compiler (2000, Moscow, Russia), Zonnon language compiler for .NET (ETH Zurich 2000–2006) with and under supervision of Prof. Niklaus Wirth (Turing Award) and Prof. J. Gutknecht, and many other projects.

His involvement in EU funded project (ONBASS 2004-09) became a next step in research and development summarized to some extent in this book.

Dr. Kaegi Thomas received his Ph.D. in 2012 in ETH Zurich in the area of system software for embedded systems (under supervision of Prof. Schagaev). He is currently a Senior Researcher at Ergon Informatics, Switzerland.

Chapter 1
Introduction

Abstract The chapter introduces what design principles and application requirements we need to pursue achieving resilient functioning of computer system. Principle of simplicity, reliability, reconfigurability, scalability, and redundancy are briefly discussed. Shown that implementation of principles with architecture might include system software modification as well as redevelopment of basis hardware blocks—processing area, storage area, and interfacing area of computer system.

Nowadays, computer systems are applied in diverse and important areas such as banking, military, aviation, intensive health care, industrial control, space exploration, etc. All these areas demand highest possible reliability of functional operation, i.e., availability. Availability is defined as the readiness to provide a correct service whereas reliability is the continuity of correct service [1].

The methods to increase reliability of a system are either based on increasing the reliability of individual hardware components or the introduction of fault tolerance in the overall system design and its basic parts.

For safety-critical systems, i.e., systems whose failure have disastrous consequences and possibly lead to human harm, the two main features such as fault tolerance (FT) and real-time capability (RT) are always implicitly required and should be reflected in hardware (HW), software in general, and, in particular, system software (SSW).

In turn, the design, development, and implementation are mutually dependent processes to achieve FT and RT and should be implemented taking all possible hardware and system software resources into consideration.

The design of fault-tolerant systems itself assumes that the required functionality of the applications is already known at design time, as well as potential faults and the behavior of faulty elements.

This work presents a conceptual approach to the design of an efficient onboard real-time system for safety-critical applications with high performance, reliability, and low power consumption.

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_1

As an illustration, a design of an aircraft onboard system will be used through the whole document. This type of system depicts a typical example of a safety-critical system where no faults and errors in the system can be tolerated.

We are using the standard taxonomy in dependable computing which is summarized in [1]. An *error* is thus a deviation from the correct system service state. The cause of such an error is called a fault and can emerge system internally or system externally.

The requirements of such an onboard system are challenging as they are almost self-contradictory: the required performance and reliability infer at first glance the need for significant power, size, weight, and maintenance.

At the same time, all of them are severely limited. In terms of functionality, onboard real-time systems usually implement control theory algorithms, require well-developed logic and math calculations, real-time data extraction from various external devices during operation, perform matrix calculations, and do a complete prognostic of the system behavior before an event impact such as an internal flight control system error or an external plane problem becomes dangerous.

There is no doubt that the requirements of RT safety-critical systems must be addressed in both—hardware design and system software design.

The conceptual challenges and requirements might be grouped as shown in Figure 1.1.

GLL on Figure 1.1 stands for Graph Logic Language and will be introduced in a special section; HW and SW stand for hardware and software, respectively.

The standard redundancy theory for fault-tolerant system design was introduced in the late 80s [2], applied for econometrics [3] and further developed in [4–6].

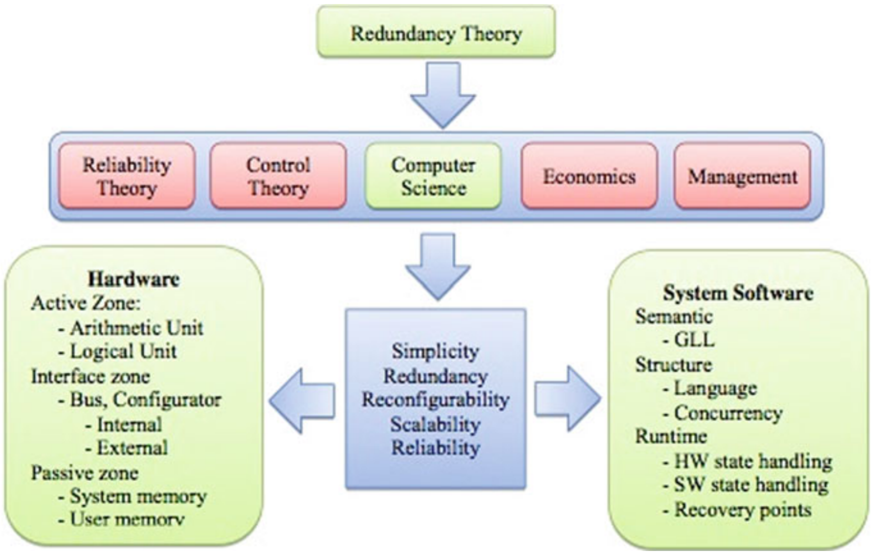


Fig. 1.1 Principles and challenges of research in FT systems

In Sect. 3.1, a short introduction to the redundancy theory related to computer science is presented. The redundancy theory acts as the basic theoretical guide we have to follow to get a fault-tolerant system with highest reliability.

However, when designing a fault-tolerant system, other non-computer science related factors set constraints on the design of such FT systems, for example, required reliability, which is tightly connected with redundancy of the system and should be proven using reliability and redundancy theories.

Also, other non-technological factors play a role in the design of FT systems, especially economics: It is simply not affordable to design and build the system with highest possible reliability and availability; therefore, trade-offs must be made in the design and maintenance as this is also a significant cost factor of the overall system cost.

The central blue box of Fig. 1.1 presents the main principles we follow throughout our designs and therefore this book:

Reliability. Highest reliability of a safety-critical system is the ultimate goal of this book and must therefore always be kept in mind.

Simplicity. We believe that the principle of simplicity must be followed in the design and implementation of SW and HW to achieve the desired reliability and reconfigurability level of HW and SW. This statement can be easily justified with the argument that complex systems in themselves are hard to manage and it is therefore even harder to reach the desired reliability level to make them suitable for fault-tolerant systems. Consequently, it is important to use only essential redundancy in the system to keep the complexity level low.

Redundancy. Making individual hardware components more reliable is out of the question for most projects, as the implications also in manufacturing techniques are severe. We therefore use in this book redundancies of different kinds to improve the system reliability, a solution which is also in practice often used.

Reconfigurability. Reconfigurability allows the system to adapt twofold: First to recover from a permanent fault (graceful degradation) and second by adjusting itself to the requirements of the running application, which helps to keep the system generic, i.e., suitable for multiple purposes.

Scalability. Keep scalability in mind when designing a system so that it can be extended, if the requirements change. In this work, scalability does not have highest priority and might be first sacrificed if one of the other principles is violated.

These principles are well established and especially simplicity and scalability proved their efficiency through a number of well-known projects and developments at ETH Zurich [7–15].

These main principles we apply to two different fields: *Hardware* and *System Software*.

In hardware, the above principles are applied at the system level and to the individual components of the hardware platform. For this, we partition the hardware in three parts: the *Active Zone*, the *Interface Zone,* and the *Passive Zone*. Each of these zones has different properties and need therefore different redundancy mechanism to tolerate faults. We do not include here internal and external devices.

Active Zone. The active zone consists of the arithmetic unit and the logic unit, both separated for better fault isolation and easier implementation of hardware tests.

Interface Zone. This zone includes all communication components, such as the processor bus, memory bus, etc. The interface zone must be protected from external radiation and needs therefore mechanisms to detect and recover from faults. A configurable bus allows the reconfiguration of the hardware to exclude failed hardware components and go into a degraded state, or replaces the failed component with a working one.

Passive Zone. These include basically the storage systems, such as memory that do not act by themselves but are used by controllers or other data processing devices. Here again, the hardware should be able to detect faults and recover from them, but as they are passive systems, the actual fault tolerance must be implemented in conjunction with the interface zone, i.e., the bus.

In software, we distinguish the following parts:

Semantic. The Graph Logic Model (GLM) [16] and Graph Logic Language (GLL) [17] are an interesting attempt for the description of system dependencies, and thus provide a framework for the design and development of the required fault tolerance in HW and SSW. The GLL and GLM are however still in their early phases, and are at the time of writing still subject for further research.

Structure. The programming language is of importance to support safety-critical features. Especially code safety can benefit from a strong types of language without direct assembler support, to force the programmer to write safe code. Another aspect here is the support for recovery points as recovery point creation can directly exploit language features (memory space partitioning, etc.).

Runtime. The main part of this work covers runtime system aspects. In addition to standard runtime system tasks such as resource management, scheduling, and inter-process communication, a runtime system supporting fault tolerance must cover additional tasks such as hardware and software state monitoring to act upon failing components or detected faults. Especially to support recovery from faults, additional features such as recovery points, recovery itself, and recovery monitoring are needed.

The modern requirements for hardware and system software for FT RT systems are as follows:

In hardware:

- High Performance—32 bit, 1–4Ghz,
- Highest possible reliability and availability,
- Means to detect and handle faults and errors (fault tolerance),
- "Zero" maintenance over the operational life span (for satellites, aircrafts, etc.),
- Intelligent power design with backup battery for fault tolerance during operation,
- Mechanical and vibration resistance,
- Graceful mechanical degradation, and
- Feasibility.

In system software:

- Extremely high reliability of software and supportive schemes for hardware fault tolerance,
- High performance and real-time scheduling with assumption of HW faults,
- Fault-tolerant concurrency monitoring,
- Language support of hardware deficiency,
- Control and monitoring of system reconfiguration in case of hardware fault,
- Program, system, and object (using applications) reconfigurability,
- Supportive mechanisms for system recovery, and
- Support for degraded hardware mode, where only the core applications are executed.

The achievable reliability of the FT RT system depends on the available technologies and the use of hardware and system software solutions. The performance also depends on existing technologies and, in particular, on the efficiency of the algorithmic solutions used for the implementation of fault tolerance.

Finally, power consumption is another overarching limit, which affects the FT RT architecture. This, in turn, demands minimum redundancy use along with limiting the processor frequency and voltage at the lowest feasible rate. Here, a solution might be the use of multiple frequency capable hardware, where the processor and other hardware components are able to operate at reduced (when necessary to save power) and full (when performance is crucial) speed.

References

1. Laprie J-C et al (2004) Basic concepts and taxonomy of dependable and secure computing, IEEE Trans Dependable Secur Comput 1(1):11–33
2. Sogomonian E, Schagaev I (1988) Hardware and software fault tolerance of computer systems. Avtomatika i Telemekhanika, pp 3–39. 2
3. Pliaskota S, Schagaev I (1995) Economic effectiveness of fault tolerance. Autom Remote Control 7. 2
4. Schagaev I (1989) Instructions retry in microprocessor recovery algorithms. In: IMEKO—FTSD symposium. 2
5. Pliaskota S, Schagaev I (2001) Life cycle economic efficiency analysis. In: IEEE TESADI-01
6. Schagaev I (2008) Reliability of malfunction tolerance. In: International Multiconference on Computer Science and Information Technology, IMCSIT, pp 733–737
7. Wirth N, Gutknecht J (1992) Project Oberon: the design of an operating system and compiler. Addison-Wesley, New York
8. Johannes M (2002) The active object system—design and multiprocessor implementation. ETH Zurich, Zurich. 4, 82, 109, 173
9. Mossenbock H, Wirth N (1991) The programming language Oberon-2. Technical report, Johannes Kepler Universitat Linz
10. Reiser M, Wirth N (1992) Programming in Oberon: steps beyond Pascal and Modula. Addison-Wesley, Wokingham
11. Wirth N (1977) Modula: a language for modular multiprogramming. Softw: Pract Experience 7(1):1–35

12. Wirth N (1985) Programming in Modula-2. Springer, New York
13. Wirth N (1971) The programming language Pascal. Acta Informatica 35–63
14. Wirth N (1977) The use of Modula. Softw—Pract Experience 7
15. Kaegi-Trachsel T, Gutknecht J (2008) Minos—the design and implementation of an embedded real-time operating system with a perspective of fault tolerance. In: International multiconference on IMCSIT 2008, pp 649–656, 20–22 October 2008
16. Schagaev I et al (2007) Applying the principle of active safety to aviation. In: European Conference for Aerospace Sciences (EUCASS)
17. Schagaev I et al (2010) ERA: evolving reconfigurable architecture. In: International conference on 11th ACIS, pp 215–220

Chapter 2
Hardware Faults

Abstract This chapter explains hardware faults, their origins, dependency on technology used, and some known solutions of fault toleration using error correction codes and redundancy hardware schemes. Shown that with growing density of hardware there is a risk of multiple temporary fault which grows at order of magnitude prime concern for designers of new computer systems for safety-critical application. Hardware faults occur due to natural phenomena such as ionized radiation, variations in the manufacturing process, vibrations, etc. We present in this chapter a short introduction to hardware faults, show the typical fault types and patterns, and also give examples of how to deal with these faults.

2.1 Introduction

Transient faults are a huge concern in silicon-based electronic components such as SRAM, DRAM, microprocessors, and FPGA. Those are devices that have a well-documented history of transient faults mainly caused by energetic particles.

An important obstacle for safety-critical systems to achieve maximum reliability is represented by the susceptibility of those systems to faults produced by radiation. In addition, as manufacturing technologies evolve, the effects of ionizing radiation become a primary concern.

Due to the reduction in size of the transistors and the reduction in critical charge of logic circuits, the susceptibility of technologies to information corruption is increasing [1–3]. A concrete example for this is given in Fig. 2.1 [1], indicating that the malfunction rate (soft error rate) is massively increasing in processor logic for decreasing manufacturing sizes.

Collisions of particles with sensitive regions of the semiconductor can change stored information in ROM/RAM and lead to logic errors, for example, in processor circuits. In this context, Single Event Upsets (SEU) are effects induced by the strike of a single energetic particle (ion, proton, electron, neutron, etc.) in the semiconductor.

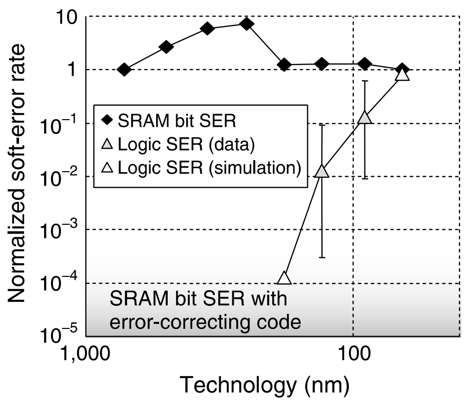


Fig. 2.1 Comparison of the SRAM bit soft error rate (malfunctions) [8]

The charged particle passes through the semiconductor material leaving an ionized track behind leaving sufficient energy in the circuit to have an effect on a localized area of the electronic device.

Single event upsets can occur either through the impact of primary particles (e.g., direct ionization from galactic cosmic rays or solar particles) or by the secondary particles generated after the strike (indirect ionization). It affects many different types of devices, designed using various technologies resulting in data corruption, high current conditions, and transient disturbances. If such single event upsets are not handled well, unwanted functional interrupts and catastrophic failures could take place.

However, single event upset is not fully representative anymore. The number of multiple bits affected by a single event was relatively small using previous silicon technologies. Modern technologies are much more vulnerable and single event can affect multiple bits, so-called Multiple Bits Upsets (MBU). The single event rate affecting multiple bits is expected to increase in the coming years [2].

Traditionally, two different types of techniques have been used to mitigate upsets: fault avoidance and fault tolerance techniques. Fault avoidance techniques, usually at the device level such as silicon on insulator or hardened memory cells, usually involve IC process changes. These techniques have drawbacks in terms of cost, chip area, and speed of operation.

Several low-level fault-tolerant techniques for memories and processor registers have been used to mitigate upsets: error detection codes and error correction codes. *Hamming* codes [4] have been largely employed to detect and correct single-bit upsets. However, these techniques are vulnerable to the increasing multiple bit upset ratio.

Reed–Solomon codes [5] can correct a large number of multiple faults but are not suitable for hardware implementation in terms of design complexity, additional memory required (area), and inherent latency (performance).

At a higher level, circuit-level mitigation techniques with some amount of redundancy, such as N-modular redundancy and voting [6], are frequently used.

Triple modular redundancy [7] is a very simple solution and perhaps the most implemented approach in RT safety-critical applications. However, the area and therefore also the power consumption penalty is large (higher than 200%).

Since the malfunction sensitivity of sequential and combinational logic is increasing dramatically [1, 2] and the multiple bit upset ratio of SRAM is also rising, the previous mechanisms by themselves will not provide the necessary reliability for safety-critical systems.

2.2 Single Event Effects and Other Deviations

Semiconductor devices experience single event upsets in two major forms: in the form of destructive effects which result in permanent degradation or even complete failure of the device and therefore also affecting functionality (permanent fault), and in the form of nondestructive effects, causing no permanent damage (malfunctions).

However, in this sense when something irregular affects the system the question must be answered which of the two cases it is. How to deal with this deviation is the part of the HW/SSW design.

In terms of events, the most common are single event upsets and multiple cell upsets, which both belong to the single event category.

Single-Bit Upset is a single event upset or single-bit upset, meaning, one event produces a single-bit error. This type of error is very common in SRAM.

Multiple Cell Upset is a multiple bit upset for one event regardless of the location of the multiple bits. For example, an FPGA where one routing bit gets an impact from a high energetic particle, affecting several memory positions. Hence, multiple cell upsets involve both upsets, the ones that can be corrected by error correction codes and those which cannot with reasonable overhead.

Multiple Bit Upset is a subset of multiple cell upsets. It is a multiple bit upset for one event that affects several bits in the same word. This type of deviation cannot be corrected by error correction codes with reasonable overhead. However, it is possible to partially avoid multiple bit upsets by using specific layout design of memory cells.

Growing density of logic elements of wafer technology, miniaturization of manufacturing processes, and high clock speeds will inevitably increase rate of so-called intermittent faults.

Inevitable variations in the manufacturing process will induce these faults at higher rate; moreover, impact of these faults will be lasting up to several seconds [2], increasing complexity of recovery. In contrast to external faults such as radiation, intermittent faults are triggered by internal events such as voltage, temperature, and timing variations.

Figure 2.2 illustrates such an intermittent fault.

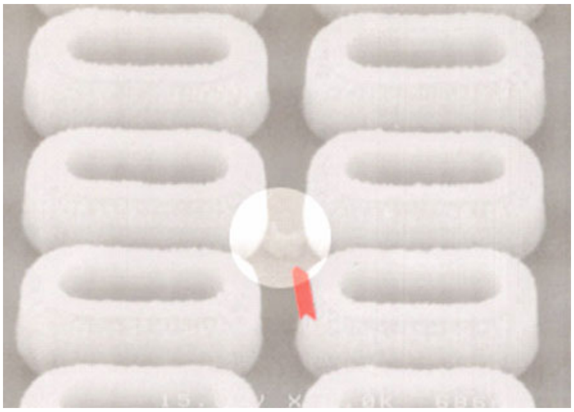


Fig. 2.2 Residue-induced intermittent fault in a DRAM chip [8]

2.3 Conclusion

This short introduction is by no means comprehensive, but illustrates that not all faults can be treated at the hardware level, especially the more complex fault types. Examples of these faults are multiple bit upsets and intermittent faults.

Both of mentioned types of faults are expected to occur more often in the future with decreasing hardware structure sizes and growing demand of higher frequencies of operation.

It is, therefore, obvious that software must assist the hardware in the fault detection and recovery process.

References

1. Baumann R (2005) Soft errors in advanced computer systems. IEEE Des Test Comput 22 (3):258–266
2. Constantinescu C (2008) Intermittent faults and effects on reliability of integrated circuits. In: RAMS 2008. Annual, pp 370–374, Jan 2008
3. Wells P, Chakraborty K, Sohi GS (2007) Adapting to intermittent faults in future multicore systems. In: 16th international conference on parallel architecture and compilation techniques, PACT, p 431, Sept 2007
4. Hamming R (1950) Error detection and error correction codes. Bell Syst Tech J XXVI(9):147–160
5. Reed I, Solomon G (1960) Polynomial codes over certain finite fields. J Soc Ind Appl Math 8 (2):300–304
6. Takaesu K, Yoshida T (2004) Construction of a fault-tolerant voter for n-modular redundancy. Electron Commun Jpn (Part II: Electron) 87(12):62–71
7. Birolini A (2004) Reliability engineering theory and practice, 8th edn. Springer
8. Schagaev I, Castano V (2015) Resilient computer system design. Springer. ISBN 978-3-319-15069-7

Chapter 3
Fault Tolerance: Theory and Concepts

Abstract This chapter briefly introduces how reliability of the system might be considered in combination with fault tolerance. Having introduced hardware faults in the previous chapter, we present in this chapter the elements of theory of fault tolerance and reliability and show how the hardware components of a computing system can be made more resilient to hardware faults. We then introduce the mathematical definition of reliability and show how to calculate the reliability of a system according to the topology of its components. Then we describe the connection between reliability and fault tolerance, i.e., we show how applying different types of redundancy, implemented in software and hardware, increases the reliability of a system. Also, some design advices are given.

3.1 Introduction to Reliability Theory

In addition to the just shown problems of faults, the HW system design for RT FT applications becomes more difficult due to the limited power envelope and requirements such as extremely reliable RT data storage, survivable packaging, possible data retrieval at any time, and a variable number of external influences such as radiation or vibration.

This results in difficult challenges in terms of power dissipation, heat management, mechanical resilience, and of course malfunction tolerance. Given the requirement of zero maintenance for onboard computer hardware and system software, the design and reliability of the FT system is extremely challenging.

There are two different known theoretical approaches to meet the abovementioned reliability requirements and specifications:

1. By developing extremely reliable devices with a substantially higher Mean Time To Failure (MTTF) and substantially higher than expected lifetime of the monitored object. In case of aviation application, an airplane control flight system should have a substantially higher availability and lifetime than the airplane itself. Birolini [1] introduced a comprehensive theoretical approach

based on the application of reliability engineering throughout the system to reach this goal.
2. By the design of fault-tolerant computer systems, with deliberate introduction of redundancy in the system along with active mechanisms for fault detection and reconfiguration to achieve continuous operation (in the sense of the application).

Developing reliable devices for system without use of redundancy for replacement of faulty element may be described as a sequential reliability diagram shown in Fig. 3.1. In this case, the whole system fails if one of the components fails.

The introduction of time into the reliability function of the system, here denoted as $R_s(t)$, is defined as the probability that the system will perform satisfactory from time zero to time t, given that the operation started successfully at time zero. This is a monotonically decreasing function with set initial value as 1.

The Poisson distribution is well suited to model rare events such as failing components and suits therefore well for our model.

$$f(k, \lambda) = \frac{e^{-\lambda}\lambda^{\kappa}}{\kappa!} \tag{3.1}$$

The general Poisson distribution shown above describes the probability of getting *k* events in a given time period. The distribution itself is determined by the value λ and the number of events that are expected in a given time interval.

λ is also the mean value and the variance of the Poisson distribution. The notion of time can be introduced by replacing λ with λt.

The reliability of a component s is defined as the probability of having no failure over a period of time. It is thus a function of time R(t) defined as a Poisson distribution with constant failure rate λ. We get R(t) by setting k of Eq. 3.1 to 0.

$$R(t) = e^{-\lambda t} \tag{3.2}$$

Applied to Fig. 3.1, this results in the system reliability being the product of its individual component reliabilities, assuming that their organization as a serial (cumulative) structure. Note that for this structure, $R_s(t)$ can be simplified as follows:

$$Rs(t) = \prod_{i=1}^{n} R_i(t) = e^{-\left(\sum_{j=1}^{n} \lambda_j\right)t} \tag{3.3}$$

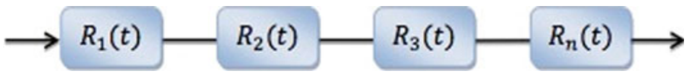


Fig. 3.1 Reliability of a system with sequential elements

Furthermore, the Mission Time Function MT(r) gives the time at which the system reliability falls below the given threshold reliability level *r*. Accordingly, this yields the following definitions:

$$R[MT(r)] = r \tag{3.4}$$

$$MT[R(t)] = t \tag{3.5}$$

The failure rate of a sequential independent element system is equal to the sum of the failure rates of its elements. In the case of a constant failure rate across all elements, the MTTF of the whole system is calculated as follows:

$$MTTS_s = 1/\lambda_s \tag{3.6}$$

Note that this equation highlights the fact that the reliability of a system is directly impacted (in practice often dominated) by the reliability of its least reliable component.

In the context of reliability, fault tolerance is considered as one of the ways to achieve a required level of reliability, i.e., the probability P that the variable R(t) is greater or equal to the value of r.

This approach is based on achieving the required level of reliability by the deliberate introduction of redundancy into the system. The sole purpose of introducing this artificial redundancy is to tolerate possible faults. This approach has been successfully applied since the original work of von Neumann [2] and Pierce [3].

3.2 Connection Between Reliability and Fault Tolerance

Note that introducing redundancy also inevitably involves some additional components and complexity and it is therefore imperative that:

The reliability benefit accruing from the redundancy scheme must far exceed the decrease in reliability due to the actual implementation of the redundancy mechanism itself.

The degradation of reliability by the introduction of redundancy can be modeled in terms of space, with the total space split into “good” and “bad” subspaces, where all redundancy solutions that provide a substantial improvement in reliability belong to the “good” subspace and all other solutions belong to the “bad” subspace.

Then using mathematics, we typically try to connect or exchange and trade-off between these space over time; all based on the ultimate constraints imposed by the natural laws of physics. The different possible redundancy solutions might not cover the same fault types, thus the selection criteria also involve selecting the solutions according to their fault type coverage.

Reliability is concerned with the duration of normal life of an ordinary nonredundant unit, whereas fault tolerance enables the possibility of “life after death”. It is assumed that there is a mechanism available that can just in time, when the fault is detected, repair the unit immediately and completely. Furthermore, the system repair is fast enough that an external observer of the operational system doesn’t even see that the fault has occurred and has been repaired.

In other words, fault tolerance assumes that fault manifestation; fault identification, if necessary; reconfiguration; and recovery are transparent to the system. After the appearance of a permanent fault, a fault-tolerant system will be dynamically restored and thought “as good as new” in operational terms, except for the fact that some of the redundancy has been used up and this may limit the possibilities for future repairs.

The classic parallel generalization of the standard redundancy model [1] describes a system of n statistically identical elements in active redundancy, when k elements are required to perform the system function and the remaining n − k elements are in reserve. The function of the system is considered as successful if during a certain time frame at least k elements of the system were available.

For a simpler example such as a 1 out of 2 system shown in Fig. 3.2, the system function is complete if at least one of the elements is known to be working correctly. The second element is redundant and only introduced for reliability purposes when the first unit is known to be faulty.

For the system in Fig. 3.2, the following reliability function can be derived:

$$R_s = R_1(t) + R_2(t) - R_1(t)R_2(t) \tag{3.7}$$

Thus, it’s obvious that redundancy even for this classic case could improve the reliability of the system considerably. Note that the redundant elements need not necessarily correspond to identical elements, but could also correspond to additional hardware used to detect and treat malfunctions.

The hardware fault tolerance and performance, as already mentioned, are limited by available technologies and the implemented hardware and system software solutions. This results in a system design process with constraints in terms of applied approaches to the above-discussed case with n elements, whereas the maximum reliability is achieved if all elements have equal reliability (Fig. 3.3).

Therefore, the new functionality of the system software which is required to make a system fault-tolerant and real-time capable is, above all, the support for equal reliability of the main components of the hardware architecture (left box of

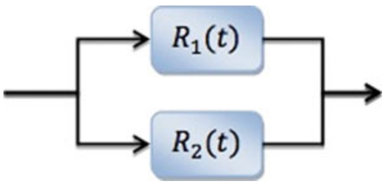


Fig. 3.2 Parallel system structure

Fig. 3.3 Maximum reliability of RT FT system

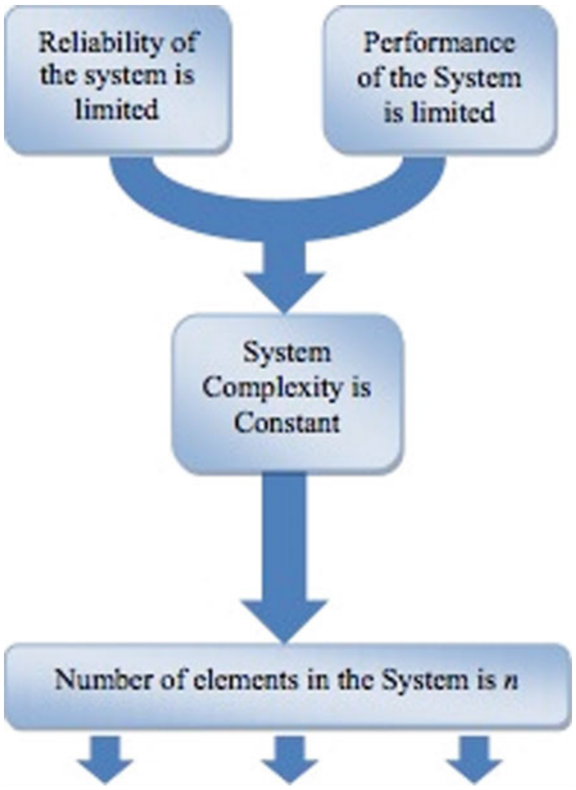

Reliability of the system is limited
Performance of the System is limited
System Complexity is Constant
Number of elements in the System is n

Fig. 1.1) by means of system software, so that in the end no “weak” component of the system reduces the reliability (right box of Fig. 1.1).

Fault tolerance as a new feature of a system requires additional overhead in both software and hardware. This area of research became visible in the early 60s [3]. Several attempts and efforts were done to explore ways how to design and classify fault-tolerant systems. Classifications of measures that implement hardware fault tolerance were presented in [4–6]. But even after these significant contributions a general question has not been answered:

Do we know enough to define the system behavior and predict its behavior from our classification in presence of faults?

In other words, fault tolerance, as a theory might be valued using following statement:

The power of any theory is in predictions and our ability to apply them.

The usefulness of the predictions depends here on timing, relevance, and practicality. Structural analysis of variants how to implement system software support of hardware deficiencies uses the classification of redundancy types and schemes of its application to achieve fault tolerance. The classification developed here is a refinement of many previous results from [4–10] and is, in fact, a further development of [9, 10].

The two top boxes of Fig. 3.4 present the way we think, while the lower box presents the way we implement fault tolerance. In general, three different redundancy types exist, namely, *structural* (S), *informational* (I), and *time* (T) redundancy.

Every concrete implementation of system redundancy uses at least one of these three redundancy types, often more than one and can be implemented in pure hardware (HW()), pure software (SW()), or a combination of both (HW(), SW()).

As an example, software-based information redundancy is abbreviated as SW(I). We use additional quantifiers together with the redundancy type to further specify the used redundancy as shown in Table 3.1.

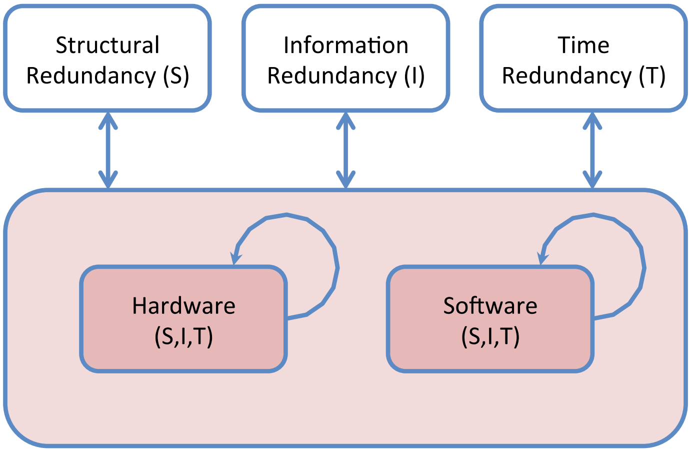


Fig. 3.4 Redundancy types and their implementation variants

Table 3.1 Redundancy classifiers

Quantifier	Example	Description
	SW(*I*)	No quantifier means general, not further specified redundancy. *SW* (*I*), for example, just indicates general software information redundancy
δ	*SW*(δ*I*)	Additional used software-based redundancy
number	*HW*(2*S*)	The number indicates duplication (2) or triplication (3) of a system if used as a prefix for the redundancy type. The original system and the copies are identical, *n* instead of a discreet number is used to mark repetition until success in case of time redundancy
indices	$HW(S_1, S_2)$	Indices are used to mark a duplicated system implementation/ hardware components

Note that the current notation does not include the implementation level. HW (2S) just indicates duplication, but not whether the whole system is duplicated or just parts of it, such as, for example, duplicated memory.

Obviously, any redundancy (hardware and software) needs additional structural redundancy for its implementation. Instruction repetition HW(nT), for example, needs additional hardware registers to store the internal state for the instruction rollback.

We call this redundancy supportive redundancy, i.e., redundancy needed for the implementation of the main redundancy technique. For the sake of clarity, we usually omit this supportive redundancy. In cases where it is not clear whether an applied redundancy type is supportive or not, more than one redundancy type can be used.

Software-based hardware checks that are performed in idle time of the system are such an example, indicated by SW(∂S, ∂T).

Tables 3.2 and 3.3 show some concrete examples of hardware-, and software-based redundancy. The reader, using own experience, can extend this list of examples.

Note that the fault tolerance of a computer system is not considered as a system feature. Throughout the whole work, we will consider it as *a process*.

Table 3.2 Examples of hardware-based redundancy

Redundancy type	Description
$HW(2S)$	Structural (material) redundancy of hardware such as a duplicated memory subsystem
$HW(S_1, S_2)$	A duplicated fault-tolerant computer system with principally nonidentical parts
$HW(\delta I)$	Redundant information bit, for example, an additional parity bit per data word in hardware memory for error detection
$HW(nT)$	Special hardware to repeat or delay computing to avoid the influence of a malfunction
$HW(\delta T)$	Special hardware to delay execution (like in a timing diagram) to avoid malfunctions

Table 3.3 Examples of hardware-based redundancy

Redundancy type	Description
$SW(2T)$	Double repetition of the same program on the same hardware, generally used to check the correctness of the results
$SW(\delta I)$	Informational redundancy of the program, for instance, a backup file, which can be used to restart and recover the program state
$SW(S_1, S_2)$	Two different programs, realizing the same functionality in diverse ways. For example, N-version programming [6]

The introduction of static redundancy in hardware and system software (HW/SSW) might be prohibitively expensive; therefore, it is much better to introduce a process that implements fault tolerance assuming dynamic interaction of existing redundancy types between elements, as it is illustrated in Fig. 3.4.

The main components of the system (hardware and system software) are themselves sources of possible internal hardware faults. At the same time, using various redundancy types, fault tolerance of the computer system can be achieved both by design and by operation. Thus, the elements of the problem become the elements of the solution.

For the implementation of fault tolerance as a process, literature usually considers a three-step fault-handling algorithm: *detection—location—reconfiguration* [11] which was extended and broadened to a more general scheme. This algorithm should be applied for each part of the system, i.e., system software and hardware. The implementation of the algorithm requires the use of the above introduced three different redundancy types (Fig. 3.4) and will be further discussed with necessary details.

In fact, a fault tolerance can be considered as three basic blocks that determine a new feature: the system model, the model of faults that need to be tolerated, and the fault tolerance model. This includes both processes: the design and the implementation of the FT system. In the next section, the fault tolerance process models are described in more detail.

3.3 Models for Fault Tolerance

Say *M* is the known model of the system to perform a given function *F*. To this model, we introduce a new feature that was not defined before: extreme reliability. To express the existence of reliability in the system, the predicates P and Q are introduced to determine the state of the model with regard to the new quality. P and Q also define the direction of the time arrow (see Fig. 3.5).

To analyze ways how to achieve a required reliability level with performance and power consumption constraints, we offer a combination of the following three models:

- The model of an object M_o or $M_{system,}$ in this case, the computer system.
- The model of the faults M_{fault} that an RT FT system should tolerate.
- The model (scheme) M_{FT} or new structure that implements fault tolerance.

The system model, fault model, and fault tolerance model are mutually dependent as it is shown at the bottom of Fig. 3.5. Note that in the here presented approach, the development and manufacturing cost of a solution is not considered.

M_{fault} is a description of all possible faults a system must tolerate. In binary logic, a typical permanent fault manifests as "stuck at zero" or "stuck at one". Behavioral faults such as Byzantine faults (malfunctions) and hidden faults (so-called latent faults) that exist in the hardware over a long period of time do not

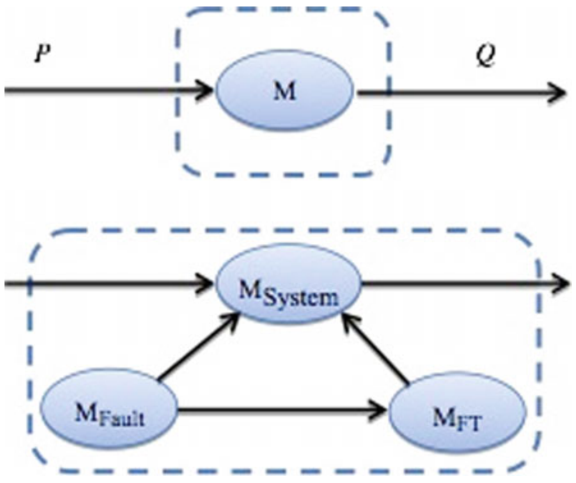


Fig. 3.5 New feature of fault-tolerant system—reliability

ease the life of a system engineer of fault-tolerant systems: all described faults should be tolerated within a limited and specified period of time.

This period actually determines the availability of the system. Fault types differ by their impact, as well as the way they are handled.

Thus, the fault model has its own hierarchy, including single bit, element, behavioral, and subsystem faults. One has to accept that the fault type is varying and some action hierarchy to tolerate them is also required. All faults types should be tolerated as there are no such systems called half- or semi-fault tolerant.

The so-called *fault encapsulation* approach to fault handling can help: due to deliberate design solutions it is possible to ensure that severe faults in the system will manifest themselves as simpler to handle faults from the system's point of view therefore making the fault handling practically possible to implement. This approach will be further developed and applied here.

RT FT system applications assume long operational life; however, fault-handling schemes are needed much more often toward the end of the device life cycle. The appropriate techniques for tolerating faults of various types are presented in Table 3.4. To tolerate malfunctions, time redundancy in hardware (e.g., instruction re-execution) might be effectively used and implemented. System software support is also needed as the hardware cannot cover all possible faults.

It is obvious that faults occurring at the bit level (stuck zero, stuck one, and similar) should be efficiently handled ASAP (as soon as possible) and ALAP (as local as possible), i.e., at the same or nearest level.

The term "level" in our case means the level in the hardware hierarchy on which the fault should be handled. In other words, when a "stuck to zero" permanent fault has happened in the register file with no corrective schemes available, the whole register file has to be replaced, if no other possible reconfigurations were predefined.

Table 3.4 Classification of HW faults in FT computer systems

Type of Fault	Description	Impact
Byzantine	An arbitrary behavior of a part of a device, hardware, or a program	The entire system is affected
Subsystem faults	An arbitrary behavior of a subsystem of the processor temporally or permanently	The entire system is affected
Open fault	Resistance on either a line or a block occurs due to a bad connection	The value associated with the line or the block is modified
Bridging fault	Crossing lines, the number of lines crossed varies	The value associated with the line or the block is modified to a different value
Stuck at 0, Stuck at 1	The result value is fixed to 0 or 1	The result value is stuck to 0 or 1
Bit-flip fault	The result value is fixed to 0 or 1	The bit is modified

In turn, when only one register file is integrated into the chip and no other reconfigurations are defined then the whole chip has to be replaced, etc. Pursuing these two principles allows limiting the fault spreading and its impact to a higher level either in the chip or the system as a whole.

Example: To tolerate bit-flip faults, hardware and system software information redundancies might be used, as well as hardware structural support. In this sense, parity checking in registers, supported and implemented concurrently by hardware, is described as HW(∂I). HW(∂S) and HW(∂T) are needed as supportive redundancies.

HW(∂S) describes the additional parity line and comparison logic, and HW(∂T) describes the additional time needed to update the parity line and executing the comparison. However, the main type of redundancy used in this approach is information.

Up to the best knowledge of the authors, there have been no representative statistics which characterize the exact distribution of faults for computer systems. The distribution of faults depends on the operational environment, for example, temperature, vibration, and radiation exposure.

Even so, it is a well-known fact that the ratio of malfunction to permanent faults can be up to 10^3–10^6. The upper bound belongs to aerospace and aviation, principally due to malfunctions, i.e., errors induced by alpha particles.

In this sense, Figs. 3.4 and 3.5 are transformed into Fig. 3.6 which presents various faults in the system and various possible solutions. M_{fault} illustrates the fact that the fault types are not separated. For example, Byzantine faults of the system might be “stuck at zero” faults of the hardware that were spread throughout the system.

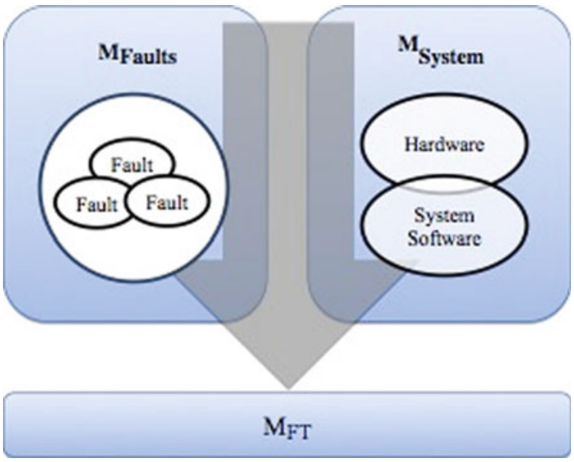


Fig. 3.6 Fault tolerance of computer system

The latency of faults thus becomes crucial in determining the reliability of the system. Consequently, different faults require different actions and mechanisms to tolerate them.

The system model of Fig. 3.6 has overlapped SSW and HW ellipses to represent the duality of the system: hardware and system software. Both of them must be involved to implement fault tolerance and real-time features.

Overlapped HW and SSW ellipses indicate that hardware and system software functions might be applied to tolerate exactly the same hardware faults. Some faults might also be tolerated by hardware or software only. M_{ft} is "a conceptual deliverer" of reliability for the RT FT system. It has to be effective during the whole operational lifetime of the computer itself.

As hardware degrades over time, the fault tolerance mechanisms are more likely to be used toward the end of the life cycle.

We assume that fault-tolerant systems are designed with the assumption that new types of faults do not appear during the operational lifetime of the system, i.e., the system must be designed to be fault tolerant for the set of faults and their types known at design time. All these solutions require careful analysis due to their impact on the system reliability.

In contrast to the usual assumption in reliability modeling, one has to assume that a fault might exist in the system over an arbitrarily long period of time (latent fault) and its detection and elimination is not possible "at once". Consequently, one should accept that fault tolerance is a process, which we present in the next chapter.

Using Dijkstra's approach of defining a function as a process described by its algorithm, fault tolerance is considered as a function that is described and implemented by an algorithm as well.

There are several options to achieve fault tolerance assuming the use of HW and SSW using various types of redundancy mentioned above. However, the use of certain redundancy types might cause system performance degradation which is especially true for software measures [12, 13]. Thus, further analysis of performance/reliability degradation should be taken into account.

The introduced system redundancy might be used in a way to cover only certain fault types a system can tolerate and thus in terms of fault coverage degrades the system, but in terms of performance works without difference to the non-fault-tolerant version.

Software-based redundancy might preserve the same type of fault coverage but with more time redundancy—delays (recovery time degrades, availability degrades) [14], but the fault coverage might not, thus the system degrades in terms of reliability.

3.4 Chapter Conclusion

In this chapter, we introduced the standard reliability theory and showed how to calculate the reliability of hardware components, assuming Poisson distributed faults. Based on this, we showed how to calculate the mean time to failure of such components.

Next, we showed that a component or a system can be made more reliable by introducing deliberate redundancy for tolerating faults.

We classify redundancy in time, structure, and information, implemented either in software or hardware. Every concrete redundancy measure, such as duplication, CRC, etc. can be described using the introduced classification (but not vice versa).

Treating fault tolerance as *a process* consisting of the basic steps of *detection*, *location,* and *elimination* of fault, involving hardware and software in cooperation, proved to be the far superior approach than considering fault tolerance as a static feature.

We then showed that only by thoroughly analyzing the interconnecting model of system, model of fault, and model of toleration of this fault using redundancy becomes possible to derive the "best" fault-tolerant system which fulfills the given application requirements in a given cost envelope.

References

1. Birolini A (2007) Reliability engineering theory and practice. Springer
2. Von Neumann J (1956) Probabilistic logics and synthesis of reliable organisms from unreliable components. In: Shannon C, McCarthy J (eds) Automata studies. Princeton University Press, pp 43–98
3. Pierce WH (1965) Failure-tolerant computer design. Academic Press Inc., New York
4. Laprie J-C (1984) Dependability modeling and evaluation of software and hardware systems. In: Fehlertolerierende Rechensysteme, 2. GI/NTG/GMR- Fachtagung, London, UK. Springer, pp 202–215

5. Avizienis A, Gilley GC, Mathur FP, Rennels DA, Rohr JA, Rubin DK (1971, November) The star (self-testing and repairing) computer: an investigation of the theory and practice of fault-tolerant computer design. IEEE Trans Comput C 20(11):1312–1321
6. Avizienis A (1971) FT computing: an overview. Computer 4(1):5–8
7. Avizienis A (1967) Design of fault-tolerant computers. In: AFIPS conference proceedings, vol 31. Thompson Books, pp 733–743
8. Laprie JC (1992) Dependability: basic concepts and terminology. Springer, Luxemburg
9. Avizienis A, Laprie J-C (1986) Dependable computing: from concepts to design diversity. In: Proceedings of the IEEE, vol 74, pp 629–638
10. Schagaev I (1990) Yet another approach to classification of redundancy. In: FTSD-FTCS congress on technical diagnostic, Varna
11. Schagaev I et al (2001) Redundancy classification and its applications for fault tolerant computer design. In: IEEE TESADI-01
12. Sogomonyan E, Shagaev I (1988) Hardware and software for fault-tolerant computing systems. Autom Remote Control 49:129–151
13. McCluskey E et al (2002) Control-flow checking by software signatures. IEEE Trans Reliab 51(1):111–122
14. Kulkarni V, Nicola V, Trivedi KS (1990) Effects of check pointing and queuing on program performance

Chapter 4
Generalized Algorithm of Fault Tolerance (GAFT)

Abstract Fault tolerance so far was considered as a property of a system. In fact and instead, we introduce a Generalized Algorithm of Fault Tolerance (GAFT) that considers property of fault tolerance as a system process. GAFT implementation analysis—if we want to make it rigorous—should be using classification of redundancy types. Various redundancy types have different "power" of use at various steps of GAFT. Properties of GAFT implementation impact on overall performance of the system, coverage of faults, and ability of reconfiguration. Clear that separation of malfunctions from permanent fault simply must be implemented and reliability gain is analyzed. A ratio of malfunctions to permanent faults is achieving 10^{5-7} and simple exclusion from working configuration a malfunctioned element is no longer feasible. Further, we have to consider GAFT extension in terms of generalization and application for support of system safety of complex systems. Our algorithms of searching correct state, "guilty" element, and analysis of potential damages become powerful extension of GAFT for challenging applications like avionic systems, aircraft as a whole. In Chap. 3, we showed that fault tolerance should be treated as *a process*. In this chapter, we elaborate further this process into a clearly defined algorithm and develop a framework to the design of fault-tolerant systems, the generalized algorithm of fault tolerance—GAFT.We also introduce a theoretical model to quantify the impact of the additional redundancy to the reliability of the whole system and derive an answer to the question of how much added redundancy leads to the system with highest reliability. A question that GAFT cannot answer is *how* the real source of a detected fault can be identified, as the fault manifestation might have occurred in another hardware element and spread in the system due to nonexistent fault containment. We will show an algorithm that based on the dependencies of the elements of a system can identify the possible fault sources and also predict which elements an identified fault might have affected. We now start in a first step by further elaborating the process of fault tolerance.

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_4

4.1 The Generalized Algorithm of Fault Tolerance

The traditional process of monitoring faults as an algorithm is generalized using approaches of [1, 2]. An extension of the algorithm of fault tolerance called the generalized algorithm of fault tolerance (GAFT) [3–5] is illustrated in Fig. 4.1 and explained below in more detail.

The primary function of fault monitoring assumes several steps such as follows:

- Detecting faults,
- Identifying faults,
- Identifying faulty component,
- Hardware reconfiguration to achieve a repairable state, and
- Recovery of a correct state(s) for the system and user software.

The process of fault tolerance considers the interaction between HW/SW elements and requires the physics of the fault itself to be presented somehow in the structure of the algorithm. From the system viewpoint, the nature of hardware faults is different; a hardware fault is considered as either a permanent fault or a temporary fault (malfunction).

When fault type determination becomes more complex the complexity of implementation of processing system grows as well.

This is especially true for multicore systems (MIMD) or processors that support vector (SIMD) instructions. Pipelined processor implementations make a considerable increase in the implementation complexity also. In principle, an asynchronous hardware fault checking solution can lead to spreading faults within the whole system and can make recovery therefore unjustifiably complex.

In practice, the complexity of GAFT implementations depends on the complexity of the system, its faults, and the fault tolerance models. Of course, hardware support for fault tolerance is the fastest way to achieve reliability and with careful design, there should be little or no performance degradation.

```
IF error is detected THEN
    Determine the fault type;
    IF the fault is permanent THEN
        Locate the faulty component;
        Reconfigure the hardware by excluding the faulty unit;
    END;
    IF the fault possibly affected the software THEN
        Locate faulty program states and find the correct
            program state from which it is possible to continue;
        Recover the system from preliminary stored correct SW state;
    END;
END;
```

Fig. 4.1 GAFT

In turn, a combination of HW and SSW implementation should be quite effective depending on the variety of tolerated faults and the acceptable level of performance degradation. It is clear that GAFT should deal with malfunctions first because they are easier to recover than permanent faults.

The closer the fault type detection is to the start of the algorithm of fault tolerance, the faster the algorithm completes in the case of malfunctions. Taking into account that the ratio of malfunctions to permanent faults is roughly 10^4–10^5 a differentiation based on fault types will obviously provide an improvement of FT efficiency. The algorithm of fault tolerance is shown in Fig. 4.1.

GAFT is initiated by external events, such as fault detection, periodic testing, or maintenance. Fault detection can be performed synchronously or asynchronously using hardware or software redundancy. Further, this chapter discusses GAFT implementation and activation by hardware and system software in more detail.

An example of asynchronous hardware redundancy is HW(∂I)—redundant data bits to check data errors. In the first step, the errors are detected and in a second step, the redundancy information is used to recover the system.

An example of synchronous hardware redundancy would be HW(∂T)—special hardware delay schemes to stabilize signals and avoid malfunctions.

Other redundancy examples may also be used for the same purpose including, for example, SW (∂S, ∂T)—execution of SW-based test programs to validate hardware integrity performed during idle time (then SW(∂T) = 0 from the system point of view) or SW(∂I)—informational redundancy of the program.

Hardware faults detected by hardware or software invoke specific fault-dependent actions and initiate the execution of a diagnosing program to identify the fault type and to locate a specific faulty chip. If the fault is permanent, the component that contains the fault is excluded from the system by a special step of GAFT called hardware reconfiguration.

As long as the hardware design supports this and there are still redundant resources available, the system is reconfigured into a properly working state. If no redundant resources are available, the faulty component can still be excluded from the system as long as it is not essential, and the system continues processing in a degraded state.

Hardware fault tolerance from permanent faults requires redundant hardware components HW(S_1, S_1) and is unavoidable for FT RT systems. Hardware reconfiguration must also be performed synchronously and invisible, if possible, to the system function to support continuous operation. If, due to already performed reconfiguration, it is no longer possible to further reconfigure the system, the faulty component might also be disabled, and the processing could continue working as long as no crucial component such as the processor or the memory is affected.

Analog to the hardware recovery, the software must also be recovered, i.e., proving that the fault did not affect the software state or excluding the effects of the fault if it had an impact. It is also necessary to make the system aware of a new possible hardware topology, in the case that an HW component failed and could not be recovered.

In addition, the system must create recovery points that can then be used to roll back program execution to eliminate the effects of the fault and resume communication.

We believe that this step of recovery point creation is also part of GAFT as a separate (concurrent) process to the standard GAFT process. Recovery point creation can either be triggered by an interrupt (for example, timer) or triggered by software. Figure 4.2 shows an adapted version of GAFT that explicitly includes the two steps such as software reconfiguration and recovery point creation.

Now, for every step in GAFT, some kind of redundancy is needed. Table 4.1 presents a framework of how to design the individual steps. Table 4.2 presents concrete examples of implementation.

Incomplete checking and uncertainty in the pattern of fault behavior are attributed possible hardware fault latency. In this case, even prearranged *recovery points* (RP) created during program execution might be damaged, as they may already contain corrupted data. This might be caused by a hardware fault (permanent or malfunction) and its consequences.

Thus, even several steps of recovery would not be enough to achieve a correct (at least consistent) state of hardware and software to continue the execution. A special phase of GAFT (step H in GAFT Table 4.1) is used to locate the correct RP, or to create a new correct state based on trusted historical application data, which is in case of a flight control system, the stored data in the Flight Data Memory. In the worst case, the system could be rebooted and reinitialized.

Note that the state of the software might even in the case of hardware malfunction be damaged, which means that the step H in GAFT (Table 4.1) is required

```
PERIODICALLY DO

     Create recovery point
END
```

```
IF error is detected THEN
  Determine the type of fault;
  IF the fault is permanent THEN
     Locate the faulty component;
     Reconfigure the hardware by excluding the faulty unit;
  END;
  IF the fault affected the software
     Locate correct state from which possible to continue;
     Recover the system from preliminary stored correct
state;
     IF hardware has been reconfigured THEN
     Reconfigure software;
     END;
  END;
END;
```

Fig. 4.2 Generalization of GAFT

Table 4.1 Template for the analysis of GAFT implementations

Step	Description	Redundancy types					
		HW (I)	HW (S)	HW (T)	SW (I)	SW (S)	SW (T)
0	PERIODICALLY DO Create recovery point END						
A	IF error is detected THEN						
B	Determine the fault type						
C	IF fault is permanent THEN						
D	Locate faulty element						
E	Reconfigure hardware						
	END						
F	IF hardware has been reconfigured OR software is affected						
G	Locate faulty software states						
H	Recover software						
I	IF hardware has been reconfigured THEN						
J	Reconfigure software						
	END						
	END						
K	CONTINUE						

for both: malfunctions and permanent hardware faults. If it is possible to prove that the software state was not affected by the fault or the recovery was successful, then the processing is resumed.

According to Table 4.1, a system is fault tolerant if it completely implements GAFT, i.e., all steps of GAFT. Every individual step can of course be implemented in various ways with different properties. Thus, fault-tolerant systems may differ in the following:

– GAFT steps execution time,
– Used redundancy types, and
– Tolerated fault types.

Table 4.1 enables the comparison, classification, and analysis of various solutions for fault-tolerant computer system implementations.

Note also that reliability, power consumption, cost, and maintenance as requirements are usually predefined in the development of safety-critical, embedded, and RT systems.

It is therefore important to define the level of required redundancy, dependent on the application, as a basis to design the system as “as reliable as possible” system which is usually much too expensive to develop. GAFT and redundancy

Table 4.2 Example of GAFT implementations

Step	Description	Redundancy types					
		HW (I)	HW (S)	HW (T)	SW (I)	SW (S)	SW (T)
0	PERIODICALLY DO		7			7	
	Create recovery point END						
A	IF error is detected THEN	2	1, 3, 9	1, 2	8	6,9	6
	ELSE						
B	Determine the fault type	2	1, 3, 9	1	8	6,9	6
C	IF fault is permanent THEN						
D	Locate Faulty Element	2	1, 3	1	8		
E	Reconfigure Hardware		10			10	
	END						
F	IF hardware has been reconfigured OR software is affected		3		8	6	5,6
G	Locate faulty software states					7	7
H	Recover software		9			7,9	7
I	IF hardware has been reconfigured THEN						
J	Reconfigure software		10			10	
	END						
	END						
K	CONTINUE						

classification together allow comparative studies of the proposed project design solutions in terms of applied redundancy types.

This provides a firm foundation for evaluating and choosing the most effective and efficient solution for the implementation of FT RT system from the many alternative approaches.

4.2 Definition of Fault Tolerance by GAFT

So far the classic term *fault tolerance* was used without an actual definition. Clearly, it needs to be differentiated from terms such as *graceful degradation* and *fail-stop*. Avizienis and Laprie [6, 7] proposed that a system is called *fault tolerant* (FTS) if it can recover itself to full performance or at least continue with sufficient specified functions and required features to fulfill all crucial tasks. The set of crucial tasks is obviously application dependent.

If the system can recover itself and continue working in a degraded mode after the occurrence of a fault, it is called a Gracefully Degradable System (GDS) [8]. In turn, if a system can stop itself correctly in an acceptable state once a fault has been detected, it is called a Fail-Stop System (FSS) [9].

The definition of fault-tolerant systems proposed for RT systems in [1] is more rigorous and uses an algorithmic definition of the feature: *A system is called fault tolerant if and only if it implements GAFT.*

In other words:
An RT system is called fault tolerant if it provides the full application functionality, executes GAFT (including recovery) when necessary within a clearly specified timeframe (between sequential data transmissions or tasks) transparently for the applications.

Applied to the example of an airplane flight system, this would mean that if the system can recover itself within defined time (between sequential data inputs, in practice about 125 ms) in failure-free mode (or with recovery that is transparent for the application) then this system is called fault tolerant.

4.3 Example of Possible GAFT Implementation

For every step in GAFT, at least one redundancy-handling mechanism is required to make the system fault tolerant. In this chapter, we give an example of a possible concept that illustrates the process. Note that one redundancy scheme can cover more than one step in GAFT, as one redundancy feature might provide means to detect faults, but also treat faults. Triplicate memory would be one example.

Table 4.2 shows the complete GAFT with used redundancy types to implement each algorithm step for our example application of a flight control system. The numbers in the redundancy columns correspond to the redundancy types applied in each step. These numbers, in turn, are briefly described in Table 4.3 to illustrate the functionality.

Of course, not all fault types can be covered by each of these redundancy mechanisms. We concentrate here on the HW faults presented in Sect. 4.3.4. Refer to [8] for a more detailed classification of possible faults not only in HW but also in SW and during the design phase.

Table 4.2 shows that the various applied redundancy mechanisms are used in single or across several steps of the GAFT algorithm. The numbers in the redundancy-type fields correspond to the redundancy mechanisms presented in Table 4.3. All mechanisms combined result in a system that implements all steps of GAFT and is, therefore, fault tolerant.

Not all components, however, are in this example equally strong fault tolerant. For example, the processor and the memory subsystem are massively covered by checking schemes, but other systems such I/O devices are only covered from faults

Table 4.3 FT features

Nr.	Name	Redundancy type	Description
1	HW checking	*HW*(δS, δT)	Each hardware component such as processor, memory, and controllers has built-in checking schemes to detect faults
2	Processor instruction re-execution	*HW*(δI, δT)	The processor itself has measures to detect faults during execution and can abort and restart the currently running instruction
3	Triplicated memory	*HW*($3S$)	The memory chips are triplicated and a voter compares the output of the three memory chips. If a deviation is detected, the majority voting is used to identify the faulty chip and the faulty value is rewritten. Read after write ensures the proper storing of the data
4	Duplicated storage device	*HW*($2S$)	Storage devices such as flash cards or hard disks are duplicated. Note that this feature does only provide fault detection but not recovery
5	Duplicated program run and input validation	*SW*($2T$)	The same program is run twice with the same input data set. The output of both programs is then compared. Prior to running the program, the input data are validated to conform to a certain pattern and range
6	Checkpoints	*SW*(δT, δS)	Periodically executed checking functions for checking software and hardware, implemented in pure software
7	Recovery points	*SW*(δT, δI)	Recovery points are points in time when the complete system state is consistently stored on a permanent storage device such as flash disks or hard disks. They are either triggered by software or an external interrupt
8	CRC	*SW*(I)	The data stored to the external storage device are protected by a CRC-32. This allows the identification of incorrect data but no recovery
9	Watchdog	*HW*(S)	As an ultimate resort, a watchdog is used to restart parts of, or the whole system. Hardware-based watchdogs can typically on restart the whole system at once
10	Reconfiguration facilities	*SW*(S), *HW* ($S1$, $S2$)	If the hardware failed, the software can reconfigure the hardware to exclude faulty components. In addition, the software can start alternative software version which needs less resources to adapt to the new hardware configuration

by periodic activation of hardware checking schemes and SW input validation. It is, therefore, possible that these devices introduce faults in software, which must be handled solely by SW without hardware support.

In our case, input validation is used to detect faulty data, which is rejected to prevent the faults from entering the system. If such an error is detected, the self-test procedure of the input data hardware device has to run to diagnose the fault type (transient or permanent).

The numbers used in the table are explained in Table 4.3. Table 4.3 shows all used redundancy mechanisms in our example above with a short description. Not every aspect of such a flight control system can be covered by GAFT, for example, real-time aspects are not present.

Thus, GAFT covers only the redundancy part of a possible implementation and other techniques, such as scheduling strategies are used to guarantee real-time constraints.

The combination of the two features: fault tolerance and real-time in one system is very challenging as they work against each other: real time requires tasks to have finished processing before their individual deadlines; in turn, fault tolerance, especially implemented using redundancies HW(T) and SW(T), requires additional time to complete GAFT.

Fault tolerance implementation using time redundancy therefore is hardly possible for real-time systems. For extremely time-critical tasks, application of HW(T) and SW(T) for fault tolerance should therefore be omitted.

The combination of duplicated storage devices (HW(2S)) and CRC-32 (SW(i)) is a typical example of combining two redundancies which on their own can only detect faults of a storage subsystem. The duplicated storage is used to detect faults, whereas the SW-based CRC-32 is used to identify the correct data, which is then used to correct the faulty instance.

4.4 GAFT Properties: Performance, Reliability, Coverage

As already mentioned above and shown in Fig. 4.1 and Table 4.1, the three connected processes checking and testing, preparation for recovery and recovery might be implemented by and within SSW and HW at the design phase and runtime phase of the whole system life cycle.

Obviously, different implementations of the three processes differ in terms of fault coverage, achievable reliability, availability, and cost. Different implementations of GAFT vary in terms of used redundancy types. This, therefore, changes the time to complete GAFT.

The runtime phase T of the system life cycle in terms of time redundancy fault tolerance can be considered at different levels of granularity that are related to the scope of the program being executed. We differentiate the following five levels: *instruction*, *procedure*, *module*, *task,* and *system* (not shown) as shown in Fig. 4.3.

The *instruction-level* scheme assumes that when a fault appears, its influence is eliminated within the instruction execution, using hardware redundancy for fault detection, fault location, and fault recovery.

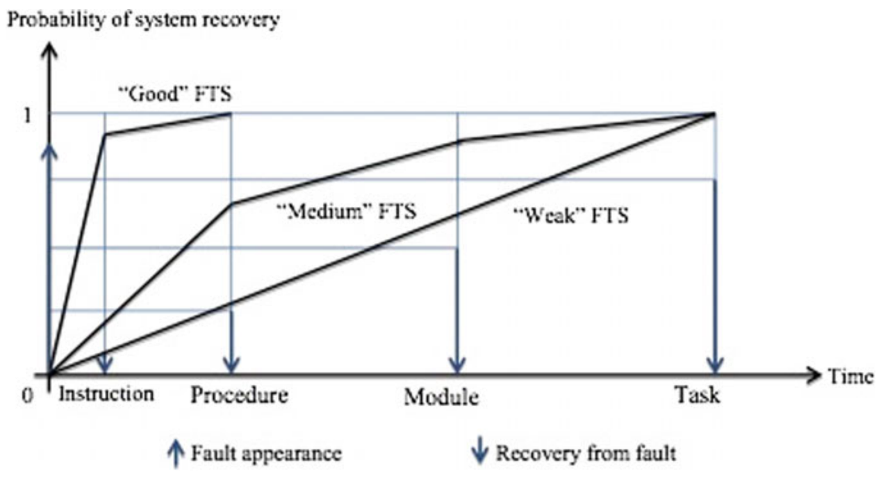


Fig. 4.3 System recovery times according to the used scheme

At this level, only hardware-based redundancy *HW(I, S, T)* can be used as software support would need too much time. Small time redundancy T is used as supportive redundancy for less than instruction time to achieve fault tolerance, or as redundancy *HW(2T)* in case of instruction repetition by the processor.

In another example, triplicate memory with voter, the voter masks the faults of one memory element while accessing the memory and thus the fault is tolerated within one instruction. The techniques such as microinstruction or instruction repetition can be used here as well to provide instruction-level fault correction within the processor.

We call this method of recovery a "good" Fault-Tolerant Computing System (FCTS) as the fault checking, detection, and recovery are done completely transparent to the software [9]. In case of permanent faults, the system can reconfigure itself and continue processing without application intervention.

Note that not all faults are detectable and recoverable within the instruction level, thus the other levels of recovery are still required and should be considered as well in design of fault-tolerant systems.

For the *procedure level* of recovery, a hardware fault and its influence is tolerated and eliminated within the scope of a procedure. For example, techniques such as recovery blocks, wrongly known as recovery points [10], can be considered as a procedure-level scheme.

In contrast to the recovery on the instruction level, the software state space must be in this case conserved and regenerated to recover from a fault. For this scheme, the state of the system that is required to save and recover is much larger than in the instruction-level approach and a significant software overhead is expected to implement fault tolerance at this level. We call a system that uses only this mechanism therefore a "medium" FCTS.

For the *module level* of recovery, a fault and its influence is tolerated within a module execution. For example, module restart or run of a simplified alternative module might be used to avoid system restart, reboot, etc. The same comments apply as for the procedure case, except that time and program overheads for achieving fault tolerance are even higher as the state space is likely to be much larger.

Redundancy on the module level might be considered as *SW(S, I, T)* where S stands for extra software element to perform hardware checking (if hardware schemes are unavailable) and prepare recovery point before module run. Other redundancies I and T define the required extra information to form a recovery point for the module and time required to generate it.

It was already shown in the late 80s that the process of checking and recovery can be implemented in parallel [3, 4], thus $T \rightarrow 0$ from the system point of view, so it could also be treated as a supportive redundancy.

The *task-level* scheme eliminates a hardware fault and its influence by a task restart after the hardware reconfiguration. Here, redundancy is obviously required much greater than in previously described schemes.

Finally, the system state may be recovered by a reboot and task repetition. Due to the high recovery time, we call such a system a “weak” FCTS.

Recovery on *the level of the system* is implicitly available on all systems, as rebooting the system in case of an error basically performs recovery. This corresponds to turning on a system on and must, therefore, be supported in any case. We consider this as the “last resort” if all other measures fail.

The initialization of these GAFT scheme implementations might also require some time especially on the higher recovery levels or depends on hardware signals from checking schemes.

The time impact of implementing GAFT at each of these levels is different, as well as delays caused by their use. Current embedded system practice indicates the following orders of magnitudes for timing:

– Nanoseconds for the instruction level,
– Microseconds to milliseconds for the procedure level,
– Hundreds of milliseconds to seconds for the module level, and
– Seconds to tens of seconds at the task level.

The different schemes have different overheads, capabilities for tolerating various fault classes, power consumption overheads, and system costs. For the instruction-level scheme, the required hardware support (such as duplicate or triplicate hardware modules) could result in serious power consumption overheads, size, and cost.

However, not all hardware subsystems need necessarily be designed in this way. For example, a cost–benefit analysis (where “cost” infers financial, power consumption, chip area, and time delays) might indicate that it is worth having the processor and system RAM protected at this level, but not the other subsystems.

The *procedure-level* scheme requires much less hardware support but, of course, has a larger timing and software-coding overhead. For the module and the task-level schemes, extra hardware and system software support for fault tolerance is actually incomplete as any permanent hardware fault of a crucial HW component would cause the system to stop working and the GAFT would fail.

The final effort of fault actions to achieve fault tolerance here becomes rebooting the system to perform the recovery; but again, even with complete system reboot without hardware redundancy specially dedicated to replace faulty components, the system can only be resilient to malfunctions (transient faults), not hard faults.

Depending on the implementation level, the systems differ in the required time frame to achieve fault tolerance (see the thick lines in Fig. 4.3). Note that several schemes can also be applied in combination, as one scheme alone might not cover all fault types.

Ideally, all possible levels should be considered for implementation within a safety-critical system due to large latencies of faults: presence of external impact inside the system can have a range of up to several seconds (see Sect. 4.2.2). Such a fault can wrongly be identified as a permanent fault on the instruction level but correctly handled as a malfunction on higher levels.

Consider the extreme case where the vast majority of malfunctions can be recovered within the instruction execution and faults that have happened within one instruction execution are invisible for other instructions (and the software). Then, only malfunctions with impact time longer than one instruction need to be detected and recovered by higher levels, for example, at the procedure level. This means that the vast majority of hardware faults is becoming invisible for the system software and system software support of hardware deficiency is used rather rarely, which corresponds to a system according to the left curve in Fig. 4.3.

At the other extreme, the system could tolerate the vast majority of malfunctions at the task level. The operating system or even user support might be needed to tolerate hardware faults.

From the user's point of view, even Microsoft Windows systems might be considered as fault tolerant as long as an application that was scheduled was completed and the results are delivered in time, although maintenance hardware or even a system engineers was involved to fix the fault "in time". Most Windows systems assume that system rebooting and restart of the applications are acceptable ways of operation.

Practical experience in the real-time safety-critical systems domain shows the opposite as the most critical reliability requirements are an MTTF of 10–25 years and system availability *A(t)* of $A(t) > 0.99$ over the whole system life cycle.

4.5 Reliability Evaluation for Fault Tolerance

From the classic reliability point of view, any extra redundancy of hardware reduces the absolute reliability of the system [11]. At the same time, the reliability of the system might be increased if the introduced redundancy is able to tolerate faults of the main part of the system by means of GAFT and itself contributes less in the system fault ratio.

Thus, part of the problem, the decreasing reliability of the system caused by hardware redundancy at some point becomes part of a solution. GAFT implementation breaks down to three processes: P1 for *checking* and *testing*, P2 for *preparation for recovery*, and P3 for *recovery* and recovery monitoring.

Clearly, there are design trade-offs to be made to achieve the optimum operational reliability and redundancy types used as we have already shown.

The reliability analysis [12], introduced in Sect. 4.3.1, shows a reliability value for each element (hardware redundancy) and assumes a Poisson failure rate λ. This allows to calculate the overall reliability as a function of time for the whole system, if the structure of the system is known.

The hardware redundancy used at various steps of GAFT degrades in reliability over time; thus, the achievable performance and reliability and their degradation within the life cycle of the RT system are dependent on each other.

Therefore, an analysis of the surface shape and evaluation of performance and reliability degradation caused by the used redundancies should be performed for every fault-tolerant system. Figure 4.4 presents qualitatively a slope where a fault-tolerant system should be located, between the plane of requirements and curves of reliability and performance degradation.

Furthermore, the introduction of the cost to implement each proposed solution allows to summarize the system overheads required to implement fault tolerance.

There is no doubt that a quantitative evaluation of reliability, performance, and cost overheads within one framework might be extremely efficient for justification of the design decisions and comparison of different approaches in implementation of fault tolerance. There is a correspondence between reliability of FT systems and steps of GAFT related to the malfunction tolerance shown in Fig. 4.4.

Two dimensions: reliability and performance correspond to requirements of the system with controlled degradation over time. This figure illustrates that it is not possible to minimize for cost and maximize all other goals and a trade-off has to be made.

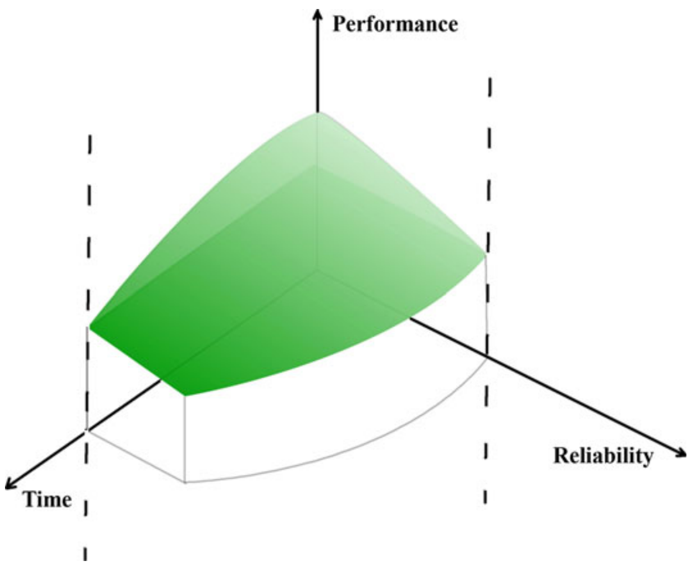


Fig. 4.4 Trade-offs to be made in fault-tolerant system design: time-, performance-, and reliability-wise

4.6 Hardware Redundancy and Reliability

Consider a system with a given reliability which is prone to transient and permanent faults (Fig. 4.5, left graph). If additional hardware is added to detect transient faults, the introduced redundancy to achieve fault detection reduces the absolute reliability of the system as more hardware is used that is prone to faults.

At the same time, if the introduced redundancy is not only used to detect transient faults but also to tolerate them, the reliability of the system increases (Fig. 4.5, right graph). Thus, part of the problem—the decreasing reliability of hardware caused by redundancy at some point (after introduction of recoverability process P3) becomes part of a solution. Note that the system is however still prone to permanent faults.

Figure 4.5 illustrates reliability degradation in presence of malfunction and permanent faults and gain for the system assuming permanent fault ration $\lambda = 10^{-5}$, coefficient of malfunction to permanent fault ratio $k = \{0, 1000\}$, and redundancy of hardware $d = 0.12$. The data applied are natural based on our own design of processor with malfunction tolerance [13], while ratio introduced deliberately lower than real, in real practice as ratio of malfunction to permanent faults, varies from 10^5 to 10^7.

Clear that analysis of effect of redundancy on the reliability of a system is worth to clarify a bit more. Note that malfunction tolerance as mentioned before can be

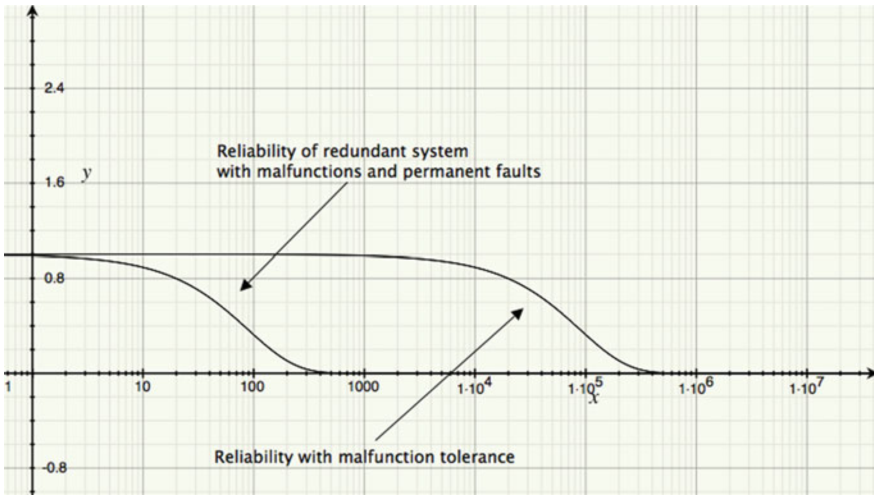


Fig. 4.5 Efficiency of a system with faults and checking schemes

implemented using hardware or system software or their combinations. Thus, reliability gain might be achieved by combination of both methods.

4.6.1 Hardware Redundancy: Reliability Analysis

We introduce a reliability value for each hardware component and assume a Poisson failure rate λ to calculate the overall reliability of the whole system, as it was introduced in previous chapters.

If time redundancy is used to implement one or more steps of GAFT, we can use the overhead of the redundancy solution to give an estimation of the performance degradation caused by the introduced redundancy.

If the additional required cost to develop and manufacture a fault tolerance feature is taken into consideration as well, it is possible to quantitatively evaluate and compare different approaches to implement fault tolerance.

Next, we derive a model to analyze the achievable reliability of possible processor and memory structures and to predict how long and at what availability level such a solution can operate [9].

We denote the permanent fault rate of a processor without redundancy as λ_{pfl} and the transient fault rate as λ_{ifl}. The probability of operation without fault within the time frame [0, T] is determined by

$$P_{nf}(t) = e^{-\left(\lambda_{pfl} + \lambda_{ifl}\right)t} \tag{4.1}$$

The mean time to failure here is

$$MTTF_{nf} = \frac{1}{\lambda_{pfl} + \lambda_{ifl}} \tag{4.2}$$

If we assume that transient faults happen k times more often than permanent faults, we get

$$P_1(t) = e^{-\left((1+k)\lambda_{pfl}\right)t} \tag{4.3}$$

$$MTTF_1 = \frac{1}{(1+k)\lambda_{pfl}} \tag{4.4}$$

As already shown above, k can be as high as 10^3–10^5 especially for space-borne systems.

If a system or processor is capable to (a) identify transient faults and (b) recover from them, additional hardware for both of these two features is required, which of course changes the reliability of the system.

Thus, it follows that

$$P_{ft}(t) = e^{-(1+d_i+d_r)(1+k)\lambda_{pfl}t} \tag{4.5}$$

$$MTTF_{ft} = \frac{1}{(1+d_i+d_r)(1+k)\lambda_{pfl}} \tag{4.6}$$

where d_i reflects the additional hardware required for fault identification and d_r the required hardware for recovery. It is clear that the smaller d_i and d_r are, the higher the reliability that can be achieved for a processor or memory subsystem.

However, the purpose of the newly introduced hardware is the identification and toleration of transient faults is not fully achievable, we have to assume that the system can detect and recover from a part of the transient faults. True, the absolute number of faults and consequently also the permanent-to-transient fault ratio becomes lower.

The system cannot, however, detect and recover all faults, thus we introduce α as an indication of how successful the recovery is. α reflects the reduction of k using fault tolerance mechanisms and must therefore be in the range of (0–1):

$$P_{ft}(t) = e^{-(1+d_i+d_r)(1+\alpha k)\lambda_{pfl}t} \tag{4.7}$$

$$MTTF_{ft} = \frac{1}{(1+d_i+d_r)(1+\alpha k)\lambda_{pfl}} \tag{4.8}$$

What is still missing in the reliability analysis is the rate of successful recovery α, i.e., the chance that a fault is detected successfully and the system can recover fully from the fault. Obviously, as mentioned above, α must be in the range of (0–1), dependent on the fault coverage c. For α, we thus introduce Eq. 4.9 with SF being the so-called *success function*:

$$\alpha = 1 - c * SF \tag{4.9}$$

This success function *SF* defines the recovery rate. A duplicated system can detect all possible malfunctions and if self-checking is implemented, it can also recover from them.

Thus, a success function should meet the following constraints, with x being the amount of used redundancy:

- The recovery rate must be in the range of 0–1;
- If no redundancy is used x = 0, the chance for recovery is 0, and therefore SF $\rightarrow$ 0. If duplication is used x = 1, the recovery is in approximation 1, and therefore SF $\rightarrow$ 1;
- If more redundancy than duplication is used (x > 1), the system gets less efficient.

The function *SF* = x^{1-x} (shown on Fig. 4.6) satisfies these conditions and seems therefore appropriate for a success function initial introduction.

The power part of this function (x^{1-x}) is some kind of penalty function that reflects the third condition the success function has to satisfy, i.e., the inefficiency of more than duplication, when x > 1.

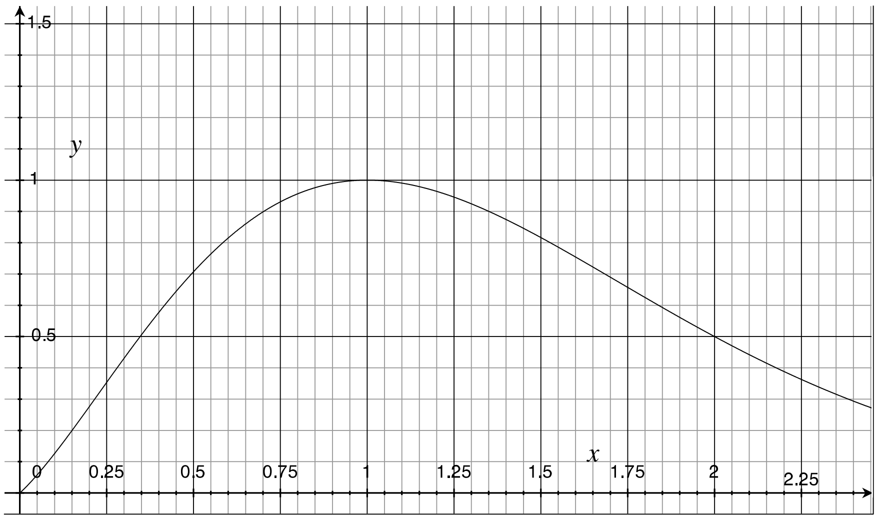


Fig. 4.6 Recovery success function

In other words, the efficiency of the fault-tolerant system declines when the used redundancy exceeds 100%.

If we introduce just defined success function in Eq. 4.9, we get the following equation for α:

$$\alpha = 1 - cx^{b(1-x)} \tag{4.10}$$

It is worth to point out that recovery might be achieved with less than 100% redundancy used, as it is shown in Fig. 4.6.

The well-known Hamming code, for example, provides recovery from a single fault with redundancy 12–13%—depending on which logic schemes and technology were used for implementation. ERRIC (Evolving Recoverable Reduced Instruction Computer [13] is fully recoverable from malfunctions in arithmetic and logic units with 11.8–12.5% of hardware redundancy involved.

To introduce this boost of checking and recovery success in Eq. 4.10, we introduce a variable b, with range {0.125, 0.25, 0.5, 0.75}. Then, success of recovery becomes dependent on redundancy used—lower family of functions in Fig. 4.7. Left-shift function belongs to b = 0.125.

The fault coverage itself depends on the amount of used redundancy; however, this fact has not been integrated into the presented formula and will be briefly analyzed further.

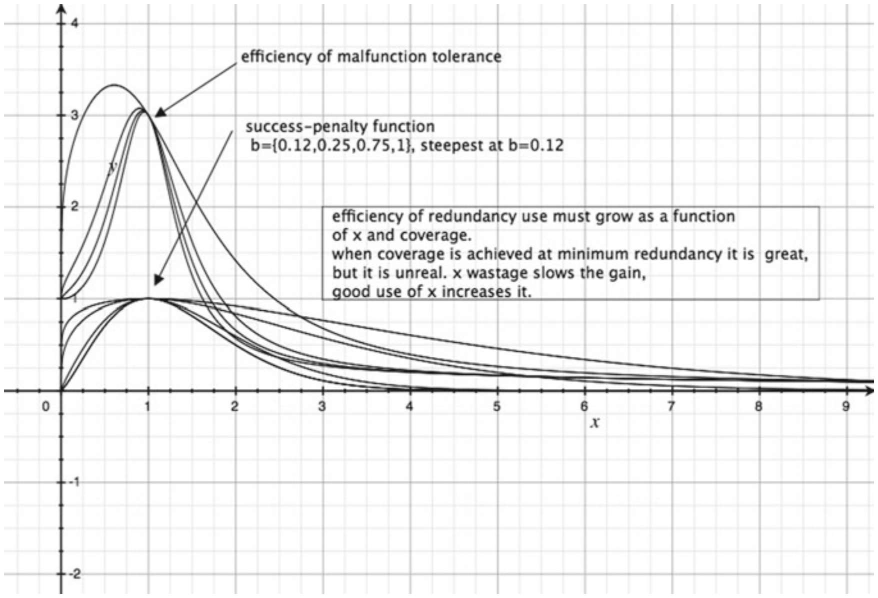


Fig. 4.7 Recovery success function

One has to consider as further research how to split redundancy x into x_d which is the amount of redundancy used for fault detection and x_r which is the amount of redundancy used for recovery.

If α is inserted into equation for mean time to failure $MTTF_{ft}$, we get the following:

$$MTTF_{ft} = \frac{1}{(1 + d_c + d_r)(1 + (1 - cx^{b(1-x)})k)\lambda_{pf}} \tag{4.11}$$

When we need to compare the MTTF for two systems, one with redundancy and one without an improvement in the order of magnitude (Efficiency E, Eq. 4.12), it is proved that the solution with redundancy introduced into the system for checking and recovery purposes is better than the system without redundancy.

$$E = \frac{MTTF_{ft}}{MTTF_{nf}} \tag{4.12}$$

Simplified, we get the following equation for efficiency of redundancy use for reliability gain:

$$E = \frac{1 + k}{(1 + d_c + d_r)(1 + (1 - cx^{b(1-x)})k)}$$

Figure 4.7 (upper family of functions) shows efficiency for different values of *b* and *x,* assuming k = 100. For convenience of illustration, permanent fault ratio is set at 10^{-3}.

Another important variable of recovery success in fault-tolerant systems is coverage of fault, denoted further as *c*. Figure 4.8 illustrates an impact of coverage on efficiency of a system design, using *c* as a variable with 0.5, 0.75, 0.98.

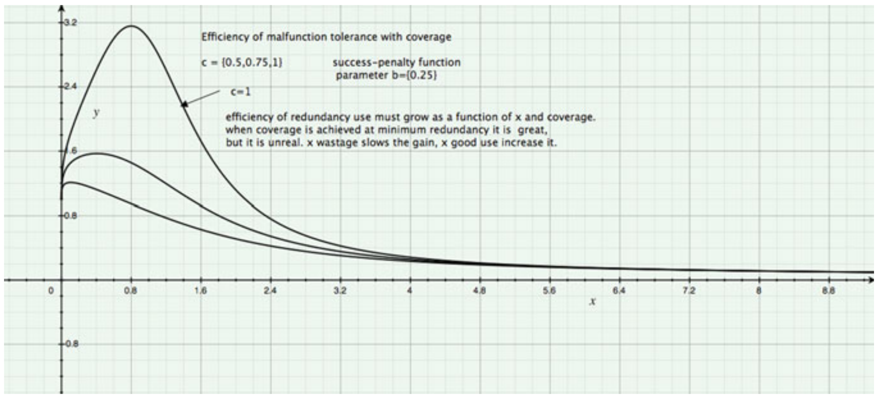


Fig. 4.8 Impact of coverage of fault on reliability gain

Naturally, the more we know about faults and able to recover from them with less cost the better.

In the used example, the chosen transient fault to permanent fault ratio k is 5, i.e., transient faults happen five times more often than permanent faults.

Note that even in this conservative case (in practice, k is in the range of 10^3 to 10^5) a significant improvement or reliability and overall efficiency of the system can be shown. A larger k results also in a higher reliability gain.

We did not separate redundancy for checking and redundancy for recovery—it is subject to further research. Nevertheless, the key outcomes are the following:

- Design of fault-tolerant systems should be revisited: malfunctions must be tolerated at the element level, leaving permanent fault handling to the system level;
- There is an optimum in redundancy level spent on fault tolerance which is less than 100% and dependent on the anticipated fault coverage level. A duplicated system is therefore not the most efficient solution;
- Fault coverage defines the efficiency of a concrete fault tolerance implementation;
- The processes of checking and recovery might be realized concurrently with the main hardware function, with literally no time delay (time redundancy).

One might argue that the introduced success function does not appropriately model the real world. A system with, for example, 2% redundancy spent on the detection and recovery is hardly implementable. In reality, a real success function might start from say 0.12, where a minimum redundancy level is required to get an implementable fault-tolerant system.

However, further research is required to define the exact minimum of redundancy, which is needed to implement a system in real life and also to refine the actual shape of the success function.

The fault coverage is also dependent on the amount of redundancy used in the system that should be integrated into the analysis.

Another aspect that we did not cover in this analysis is manufacturing. With adapted manufacturing techniques and conscious placement of the individual hardware elements such as arithmetic unit, registers, etc. the fault coverage can be increased and the fault effect on the system lowered.

As there are many open questions, an estimation presented here should be considered as indicative and not absolute.

4.7 Conceptual Summary

In this chapter, we introduced GAFT, which describes in an algorithmic form a sequence of required steps to make a system fault tolerant. We then introduced GAFT in tabular form (template), useful to design a fault-tolerant system and prove that every step of GAFT is implemented by the chosen design. A specific redundancy means such as implementation by hardware redundancy re-execution of all processor instructions is able to cover multiple steps of GAFT simultaneously.

The template also identifies which redundancy type is used for the implementation of every GAFT step. We then illustrated the use of the template with an example. Further, we showed that although a complete GAFT implementation results in a fault-tolerant system, the different solutions vary in different aspects: performance, reliability, coverage, and cost. The usage scenario of the system defines the envelope of performance, reliability, and degradation of the system over time, which must be met by the fault tolerance solution within given cost limits.

Different redundancy schemes have different properties, depending on which level (instruction, procedure, module, tasks, and system) they are implemented. We show that as a general guideline, faults should be tolerated as fast and local as possible (ASAP and ALAP), favoring therefore the instruction level for the majority of malfunctions.

We extensively covered the three main processes of GAFT. For testing and checking, we showed that the combination of hardware- and software-based checking is the most efficient approach:

- Hardware-based checking process works well for covering short time malfunctions;
- Software-based checking of hardware for latent malfunctions and permanent errors.

However, for multiple bit and latent transient faults, software must be involved in addition to the hardware, as a pure hardware solution might be prohibitively expensive and inefficient. We also conclude that permanent errors should be handled by software as they involve hardware reconfiguration which needs software support.

A designer of a fault-tolerant system might be tempted to add as much redundancy to the system as possible, thinking that the system is more reliable the more redundancy is used. With a carefully designed reliability evaluation we were able to prove that this approach is wrong as the newly introduced redundancy leads to a decrease in reliability. In fact, we were able to prove that there exists an optimum in redundancy that should be used for a system, which is in fact less than duplication. This result has actually a fundamental importance for system design: so far, there were known approaches in structural and engineering of reliable design and analysis widely applying duplicated, tripled, and quadrupled systems. Mechanical copy of results from mechanical engineering was wrong—in the electronic design, this principle is unjustified and makes computer systems absolutely not necessary redundant and less reliable!

As chapter shows when element of fault-tolerant system is able to tolerate own malfunctions while system level is dealing with permanent faults of hardware by reconfiguration the system becomes at the order of magnitude more reliable. Redundancy of element required to achieve malfunction tolerance at the level of element was proved to be 12%—much less than duplication of the whole system.

Thus our approach should be used for both: making standard computers malfunction tolerant, while special purpose system fault tolerant using malfunction tolerant hardware elements.

References

1. Avizienis A, Gilley G, Mathur FP, Rennels D, Rohr J, Rubin D (1971) The star (self-testing and repairing) computer: an investigation of the theory and practice of fault-tolerant computer design. IEEE Trans Comput 20(11):1312–1321
2. DeAngelis D, Lauro J (1976) Software recovery in the fault-tolerant space borne computer. FTCS-6 26
3. Schagaev I (1986) Algorithms of computation recovery. Automat Remote Control 7
4. Schagaev I (1987) Algorithms for restoring a computing process. Automat Remote Control 48 (4)
5. Schagaev I et al (2001) Redundancy classification and its applications for fault tolerant computer design. In IEEE proceedings of man system cybernetics, Arizona Tucson
6. Avizienis A (1985) Architectures of fault tolerant computing systems, 1975. FTCS-5. In 5th international symposium, pp 3–16
7. Laprie J-C (1984) Dependability modeling and evaluation of software and hardware systems. In: Fehlertolerierende Rechensysteme, 2. GI/NTG/GMR- Fachtagung, pp 202–215, Springer, London
8. Laprie J-C et al. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secure Comput 1(1):11–33
9. Schagaev I (2008) Reliability of malfunction tolerance. In International multi-conference on computer science and information technology. IMCSIT 2008, pp 733–737
10. O'Brian F (1976) Rollback point insertion strategies. In Digest of papers 6th international symposium on fault-tolerant computing, 1976, FTCS-6
11. Vilenkin S, Schagaev I (1998) Operating system for fault tolerant SIMD computers Programmirovanie, (No. 3)
12. Birolini A (2014) Reliability engineering theory and practice, 7th edn, Springer, London
13. Castano V, Schagaev I (2015) Resilient computer system design. Springer, London ISBN 978- 3-319-15068-0

Chapter 5
GAFT Generalization: A Principle and Model of Active System Safety

Abstract Fault tolerance so far was considered as a property of a system. In fact and instead, we introduce a Generalized Algorithm of Fault Tolerance (GAFT) that considers fault tolerance as a system process. There is no doubt we have to analyze GAFT implementation using redundancy types of computer systems. Properties of GAFT implementation impact on overall performance of the system, coverage of faults, and ability of reconfiguration.

It is clear that separation of malfunctions from permanent fault simply must be implemented and reliability gain is analyzed. A ratio of malfunctions to permanent faults is achieving 10^{5-7} and simple exclusion from working configuration a malfunctioned element is no longer feasible.

In this chapter, we consider GAFT extension in terms of generalization and application for support of system safety of complex systems. Our algorithms of searching correct state, "guilty" element, and analysis of potential damages become powerful extension of GAFT for challenging applications like avionic systems, aircraft as a whole.

Until now, we were introducing the GAFT as the algorithm of achieving property of fault tolerance in a system as *a process*. What we did not yet cover is how to figure out a system approach or method of finding which hardware or software components are affected by a fault. We want to tackle this problem in this chapter.

The Method of Active System Safety (MASS) [1] provides a mathematical approach for the diagnosis and prognosis of faults in real time during system execution. It is based on the description of a system (of objects) and the dependencies between components and elements of the system represented as a matrix.

Every entry in the dependency matrix represents a dependency of one element to another. Represented as a graph, dependencies are just directed edges between elements. Two elements *a* and *b* can also be mutual dependent on each other which is represented by two edges with opposite direction between these elements.

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_5

Each edge has an assigned weight, corresponding to the probability that "element ***a*** has an influence on element ***b***." Note that influence and dependence might not be symmetrical.

Every element in the system is described by a pattern function that compares the received (monitored) data with predicted data. During system operation when a discrepancy is detected between real and the predicted (expected) data, within a given threshold, the MASS performs the following steps:

- Using the dependency matrix, starting from the "suspected" element, all possible scenarios, i.e., the full graph of all dependencies the suspected element has, are evaluated and their probability is calculated. Probabilities along the paths are cumulatively multiplied. The resulting scenarios are then ranked according to likeliness using a predefined threshold;
- The structure of the dependency matrix and all probabilities are assumed to be constant during the graph unfolding;
- The resulting graph with the suspected root element is then treated as a fault tree and evaluated either automatically or by a human to find the real reason for the detected anomaly.

Hidden faults, faults that do not immediately result in errors but are triggered later in time, must be given special attention. The introduced fault latency occurs because the manifested discrepancy between the real and the expected outcome of an element (i.e., program output) might be either wrong due to wrong behavior of a suspected element itself or due to faulty elements the suspected element depends on.

In other words, an element which shows faulty behavior is not necessarily faulty, and the real source could also be another element that influences the first element. We therefore introduce a search algorithm that recursively goes through all dependencies starting with the suspected one.

The algorithm uses the assigned probabilities on the dependencies to calculate the likeliness that an element is the source of the fault resulting in a list of possible sources and their likelihood. Every element of this list is then checked to efficiently find the real cause.

We propose to use the dependency matrix as data structure to store all element dependencies. Fault tree schemes used for description of a system for diagnostic purposes do not look like reasonable alternative to the dependency matrix for several reasons:

- Fault trees need more space and more effort to create them manually.
- Fault tree is static by design and for each entry—new case of fault we have to have new fault tree formation.
- Application of fault tree in real time of system operation is impossible.

We have proposed a dependency matrix with introduced GAFT. This matrix is then used to derive the fault tree and all dependencies in real time [1].

Furthermore, MASS assumes concurrent processing of present and incoming information about system elements to update the dependency matrix after each iteration of analysis. Therefore, MASS executes the following algorithm, introduced below as a special section.

5.1 GAFT Extension: The Method of Active System Safety

Method of active system safety assumes existence of internal knowledge about an object in the form of dependency matrix.

- Formation of initial dependency matrix for a node using expert opinion;
- Assigning for each link between elements a direction and weight (probability);
- Formation of a list of possible recovery actions for every element;
- Set threshold ε to analyze and terminate dependencies;
- Set threshold ρ to analyze and terminate backward search;
- Repeat: During application execution, validate the program output according to application dependent rules;
- If discrepancy is detected, then
  - Use dependency matrix to build a dependency graph using the suspected element as root node. Multiply probability along edges and stop when the probability is smaller than ε;
  - Form a sorted list of all scenarios according to the probability;
  - Start backward search of all possible faulty elements that are predecessors of the suspected element;
  - Continue until multiplicative probabilities of dependencies are bigger than ρ;
  - Form list of possible reasons of fault manifestation;
  - Choose further actions from list of recovery;
  - Use new node data and already existing data to update the dependency matrix.

This sequence allows controlling an object in case of deviation of behavior and recovering its condition or performing graceful degradation when full recovery is impossible.

5.2 GAFT Derivation: A Principle of Active System Safety

This section introduces the Principle of Active System Safety (PASS) as logic derivative of GAFT. In this approach, all key elements of a system (hardware and software behavior) are modeled as a set of interconnected elements with dependencies reflecting the impact on each other from the recovery point of view.

Schagaev et al. [1] applied the principle of Active Safety to aviation. In their approach, information about the proper functioning of an airplane component is monitored in real time and compared to a list of fault symptoms.

The idea is then that deviations from the expected behavior can be used to detect malfunctions in these aircraft components in *real time* of operation. This principle can easily be applied to other systems such as a computing system. In this case, the malfunctions are not detected in aircraft components, but hardware components.

Applied to a software system, the "components" are tasks, which communicate with each other and thus depend on each other and also depend on the hardware they use. In fact, MASS and PASS could be considered as *an extension of system-level generalization of GAFT.*

The questions that arise are the following:

– How can the component that is responsible for a deviation be found?
– How can the fault type (permanent fault or malfunction) be determined?

True, the location of faults and faulty element(s) from the set of their manifestations is one of the fundamental problems of complex technical systems [2].

The complexity of the problem can be illustrated by the fact that literature often uses simplifying assumptions that eases the analysis of the problem and allows to introduce solutions which fit the assumptions but unfortunately not the real world.

Solutions to the fault localization problem, for example, are in literature based on very strong assumptions about the system and a priori information about the set of possible faults and the fault modes [3]. Often faults are considered as so-called simple faults, i.e., single non-repeatable ones or mutually independent ones.

However, the miniaturization and imperfection in current technologies and manufacturing processes force us to face the situation where multiple faults of elements must be considered if we are willing to make practically useful solutions, based on type of fault detection and further malfunction analysis and toleration.

The interdependence between impacts is reflected in the matrix of mutual dependence, the so-called Dependency Matrix that uses a directed graph to represent this information.

The alternative way to react to the object's condition, composed of the condition of all its elements, is defined in the Recovery Matrix. When a particular fault occurs, the Recovery Matrix allows to analyze and describe "what the system is going to do." There is a close relationship between the nodes of the Dependency Matrix and those of the Recovery Matrix.

Figure 5.1 illustrates how the three features behave in relation to each other. Each particular condition of the node might in some way be related to the collected parameters. In turn, a trend can be detected if the data of several nodes are taken together [1].

5.3 Dependency Matrix

The Dependency Matrix D defines dependencies between elements of the system. A node of the dependency graph corresponding to its dependency matrix represents every element.

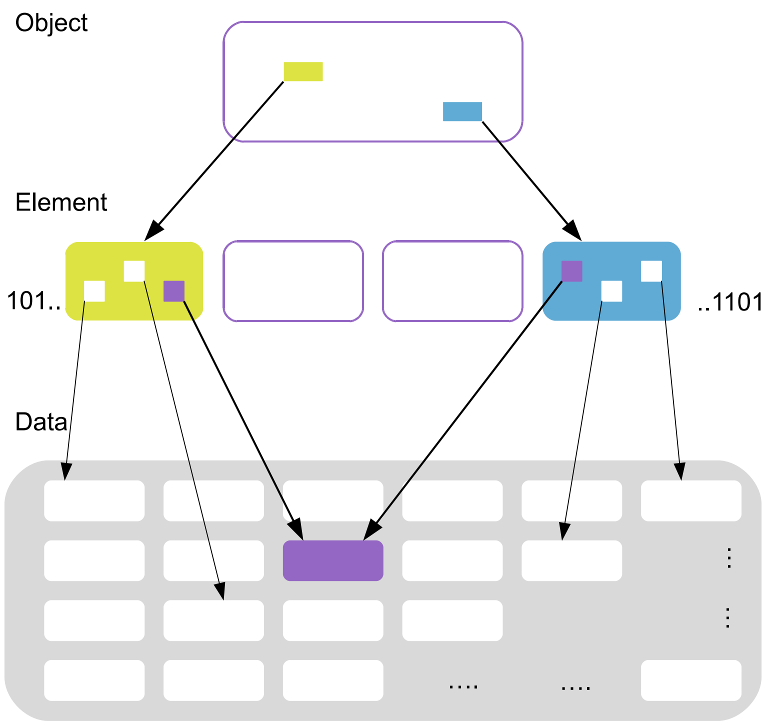


Fig. 5.1 Dependencies of data and elements

A simple example of a dependency graph is shown in Fig. 5.2 and its corresponding dependency matrix D in Table 5.1 as a two-dimensional square matrix of k columns and rows. Such a dependency matrix can cover k elements.

The assigned probability reflects the probability that a fault in one element affects another element which depends on the first. In other words, for elements i and j, two probabilities p_{ji} and p_{ij} are defined.

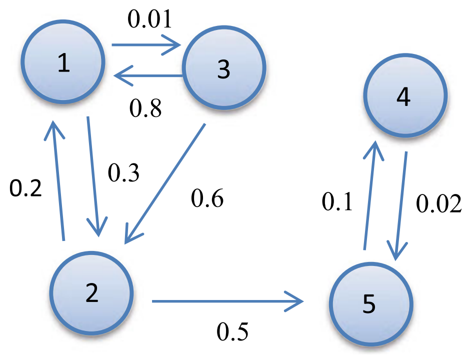


Fig. 5.2 Dependency graph example

Table 5.1 Dependency matrix example

	1	2	3	4	5
1	1	0.3	0	0	0
2	0.2	1	0	0	0.5
3	0.8	0.6	1	0	0
4	0	0	0	1	0.02
5	0	0	0	0.1	1

If p_{ji} is the probability that element j induces a fault in element i, one cannot conclude, in general, that $p_{ij} = p_{ji}$ as element i might depend on element j but not vice versa.

The sum of all probabilities of element i is also not necessarily 1 as the dependencies to other elements are independent of each other.

An even simpler version of the dependency matrix contains a 1 in location i, j if element e_i depends on element e_j, else 0.

However, this version of the matrix is usually not used as it is too limited in power. It is possible to apply statistical analysis to the dependency matrix in order to adjust the element dependence probabilities with newly discovered ones and possibly excluding existing ones that have become obsolete [4].

Thus, every element dependency should be updated autonomously after each detected error.

Such a dependency matrix is a powerful tool used in two algorithms:

1. The search for possible consequence a fault in one element can have on others.
2. Determination of which element originally caused the fault.

In the first case, the dependency matrix is used to make a prognosis of how the fault spreads in the system.

Therefore, if a fault is detected in element i, it is possible to derive the set of elements that still can be trusted and the set of elements that might be affected by the fault. In the second case, the dependency matrix is used to derive the set of elements which might have caused the fault in element i and the set of elements which did not cause the fault as element i does not depend on them, neither directly nor indirectly.

5.4 Recovery Matrix

The Recovery Matrix RM defines the actions to the detected or suspected faults and has the same dimensions (N×N) as matrix D. Each cell of RM contains two entries: first, the program or program entry point that is executed when the respective element and second, the rules used to decide whether an attempt to recovery is made.

A recovery attempt might not necessarily lead to a fault-free system. A recovery procedure is therefore considered to be successful if the probability of recovery P_{ji} is high.

In other words, $P_{ji} \gg 1 - P_{ji}$.

5.5 PASS Tracing Algorithm

Having introduced the method of active safety, the question arises now how to turn this process into useful algorithms for detecting possible consequences and possible problem sources in case of errors. We assume that the probability on the edges of every node in the dependency graph is ≤ 1.

Starting from a suspected node, the PASS algorithm evaluates the possible paths and ranks them according to their possible consequences in terms of safety (risk and potential damage). The tracing for every path continues until the multiplicative probability along it is less than the threshold ε, where the value of ε is statically set using engineering expertise.

The value ε does not change during the lifetime of a system. For every found path, the elements along the path are added to the set of potentially unsafe consequences. The algorithm uses two distinctive processes, namely, Forward Tracing and Backward Tracing [4]. Both of these processes are applied to the Dependency Matrix.

5.5.1 Forward Tracing Algorithm

The forward tracing algorithm is used to find all possible consequences of a fault in the suspected element *i* on the other elements in the system. This algorithm has two termination conditions: First, the algorithm stops if every node in the dependency graph (described by a matrix ($N \times N$) with *n* elements) has been visited and processed.

Cycles in the graph must be given special attention in the implementation of the algorithm for this condition to hold. The second condition is the probability of the analyzed path. The algorithm stops if the probability is less than ε.

We define the probability of the paths from the suspected node d_i to node d_j as $\prod(p_{i,j})$. If multiple paths lead from node d_i to node d_j, all possible $\prod(p_{i,j})$ are ranked and the nodes along the paths are included into the set of suspected nodes. Figure 5.3 describes the forward tracing algorithm in more detail.

Initially, every node in the graph is considered as not affected by the fault and the result set D_s is initialized to the empty set (not shown in the code). For the suspected node, the probability is initialized to ε; all nodes of the graph are put into a priority queue Q. We assume that the queue has a function called GetMax, which

```
01 PROCEDURE Tracing ($s$, $D(N)$ , $D_s$ , $\prod(p_{s,x}), xR_s$ )
02 Input: Dependency matrix $D(N)$ with $N$ elements of a weighted connected graph $G =< V, E >$
03 Input: The start node $s$
04 Output: The set of nodes $D_s$, where $\prod(p_{s,x}) > \epsilon$
05 Output: The highest probability $\prod(p_{s,j})$ of node $j$ reached by node $s$
06
07 $Q$; (* a priority queue sorted in descending order of the probability of nodes reached by $s$ *)
08 $L$; (* the set of already visited nodes, used to avoid tracing loops *)
09 Initialize ( $Q$ ); (* initialize nodes priority queue to the empty queue *)
10 FOR each node $v$ in $V$ DO $p_{s,v} \leftarrow \epsilon$; (* set default probability to $\epsilon$*)
11    Insert( $Q$ , $v$ , $p_{s,v}$ ) (* initialize the priority queue *)
12 END;
13 $p_{s,s} \leftarrow 1$; Increase( $Q$ , $s$ , $p_{s,s}$ ) (* update priority of $s$ with $p_{s,s}$ *)
14 $D_s \leftarrow Empty$ (* presume all elements are safe *)
15 $L \leftarrow Empty$
16 FOR $i \leftarrow 0$ TO $N - 1$ DO
17    $a^* \leftarrow GetMax(Q)$ (* get the element with the highest priority *)
18    IF $p_{i,a^*} > \epsilon$ THEN
19       $D_s \leftarrow D_s \cup a^*$; $L \leftarrow L \cup a^*$; $\prod(p_{s,a^*}) \leftarrow p_{i,a^*}$
20       FOR every node $a$ in $V-\ D_s-\ L$ that is adjacent to $a^*$ DO
21          IF $p_{s,a^*} * D_{a^*,a} > p_{s,a}$ THEN
22             $p_{s,a} \leftarrow p_{s,a^*} * D_{a^*,a}$;
23             Increase( $Q$ , $a$ , $p_{s,a}$ )
24          END
25       END
26    ELSE (* corresponds to $p_{i,a^*} < \epsilon$*)
27       EXIT;
28    END
29 END
```

Fig. 5.3 Forward tracing algorithm

returns the element with highest probability and removes it from the queue. This node is then updated with the function "increase" by the adjacent nodes.

Initially, the first tracing node ***s*** is the active node. Inside the tracing loop, for every node, the probabilities for all possible paths are calculated. The highest probability for the traces defines the start for the next step of the algorithm; the node along the highest probable trace is assigned as active and deleted from the queue. Therefore, the probabilities of all the adjacent nodes (minor to major (i.e., high risk)) with active node are calculated.

To avoid looping during tracing analysis, the adjacent nodes that have already been visited are excluded from further tracing (line 17) during each particular analysis of the matrix. When a loop is detected, a production $\prod(\mathrm{p_{i,j}})$ is calculated excluding the last probability. All the remaining nodes will be traced; probabilities along their paths (from starting node) are updated (line 20, 21).

As already mentioned, tracing terminates when the probabilities $\prod(\mathrm{p_{i,j}})$ of reaching the remaining nodes are less than ε. To summarize, starting from the "suspected" element all possible paths and their probabilities are traced, their cumulative probabilities are calculated, and their sequences are ranked according to their relative probabilities.

Relating to standard algorithm and data structure theory, this algorithm is basically a modified version of a breadth-first search.

5.5.2 Backward Tracing Algorithm

The backward tracing algorithm is used to determine which elements originally might have caused a fault in the suspected element s. This is required for two reasons: at first, not all elements might have good quality of sensors or precise models and a fault of an element might not be manifested as a data corruption or corrupted behavior at another node.

Second, recovery, if possible, should be started not from manifested element but from the faulty element—i.e., the reason, treating the reason not just the symptoms. Reverse tracing of dependency matrix allows finding the elements, which are likely to be the cause of the faults manifested by the suspected element.

The Backward Tracing Algorithm is described in pseudocode in Fig. 5.4 and is in essence equal to the forward algorithm except that all edges are traversed in the opposite direction.

If the result of the reverse algorithm shows that there was no fault propagation, then the suspected event is logged, but no further actions are taken.

If recovery actions are executed, these should be logged to the system log, for later traceability of occurred events. The PASS algorithm itself is triggered either

```
PROCEDURE Backward Tracing ( $s$ , $R(N)$ , $S_s$ , $\prod(p_{s,x})$ , $xR_s$ )
Input: Recovery matrix $R(N)$ with $N$ elements of a weighted connected graph $G =< V, E >$
Input: The suspected node $s$
Output: The set of nodes $S_s$, where $\prod(p_{x,s}) > \epsilon$
Output: The highest probability $\prod(p_{j,s})$ of node $j$ reaching node $s$

$Q$; (* a priority queue sorted in descending order of the probability of nodes reaching $s$ *)
$L$; (* the set of already visited nodes, used to avoid tracing loops *)
Initialize ( $Q$ ); (* initialize nodes priority queue to the empty queue *)
FOR each node $v$ in $V$ DO
    $p_{v,s} \leftarrow \epsilon$; (* set default probability to $\epsilon$*)
    Insert ( $Q$ , $v$ , $p_{v,s}$ ) (* initialize the priority queue *)
END;
$p_{s,s} \leftarrow 1$; Increase( $Q$ , $s$ , $p_{s,s}$ ) (* update priority of $s$ with $p_{s,s}$ *)
$S_s \leftarrow Empty$ (* presume all elements are safe *)
$L \leftarrow Empty$
FOR $i \leftarrow 0$ TO $N-1$ DO
    $a^* \leftarrow GetMax(Q)$ (* get the element with the highest priority *)
    IF $p_{i,a^*} > \epsilon$ THEN
        $S_s \leftarrow S_s \cup a^*$; $L \leftarrow L \cup a^*$; $\prod(p_{a^*,s}) \leftarrow p_{a^*,i}$
        FOR every node $a$ in $V-$ $S_s-$ $L$ that is adjacent to $a^*$ DO
            IF $p_{a^*,s} * R_{a,a^*} > p_{a,s}$ THEN
                $p_{a,s} \leftarrow p_{a^*,s} * R_{a,a^*}$;
                Increase ( $Q$ , $a$ , $p_{a,s}$ )
            END
        END
    ELSE (* corresponds to $p_{a^*,i} < \epsilon$*)
        EXIT;
    END
END
```

Fig. 5.4 Backward tracing algorithm

externally, or by a signal from an element, which is part of the model, indicating that there is a discrepancy in behavior of one or more elements.

5.6 Chapter Summary

In this chapter, we presented the model of active system safety, which applied to fault-tolerant systems serves as an extension to GAFT, i.e., it shows how to deal with latent faults that stay for a long time in the system until they manifest somewhere. To identify the true source of a fault, we propose therefore to map all elements of the system and its dependencies to a dependency matrix.

Faults can only be spread in the system along element dependencies; therefore, the shown tracing algorithms can trace a fault manifestation (error) to the true source of the fault. The appropriate recovery actions are stored in the recovery matrix, which is then executed to recover the system, i.e., eliminate the fault. In reverse direction, we showed that the tracing algorithm can also be used to derive the set of elements that might have been affected by the fault and run diagnostic routines to ensure the integrity of the elements.

Conclusion for Part I, Chaps. 1–5

- In Part I, we introduced the topic of hardware faults and showed that safety-critical systems, especially in high altitude, are prone to faults by external events and that the number of these elements is expected to rise in the future. Based on this finding, we using the standard reliability theory show how reliability and fault tolerance are connected, i.e., the deliberate introduction of redundancy increases reliability if the reliability benefit resulting from the redundancy scheme use exceeds the decrease in reliability due to the actual implementation of the redundancy scheme itself.
- We then presented our fault tolerance model, where we showed that the three models: system, fault, and fault tolerance are connected and only the thorough analysis of the system and the expected faults lead to the efficient fault tolerance solution. GAFT serves as a model and guide for the implementation of fault tolerance and defines all required steps to make a system fault tolerant.
- GAFT in a tabular form which serves as a template for the implementation of a fault tolerance solution. We defined that a system is fault tolerant when it implements every step of GAFT. We will further go into details about possible implementations in Part II of this work. Different fault tolerance solutions have different properties in performance, reliability, fault coverage, and cost and are constraint by the requirements of the usages scenario of the system.
- In our reliability evaluation of hardware redundancy, we clearly showed that there exists a optimum in achievable reliability gain using redundancy which is in fact less than duplication. Therefore, the usual approach to flight control systems in airplanes (Airbus, Boeing), i.e., massive redundancy using triplicated or even quadruplicated systems does not result in the most reliable solution.

– As an extension to GAFT, we then introduced the principle of active safety, which is able by modeling the dependencies of the hardware elements and using the introduced tracing algorithms to derive in case of latent faults the originating element of a fault and in reverse order the elements that might be affected by a fault.
– In essence, we introduced in Part I of this work the field of fault tolerance and give the user a guideline to the design of fault-tolerant systems.

References

1. Schagaev I et al (2007) Applying the principle of active safety to aviation. In: European conference for aerospace sciences (EUCASS)
2. Schagaev I (1989) Computing process recovery algorithms. Avtomat Telemekh (4)
3. Schagaev I (1998) Concept of dynamic safety for aviation. In: Proceedings of ISSC, pp 448–453, Seattle
4. Schagaev I et al (2007) Method and apparatus for active system safety, UK Patent GB 0707057.6

Chapter 6
System Software Support for Hardware Deficiency: Functions and Features

Abstract This chapter explains what is required to be changed or modified at the level of system software (runtime system, languages) to address new properties of computer system fault tolerance, resilience, and reliability. From the first principles, we think through the system design attempting to reflect new system properties "as early as possible".

The rapidly developing area of IT applications in safety-critical real-time embedded systems, up to now, was equipped by standard modifications of existing software (Win CE [1], VxWorks [2], Linux, …) or for safety-critical applications designed OS, such as Integrity OS [3].

Interestingly, no commercial fault-tolerant operating system is available on the market. Integrity OS, for example, which is certified to be used in airborne applications (DO-178B, SKPP, EAL6+ High Robustness, according to their homepage "the highest security level ever achieved for an operating system"), focuses as the name implies on the integrity of the OS and its applications, i.e., memory protection, guaranteed resources (memory, CPU), and as they call it highest reliability by guaranteeing system resources even in case of malfunctions.

The OS is responsible for isolating misbehaving applications due to malfunctions, but provides no further means of fault tolerance. The OS itself is thus prone to malfunction and can according to our definition (GAFT) not be called fault tolerant as it only provides application fault isolation.

None of these OS has been theoretically analyzed to answer the question what system software can and must do to meet real-time constraints, while still providing fault tolerance features of the system. This lack of theory is reflected in the misleading variety of solutions, huge overheads, and nearly impossible or prohibitively expensive maintenance. Therefore, the aim of this research is

> To design a new concept and its theoretical framework for further development of system software for safety critical real time embedded systems, assuming possible hardware deficiencies.

In fact, we should answer one question: What is system software able to do and should do to fulfill the two requirements—real-time processing and fault tolerance. To achieve this goal, one has to present constructive design principles to follow and apply at all steps.

A number of principles will be introduced on the way of further development, but the first principle of the system development here is simplicity. The rational behind this is obvious: complex system even without fault tolerance is hard to manage and understand, thus making the fault tolerant even harder. Simple systems, i.e., systems that use a simpler architecture but are still able to meet the system requirements are easier to implement, control, and make fault tolerant.

However, the newly developed system software will need some extra features, thus it will not be absolutely simple (not simpler than simple). But again, new features should be designed and developed with the same approach—simplicity, better say, maximum possible simplicity.

From all possible options to support the new features of the system (will be specified further), we will choose the simplest and the most promising choices. This means that if we have some options to realize a new feature at the language, OS or application program levels, we will always choose if possible the language level. This allows the omission of complex dynamics and uncertainty during program execution. At the same time, if a new feature requires runtime support, this support will be provided by the OS, again, by the simplest way.

Therefore, the additional overhead by the runtime support will be as minimal as possible. It is expected that performance and other additional overhead caused by the support of new features will not be significant and can be in the first consideration ignored. We call this principle essential redundancy.

In this work, system software is considered as the combination of a programming language and an operating system with runtime support for all languages features. System software for embedded systems usually provides some RT features.

Regarding RT, we will focus on the optimization of already existing system software specifications, efficiency analysis of existing solutions, and where possible the reduction of complexity in these solutions. The logic behind the research for new features is shown in Fig. 6.1 as a comparison of RT and FT.

(A possible realization with required features is shown in Fig. 6.1 with the following meaning of capital letters: HW—Hardware, OS—Operating System, AP —Application Program.)

Next, we present some comments on real-time and fault tolerance features and the supportive means by the operating system. Assume that a program requires RT access to program data.

To guarantee the required real-time constraints, it is important to exclude file structures, as these do, in general, not allow direct and equal (in time) access to the data. Instead, simpler data structures with guaranteed design equality to access each data element or record should be introduced.

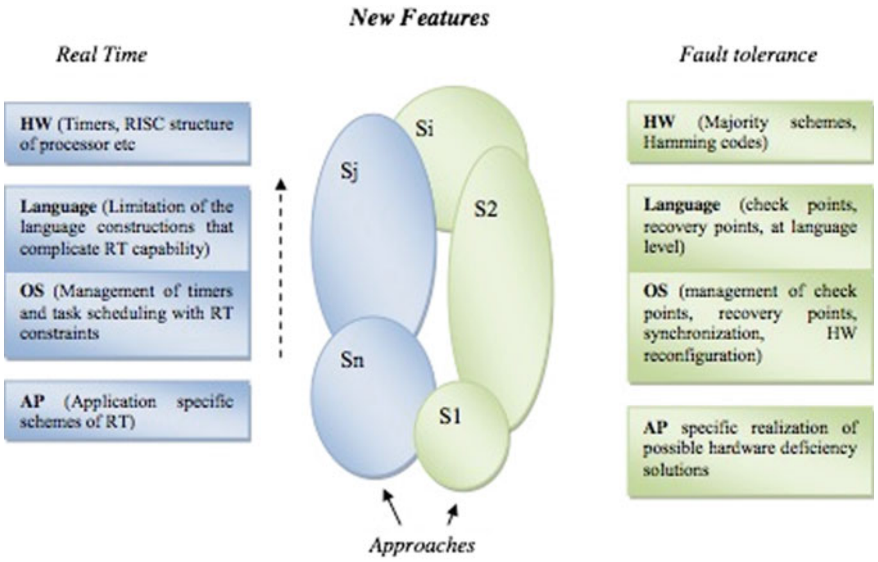

New Features
Real Time
Fault tolerance
HW (Timers, RISC structure of processor etc
Language (Limitation of the language constructions that complicate RT capability)
OS (Management of timers and task scheduling with RT constraints
AP (Application specific schemes of RT)
HW (Majority schemes, Hamming codes)
Language (check points, recovery points, at language level)
OS (management of check points, recovery points, synchronization, HW reconfiguration)
AP specific realization of possible hardware deficiency solutions
Si
Sj
S2
Sn
S1
Approaches

Fig. 6.1 System presentation and implementation hierarchy

Clearly, RT as new feature requires modification of almost all elements in Fig. 6.1. RT and FT are synthetic, not elementary feature of computer systems, and possible implementations can be located at different layers in system hierarchy.

Some well-known solutions exist for achieving RT in a computing system, such as limiting the use of data constructs, deliberate introduction of time-limit program control structures, exclusion of complex instructions from the processor architecture, limitation of pipelining, strong extension of timer schemes, etc.

The ellipses in the middle of Fig. 6.1 represent possible implementations of new features. As a general rule, the top-down principle should be applied, i.e., every new feature should be implemented at the top level of the system hierarchy if possible. This rule does also imply that implementations of new features like S_1, which is implemented on the application level, are excluded from this research.

Instead, we concentrate on new features implemented at the top layers of the presented hierarchy, assuming that the resulting dynamic system behavior will be under full control.

Let's take language features as an example. A programming language can be described by means of control structures, presentation of data types, and the realization of sequential and conditional expressions. For example, typical data structures are arrays, strings, files, and records (Fig. 6.2).

If hard real time is required, file data structures should be excluded from the set of possible data structures in RT applications because files do not guarantee equal access time over all files and all file data.

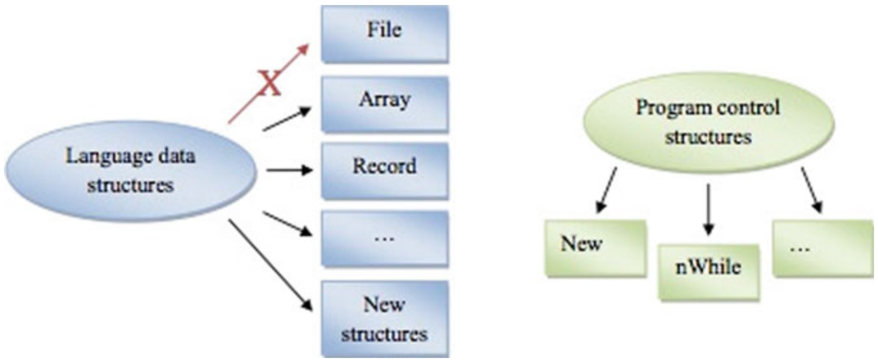


Fig. 6.2 Modification of data structure limited by application domain

New data structures with new features might also be developed. For RT applications, the language control structures might also be modified to provide a higher level of control and timing conformance during program execution.

A programming language can be described by means of control structures, presentation of data types, and the realization of sequential and conditional expressions. For example, typical data structures are arrays, strings, files, and records as above.

Implementation of the property limits or excludes some of currently “conventional” solutions or structures: files as data structure are no longer applicable and, for example, should be replaced by new array structure. The best-fit idea so far was suggested by Dr. Felix Friedrich ETH Zurich [4].

Implementation of this language data construct and support for it at the hardware level enables us to activate (and power) only sections of memory that will be in current use, thus providing when necessary energy-smart functioning.

At the same time, we are able to reconfigure data and duplicate data segments when we feel it is required for reliability purposes.

Above all access, with fine-tuning of addresses—smaller domain on Fig. 6.3 makes performance of program with this data structure an order of magnitude faster than even assembler code in conventional system.

Then development of data structures for evolving architectures would now seem to lay in extension of array structures with different data types of elements, where each element of data sequence is accompanied by type specifier providing enormous flexibility and even dynamic modification by the application or runtime system and hardware conditions during program operation.

For fault-tolerant systems, where external events cause internal malfunctions and/or necessitate reconfiguration for PRE requirements of applications, we have to avoid hangs of the program and other uncertainties of the system. Thus, language control operators and their interpretations through hardware instructions (as well as operators themselves) should be organized with the following objectives:

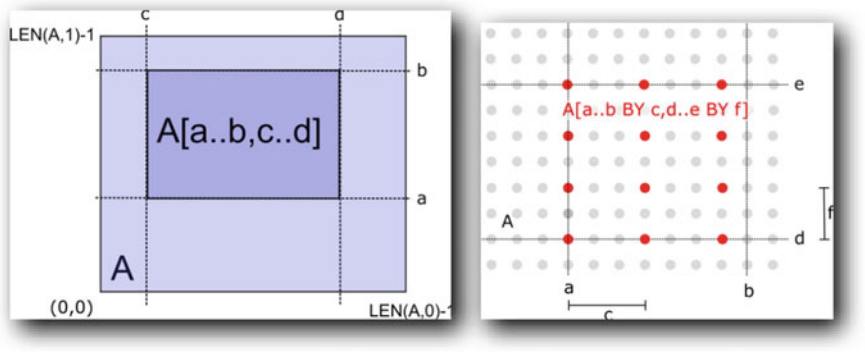


Fig. 6.3 Arrays with domain extraction [4]

(a) Minimization of the state space that needs to be saved before each instruction's execution;
(b) Simplest possible graphs of predicates, addresses, and data for program;
(c) Control of execution when hardware state changes during repetitive operators.

Clearly, even using a kind of standard program schemes, we have to control potential changes of control, condition of the data dependency graphs of the program (Fig. 6.4). Size of plane of i-th layer defines amount of states and parameters we have to save to provide an ability to return to the state of program.

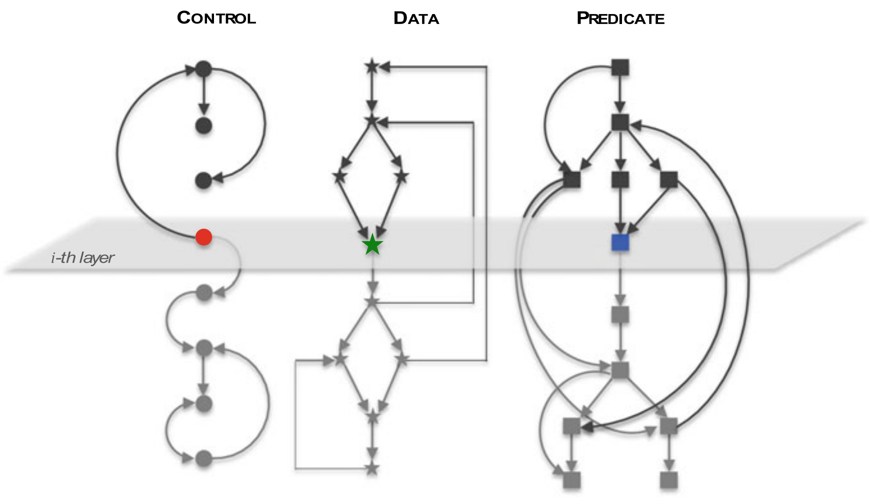
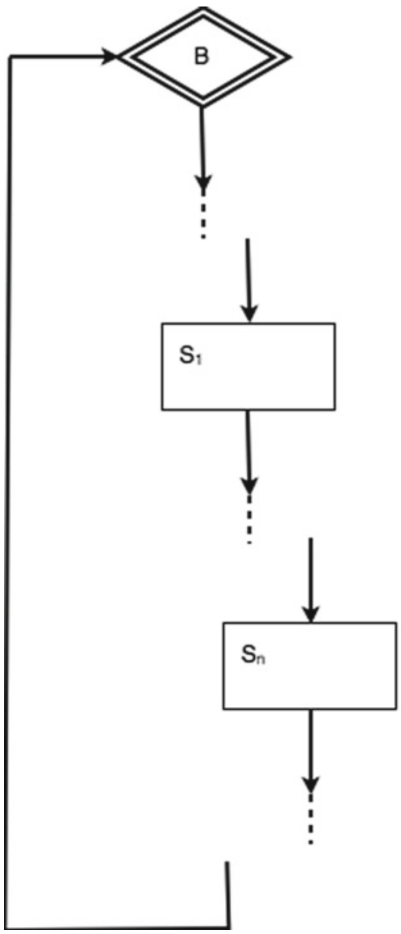


Fig. 6.4 Control, data, predicate model of program

Also, if possible, control structures should be simplified as much as possible due to the strict timing requirements and as small as possible overheads. One promising feature would be the *nWhile* [5] introduced by T. Takaoka. Thanks to Takaoka [with his idea of new *nWhile* loop we can introduce various conditions of loop execution inside the body of a loop, Fig. 6.5.

Introducing a pre-condition P and post-condition Q for this structure, Takaoka suggested to use several assignment statements S1, S2, S3, … Sn in S which affect the condition B, and therefore P1, P2, P3, … Pn that held immediately before S1, S2, S3, … Sn under pre-condition P.

What does it gives us in terms of fault tolerance and even system evolvability? Actually, a lot. It enables us to write a program using loop operators as usual, but during compilation of the program one is able to introduce other S2–Sn conditions



$$\frac{\prod_{i=1}^{K}\{P_i \wedge B\}S_i\{Q_i\}, \prod_{i=1}^{K}(\neg B \wedge Q_i \supset Q), P \wedge \neg B \supset Q}{\{P\} nwhileBdoS \{Q\}}$$

Fig. 6.5 Semantics of a control loop for evolving system

that might be connected with computer state changes including hardware faults, time-outs, or other interruptions, including interaction with other processes.

Then, if of the loop "hangs" due to problems within hardware an arrival of another signal able to break loop execution might become visible—hardware state change reflects immediately within the control construction and we are not using "brute force" of conventional waiting or wasting resources of another type.

FT as a new feature of embedded systems, implemented on the level of system software should be analyzed in some detail. First, let us consider fault tolerance as a sequence of steps as introduced by GAFT in chapter.

When we defined FT as GAFT it became possible to investigate how system software should be involved to realize this algorithm. Then, all required features and mechanisms of system software to support fault tolerance of embedded systems can be derived from GAFT.

There are several processes and functions in the system software to provide fault tolerance, Fig. 6.6.

Research in the area of program recovery after a hardware fault has occurred is known for 40 years. Google search shows millions of related links, including some even from Microsoft. But none have been found that is concerned with the actual semantics of checking and recovery at the system software at programming language level and runtime system level.

Regretfully, checking and recovery sometimes are used as synonyms. Checking is the process of analyzing the hardware state to prove an existence (or absence) of faults from predefined classes. In turn, recovery is the process of preparation and storing of several states of the hardware aimed at being able to recover from any predefined and arranged state or states.

Both processes, including recovery schemes, are important and unavoidable elements of GAFT. Their combination improves the actual reliability of RT FT systems and the efficiency of solutions taken at least at the model level.

Various schemes for program recovery after a hardware fault were introduced and described in [6]. Several methods of correct recovery searching with supporting hardware are described in [7, 8] including scheme of recovery point formation; algorithms of recovery and their efficiency, and management and efficient supportive schemes.

It is worth to mention that recovery points might be arranged at the level of tasks (that are mutually dependent either with data or control or both, or interrelated by conversations). These recovery algorithms become much more complex and almost prohibitively slow for RT applications. This kind of system recovery might be a

```
o Check points, RT control & checking;
o Hardware Reconfigurability;
o Recoverability and support of its hierarchy;
o Fault resistance for concurrent processing
```

Fig. 6.6 Processes and functions for system software to implement FT

subject of special research at a later date, taking into account complexity of the task dependency organization.

6.1 System Software Life Cycle Versus Fault Tolerance

The system life cycle of an FT system can be split into two phases: first, the development phase (design and implementation) and second the operational phase (deployment, operating use, and maintenance).

The first, and in terms of an FT system almost most important part is the design phase. In this phase, the application scenario is thoroughly analyzed and the system requirements are derived. In general, this conforms to the standard project development process [9], but in our case, we would like to focus on the fault-tolerant aspects.

Every scenario where a system is deployed has different requirements in terms of reliability, availability, and lifetime of the system. A satellite system, for example, has highest demands in all three properties: highest reliability as maintenance is almost impossible or extremely expensive, highest availability as satellite services are typically crucial (e.g., GPS system) for the users, and a long lifetime as they cannot easily be replaced.

It is obvious that these extreme requirements can only be met with a rigorous design and implementation phase. However, the higher the required level of reliability and availability, the higher is also the cost. In fact, the cost for the design and implementation of a dependable system rises exponentially with the degree of required dependence.

It is thus very important not to design a system that is as reliable as possible, as such a system would be extremely expensive, but to derive from the applications and the usage scenarios of such a system the required level of reliability.

As shown above, fault tolerance is the main method of increasing the dependability of a system, it is therefore necessary to keep during all phases of software development (specification, design, implementation, testing, deployment, and maintenance) fault tolerance in mind.

An example: humans are not perfect, this is a well-known fact. Therefore, a system does always have some internal faults, for example, software implementation bugs that are also not identifiable with verification tools. Individual applications should therefore be programmed in a defensive way, which allows a software component, for example, to detect invalid data or invalid software behavior that does not correspond to its specification.

Even though this might be an obvious way of how to write programs, it is nevertheless a method of fault tolerance. And it is only of use, if this principle is used in all applications and of course the runtime system itself.

System testing needs another word of explanation. In the standard development process, using automatic test cases or manual testing by an engineer tests the software and hardware. Testing of fault tolerance proves to be a challenge due to inability to repeat the same impact, for example, of alpha particles and the need to use a special purposes build radiation facilities.

To test the software, the use of a simulator often proves to be far cheaper and much more efficient as specific faults can be modeled in software. In this case, real radiation tests are only used to test the hardware and to verify the chosen fault models.

The second life cycle phase, namely, the operational phase of a safety-critical system is in principal similar to a standard embedded system with one big difference, where maintenance of a standard embedded system in case of failure is usually possible although costly, and the situation typically changes for safety-critical systems.

If one of these systems fails, the consequences can be the loss of human lives (failure of a flight control system in an airplane) or physical maintenance might not even be possible (satellite system). Therefore, maintenance, especially physical maintenance cannot be considered as a part of life cycle for this kind of systems.

The maintenance interval of a system is usually derived from the life span of such a system, in other words, from the MTTF of such a system, which can often not clearly be derived. An optimal solution would be a system that has the power of self-diagnosis up to the level where it can predict its failure in the near future. In this case, it could inform the operator which in turn could initiate the maintenance procedure before the system fails. This would lower maintenance cost significantly.

6.2 System Software Phases

System software consists of the operating system and all other hardware-related software, which provides services to user programs. It is thus responsible for abstracting and managing hardware resources and gives application software controlled access to these resources. Some examples are interruption handlers, memory management, device drivers, scheduler, garbage collector, etc.

These traditional system software components are already extensively explained in literature [10–12]; therefore, we do not go into too much details about these functionalities, but introduce here instead some new system software concepts which are required in the field of fault-tolerant computing.

As already shown, hardware is suspect to malfunctions and failures despite all efforts on the hardware side. Responsible of managing faulty hardware should be the operating system itself as application programs have no direct access to the hardware and usually lack information to deal with hardware errors. Thus, the hardware state (see Chap. 7) must be represented in the system software or even in the programming language. We state here that a fault-tolerant system must, according to GAFT, support the following three additional processes:

Checking. The system software must collaborate with the hardware to perform the system checking, i.e., show that the hardware is fault free. Checking procedures can either be executed by the hardware itself, or more thoroughly but also slower by software. It is up to the system software to perform the software-based checks and react on detected faults. Task base checking (see Sect. 7.1) is an example of when to perform the checks. As shown in Fig. 4.3, hardware-based checks, especially if they are performed in parallel to the current instruction execution, are ideal in terms of time overhead.

Reconfiguration. In case of permanent faults, it might also be necessary to reconfigure the system to exclude a failed component from the currently used working set. In that sense, the state of the software and hardware is directly dependent as hardware faults are reflected in software. If a failed hardware component is excluded from the working set, all the software components that use this hardware component must be reconfigured accordingly (see Sect. 7.4).

Preparation for recovery. The system software is also responsible for taking measures to react on faults. Thus, it is responsible for the preparation for recovery, which consists of creating recovery points. A recovery point is a snapshot of the whole or part of the system at a specific point in time. In case of a fault, the system can rollback to the stored snapshot and continue processing. Of course, recovery is only successful if the fault was not present in the system when the recovery point was created. Various implementation schemes exist for creating recovery points with various time and space constraints. See Chap. 8 for more details.

Recovery. If a fault is detected in the system, the recovery process is responsible for the elimination of the fault impact in hardware and software. This process depends therefore on the previous two processes for fault identification and elimination. Thus, after using the checking schemes to identify the fault and eliminate the fault impact, the software must in a second step be recovered to the last consistent state using the recovery points. See Sect. 9.1 for more details.

References

1. Microsoft Corporation (2008) Windows embedded CE overview. World Wide Web Electronic Publication
2. VxWorks (2008) Vxworks overview. World Wide Web Electronic Publication
3. Green Hills Software (2008) Integrity, the most advanced RTOS technology. Technical report, Green Hills Software
4. Friedrich et al (2006) Array-structured object types for mathematical programming. In: Lecture notes in computer science, vol 4228, pp 195–210. http://link.springer.com/book10.1007/11860990
5. Takaoka T (1986) The semantics of new while loop. Comput J 29(1)
6. Russell D, Tiedeman M (1979) Multiprocess recovery using conversations. In: Digest of papers FTCS-9, 9th international symposium
7. Schagaev I (1986) Algorithms of computation recovery. Autom Remote Control 7
8. Schagaev I (1987) Algorithms for restoring a computing process. Autom Remote Control 48 (4)

9. Zehnder C (2001) Informatik-Projektentwicklung. vdf - Hochschulverlag AG an der ETH Zurich, Zurich (in German)
10. Wirth N, Gutknecht J (1992) Project Oberon: the design of an operating system and compiler. Addison-Wesley
11. Silberschatz A et al (2002) Operating system concepts. Wiley
12. Stallings W (2008) Operating systems: internals and design principles. Prentice Hall

Chapter 7
Testing, Checking, and Hardware Syndrome

Abstract In previous chapters, we introduced the processes of checking and testing, the first of the three main processes of Generalized Algorithm of Fault Tolerance—GAFT. In this chapter, we further discuss the process of checking hardware, at first software-based hardware-checking and at second hardware-based checking. For the software-based hardware checking, we show what a software-based test should include, when they are the preferred choice over hardware-based checking schemes, and especially how such tests can be scheduled in the system without interfering with ongoing real-time tasks. Further to support handling of hardware-based checking, we introduce a new system condition descriptor—so-called a *syndrome*, and illustrate how it can be used as a mechanism to signal to the operating system the hardware condition, including manifestation of detected error. We then show the steps the runtime system performs to eliminate the fault and in case of permanent errors how the software can reconfigure the hardware to exclude the faulty element. We also explain in which cases software has to adapt to the new hardware topology. We start by explaining how software-based checks can be used to detect hardware faults. Runtime systems use online or offline scheduling mechanisms for task management of programs—own—system software ones and user application ones. Since [1–4] it is expected that runtime system provides a special session of tasks scheduling (offline or online during execution) for the purposes of diagnostic of hardware conditions—recall Apple and Microsoft system starting delays. Later for some systems that operate in domain of real-time monitoring scheduling of tasks, critical in time of execution especially criticality of hardware availability and efficiency of process scheduling become crucial. In turn, testing itself becomes "hot" in terms of required time and coverage of hardware. Thus in this chapter, we initially analyze simple sequences of testing of hardware elements of computer systems. Further, we introduce a concept of *transparent* for user application procedure of hardware testing. This enables to prove integrity of computer system hardware, and guarantee it within a reasonable time, without delay of service of execution of user tasks.

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_7

7.1 Hardware-Checking Process

Consider a sequence of tests and programs denoted as T and P in Fig. 7.1. The initial test T is executed before a task execution and guarantees the hardware consistency and integrity, i.e., it confirms that there is no hardware fault at that time in the system.

However, if a permanent fault happens, for example, a “stuck bit”, the effect of the fault is basically permanent. When a permanent fault occurs immediately after the first test or during the program execution, it might be in principle invisible for an arbitrary long time (latent period).

Therefore, a second sequential test is required right after the program execution to guarantee *that no permanent fault occurred* since the last test. For periodic tasks which are often used in control systems, we slightly adapt this scheme as shown in Fig. 7.2.

But what happens if a malfunction occurs during the execution of program P? The effect of the malfunction might not last until P finishes and T, therefore, cannot detect the fault, leaving the malfunction undetected at all.

Malfunctions can be detected by double execution of the same program with comparison C of the result and the result state space. Figure 7.3 illustrates this scenario.

It is important to note that for periodic tasks, the persistent state of the program, i.e., the program state which is used in the next computation as input data must also be compared, as malfunctions might affect data which is no longer used in the current computation but in the next.

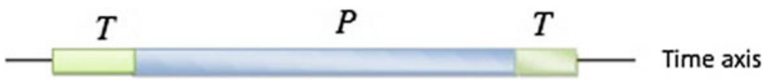


Fig. 7.1 Ensuring of hardware integrity through program execution

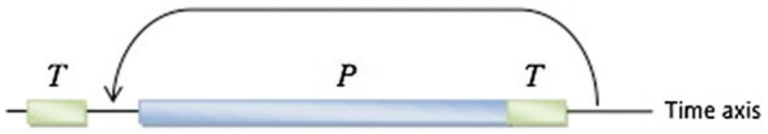


Fig. 7.2 Regular sequence of program with test of hardware integrity to detect permanent faults

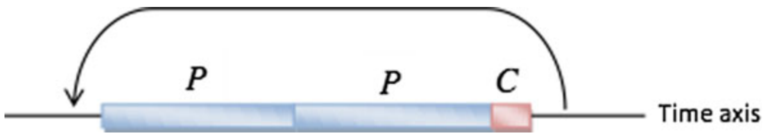


Fig. 7.3 Ensuring the hardware integrity to detect malfunction faults

Permanent faults, however, cannot be detected with the comparison scheme alone, as they might affect both executions of P. In other words, the scenario in Fig. 7.2 can only detect permanent faults whereas the scenario in Fig. 7.3 can only detect malfunctions.

The combined power to detect malfunctions and permanent faults is illustrated in Fig. 7.4 where C is used to detect malfunctions and T to detect permanent faults. Assuming that C triggers an error but T does not, it is clear that a malfunction occurred. Another run of program P with comparison to the previous two runs can identify the run where the malfunction occurred.

In the following analysis, we concentrate on the detection of permanent faults only and use only T in the analysis. The detection of malfunctions can be considered as included in the following analysis if the double execution of P with following C as a whole is treated as task P in the following analysis.

A testing phase is required initially at boot up time to guarantee the correctness of the hardware and also a periodic test before and after the execution of a program. The applied tests might vary in depth (coverage), type of faults and the set of the tested hardware.

Every hardware component has typically at least one assigned test but might also have more than one that could differ on the implementation level.

Software-based tests need a processor and memory for the test execution even if a peripheral device is tested. In order to guarantee that faults in other hardware components (that are not subject of the current test itself) do not have an influence on the outcome of the test, the order of the tests must follow the principle of growing core:

If a test of a hardware component u_i has implicit dependencies on another hardware component u_j, the test of u_j must be executed first.

If the resources needed by a task are known in advance, it is sufficient to run after the execution only the testing procedures of the accessed hardware resources (selective testing), again by using the principle of growing core.

This way, the system stays fully operational even in the case of present faults in some hardware components that are not in use. Spare components can be used for the relocation of programs that were running on faulty hardware components. Sure the state of a system should be reflected somewhere for convenience of timely decision if required.

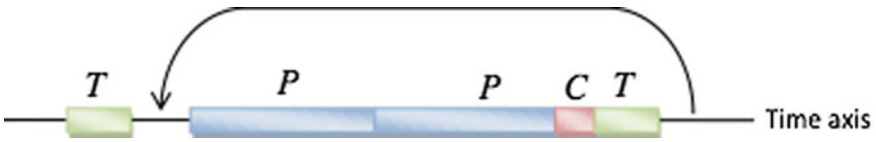


Fig. 7.4 Program repetition and hardware test detect both types of faults

Of course, it is also necessary, independently to this scenario, to test periodically the hardware in full as; otherwise, faulty spare components are considered as fully operational and might be used again in a subsequent reconfiguration process.

A full hardware test also allows the system software to monitor the current state of the hardware as a whole and take appropriate actions if necessary. If no spare components are available in the system, all programs depending on this component must be obviously terminated. If no essential program is affected by this component, the system can continue operating in a degraded mode.

For diagnostic and monitoring purposes, the results of the tests should be available for the software or even external systems. We propose therefore to organize the test results of hardware-based test in the so-called test *syndromes* (see Sect. 7.3).

For every hardware component, for example, the register file, ALU, internal bus or device controllers, the checking procedures present their result in the form of a syndrome to the software, indicating in binary form the state of the device. By grouping all syndromes together in one register, the software has a very effective way to check the integrity of the system. In case of a nonzero syndrome, further analysis of the hardware conditions is required, especially when the malfunction duration is long.

Dependent on the used hardware-checking scheme, it is not only possible to signal a fault to the runtime, but also provide more information to the runtime system to ease recovery. If, for example, the testing schemes discover stuck bits in memory, it is sufficient to recover programs that access the affected location and not all programs that are using this memory module.

Device drivers could, for example, provide their own testing schemes for their respective device. Especially for devices, one could think of having a combination of hardware and software-based testing. I/O devices such as UARTs could effectively be tested by cross connecting the input and output wires by very simple additional hardware logic and sending various bit patterns over this loopback connection.

Timely task completions in real-time systems is a key requirement; therefore, the testing overheads should be reduced as much as possible when and where possible. Figure 7.5 shows an example of three tasks with corresponding tests.

The assumption in this case is a time slice-based scheduler, which distributes time slices to the running processes. In this example, the processes run to

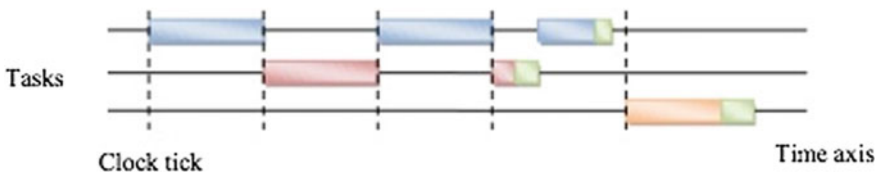


Fig. 7.5 Tasks and tests combined

completion and are called periodically by the scheduler. Three tasks are running, each with its own test (the green boxes) at the end of the task execution.

The test only checks the condition of hardware resources the respective process needs, which results in different test execution times. The task execution is only considered as successful if the test at the end of the task is successful.

If the test failed, the task is re-executed by using the same input data set as in the first try. Difficulties arise if the task performs I/O on hardware devices or communicates with other tasks, which we discuss in Chap. 8.

7.2 Analysis of Checking Process

Applications are nowadays so complex that they tend to saturate the computing system they are running on, which limits diagnosis possibilities. Especially in multiprocessor systems, a high interest arises to test some hardware units when other hardware units execute tasks. This approach has been called the Sliding Dropping Diagnosis (SDD) [5].

Multiprocessor architectures fit very well the scheme of concurrent software-based checking. In principle, two SDD types can be distinguished, namely, synchronous and asynchronous. An example: The CTSS Operating System for the CRAY-1 [1] supports synchronous SDD with interruption of user task for testing hardware. Periodically every 15 min, all tasks are interrupted and the diagnostic routines are executed. These routines are "invisible" from the user tasks point of view, as the entire system is stopped while the tests are executed.

The Synapse N+1 [2] fault-tolerant operating system serves as an example for asynchronous SDD. The OS implements a self-loading algorithm to schedule tasks for processors. The same algorithm can be applied for SDD too, as diagnostics routines could be loaded as tasks and run on free processors. The actions required for the implementation of the synchronous SDD mode are as follows:

- Unloading the current state of the currently running user process from some hardware parts to load diagnostic routines (RAM in a multiprocessor OS is an example of such a hardware unit).
- Loading and initialization of the diagnostic routines.
- Execution of the diagnostic process.
- Unloading the diagnostic routines and perform actions in case of found faults. If high priority interrupts occur during testing, some temporary data might be needed to continue the testing after processing the interrupt.
- Reloading of the user process to continue.

As asynchronous SDD does not interrupt running processes, but is allocated on free processors; only steps 2–4 are required for asynchronous SDD. It is important to see that the task unloading time is much higher than a task switch time, as the resources occupied by this task must be released, which might also involve moving the code and data of this application to another part of the memory. The same applies of course for reloading a task.

To further analyze the SDD timing constraints, a certain assumption about the task times must be made. In general, two cases must be distinguished: The task completion times are known, or the task completion times are unknown. In a time-sharing OS as well as real-time OS, the task time may not be known in advance. An SDD process that ends in checking all hardware units is called a diagnostic cycle and requires time Tc to complete.

We will further discuss the following three cases:

- Tc time and all task completion times are known.
- Tc time is known but the task completion times are not known.
- Tc time is unlimited and all task completion times are not known.

7.2.1 The System Model

The system model, diagnosis algorithms, and all analysis in this chapter are done on the basis of a multiprocessor system with a set of *U* identical processors (units). A single processor system is included in this analysis as it corresponds to the special case of

$$U = \{u_1\}$$

which would, of course, considerably simplify the analysis.

In addition, we assume that all scheduled tasks are independent, i.e., they have no time, control or information dependencies. Further, the task switch time (not to be mistaken with the task load and unload time in case of synchronous diagnosis) is considered to be so short in contrast to the task execution time that it can be safely ignored. We also suppose that at any given point in time a task is ready to be executed.

The assumptions about the hardware are simple: All hardware units are considered to be equal (equal processors) and are processing units. Storage units are not considered here as they are passive elements. Further, all involved hardware units become free immediately after the task completion. The runtime automatically assigns user application or diagnostic routine to a processor as soon as it becomes free.

We define the required time for a single unit (processor) to complete the asynchronous SDD as T_{ad} and to complete the synchronous SDD as T_{sd}. As all processors are considered equal, all T_{ad} and T_{sd} are equal as well. Both diagnosis modes share the common task of performing the actual test, whereas the synchronous mode also involves unloading and reloading of the interrupted task.

T_{sd} is therefore calculated as follows:

$$\boldsymbol{T}_{sd} = \boldsymbol{T}_{ad} + (\boldsymbol{T}_u + \boldsymbol{T}_r) \tag{7.1}$$

where T_u is the time of unloading, and T_r is the time of reloading the user task in synchronous diagnostics. In order to have an upper bound on the checking overhead at any given point in time, we propose here that at any given moment in time, the diagnostic process can run on at most one processor.

Otherwise, as shown above, by accident, a simultaneous testing of all processors would turn the system completely unresponsive. In this spirit, it is also necessary to relax the strict order of the testing explained in Sect. 7.2, and allow the testing to be done during task execution.

The recovery in this case would involve higher cost as the outcome of the last task run cannot be trusted and must be repeated as well.

7.2.2 Diagnostic Process Algorithm

Suppose a computing system consists of n processes. If only synchronous SDD is used, a loop embedded in the firmware or the task scheduler chooses the next unit, which will be diagnosed by chance. The choosing overhead itself is considered as negligible.

The real value of the asynchronous SDD depends on the sequence in which the units are diagnosed. The most natural way to appoint a diagnostic process for a unit is when the unit gets free. However, if a long-running task is assigned to a single processor, it is not diagnosed for a long time. Therefore, in this special case, the synchronous SDD are preferable.

On the other hand, time-critical tasks in a real-time system should not be interrupted, and thus the asynchronous SDD should be applied. To decrease time overheads but still provide completed testing, a combination of both diagnosis modes seems to be ideal.

According to Blazewicz [3], even for the simple case of the problem P || Cmax where a set of independent tasks is to be scheduled on identical processors without preemption in order to minimize schedule length can be proved to be a problem of complexity NP-hard [6]. Thus, our problem which imposes further restrictions on the scheduling algorithm and is therefore is also NP-hard cannot be perfectly solved in a real system with a high number of tasks. The use of an optimization algorithm seems to be therefore the way to go.

In general, two main scheduling principles can be used: offline scheduling and online scheduling. In offline scheduling, the whole schedule is computed during the system design phase. In this case, the time efficiency of the used scheduler is of no importance, and therefore, more complex algorithms can be used to calculate the schedule. Chances to get a schedule that satisfies all constraints are higher, and it is relatively easy to incorporate constraints such as deadlines or precedencies. During program execution, the tasks are executed according to the precomputed schedule.

In online scheduling, the decision when to run a task is done at runtime. The scheduler has more work to perform at runtime and might find a proper schedule that could be found with offline scheduling. In addition, it is also more difficult to take all task constraints into account.

However, online schedulers are much more flexible to match requirements of performance—or reliability—or energy-smart functioning. Under reliability “flag”, we can consider an ability of the system to react on unforeseen or exceptional situations. Dynamic allocation of new tasks for testing purposes introduced by just standard user or us here tasks both at runtime is also only possible using online scheduling. In this first example, we would like to use static scheduling, as this better shows our approach.

Consider now case (a) of Sect. 7.2 where T_c and all task completion times are known. Assume there exists a deadline for a maximum value of T_c.

In other words, in the time interval 0 … T_c, all hardware units (processors) of the set U must be diagnosed where T_c has a given upper bound. We now divide the set of units $U = \{u_i, \ldots, u_n\}$ into two subsets U_1 and U_2 taking the relation between the user task finishing time T_{compl} and T_c into account.

In a first step, we add every unit u_i in U either U_1 ($u_i \in U_1$) if the completion time t_i of the task running on unit u_i satisfies $t_i + T_{ad} \leq T_c$, or we add u_i to U_2 ($u_i \in U_2$) if $t_i + T_{ad} > T_c$. Obviously, it is impossible to test any unit in U_2 asynchronously, so this step seems logical.

But even if $U_2 = \{\}$, it is not possible to reliably guarantee testing of all units in U_1 asynchronously. In fact, when a user task has finished on one unit while a testing process is running on another processor, the scheduler will assign a new task to the just released processor according to the constraint that only one diagnostic procedure should run concurrently. This case is triggered if one task finishes while another is diagnosed: $t_j - t_i < T_{ad}$.

Even though both tasks finish before $T_c - T_{ad}$, there is not enough time left to test both in asynchronous mode. Therefore, the separation of units to two subset U_1 and U_2 is necessary but not sufficient.

We present below the procedure T1 that chooses a subset of units R for thef synchronous SDD in accordance with T_c and the task completion times. The subset $R \subset U1$ formed by procedure T1 is tested synchronously as well as all units from subset U_2. $R \cup U_2$ is, therefore, tested synchronously.

To find a solution to the mode problem for every unit of the system, the user should consider two extreme positions. The first concerns with the minimization of the diagnostic time boundary, i.e., the length of the time span in which all units should be diagnosed and the second deals with the minimization of the performance overheads, i.e., the minimization of the number of required task load and unload operations and having only one checking process running at the time. We propose here a natural approach to this problem.

7.2.3 Procedure T1

First, it is necessary to sort the units $u_i \in U$ in increasing order according to their completion time. This step is skipped in the algorithm as any standard sorting algorithm could be applied. Keep in mind that all units (processors) are identical, which means that T_{ad} is constant for all units.

The most time-consuming part of the algorithm T1 (Fig. 7.6) is the sorting which has an average complexity of *O(nlog(n))* [7].

We discuss the introduced algorithm according to the example in Fig. 7.7. First, we discuss case (1) where T_c and all task completion times are known. In this case, it is heuristically checked if $t_i + 1 > t_i + T_{ad}$. If this is the case (case a. of Fig. 7.7), no testing process will be ongoing when the task t_i ends, and therefore, the testing process can immediately be started. This case corresponds to step 3 of procedure T1, when unit $u_j = u_{i+1}$ is set to asynchronous testing mode.

Now, if this condition does not hold (case b. of Fig. 7.7), the task on unit $u_j = u_{i+1}$ finishes while the unit u_i is diagnosed. Remember that the task on unit u_j

Fig. 7.6 Procedure T1

```
$i = 1, k = 1, R = \{\}, j = i + 1;$
1. if $t_i <= T_c - T_{ad}$ then
    2. while $t_j \le T_c - T_{ad}$ do
        3. if $t_j > t_i + kT_{ad}$ then
            $k = 1; i = j;$
        else
            4. if $t_j < t_i + kT_{ad} - (T_u + T_r)$ then
                assign user task to $u_j$;
                set new completion time of $u_j$;
                restore ordering;
            else
                5. $k = k + 1;$
                if $t_i + kT_{ad} > T_c$ then
                    $u_j \in R;$
                end;
            end;
        end;
        $j = j + 1;$
    end
end

if $j < n$ then
    $u_k \in R, k = j \dots n$
end;
```

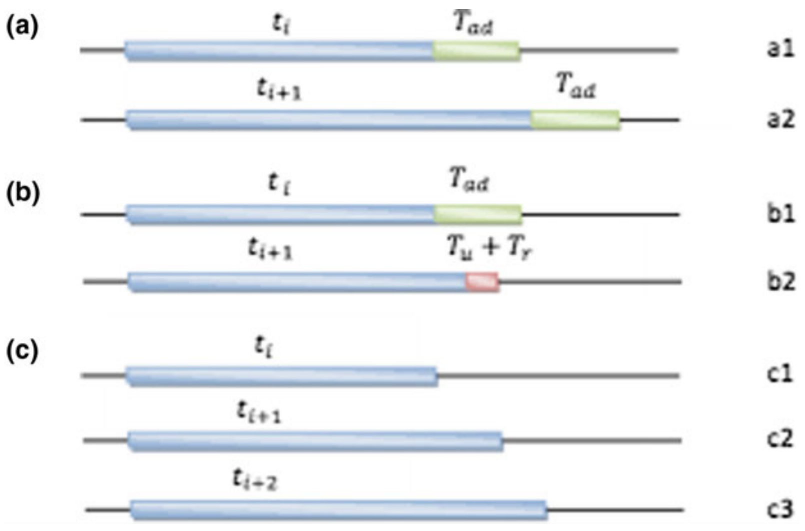


Fig. 7.7 Task example 1

cannot finish before the task on unit u_{j1} as all tasks are ordered according to their finishing time.

If the task time including $T_u + T_r$ of this task is still smaller than $T_{compl.i} + (T_u + T_r)$ (line b2) then the idle time of this task is considered as too long or in other words the performance impact would be too high, and thus a new user task is assigned to u_j.

Of course, the finishing time of u_j has to be adapted as well. In the next step, the just updated u_j must be placed at the correct position in all units to restore the ordering property.

For the cases where we still have no decision whether a test should be performed synchronously or asynchronously, i.e., $t_i + T_{ad} - (T_u + T_r) \leq t_i + 1 < t_i + T_{ad}$, the decision is made easy.

If enough time is available to perform the testing asynchronously it is done so, otherwise synchronously (step 5 of Fig. 7.6).

Even if u_{i+1} is scheduled for synchronous testing and has to be included in set R, the time overheads will be less than with asynchronous testing but more waiting takes place.

We just illustrated the algorithm on the example of two threads. The algorithm T1 in Fig. 7.6, however, was extended to support n units. It is also explained below why not the true completion times of the tasks are used but an approximation instead.

A still unanswered question is when the synchronous diagnosis of $R \cup U2$, should be performed. If possible, the tests for these units are executed within the gaps between the asynchronous diagnosis of the units $U1 \backslash R$, when the gaps are larger than T_{sd}. If not enough gaps are available; the synchronous diagnosis could also be scheduled in gaps smaller than T_{sd}, as long as T_c is not exceeded. This approach is especially useful if the difference between the gap and T_{sd} is small. If this happens, then the problem is not able to schedule, and it is necessary to extend T_c. Of course, a diagnosis cycle is over when all units are tested.

If one wants to minimize performance losses when T_c is not limited, every unit except one gets a user task and the remaining one performs the asynchronous testing.

Everything that was discussed above covered the first case where both the task times and T_c are known in advance. We want to discuss now case (2) where T_c is known but the task completing times are unknown. In this case, it is possible to test all units of $t_{i,j} = 1\ldots n$ if $T_c \geq nT_{sd}$ holds. This ensures that even in the case that all tasks are diagnosed in the more time-consuming synchronous mode enough time is available.

For case (2), the testing algorithm is as follows.

Tests are performed asynchronously as long as enough time is left to perform all remaining tests synchronously, i.e., $(n - i)T_{sd} < (T_c - T^*)$ with T^* as the current time and i the number of the currently executed task.

In case (3) where T_c is not known and therefore unlimited and the task completion times are also not known, it is hard to optimize the system diagnosis due to the lack of information. However, it is still possible to apply the here presented approach.

Suppose that the system currently tests unit u_i. Suppose now that unit u_j completes a user task while u_i is tested. By using the testing completion time which is known, it is possible to find out whether it is worth waiting for the test to complete or not.

Thus, if the remaining testing time for u_i is less than $(T_u + T_r)$, it is worth for u_j to wait for the completion of u_i and then perform an asynchronous test. Of course, if u_j finishes and no testing is currently ongoing, u_j can immediately initiate its asynchronous testing. In other cases, uj has to be tested synchronously. The situation gets worse when all units of the system are supposed to be tested synchronously, i.e., $\max(T_c) = NT_{sd}$.

Although the diagnosis period for the whole system in case (3) cannot be strictly determined, the diagnosis process has a finite character, as over time, every task will finish eventually and thus every unit will get the opportunity of to be tested asynchronously.

One might also think of having tasks that are supposed to run permanently on one CPU. For this purpose, it is sufficient to dedicate one CPU for this task and periodically test it in the synchronous mode.

7.2.4 Extension of the Diagnostic Procedure

One of the constraints until now was the fact that all tasks are ready at t = 0, the boot up time. In a real system, especially in control systems with specific task timings, this might not be always the case. We suggest therefore a modification of the above algorithm to take this into account.

The main idea is to include a task ready time t_{ri} for every task in the system. The task finishing time changes accordingly to $t_i = t_{ri} + t_{pi}$, i.e., the ready time (t_{ri}) + the task processing time t_{pi} of task number i.

In addition, we want to extend the task checking from the processor only to all the hardware that is used by the task and introduce t_{adi}, the asynchronous task checking time for task number i.

Taking this change into account, we modify the procedure in Fig. 7.6.

First, we want to keep the real-time nature of the algorithm and sort the tasks according to their finishing time (earliest deadline first). Except for the continuously running tasks, which can be tested at any time, it is advisable to move the testing to the end of a task processing to minimize recovery time in case of a fault. The modified algorithm is shown in Fig. 7.8.

Fig. 7.8 Procedure T2

```
$i = 1, R = \{\}, j = i + 1, t_{sum} = 0$;
if $t_i <= T_c - T_{adi}$ then
    $t_{sum} = t_{adi}$;
    while $t_j \leq T_c - T_{adj}$ do
        if $t_j > t_i + t_{sum}$ then
            $i = j$; $t_{sum} = t_{adj}$
        else
                if $t_j < t_i + t_{sum} - (T_u + T_r)$ then
                    assign user task to $u_j$;
                    set new completion time of $u_j$;
                    restore ordering;
                else
                    $t_{sum} = t_{sum} + t_{adj}$;
                    if $t_i + t_{sum} > T_c$ then
                        $u_j \in R$;
                    end;
                end;
            end;
        $j = j + 1$;
    end
end

if $j < n$ then
    $u_k \in R, k = j \ldots n$
end;
```

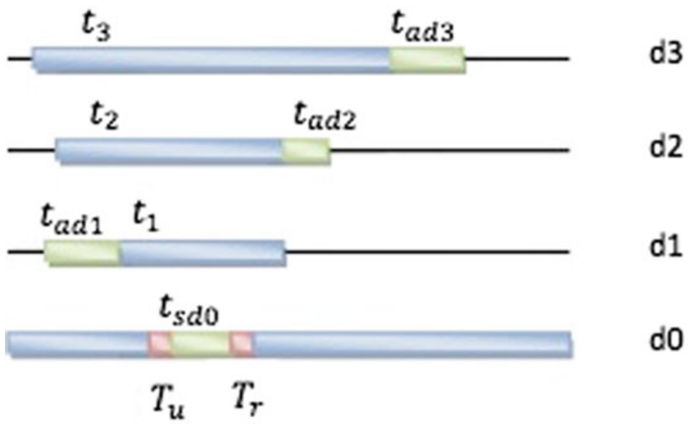


Fig. 7.9 Task examples for extended testing

The changes in the algorithm are twofold. First, the task checking time is no longer constant, thus we replace k_{Tad} by the sum of the task checking times t_{sum}.

Second, as not all tasks are ready at t = 0, it is possible that some processors are free at some given time, which allows us to test the hardware used by tasks asynchronously even if they are not currently running. This is possible due to the nature of the applications in our case, namely, reoccurring periodic tasks in a closed control system.

Case d0 in Fig. 7.9 shows the special case of a continuously running process. This task is synchronously tested at a time where no other processor is being tested.

7.2.5 *Testing of Time-Sharing Systems*

In the above introduced algorithm T1 and T2 and the sketched scheduling algorithm, task preemption was not allowed. In contrast to scheduling without preemption where it can be shown that the problem to find a schedule with minimal scheduling length is NP-hard, a static schedule with minimal length for systems that allow preemption can be found in polynomial time [3]. For dynamic scheduling with preemption, even O(1) schedulers are known and implemented [4, 8].

Consider the well-known Rate Monotonic (RM) scheduling where the task with highest priority is executed. The priorities in are derived from their deadlines, which correspond to their period. Thus $w_i = 1/\pi_i$ with w_i representing the priority of task i and π_i the period of task i. As the deadlines and therefore also the priorities do not change during the lifetime of a system, this is a static algorithm.

Liu and Layland [9] proved that for a set of n periodic tasks with unique periods, an RM schedule can always be found as long as the processor utilization U is below.

$$U = \sum_{i=1}^{n} \frac{c_i}{t_i} \leq n\left(\sqrt[2]{2} - 1\right)$$

Taking into account that $\lim_{n \to \infty} n\left(\sqrt[2]{2} - 1\right) = ln2 \approx 0.6931$ we see that every problem is schedulable with this scheduler, as long as the system utilization is below ln2. This does not necessarily mean that for a system with higher utilization no feasible scheduling can be found with this algorithm, it just proves that for every system with lower utilization than ln2, the rate monotonic scheduler finds a solution.

We now want to apply the above introduced principle of task and test to a system with preemption, based on the static RM scheduler. We use here the same constraints as applied for T2, i.e., the tasks are ready at an arbitrary time t_{ri} and every task has a given running time $t_i = t_{ri} + t_{pi}$ and an assigned hardware-checking time t_{adi}.

We do not consider here any dependencies between tasks, or any task switch times. We assume that the system provides a periodic time tick that calls the scheduler. As the scheduling must be calculated for every time slot, we introduce the remaining task processing time r_i for every task and the deadline d_i for each task. The current time is given by T^*, the scheduler interval by *dt*.

The static algorithm that we show here calculates for every scheduler tick whether a test for a specific task should be started or not. We further calculate only a schedule for one T_c cycle, with the assumption that the T_c cycle corresponds to the global period of a control loop system.

Under this assumption, we get $T_c = max(d_i)$. The deadlines themselves must for the sake of simplicity be a multiple of dt.

Performing an asynchronous or a synchronous test involves the same steps as shown in Sect. 7.2, i.e., the used resources of a task still must be freed even if the task is currently preempted.

Thus, it is preferable to test the resources in asynchronous mode. As tasks in general have a smaller finishing time than their respective deadline ($t_i \leq d_i$), the main idea is to schedule their asynchronous test in the timeframe t_i to d_j.

Of course, $t_i + t_{adi} \leq d_i$ must hold; otherwise, the task is not schedulable. In case of synchronous test, even $t_i + t_{sdi} \leq d_i$ is required. Therefore, we formulate the following rule:

The test of task i is performed asynchronously, if it is possible to schedule it in the timeframe t_i to d_i as long as all other tasks can still meet their deadlines and only one test is executed simultaneously. Otherwise, execute the test synchronously.

In order to keep the algorithm T3 (Fig. 7.10) short and understandable, we separate the scheduling itself from the decision algorithm. By doing this, the task finishing times are known and can thus be used for the sorting of the tasks.

Therefore, we first perform the scheduling according to the RM scheduler and then sort the tasks according to two criteria, first according to the deadline of the

```
$i = 1$,$R = u_1 \dots u_i$, $t_{last} = 0$
while $i \leq n$ do
    If $(t_i \leq T_c - t_{adi})$
        $t_i$ possible candidate for asynchronous testing
        if $(t_{last} < t_i)$ then
            start test of task $i$ asynchronously
            $R = R - \{u_i\}$
            $t_{last} = t_{last} + t_{adi}$
        else
            possible candidate for future asynch. testing
            if $t_{last} + t_{adi} \leq d_i$ then
                starting time of asynch. testing of task $i = t_{last}$
                $t_{last} = t_{last} + t_{adi}$
            else
                search largest timeslot in $t_{ri}$ to $t_i$ with no ongoing testing
                schedule test
            end
        end
    end
    update scheduling;
    $i = i + 1$
end
```

Fig. 7.10 Procedure T3

tasks and in case of the same deadline according to the task finishing time derived by the scheduling algorithm, both in increasing order.

These separate steps can of course be combined in one algorithm that works incrementally and therefore much faster. However, we believe that for the sake of understanding the algorithm, separating these steps is useful.

If T_c proves to be too short, for example, due to a very short system control loop, T_c can also be extended to cover multiple loops of the control system with the disadvantages that more data must be stored for recovery and increased recovery time.

Figure 7.11 shows an example of tasks sorted according to the just given rules.

After the two (here not shown) first steps, all tasks are scheduled according to their priorities and sorted. As preemption is allowed, the task times t_i do not reflect the needed processing time of a task, but the absolute time when a task will finish its processing. Due to preemption, t_i can be much longer than the pure processing time of task i.

For every task, it is now inserted its test in the schedule based on the rules given above. Whenever a test is inserted, however, the remaining tasks on this processor must be possibly shifted further in the timeline to get a large enough timeframe for the test. As this shifting might violate the rules of the RM scheduler, the schedule on all processors after the test insertion must be checked and updated.

If no time slot can be found to perform the testing asynchronously in the timeframe t_i to d_i the task test must be performed synchronously. A timeframe of

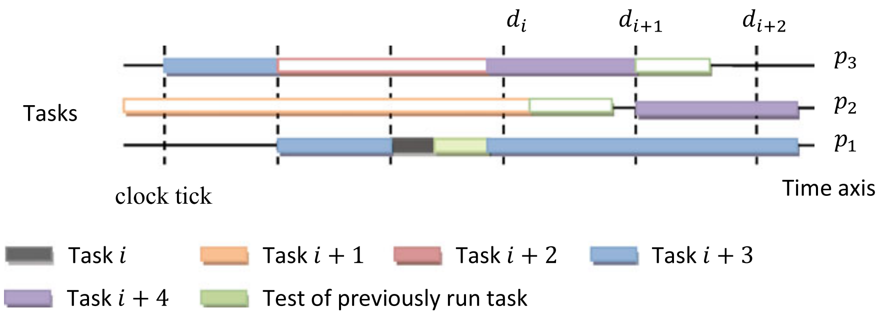


Fig. 7.11 Task examples with applied T3

minimum length t_{sda} is searched where no test is ongoing on other processors. If one is found, a synchronous test is inserted, and the scheduling of all further tasks adapted. If no timeframe is found, the largest frame is used to place the test. In this case, two scenarios are possible:

1. If it is very important to have at no point in time two tests ongoing, the schedule of the ongoing other tests could be moved to the past.
2. If this is not possible due to timing constraints, one could also move the test in front of the task that belongs to the test. If still no space can be found, another test ongoing on another processor could be moved further to the past to generate the space.

Figure 7.11 shows a possible schedule of some tasks with tests. The x-axis represents the timeline and the y-axis represents three processors (p_1, p_2, p_3). The dotted vertical bars represent the clock tick, which trigger an interrupt that calls the scheduler. The scheduler reassigns the tasks for the next time slot according to the pre-calculated schedule. Shown on the timeline are five tasks, task i to $i + 4$.

Also shown in the diagram are the deadlines for all the tasks, i.e., the point in time where the task and its associated test must have finished. Task i has the shortest deadline and fulfills it. No test is ongoing, and thus, the test is scheduled right after task completion. More interesting is the case of task $i + 1$ and $i + 2$.

Although task $i + 2$ has completes before task $i + 1$, the test of task $i + 1$ is scheduled first, as this task has the shorter deadline and would also miss it if the test was delayed. The test for task $i + 2$ is scheduled right after the test for task $i + 1$.

In the timeframe between the completion of task $i + 2$ and its test, another task could be executed if one is due. Illustrated is this by task $i + 4$ which is using exactly this empty space for processing.

If background tasks are present in the system, i.e., tasks with a deadline of 1, one synchronous tests should be scheduled for every T_c cycle at an arbitrary point in time where no other test is running.

7.2.6 FT Scheduling

Scheduling in the event of faults is challenging, as the system must immediately diagnose the fault and define further actions that depend on the diagnosis outcome. As faults can happen at any point in time, and cannot be predicted, dynamic scheduling is required to have the necessary flexibility in the system to react on the fault.

Static scheduling would not allow interrupting the currently running tasks and performing system diagnostic.

Before we introduce a scheduling algorithm that combines tasks and tests in the event of faults, we introduce some general guidelines we want to follow. Note that we assume a rate monotonic scheduling algorithm. This algorithm also includes the steps of GAFT. To illustrate the whole process, we first want to describe the process in words as it is shown below:

1. In case of a manifestation of fault, the process on the processor with the least critical load (idle, or the task with lowest priority) is preempted and the diagnostic routines are loaded.
2. If the fault affects hardware that is not in use, ignore the fault if the preempted process in Point 1 was a real-time task. If one processor does not run a real-time-critical task, the system can use this processor to execute further actions (Point 4). If the hardware is affected that is in use, continue at Point 4. Otherwise, continue processing and delay the fault handling until later.
3. Perform GAFT steps B–E on one of the processors.
4. GAFT Steps F–J: Tasks that use the affected hardware and are running since the last hardware check are considered as faulty, and recovery actions must be undertaken (see Chap. 9, Sect. 9.1). If the remaining execution time is not sufficient to execute recovered tasks before their respective deadline, consider executing alternate versions, depending on their execution times.
5. Reschedule tasks. Continue processing.

Now, as seen above, the tests are scheduled by the scheduler and executed by one of the processors. If a test found a fault in the system, the test invokes the fault monitor, which then in turn informs the scheduler and performs the above described actions. After doing this, the scheduler can reschedule the tasks and continue processing. For a detailed description of the recovery, see Chap. 9, Sect. 9.1.

7.3 System Monitoring of Checking Process: A Syndrome

In the last section, we discussed software-based tests. If the hardware itself has built-in self-diagnosis capabilities, a mechanism is needed to inform the software about detected faults. For this, we have mentioned earlier a scheme to present a snapshot of a system for purposes of reconfiguration, diagnostics, performance or energy saving, etc., we have introduced the syndrome.

Taking into account the conceptual importance of a syndrome, it is worth to elaborate conceptual importance and its possible implementations.

As it was already mentioned in [10], a system flexibility assumes the presence of hardware flexibility and system software flexibility.

System flexibility can be considered as several connected properties and supporting features, as described in [10–13]. We list the following system requirements that need to be reflected and handled by a syndrome:

- flexibility (of hardware and system software);
- resilience;
- scalability (task-wise, frequency-wise, technology-wise);
- performance-wise functioning;
- reliability-wise functioning;
- energy-smart functioning.

We aggregate these properties by the name of PRE-smart systems (performance-, reliability-, energy-). We argue that all mentioned properties are required and should be implemented within a computer system. Good point that it is possible: all of them are based at some level on system *reconfigurability*.

System reconfigurability must be introduced at the design level and pursued along at other levels, including maintenance, especially for critical applications. System reconfigurability should also be considered, throughout the entire life cycle.

Therefore, system reconfigurability becomes not only a *feature* or *property* of a system, but *a process*. This process serves the need of introduction and maintenance of reconfigurability, including ways of changing configurations and ability to reconfigure.

Therefore, efficient design and implementation of *reconfigurability* becomes a task of utmost importance for the next generation of computer systems.

For example, during critical missions, reconfigurability should be executed within real-time constraints, *invisible* for applications.

In turn, during regular operation, system reconfigurability should be used to adapt the system to different requirements in terms of efficient performance, reliability, and power consumption. We call this PRE-smart functioning as defined in [11].

Reconfigurability for reliability should be implemented with supports the ability of system to recover with minimum time overheads. Here it is worth to mention that reconfiguration might have internal and external reasons. For instance, the system might exclude or isolate some hardware elements from the configuration due to a transient/permanent hardware fault detected via checking schemes (external reason). This isolation should be considered as *a process* and be “fine-tuned” by minimizing the hardware loss.

Additionally, with energy saving in mind, the system could set up a simple hardware configuration for particular task execution (internal reason). In energy-saving functioning, reconfigurability has to provide a mechanism to disconnect or switch to a lower consumption mode all hardware elements that are not required for the active program processes.

The term Syndrome is new Latin (origin 1535–45) and was originated from Greek “Syndrome” where “Syn-” from combination, concurrence. For our purposes, a Syndrome is not just passive, i.e., presenting “a snapshot status” of a system but also active, a serving tool to control the system configuration. For us, a Syndrome is

a group of related or coincident things, events, actions, signs and symptoms that characterize a particular abnormal condition.

A Syndrome also might help to answer the questions that have been omitted in the vast majority of research on fault tolerance and performance: “*what provides the fault tolerance of the system?*” and “*how big a performance, reliability, energy-saving gain might be achieved?*”

It is usually assumed that the hardware core logic is ultrareliable and guarantees control of configuration and reconfiguration. Unfortunately, using homogeneous redundancy limits the reliability gain—since techniques based on the same type of redundancy are vulnerable to the same threats. Hybrid techniques based on heterogeneous redundancy can be more effective.

Thus, even when memory or processor checking schemes detect error and transfer information to the Syndrome, this information might not be useful if the system does not include either one or both: “External elements” responsible for exercising reconfiguration and making decisions on configuration/reconfiguration. Reconfiguration might be initiated externally, by other system elements to create best fit for task configuration, or, if necessary, by “Internal elements” that are capable to initiate the required sequence of reconfiguration for internal purposes and reasons—faults, errors, or power saving.

Indeed, in regular computing systems, when there are faults in the processing logic, to expect that it is able to perform self-healing and then control and monitor the configuration of the rest of the system looks like a part of fairy tale, not engineering.

There is a solution though, as described below. To be able to absorb any trustworthy information about the status of system elements we have to aggregate all checking and status signals about the condition of registers, memory, AU, and LU as well as control unit. This aggregating scheme is implementation of *a Syndrome* concept.

Clear, reconfigurability of different hardware areas passive where information is stored, interfacing and active zones are different. Therefore, a scheme of implementation of reconfigurability should separate the passive zone and active zone of the proposed architecture.

A separation of the functions of processing (of data operation) and storing (memory) enables to apply various checking, recovery, and reconfiguration solutions and making system more flexible. The Syndrome acts as a control center for three main functions including *fault monitoring*, *reconfigurability*, and recovery.

A syndrome is conceptually speaking can be considered as a real-time status for every element of the system. However, its function is not only passive, presenting a “snapshot-status”, but also active, serving as a control manager of the system.

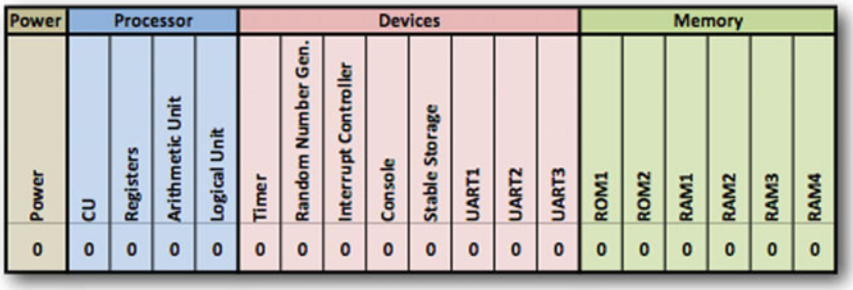
In the following, we present how the concept of syndrome is implemented using Embedded Reconfigurable Architecture (ERA) [11] as an example. From the software point of view, the syndrome is represented as a set of special hardware registers that shows the current hardware state (current configuration, detected faults, power) and can, in case of detected faults, signal them to the software via interrupts.

NB. Pictures of syndrome (Figs. 7.12, 7.13 and 7.14) for our proposed architecture ERA [10] were prepared by Dr. Victor Castano and with his kind permission are introduced here.

As an example platform to illustrate the syndrome, we use here the simulator of the same processor architecture ERA with all its devices as it is implemented. See Appendix A for more details about the simulator. The syndrome implementation in hardware is currently ongoing but we still give in this section some implementation guidelines.

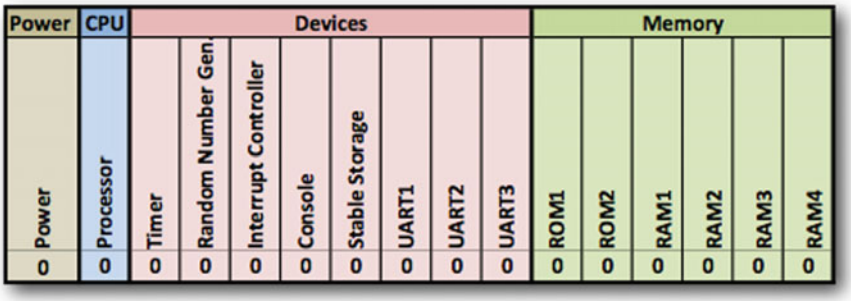
The structure of the syndrome is subdivided into three different management areas: fault management area, configuration management area, and power management area. The three management areas are each reflected by a hardware register visible from software in memory mapped I/O. The respective registers for the simulator are shown in Figs. 7.12, 7.13 and 7.14.

The *fault management area* reflects the hardware status of the different areas of a computing system: *processor*, *memory*, and *interfaces*. A full “Zero” syndrome in this area indicates that the hardware-checking schemes or any other actions (program testing, external control) did not detect any fault in the system. When a fault in a specific location of the system architecture is detected, the value of the bit



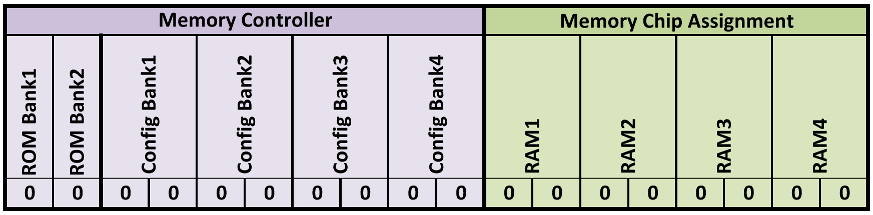
Power	Processor				Devices								Memory					
Power	CU	Registers	Arithmetic Unit	Logical Unit	Timer	Random Number Gen.	Interrupt Controller	Console	Stable Storage	UART1	UART2	UART3	ROM1	ROM2	RAM1	RAM2	RAM3	RAM4
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 7.12 Syndrome fault management



Power	CPU	Devices								Memory					
Power	Processor	Timer	Random Number Gen.	Interrupt Controller	Console	Stable Storage	UART1	UART2	UART3	ROM1	ROM2	RAM1	RAM2	RAM3	RAM4
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 7.13 Syndrome power configuration



Memory Controller										Memory Chip Assignment							
ROM Bank1	ROM Bank2	Config Bank1		Config Bank2		Config Bank3		Config Bank4		RAM1		RAM2		RAM3		RAM4	
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Fig. 7.14 Syndrome memory configuration

corresponding to the area is set to "1". The ROM group in our example consists of two modules, and therefore, the syndrome of ROM condition has two positions with zero when the ROM works correctly. In turn, the static RAM has four hardware elements. For tests between program runs, it is possible to run tests of the memory elements that were used in the last program run or will be involved in the next execution.

The *configuration area* reflects the current memory configuration and power status of all devices. In the simulator hardware configuration, four 32-bit memory modules are attached to the memory controller. These four modules can be configured in four banks according to the following table.

Memory Controller Bank Configuration

0 =	Disabled
1 =	Linear
2 =	Duplicated
3 =	Triplicated

Each of the four memory modules can be configured to be internally connected to at most one of the banks. Dependent on the configuration selected for the bank, the resulting memory configuration varies in available space, reliability, and power. More details about this Performance-, Reliability-, and Energy (PRE)-smart systems see [11].

The ROM Bank entry has only one bit indicating duplication with voter (bit set to 1) or linear (bit set to 0), as the simulator features only two memory modules. All banks have fixed addresses in the memory space, with ROM bank 1 starting at address 0 and bank 2 continuing adjacent to bank 1. At system boot up, the processor starts executing code from address 0, which means that bank 0 must always be populated with at least a single ROM module. The same applies for the memory banks. Every bank has its own unique address, with all memory banks being adjacent in the address space.

The *power management* area reflects the power status of the hardware components. For every component (RAM, Devices, etc.), the power can be enabled or disabled. If the power is disabled, the device is ideally also physically isolated, for fault containment.

The power settings for every device (one bit per device in the register syndrome power configuration) are as follows: 0 = *Power Off*; 1 = *Power On*. Note that the power and memory bank configuration are tightly connected. A memory chip module can only be used in one of the memory configurations if it is powered on.

The combination of those three areas: fault, configuration, and power management determines the state and configuration of the system and every component. However, the syndrome is only capable to signal faults but gives no details about the fault type (malfunction, permanent fault). Therefore, additional diagnosis routines are required to distinguish the fault type. In case of a permanent fault, software support is required to reconfigure the system, and the software can replicate the syndrome in software to keep the current system state or to track changes.

Without doubt, the syndrome is one the most critical parts of the system. For reliability purposes, we suggest using internal three copies of the three syndrome registers connected to a voter. Without this replication, a bit flip in the faults area of the syndrome would lead to wrongly initiated fault processing, or even worse, undetected faults, whereas a bit flip in the configuration area would likely end up in a catastrophic failure.

The same applies for misbehaving software. If an undetected fault affects the software in a way that the processor accidentally writes a value in the configuration section, the result could be a dead system. To prevent this, a special access pattern is used, which we explain later in this chapter.

7.3.1 Access and Location of the Syndrome

Triplication of the syndrome registers involves more complexity in terms of logic. In addition, the voters are vulnerable as well (unless we have triplicated voters as well). Low-level hardening could be used for the syndrome registers and voters to decrease the complexity but this would need an adapted manufacturing process and would therefore result in much higher costs.

As faults in hardware can either be detected by the hardware itself (mismatch and voters) or software-based checks, both the hardware and software need access to the syndrome registers.

Hardware checks. If the hardware detects a fault, the hardware updates the syndrome and raises an interrupt. The software can then handle the interrupt and further diagnose the hardware. If multiple faults are detected, the principle of growing core must be applied, mode details are presented in [10].

The same is valid if consecutive faults occur during fault handling. The syndrome interrupt should therefore be priority-based and reentrant. For instance, a fault in a peripheral device must have a lower priority than a fault in the ALU. When the software finished handling the fault, it must clear the handled syndrome bit.

Software checks. Software-based hardware checks use specific hardware properties for checking, i.e., clearly defined hardware behavior or software redundancy when checking for faults in memory. Although these checks are initiated by software, they can also trigger a hardware-checking scheme which then signals the fault via interrupt and syndrome. An example for this is memory scrubbing that triggers a latent fault (modified memory cell) on triplicated memory.

As the interrupt controller itself can also be affected by a fault, which could result in a nonworking syndrome, interrupt, the software must periodically poll the fault syndrome to check for faults.

The ERA instruction set (ISA) (see Table 14.1), which the simulator uses, does not include special purpose instructions to access the syndrome. Using some of the special registers instead of general purpose ones for the syndrome would contradict the design principles of the instruction set.

Thus, the preferred way to operate and access the syndrome that avoids changing the ISA is the use of the I/O memory lines to a reserved fixed location in the address space.

In hardware, the syndrome scheme could be implemented in a triplicated scheme, either connected upstream to the memory controller or in parallel. Regardless of the triplicated implementation, only one syndrome register set is visible to the software. An error in one of the syndrome registers is therefore corrected by the hardware without software intervention.

However, it seems useful that the runtime is aware of errors in the syndrome itself for monitoring purposes. In addition, the monitoring of all occurring errors indicated by the syndrome provides useful information for a potential contingency plan (e.g., setting the fault tolerance of the system to a higher level in case of recent particle impacts).

Although both hardware- and software-based checks are required for full possible fault coverage, the hardware implemented schemes of detection fault and signal to the rest of the system are clearly the superior approach.

The hardware never clears a bit in the fault syndrome on its own, as it cannot know when the software finished handling the fault; therefore, the software must clear all handled faults manually before it jumps back from the syndrome interrupt.

If the fault is still present in the system, the respective bit in the syndrome will reappear, indicating that the recovery has failed which means that the software either misinterpreted the fault and performed the wrong recovery actions or that the fault is a permanent fault. Software access to the syndrome must be controlled in a way that ensures that no misbehaving program can render the system unusable.

We suggest, therefore, use of an access pattern to unlock the syndrome for write access:

1. Write the unlock constant (a fixed predefined bit pattern) and in the next instruction the inverse of the unlock constant to a helper syndrome register to enable write access.
2. Update the syndrome registers by writing the value and its inverse in two consecutive instructions.

This access protection ensures that the syndrome is not accidentally modified by software. The syndrome unlock command must be rewritten every time software wants to write to the syndrome. If the software does not update the syndrome right after the unlock command, the syndrome locks again, assuming the software failed.

7.3.2 Memory Configuration

The proposed memory scheme may be regarded as a collection of four memory modules, 32-bit wide, with identical size of 4 Mb each. Using 32-bit memory modules instead of often-used 16-bit memory modules increases reliability and flexibility due to additional supported memory modes. The proposed scheme allows 12 different memory configurations as shown in Table 7.1.

The configurations in the table are sorted in decreasing order according to their reliability with triplication + spare on top and four linear memory modules with no fault tolerance at all on the bottom.

The flexibility of the memory controller allows the platform to adapt to different application requirements. A flight control system, for example, requires highest reliability, which is achieved by using duplication for ROM and triplication + spare for RAM.

On the downside, the available resources for the program are much smaller, i.e., only one-fourth of the total amount of available memory. This also implies that only the most critical programs should run on this system, all non-safety-critical programs should be moved to another system. The chosen configuration even allows tolerating permanent faults by reconfiguring the memory configuration by excluding the faulty unit and if possible including a spare one.

An example: If the current selected memory configuration is 1 according to Table 7.1, then the system can reconfigure the system to exclude the faulty unit and include the spare unit in the working configuration.

Table 7.1 Possible memory configurations

Mode number	Number of used banks	Redundancy mode	Number of used memory modules	Usable size (Mb)
1	1	Triplicated + 1 Spare	4	4
2	1	Triplicated	3	4
3	2	Triplicated + 1 Linear	4	8
4	1	Duplicated + 2 Spare	4	4
5	1	Duplicated + 1 Spare	3	4
6	2	Duplicated	4	8
7	3	Duplicated + 2 Linear	4	12
8	2	Duplicated + 1 Linear	3	8
9	4	Linear	4	16
10	3	Linear	3	12
11	2	Linear	2	8
12	1	Linear	1	4

The question arises why configuration 3 is useful. We anticipate here two scenarios:

First, the software could use the linear space for scratch memory, i.e., nonredundant memory where either faults do not matter or are protected by some software schemes.

Second, the software could be used for checking the integrity of the memory module. If a memory module is configured as a spare one, and not used over a long time, the integrity of the module is uncertain. The software could, for example, switch periodically from mode 1 to mode 3 and check the integrity of the spare module, preferably in idle time of the system.

If no safety-critical applications run on the system, the memory configuration can be set to mode 9 where maximum capacity is available but no HW fault tolerance.

16-bit wide memory modules could also be used instead of 32-bit modules. In this case, two memory modules must be combined to allow 32-bit memory access.

The possible configurations with four 16-bit modules are limited to duplication only as triplication would need at least six memory modules.

If 16-bit modules are used, an emergency mode could be implemented, using only one 16-bit module, mainly for signaling the need for maintenance. Or, if space and speed (two memory accesses for loading one 32-bit word) are sufficient, this mode will be useful to run the most critical applications.

7.3.3 Interfacing Zone: The Syndrome as Memory Configuration Mechanism

Until now we just showed the properties and advantages of such a configurable memory controller. What we did not explain is how such a controller could be implemented while still providing interconnection and dynamic exclusion of faulty components from the operational system.

For this, we suggest to use a so-called T-logic inter-connector, illustrated in Fig. 7.15, an idealized concept of a hardware switch in the form of a “T” that can connect or disconnect (for fault containment) from the memory controller by “rotating”.

This logic is used in the hardware architecture to form a hardware configuration scheme adjustable to the software or hardware requirements when a hardware element itself detects faults and thus can’t be involved in further program execution either on a temporary or permanent basis.

The four T-logic inter-connectors, one per memory module, are physically contained in the T-logic Management Unit or TLMU, which is basically an extended memory controller with reconfiguration support at runtime. Using the TLMU enables the memory to be configured and reconfigured according to all supported modes shown in (Table 7.1) and supports memory module isolation and power management.

The syndrome acts as a control and monitoring unit for the TLMU. On the one hand, the syndrome configures the TLMU, but on the other hand, it also shows the current state, i.e., whether faults were detected.

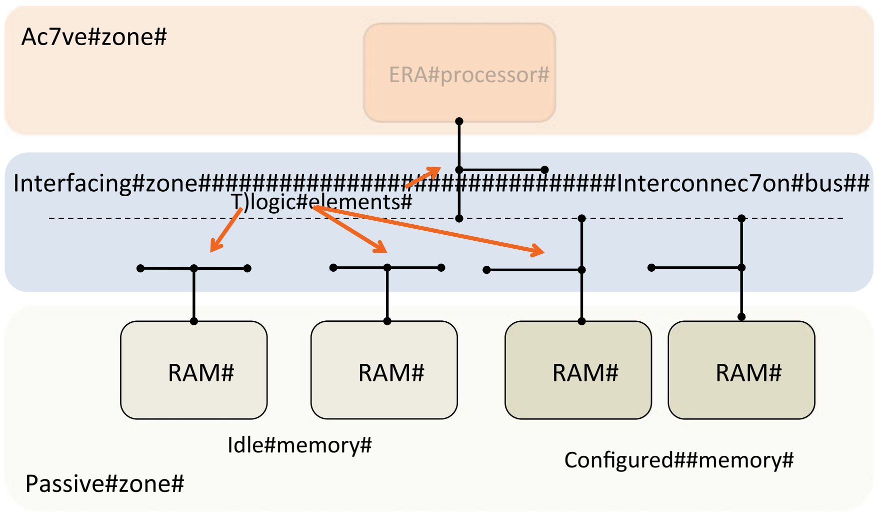


Fig. 7.15 Illustration of T-logic connector use

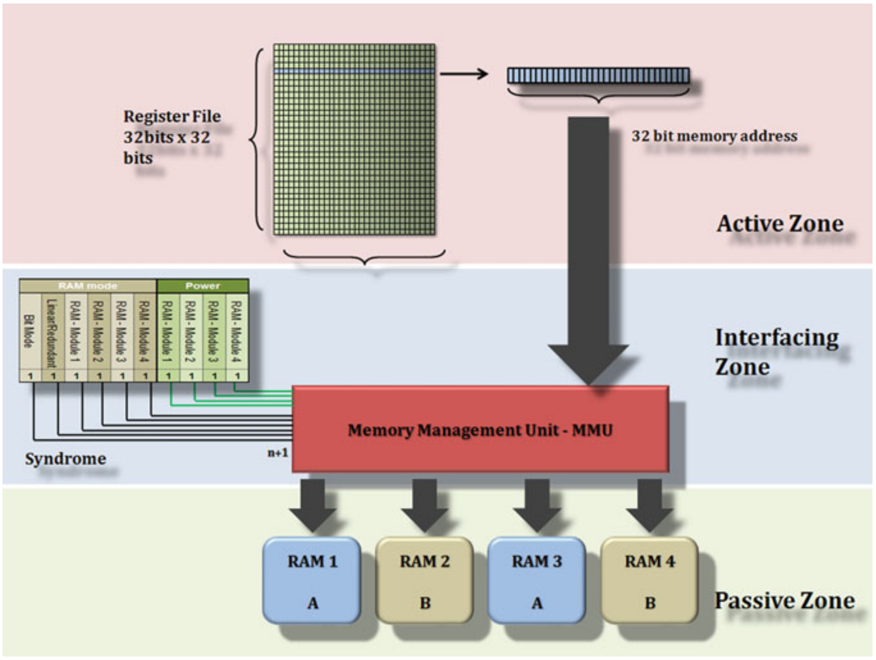


Fig. 7.16 TLMU and syndrome as memory controllers

One example of a possible memory configuration (Mode 4, Table 7.1) arranged by “T”-logic is presented in Fig. 7.16. The example reflects a 32-bit configuration with four modules of which two are used as spares (not connected) and two duplicated.

7.3.4 Graceful Degradation Approach and Implementation

If one of the memory modules fails, the system can be reconfigured to exclude it from the current configuration. Depending on the original configuration and the need of the application, several degradation scenarios are possible. Please keep in mind that a reconfiguration is only required to recover the system in case of a permanent fault and not in case of a malfunction.

We present here two scenarios, the first starting from a triplicated system+spare (Fig. 7.17) and the second starting from a duplicated system (Fig. 7.18). We then use these scenarios to discuss in which cases further software intervention is required.

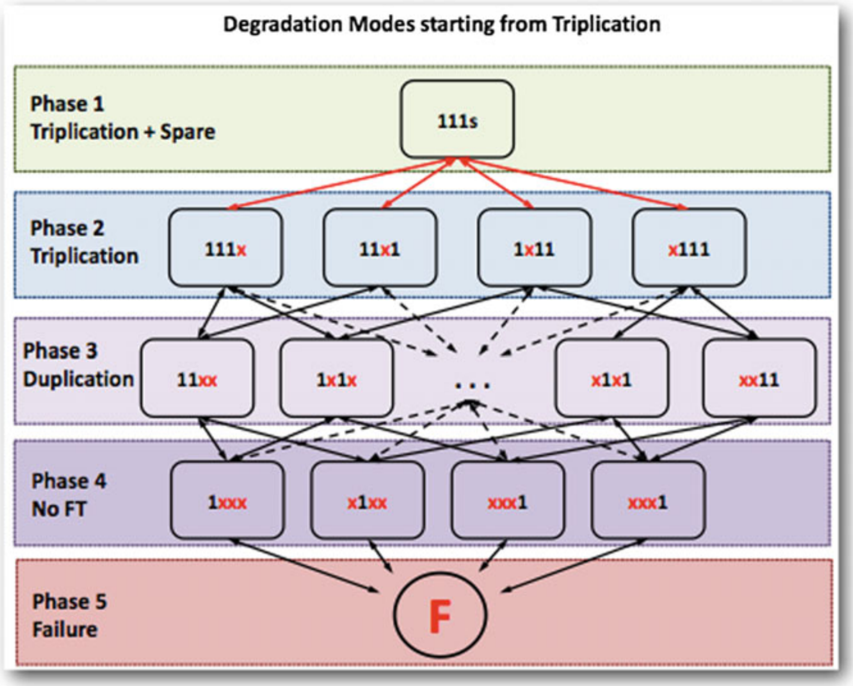


Fig. 7.17 Degradation phases of a triplicated memory system

7.3.4.1 Degradation Phases of a Triplicated System

The most reliable configuration in terms of fault detection and transparent recovery is triplication. As we have four memory modules available, we use the last one as spare. The proposed degradation phases of the triplicated system are shown in Fig. 7.17. Every box represents one possible configuration of the system. The four numbers in the boxes represent the four memory modules and their current configuration.

The position in the four numbers identifies the memory module and the value the memory bank the module is attached to. x stands for failed, and s for spare chip. An example: *1xx1* means that memory modules number 1 and 4 are connected to bank 1, and the memory modules 2 and 3 are failed or are at least disabled. Bank 1 is duplicated, as two memory modules are attached to this bank.

By convention, we assume that bank 1 is always in use as ERRIC supports only absolute addressing. In other words, if code is supposed to run directly from ROM, all memory addresses must either be absolute or calculated relative to a base address stored in one of the processor registers, as the addresses cannot be updated in the ROM. For performance reasons, the former addressing approach is preferable, but

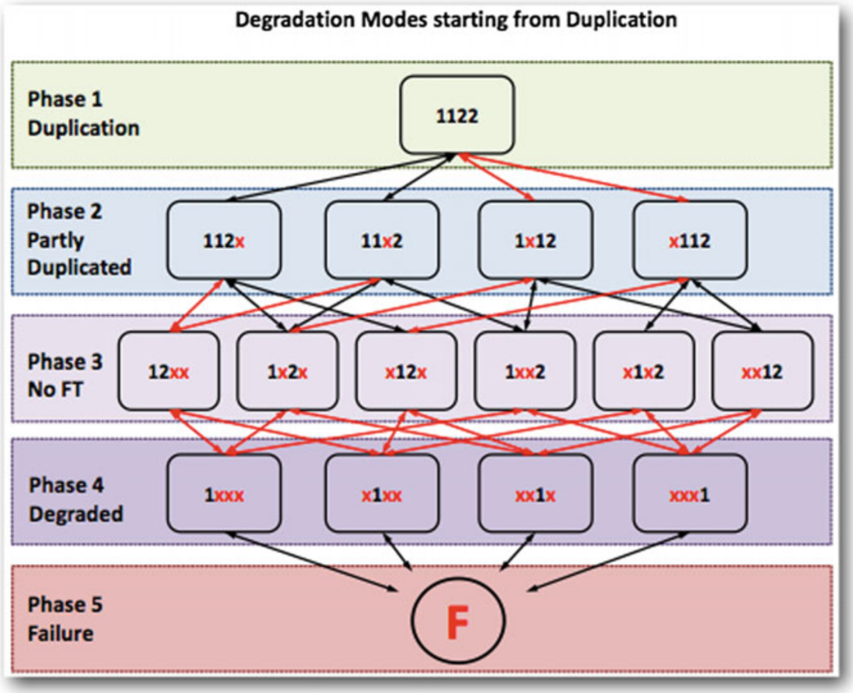


Fig. 7.18 Degradation phases of a duplicated memory system

testing code or code, which has to be relocatable, could indeed use the second approach.

The degradation phases are supported starting from "triplication+spare" state:

Phase 1 The initial state with full checking and triplication of every single bit. The single state in this phase is the initial state and has maximum redundancy. Module 4 is used as a spare module.

Phase 2 Triplication of every single bit. A switch from Phase 1 to Phase 2 includes the replacement of one memory module, with software intervention.

Phase 3 Duplication of every bit. A switch from Phase 1 to Phase 2 is simply done by excluding the faulty element. This mode supports fault detection with hardware read mismatch and recovery from malfunctions with software recovery points. In case of 36-bit wide memory modules, recovery from malfunctions can be done by using the additional 4 bits as checksum for the identification of the correct version.

Phase 4 No hardware fault tolerance. The software-based checking is required with hardware watchdog to detect a dead system due to faults.

Phase 5 Failure. The system fail stops.

The switch from Phase 1 to Phase 2 involves some software intervention to prepare the module before it can be included in the working set. We discuss all cases where additional software intervention is needed (red arrows in Fig. 7.17) in Sect. 7.4.1.

7.3.4.2 Degradation Phases of a Duplicated System

The proposed degradation phases of a duplicated memory system as shown in Fig. 7.18 pose some interesting aspects regarding software intervention which we discuss in Sect. 7.4.

The discussed scenario uses bank 1 and bank 2, each having two modules in duplicated mode.

Phase 1 States with full checking and duplication of every single bit. The single state in this phase is the initial state and has maximum redundancy for 32 bit.

Phase 2 States in which at least 50% of the bits are replicated. The transition to one of the four different states available in this phase is due to a failure in one of the memory modules of Phase 1. Software-based checking is required to detect faults in the nonredundant section.

Phase 3 No redundancy but full memory space available. This is the last mode in which the software has the same memory space available as in Phase 1.

Phase 4 Degraded State. The software must adapt to the new situation of having less memory space available either by running alternative, simpler program versions that need less resources or by disabling some of the tasks completely.

Phase 5 Failure. The system has to fail stop.

The question arises whether Phase 3 of Fig. 7.18 is reasonable or whether Phase 3 of Fig. 7.17 plus one spare element should replace it.

This is actually up to the system designer, dependent on the memory needs and the type of the applications. During degradation, it is assumed that the more important software parts such as the runtime system and important user tasks are located if possible in bank 1.

In Phase 2 as shown in Fig. 7.18, the system therefore keeps bank one duplicated although this involves in some cases (all degradations indicated by red arrows) software intervention.

7.3.5 Reconfiguration of Other Hardware Devices

We focused in this section on the memory reconfiguration aspect, but it is also possible to reconfigure other devices. For duplicated ROM, as used in the simulator, the one and only degeneration scenario is going from duplication to nonredundant mode.

However, as ROM is much less susceptible to ionized particles and longer lasting than memory, this approach seems reasonable.

The reconfiguration of peripheral devices is currently limited on disabling and isolating the device. In case of duplicated I/O controllers (two controllers attached to the same external wires), reconfiguration could be done analog to the above presented approach.

7.4 Software Support for Hardware Reconfiguration

Until now, we did not clearly define what software intervention comprises, but showed how such a reconfiguration with degradation is applied to a system. In this section, we discuss the needed software support for degradation and hardware reconfiguration and show how the software itself can steer the reconfiguration process.

7.4.1 Software Support for Degradation

In case of a detected hardware fault, the syndrome raises an interrupt and software takes control of the reconfiguration process. The whole process is basically identical if software-based schemes detect the fault, with the difference that no interrupt is raised and the fault is not signaled via syndrome.

If the current memory configuration does not use a redundant mode, software-based checking is the only possible approach. The following sequence shows the general procedure of software support during hardware reconfiguration:

1. The hardware-checking scheme triggers the syndrome interrupt.
2. The syndrome interrupt handler then either initiates a hardware Built-In Self-Test (BIST). The procedure of self-test for the device is software-based and serves to distinguish the fault type. For a software-based test, writing different memory patterns to the faulty memory address can be used to derive the fault type.
3. The syndrome bit indicating the fault has to be cleared before doing that. If the syndrome does, after the test, still show the fault, a memory module is faulty. The affected memory address is still present in one of the processor registers and based on the IRQ return address, the correct register number can be derived by decoding the memory instruction that triggered the fault.
4. In general, all software-based testing procedures that test memory must not use the stack (no procedure calls, no data pushed on the stack) until the proper functioning of the used stack locations is ensured.

5. In case of a malfunction, the event is logged and the program execution resumed. Logging the events is important as an accumulation of malfunctions in a module or a specific memory location could hint to a permanent failure in the near future.
6. In case of a permanent fault, the current memory configuration is extracted from the syndrome, and the next degradation state is calculated according to the application needs and predefined degradation tables.
7. The new calculated memory configuration is written to the syndrome registers and the power of the faulty unit disabled. In some configuration transitions, the software has to adapt to the new situation and recover after excluding the faulty unit but before including the new one.
8. Software clears the handled fault in the syndrome and continues processing by returning from the interrupt.

Some of the presented transitions in Sect. 7.3.4 need software intervention to fully recover from a permanent fault and to establish a new working software state. We show here a list of all situations where software support is needed.

Adding/Replacing a module of an already populated bank If a memory module of a duplicated or triplicated memory mode is replaced by another module, the memory content must be replicated to the new module before it is included in the configuration. If this is not done, every memory access would yield an error.

A small loop in software, using only registers is sufficient to perform the copy without modifying the memory during the copy operation. To have access to the spare module, the software configures the syndrome to attach the spare module temporarily to a free memory bank. After the copy operation, the spare module can be included in the working set. These actions must be performed, for example, when the system switches from Phase 1 to Phase 2 in Fig. 7.17.

Failed module of bank 1 Bank 1 contains by convention all runtime system data structures and is therefore critical for system operation. If the memory module of bank 1 fails, the software crashes ultimately.

The only way to treat this is by hard resetting the system either via a hardware watchdog or a software initiated power cycle. The built-in self-test of the hardware automatically identifies the failed module and configures another still working module to bank 1. The runtime system can then restart all critical applications.

Failed module of another bank Assuming that the runtime system structures do not cover more than one module, the runtime system can simply free all resources used by all applications and restart them appropriately.

Instead of graceful degradation, the software can also decide to “regenerate”, i.e., going from a mode with less redundancy to a mode with higher redundancy. The inclusion of a spare module corresponds to Point 1 in the list above; if an already used module is moved to another bank, software has to release all data structures residing on that module in case it is in nonredundant use and repopulate it with data according to Point 1.

Intentional change of the operating mode to a less redundant mode is of course also possible and needs no special software measures. As soon as the module is reconfigured to a free bank, the runtime system can start using it.

7.4.2 Hardware Condition Monitor

A hardware monitor, which is part of the runtime system, is responsible for keeping track of the hardware state. The syndrome also can manage every hardware component represented in syndrome configuration, the hardware monitor tracks the state in more detail than the syndrome alone can provide.

It is also responsible for the execution of all software checking schemes and performs the actual hardware reconfiguration.

Thus the hardware monitor must be accessible by the syndrome interrupt handler as well as the runtime system.

This monitor should however not directly be accessible by applications; only drivers, which are part of the runtime system can register checking procedures for their respective hardware component.

When the system is turned on, the Built-In Self-Test procedures (BIST) embedded in the system are executed.

These self-test runs are required for all devices, using the principle of growing core, ensuring the integrity of all devices. If a failure is detected, the syndrome sets the appropriate fault bits. The BIST is also responsible to initiate the system to a predefined working state, i.e., the most reliable mode with all working available resources.

When the BIST finishes and passes control to the runtime system, the runtime passes control to the hardware monitor for the first mirroring the current state in software and then reconfigures the system according to the need of the program.

As the syndrome might trigger an interrupt right after boot up, the syndrome interrupt handler has to ensure that the stack pointer is valid and if not initialize it.

Every hardware component, which is managed by the syndrome, is in exactly one state of Fig. 7.19. This state diagram shows also all possible transitions between states, allowing the hardware monitor to reconfigure the system in a consistent way.

In fact, all of the above presented cases in the degradation scenarios where software intervention is required, are clearly identifiable in Fig. 7.19. Intervention is only required if the state transition goes from Standby to one of the active cases (marked in blue).

After boot up, all devices are either in state OFF or in one of the blue operation modes. As the BIST automatically configures the most reliable possible memory configuration, the initial states of all devices must be acquired by reading the syndrome. Here is a short list of all possible states and a short description.

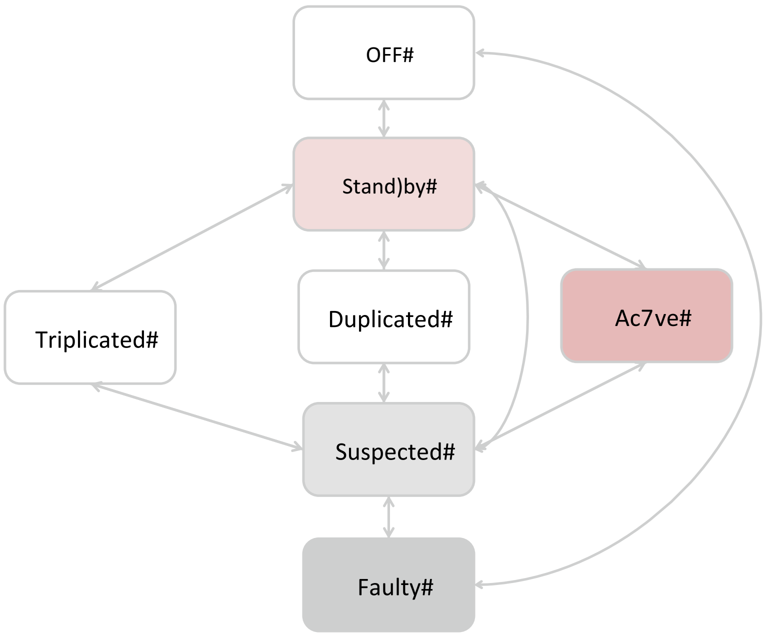


Fig. 7.19 HW state diagram

OFF The device is currently not in use, powered off and isolated for fault containment.
Standby The device is powered on but not yet in use, i.e., in case of memory not yet assigned to a memory bank. In case of reconfiguration, all transitions go through this state.
Active The device is in use in a nonredundant mode. In case of memory, the memory module is assigned to a bank in linear nonredundant mode.
Duplicated The device acts in duplicated mode.
Triplicated The device acts in triplicated mode.
Suspected As soon as a fault in the hardware is detected, the state of the affected hardware component is set to suspected and the testing procedures are initiated to diagnose the fault. If a device is often in this state, this could be a hint that the device might fail in the near future. For reliability purposes, it might therefore be sensible to replace the component with a spare one.
Faulty Dependent on the analysis outcome, the state is then set either to Faulty if a permanent fault was diagnosed or back to the previous state if it was only a malfunction. A device in the state Faulty is powered off.

The state transition diagram of Fig. 7.19 is not directly applicable to all devices. A memory bank, for example, should during the transition not go through standby to make sure that the stored data in the attached memory modules are not lost.

Periodical hardware checks should be performed on every single hardware component despite its state. This ensures that no hidden faults can stay undetected in the system and possibly spread. Thus, from any state exists a transition to the so-called "suspected" state; also from Standby as currently not used devices must also be tested regularly.

It is even possible to revive faulty components, as, for example, environmental changes could allow a component to function correctly again.

7.4.3 Hardware Condition Monitor—System Software Support

One of the distinguished features of our platform is reconfiguration ability. As already explained, the system is able to reconfigure the system in the case of faults. Different scenarios are anticipated for the reconfiguration:

1. hardware malfunction,
2. permanent hardware failure, and
3. software failure.

All these scenarios share the same basic functionality, namely, the reconfiguration of software and hardware. In contrast to standard operating systems where all such reconfiguration has to be performed by the applications themselves, we propose that the runtime itself has the capability and responsibility of performing the reconfiguration.

This way, applications don't have to be fully aware of all possible reconfiguration possibilities, which would increase application complexity and reconfiguration significantly. In contrast, by moving the reconfiguration ability from the application directly into the operating system, we decouple the actual application functionality from the fault tolerance reconfigurability mechanism.

In order for this to work, all applications must provide additional metadata such as the required services (dependencies), processor time requirements, memory requirements, etc. Such metadata information could be added to each application during runtime or preferably already at compilation time.

This way, the system is always aware of all requirements and dependencies of the running applications. Here an example: An application is typically dependent on other services or resources, such as a logging service. The application just needs to know how to access the logger, but the actual implementation is of no importance to the application.

Thus, a first logger could log over a serial connection to another computer, and a second logger writes the log to disk. Now, the reconfiguration in this case could be applied on different software levels:

1. A common logger component preprocesses the log data and saves it on a device. Two hardware configurations in this case drivers that have the same interface and provide thus also the same service are present in the system. If now, for example, the disk fails, the system could transparently replace the faulty disk device by the serial device.
2. Two different logger components that provide the same service are present in the system and thus if one fails, the other one could take over.

As shown by this small example, the reconfigurability feature can be applied on different levels in the system with their respective advantages and disadvantages. The implementation on a lower level in the software hierarchy results in less dependencies and in general smaller components.

However, the first version shares a single point of failure, the preprocessing component of which no alternative version is available. In that sense, version two promises to be more reliable as it has no single point of failure. As a conclusion, it can thus be stated that the granularity of the implementation of an alternative version should be chosen on the lowest level as possible, as long as an alternative version is available. Single points of failure must be omitted.

A *Reconfiguration Monitor* (RM) in the runtime is aware of the current and all possible configurations in the system and is also responsible for executing the reconfiguration. The RM can, based on the available metadata of every resource or component, generate the map of all possible configurations.

The number of these configurations could theoretically be quite large in large software systems as even for n instances of the same entity, *n!* possibilities exist to satisfy the system.

However, in concrete systems, it is not expected to have much less possibilities. The total number of configurations is shown in Eq. 7.2.

$$c_{tot} = \prod\nolimits_{i=1}^{j} \binom{n_i}{k_i} \tag{7.2}$$

where j is the total number of distinct resource entity types, n_i the number of available entity alternatives of type i, and k_i is the number of instances the system requires to operate. The assumption here is that every implementation of one type is only instantiated once and that the implementations are distinguishable from each other.

This is reasonable as not all implementation of one type might be equally powerful and efficient.

If all metadata information is already available at compilation and link time, in other words, if the programming language is powerful enough to express all dependencies and requirements at the language level, all possible reconfigurations could be pre-calculated and stored in the boot image.

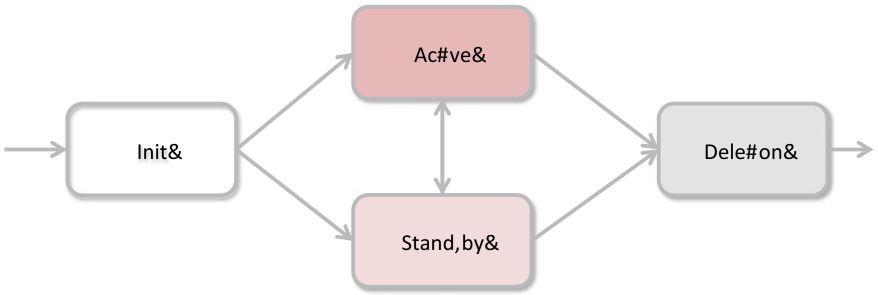


Fig. 7.20 Software states as seen by runtime system

If this is not possible, the system could either calculate during runtime all possibilities or find new configurations on demand. The overhead of finding a new configuration is low, as typically only a small number of reconfiguration entities are affected.

Every software reconfiguration entity is in one of the state in Fig. 7.20. Upon boot up, every reconfiguration entity is in the state Init, which means that the module itself is already linked and ready for execution (The modules of all applications might be linked already at link time and not at runtime). However, what is missing in this state is a concrete working configuration of all reconfiguration entities.

The reconfiguration monitor then finds the first working configuration for the system and starts the initialization sequence. The reconfiguration entities must not have circular dependencies, as this would complicate recovery considerably.

After the first reconfiguration, all reconfiguration entities that are currently in use have the state Active, the ones that are not used are still in the state *Init*.

Now, if an alternative entity is needed, it is initialized, and set to Active, the one that is being replaced is set to Standby. If the entity is no longer used, it is set to Deletion and removed by the runtime system.

Upon boot up, the main processor and the main memory are initialized and transferred to state Active.

This state is the standard state of a component that is actively participating in the system. If no reconfigurability is available in the system, this would be the one and only state of the hardware. Any active hardware component can and should have checking schemes built-in which allows the hardware component to check itself.

The checking procedure could be either software or hardware initiated and consists of hardware and/or software checking schemes.

In Chap. 8, we introduce some language constructs that support this hardware reconfiguration, which is then in explained in detail in Chap. 9.

7.5 Hardware Reconfiguration Outlook

Until now, we assumed a single processor system. The here introduced reconfiguration principles are extendable to cover multiprocessor systems or even multiple systems.

Especially for multiprocessor systems, the single point of failures, the processor, and T-logic LMU could be removed by using a TLMU per processor with its own attached memory. If one of the processors fails, the other processor could reconfigure the TLMU to disable and exclude the other processor from the working set.

If the two TLMUs are connected, the memories of the just disabled processor could be reconfigured to be used by the working second processor, doubling in fact the available resource of that processor.

From the current perspective, it is not feasible to apply this reconfiguration principle further to multiple systems, as the high-speed interconnection between the systems seems problematic. On a diagnostic level, i.e., one system diagnoses and reconfigures another, this approach is very well applicable.

7.6 Summary

We showed that software-based checks can indeed be used to detect malfunctions and permanent faults.

For malfunctions, however, the overhead is quite significant, as duplicate execution of the task is needed to detect discrepancies between the program runs.

For permanent faults, however, the software-based checking can be very useful, under the assumption that the tests can reliably detect permanent faults.

This is most often the case, as software can check the predefined behavior of the hardware.

Then we showed, for the detection of permanent faults, how to efficiently schedule the tests with almost no interference with ongoing tasks.

We proposed to define tests on a tasks basis, where the tests include only the hardware that the respective task used to minimize test execution time. Tests should be performed according to the principle of growing core, where the most critical systems are checked first.

We also introduced several algorithms (T1, T2) for real-time systems with several CPUs and non-interruptible tasks and T3 for time-sharing systems. Synchronous and asynchronous testing are used with asynchronous testing being the preferred choice, as the test does not interfere with its task.

For background tasks, e.g., services that do not finish, we propose to schedule the test regularly when there is no other test ongoing. We therefore proved that testing hardware even with software-based tests can be done efficiently without, in most cases, interfering with ongoing tasks by using processor idle time in a clever way.

For hardware-based checking, we introduced the syndrome a mean of configuring the hardware and signaling detected faults to the software. We discussed possible hardware reconfiguration scenarios and analyzed in which cases software has to adapt to the new topology.

Hardware-based checking scheme the so-called syndrome, a hardware controller that can configure the hardware (power, memory topology) and signal detected faults to the software. We showed which steps are necessary for the software to execute in order to identify the fault type (malfunction and permanent error) and in case of permanent error how software can reconfigure the hardware into a degraded but still working state.

Based on concrete degradation scenarios, we showed that the software can exclude faulty memory modules but has to adapt itself in two cases:

- First, if a memory module in a redundant mode is replaced by another one, software has to repopulate the new module with the data of the working one.
- The second if the main module containing the runtime system data in linear mode is replaced by another one, the whole system must be rebooted for recovery.

For a possible hardware implementation of the reconfiguration mechanism, we propose to use the T-logic embedded in a modified memory controller which also includes the syndrome logic.

The benefit of such a system is twofold:

- As long as enough hardware elements, e.g., memory modules, are working, the system can continue processing.
- Hardware can adapt to the requirements of the application: Highly reliable or more space and more performance as the application has more space available for processing.

References

1. Kirby W et al (1985) The NMFECC cray time-sharing system. Softw Pract Exper 15(1):87–103
2. Serlin O (1984) Fault-tolerant systems in commercial applications. Computer C 7(8):19–30
3. Blazewicz J et al (2007) Handbook on scheduling, from theory to applications. Springer, Berlin, Heidelberg
4. Ingo M (2002) Linux kernel archive. World Wide Web electronic publication, January 03, 2002
5. Bogdanov J, Schagaev I (1990) Sliding slotting diagnosis in multiprocessors. In: IMECO congress proceedings, pp 141–150
6. Garey M, Johnson D (1979) Computers and in-tractability: a guide to the theory of NP-completeness. W.H. Freeman and Company
7. Knuth D (1998) The art of computer programming 3. Sorting and searching, vol III. Addison-Wesley Longman, Amsterdam

8. Johannes M (2002) The active object system-design and multiprocessor implementation. ETH Zurich, Zurich
9. Liu CL, Layland J (1973) Scheduling algorithms for multiprogramming in a hard-real-time environment. J ACM 20(1):46–61
10. Castano V, Schagaev I (2014) Resilient computer system design. Springer. ISBN 978-3-319-15069-7
11. Blaeser L, Monkman S, Schagaev I (2014) Evolving systems Worldcomp 2014. In: Proceedings of the international conference on foundations of computer science FCS'14, 2014. CSREA Press. ISBN 1-60132-270-4
12. Monkman S, Schagaev I (2013) Redundancy + Reconfigurability = Recoverability Electronics, 2, pp 212–233. ISSN 2079-9292. https://doi.org/10.3390/electronics2030212
13. Buhanova G, Schagaev I (2001) Comparative study of fault tolerant RAM structures. In: Proceedings of IEEE dependendable system networks Conference, Guetebog, July 10, 2001. https://www.academia.edu/7140850/

Chapter 8
Recovery Preparation

Abstract In the last section, we showed how hardware integrity of a computing system can be efficiently ensured using hardware-checking schemes and system software testing procedures and their sequences. However, to recover from faults, it is necessary to eliminate the effects the error had on the computation, i.e., the software code and data space. In GAFT, this corresponds to preparation for recovery. We now want to show how software has to be organized to be able own recovery or in other words, we want to revise different strategies how software can, after the detection of an error, ensure that the error did not affect the software state, or if this cannot be ensured, what precautions software has to conduct to be able to re-establish a correct software state. First, we revise the state of the art and then introduce a new technology and show its power and limitations. In the next step, we will show how hardware can assist software in the process of recovery preparation. For all generic approaches to recovery preparation, so-called stable storage, a nonvolatile, reliable, and fast storage is needed. If no direct hardware support is available, stable storage must be implemented in software. We will present a possible software implementation of such a stable storage.

8.1 Runtime System Support for Fault Tolerance and Reconfigurability

The runtime system, often also called kernel of an operating system, provides low-level software functionality such as threading, inter-process communication, and memory management. These three tasks are the minimum functionality every runtime system must provide; otherwise, an operating system cannot be implemented [1]. Since we analyze design and develop of a single (not distributed), embedded computer system with properties of fault tolerance, reconfigurability, etc. and abilities of implementation of these properties at the level of hardware and especially system software we do not pursue the analysis if distributed systems. Some of aspects of implementation of fault tolerance in distributed systems are covered in our previous work [2].

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_8

We argue that the first-level interrupt handler is also required to act in the kernel, as the hardware typically switches into a privileged processor mode in case of an interrupt.

For a fault-tolerant safety-critical operating system, GAFT is also part of the runtime system, as already shown in Chap. 4. Specifically, the tasks fault detection, system reconfiguration, preparation for recovery, and recovery become essential parts of the runtime system. In this chapter, we focus on the task preparation for recovery.

- We divide possible operating system architectures into three classes: Simple unified memory architectures, microkernels, and monolithic kernels.
- Microkernel operating systems such as L4 [1] implement only the three classical mechanisms in the runtime system itself and move everything else out of the kernel, with the advantage that a failed software component be it a program or a driver, does not influence the runtime system stability by design.

Unfortunately, the message-passing mechanism that is required by such systems had in earlier times significant negative influence on the performance as many task switches are required by such a system [1]. However, in recent developments, the overhead could be lowered [3, 4] but is still significant.

Monolithic kernels such as Linux or Windows integrate the three low-level tasks including all drivers in kernel space. Therefore, even a third-party driver can, in these operating systems, corrupt the whole kernel space and lead to fatal crashes. More than 80% of all Windows XP blue screens (traps) are caused by driver errors.

However, less context switches are required in such an operating system in comparison to the microkernel approach, and thus a general improved performance is observable. Applications usually run in distinct memory areas, i.e., they are isolated from each other, but have the kernel mapped in the address space, protected by the memory management unit (MMU).

Simple unified memory architectures use one memory space for runtime system, applications, and data. Their big advantage is their simplicity, i.e., they are easily understandable and manageable. Clear interfaces between parts of this architecture with reduced number of waiting states in the scheme of interaction enable performance gain that is unachievable on complex instruction computer systems (CICS). Another advantage of RISC and simple unified memory architectures is the ability to execute fix time instructions useful especially in real-time applications. Arguments pros and cons about RISC and CISC architectures are not a subject of this research, comprehensive analysis of these architectures might be found in [5], and nothing new was discovered since then.

We do not discuss or consider her any applications based or design done using Java due to limitations of them for real-time applications. In spite of further analysis of language properties based on Oberon and Oberon-like languages [4, 6–13], we do not consider application protection.

It is obvious that the first two approaches need a sophisticated MMU to support flexible memory partitioning and are not suited for fairly simple systems.

In the context of safety-critical systems, where no faults at all can be tolerated, it is critical to eliminate all possible sources of failures and thus the additional overhead of the microkernel approach can happily be traded against higher reliability. All subsystems, such as hardware drivers, network stack, and file systems, should be moved out of the kernel so that faults cannot spread in the system (Fig. 8.1).

If the hardware platform provides a memory management unit (MMU), this unit is used to implement process isolation either by having a distinct address space per process or by having one shared address space and a capabilities mechanism [14].

If the platform does not have a memory management unit, the process isolation relies solely on the programming language. It is possible to structure the language in such a way that processes can only access their own modules and therefore not unintentionally interfere with others. In addition, this clear separation allows simplified process recovery, as the assigned memory space can simply be erased and the process restarted.

If the programming language does not provide any means to implement process isolation, a strict coding policy could be used instead. It is then up to the responsibility of the programmer to implement proper isolation, which is unfortunately error-prone and therefore not practical.

In our case, the anticipated ERRIC hardware platform does not support virtual memory, which means that one large address space for all applications is used and the language is responsible for implementing the isolation. As long as the memory is not corrupt, this is perfectly sufficient, but in case of memory corruption,

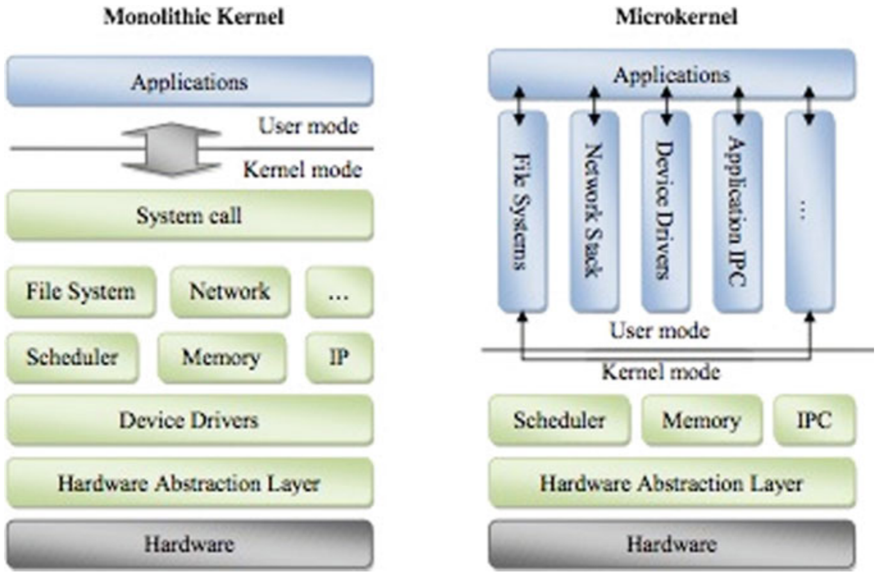


Fig. 8.1 Monolithic kernel versus microkernel

modified memory addresses could result in accessing memory outside of the assigned memory address space. Various variants exist of handling these faults:

1. *SW(T, S)*: Control flow checking by software signatures, introduced in [15] and elaborated in [16]: The compiler introduces branch checks by using signatures to validate the control flow. Although many faults are found by using this technique, the overhead in size and time is significant (size: 25–60%, time: 16–70%). Note that this approach does not rely on specialized hardware and constitutes a pure software solution. Variants do also exist that involve an additional hardware processor to calculate these signatures. Alternatively, duplicated instructions or double program execution SW(2T) with output comparison [17–19] could be used, also which high execution overhead of 40–200%
2. *SW(T), SW(T, S)*: The use of multiple independent runs of the same or different programs with output comparison, sequential or concurrent [20–22].
3. *SW(T, I)*: Use hardware/software fault detection (checkpoints, syndrome) together with recovery points to perform recovery.

In our context, a complex virtual memory addressing scheme is not desirable. We therefore believe that the adaption of the language could help to support an efficient implementation of recovery points.

In the next sections, the requirements to fault tolerance for each software component of the runtime system are shown.

8.2 Overview of Existing Backward Recovery Techniques

Two different types of fundamentally recovery are possible: *backward recovery* and *forward recovery*.

In the former case, the system uses previously stored system states to rollback program execution and restore a consistent fault-free software state in case of a failure.

In the latter case, the system is recovered by finding a new possible, consistent state without using previously stored information. Forward recovery is always application-dependent and needs, therefore, additional overheads. Backward recovery, in contrast, can be applied in both ways: application-dependent *semi-transparent* and *application transparent*:

Semi-transparent The application programmer is responsible for storing the application state in a consistent state for later rollback [23]. More recently, a semi-automatic recovery process called Microreboot [24] was presented, which does not require the application to create recovery points, but requires a specific software architecture with stable storage such as a database.

Application transparent The programmer is not involved in the process of creating recovery points, i.e., application need not be modified. In this case, the compiler and runtime system are responsible for the recovery point creation, which is part of this research.

Using [20, 25, 26] as a guideline, we present here a short survey of possible backward recovery mechanisms only, as this can be applied in a general way to any program without application or programmer intervention.

A *recovery point RP* is at minimum the states of the participating processes [20, 26]. In other papers, the term *checkpoint* is used instead of *recovery point*. We prefer, however, to use the term "recovery point" for stored information used the recovery from faults and "checkpoint" the point in time when the hardware is checked for faults.

Recovery points are always stored on a stable storage [27], which means that the storage survives any system fault without corruption. Such a stable storage can either be memory, a hard disk, or better flash disk or be located on a physically separated device.

In the process of recovery point creation, the problem of inter-process consistency must be taken into account because the rollback of a failed process can force non-failed processes to rollback as well for the purpose of restoring program state consistency.

Let us have an example: In a system, with two processes *a* and *b*, a sends messages to process *b*. If *b* fails and has to rollback, it might rollback to a point in time before the arrival of the last message of *a*.

To fix this inconsistency, process *a* has to rollback as well even though it did not fail. This effect is called rollback propagation. If badly implemented, the rollback propagation can continue up to the initial software state, which is called *domino effect* [28, 29]. The *domino effect* often occurs if processes create their recovery points independent of each other.

Figure 8.2 shows a simple example of the domino effect. Two threads P and Q that communicate with message-passing run concurrently and create their respective recovery points as shown in the figure.

In case of a recovery, the system now tries to find a consistent state space which does not exist in this example except for the initial state. The reason is simple: The global state is only consistent if the threads receive only messages that were sent. If R_Q3 is rolled back, *P* receives a message from *Q* that *Q* never sent, and therefore P must be rolled back as well. The process continues until the initial state is reached.

A *rollback line* [28] is defined as the most recent consistent set of process recovery points. If the recovery points of all processes are consistent, the domino effect cannot happen, as it is only necessary to recover the system up to the last

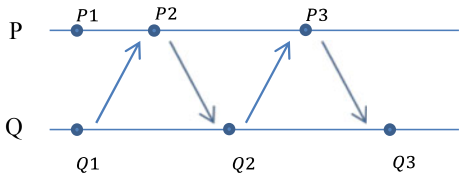


Fig. 8.2 Example of *domino* effect

rollback line. Under the assumption that the system is a fail-stop system and faults manifest themselves immediately, it's just necessary to keep that last recovery line. Every earlier recovery point can be freed.

The classic system model which is used when analyzing recovery points is a fail-stop system [29], i.e., they either work according to specification or crash (stop working) without corrupting data, especially memory.

Most of the work which has already been done in this field is based on message-passing systems, which can either be distributed system with a network interconnection, but also single multiprocessor systems where messages are passed via shared memory or other local interconnection facility. In fact, the single multi-process message-passing system is just a special case of a distributed message passing, and thus the same algorithms can be applied.

The commonly used approaches for creating recovery points are either implemented on the level of the system or on the level of tasks. The former is also called *global checkpointing* [30, 31]. The global checkpointing assumes that the recovery points are created when all other operations of the system stop, i.e., the whole system operations are "frozen" for creating a recovery point of the whole system. This approach has, of course, the huge disadvantage that the system must be completely stopped while the snapshot is made. In this sense, recoverability action reduces reliability—long-range preparation of recovery from future malfunctions or permanent faults reduce point availability by adding to maintenance time and reducing operational time.

Recovery points on the level of tasks have already been extensively covered in literature and can be classified as follows:

Uncoordinated recovery points Every task creates a recovery point whenever it suits the task [28];

Coordinated recovery points Chandy and Lamport [29] introduced coordinated recovery points by saving a system-wide consistent state. Some processes [30] save their state in a coordinated way;

Message logging In these systems, recovery points of threads are created independently from each other. Logging all messages between threads retains consistency between threads. In case of a failure, the threads are rolled back and the communication is replayed and thus consistency restored.

The systems assume piecewise determinism [31], which requires that all determinants [32], i.e., all non-deterministic event, can be identified and logged. Typically, messages between threads and I/O data are determinants. Examples for message logging systems are [33–37].

These mechanisms are all application transparent, as we have mentioned the application-dependent techniques before. Recovery on the level of procedures is covered in Sect. 8.3.

8.2.1 Uncoordinated Recovery Points

Uncoordinated recovery points approach belongs to the first implemented recovery techniques [28]. This technique sets no constraints on the creation of recovery points with the advantage that the thread can initiate a recovery point creation when the state space is small which results in small recovery points.

The implementation is also quite simple. The downside, however, is the susceptibility to the domino effect, which reduces the practicability of this approach considerable. It might also happen that a thread creates an RP that is not part of the global system state and wastes therefore processing time.

As the global consistent system state is in general not known, every thread must maintain several RPs, which need to be garbage collected over time to regain space.

During recovery, it is necessary to determine a global consistent system state to which the system is then recovered. When a thread sends messages to another, it adds the current checkpoint number to the message. The receiver then stores this number locally. In case of a rollback, these numbers are used to calculate a dependency graph showing the latest global consistent state [38].

8.2.2 Coordinated Recovery Points

In contrast to the uncoordinated recovery points approach, threads are synchronized if coordinated recovery points are used. The main idea is that if threads are synchronized between each other, the state is always system-wide consistent. Therefore, every thread needs to keep only one stored recovery point. One of the earliest approaches [39] implements *blocking recovery point coordination* approach that uses two-phase commit to force all threads to create a recovery point at the same time.

Synchronization then is achieved by blocking all threads at the same time (message broadcast). This accompanied by flushing of all communication channels and creating the recovery points. The disadvantage of this approach is obvious: While the recovery points are taken, the whole system is blocked and unable to react to external events, i.e., become nonresponsive.

Between the recovery point creations, there is no additional overhead involved in this approach and the total overhead is adjustable by the recovery point interval.

The *non-blocking recovery point coordination* approaches use other synchronization mechanisms that do not force the system to stop. An early approach [29] solves the problem of inconsistency by creating a recovery point prior to sending a message to another thread.

Extension to that use markers that are sent from one thread that generated a recovery point to all others. These generate their checkpoint and broadcast the marker further. This is an example of a *non-blocking recovery point* approach, as the threads do not immediately stop program execution but create the recovery point when they remove the RP message from the message queue. The message queues must be reliable and FIFO for this to work.

Another approach sends the latest recovery point number together with the actual message. If the latest RP number of the receiver is smaller than the received number, the receiver generates a new recovery point [35].

In a distributed system, synchronized clocks can be used to synchronize the RP creation. The systems generate their RPs at a specific predefined interval and wait for a predefined interval (clock inaccuracy) before continuation [40]. In a distributed system, the communication properties (reordering, delay, reliability) play an important role. We ignore these problems, as we don't want to go into further details about recovery points in distributed systems.

All introduced approaches force all threads to create a recovery point. In real systems, however, threads or more specifically programs often exclusively communicate in related groups. The fact can be exploited as Koo and Toueg [41] did. They introduced a checkpoint initiator that partitions the threads in groups that then individually create their recovery points.

These groups can be further limited if the type of message is taken into consideration. Only messages sent from one thread to the other introduce dependencies between the threads, if they contain shared data. Janakiraman et al. [42] therefore record a dependency only if a message containing shared memory that was modified since the last recovery point is sent from one thread to the other. This approach progressed by introducing page stamps [43], resulting in further reduced dependencies.

Communication-induced recovery pointing basically generates a recovery point whenever inter-process communication occurs. A checkpoint needs to be generated if modified data is sent to another thread. Incremental recovery pointing is often used in this approach to limit recovery point size. Obviously, the number of performed recovery points is linear to the number of occurring inter-process communications, i.e., this approach is not well suited for communication-heavy applications. An example of this approach is [42].

The *coordinated recovery point* approach has one significant drawback. When programs heavily communicate with the outside world using input–output schemes (I/O), this approach is not well suited, as I/O does not fit directly this scheme and extra application-dependent efforts are required.

8.2.2.1 Limiting Checkpoint Size

The simplest approach to generating a recovery point is to store the whole memory area the program has access to. However, this can be very expensive if the program is large or needs a lot of data.

We present here two existing approaches of how to reduce the recovery point size. Li [44] introduced a page-based approach that uses the memory management unit of the computer to identify memory pages that were written to (copy on write) since the last recovery point was taken.

When the thread creates a new recovery point, it is sufficient to store just the modified pages, resulting in much smaller recovery points. This approach has also its drawback. Accessed data are often much smaller than the page size; the recovery point still contains much unnecessary data.

Such recovery points that are not self-contained but rely on other previously stored memory points are called incremental recovery points.

Another approach [45] uses programming language extensions to identify shared data and synchronize access to it. Additional metadata (type, etc.) helps to further improve the efficiency of that approach.

8.2.3 Message Logging-Based Rollback Recovery

In contrast to the recovery point approach, the *message logging rollback recovery* is based on the idea that the execution of a program is deterministic as long as it does not communicate with other threads or performs I/O. In other words, programs are therefore assumed to be *piecewise deterministic*, i.e., programs are deterministic as long as all determinants; events (internal or external) that affect the programs can be identified and logged.

RPs of all threads must still be created and stored on stable storage, but messages and I/O are stored as well. Thus, this approach is particularly suitable for I/O heavy applications. One can consider the message log-based approach as an extension of coordinated recovery points where inter-process messages do not trigger the recovery point creation but are stored for later use.

In contrast to the pure recovery point approach, message logging allows the replay of the program execution after the last recovery line by replaying all stored messages. The piecewise deterministic assumption must hold also while replaying the messages, and thus any non-determinism must be avoided. One of the simplest approaches just stores the number of executed instructions together with the message, which allows, during recovery, the timely correct arrival of the message in the system, avoiding therefore any non-determinism. A backup computer is used to store the recovery points as well as the messages and controls the recovery in case of a failure.

Message logging protocols can be partitioned into three categories: Protocols with *pessimistic logging*, *optimistic logging*, and *causal logging*. We don't go into the depths of these approaches but just give the main idea, advantages, and drawbacks.

Pessimistic logging requires that the message logging completed before the message or determinant is delivered to the receiver (synchronous logging), which leads to a performance impact in the failure-free case, as logging and message processing cannot be executed in parallel.

The pessimistic approach assumes that failures can occur during message logging but as messages are logged before message delivery, recovery is always possible to the latest state and cannot lead to orphans, i.e., messages and related tasks unable to recover due to the logging failure. Further advantages are simple recovery as failures do only affect the processes that fail and as no orphans can occur, recovery always recreates the state just before the failure happened which is extremely useful in real-life applications [46].

Garbage collection of old messages and recovery points is also easy as all recovery point except the last can be recycled.

Optimistic logging stores messages mostly asynchronously (in parallel) to the message delivery and are based on the optimistic assumption that the logging has finished before a failure occurs. This approach is obviously not suitable for safety-critical systems where it is assumed that failures occur anytime. In contrast to the pessimistic approach, simplicity and speed in case of failure are traded to performance in the failure-free case with considerable more complex failure handling.

To keep the logging performance impact small, the determinants are usually stored in volatile memory (RAM) and periodically flushed to disk. In case of a failure, the log in volatile memory is lost and a full message replay is not possible. If the process sent a message to another thread during the not recoverable process execution, the receiver of the message becomes an orphan and must be rolled back as well to restore consistency.

Such a rollback might go beyond the last recovery point, which means that several recovery points are needed. It's obvious that this recovery considerable complicates garbage collection of determinants and recovery points.

The necessary information to assert consistency is usually recorded together with the determinants during failure-free operation.

Optimistic logging allows synchronous and asynchronous recovery; an example of the former is [47] and for the latter is [31].

Causal logging approach attempts to combine the advantages of the pessimistic and optimistic loggings [32, 35]. Logging is performed asynchronously (optimistic), i.e., using volatile storage for message logging and persistent storage for the recovery points and the message log. The volatile message log is flushed to stable storage before a message is sent to the outside world (output commit). Orphans are omitted by ensuring that all casual determinants are either stored on disks or are local to the respective task.

Causality is defined by Lamport's *happened-before* [48] relationship. The locality of determinants is now achieved by piggybacking them on messages sent to other threads. The advantages of causal logging are traded in for a more complex recovery protocol.

8.2.4 Other Approaches

There are other approaches, which do not belong to the above presented categories. Brown [49] uses dedicated machines that snoop the inter-process communication and store the information to restart the tasks in case of a failure. This approach is particularly suitable for a distributed system connected over Ethernet, i.e., a distributed system with physically distinct partners.

8.2.5 Implementation Aspects

The main focus in research during the last centuries focused more on the theoretical aspect of recovery point protocol than actual implementation. Especially in commercial systems, recovery points are rarely used and if at all mostly based on shared library implementations.

Classic Unix-based system that is often used for the implementation of recovery points uses different processor modes to protect the kernel with its kernel data structures from the application. In case of an implementation on the application level (shared library), a recovery point contains user mode data structures only.

If the recovery point mechanism is directly implemented on the kernel level, it is possible to store kernel data structures together with the user mode data, which helps in recovery significantly, as the kernel resources occupied by the application can be restored as well. This is especially useful if the application is heavily I/O based. An example of kernel-level implementation is [26] and for a user mode, implementation is [50].

Another approach replicates relevant process kernel structures in the application and stores them together with the application data in a recovery point. In case of recovery, the recovery process restores the application kernel structures and uses this information to continue processing.

An approach of creating recovery points is presented in [15, 18, 51–53]. This approach heavily relies on the memory management unit of the processor. When a recovery point of a process is created, the system write protects all memory pages of the specific process and concurrently starts saving the memory pages and continues processing.

In case of a memory written by the application, the memory protection violation handler copies the affected memory page to a separate buffer and removes the write protection from the accessed page. The content of the new buffer is then stored to disk instead of the original page. Using the dirty bit of the memory protection unit or software-based dirty bit, this approach can be adapted to use incremental recovery points.

Another approach simply compares the content of a recovery point with the current memory content and stores only the difference. However, the significant comparison overhead together with the storage access time can jeopardize the advantages of this approach.

In a probabilistic approach, hashes are calculated for all memory pages and stored together with the recovery point on stable storage. Recalculating the checksums can identify changes to the pages. Further comparison of current and stored ones can be useful as it was presented in [51–54]. This approach requires less computation than comparing the contents of the recovery points directly.

8.3 Recovery at the Level of Procedures

As shown in the overview above, most of the current approaches to generating recovery points are either located on the level of the system (one recovery point covers the whole system) or on the level of tasks, which is the most frequently used approach.

As shown in Sect. 8.2, the standard system model that is usually applied for systems supporting recovery points is a fail-stop system where faults are eliminated just by recovering the last consistent recovery point and resume processing. The assumption in this case is that the checking schemes can detect faults either during instruction execution or while accessing hardware devices and fail-stop.

Now, we present in this chapter our approach of generating recovery points: Recovery points on the *level of procedures*. This approach tries to exploit language features limiting the required space to save a current state of a program in every recovery point.

Especially for the model of a fail-stop system, the fast recovery time due to the small recovery points is tempting. If faults are masked, i.e., present but hidden in the system, affect the program execution but are detected much later (not a fail-stop system), recovery on the level of procedures might not be the right approach as considerable time is required to find the last correct recovery point or no correct recovery point at all might be present and the system must be restarted to recover.

Upon every procedure entrance, a recovery point is created which stores all parameters, all write-accessed variables the stack and the base pointer, and the current program counter.

This allows minimizing the footprint of each recovery point. The compiler analyzes the program code and emits additional code in the procedure preamble to create the recovery point. Keep in mind that temporary variables on the stack do not need to be included in the recovery frame. The same is valid for the processor registers, assuming parameter passing via stack and not registers. The actual storing mechanism needs runtime support, especially if it is stored on external stable storage.

A recovery block entry usually has the format shown in Table 8.1.

In the *field address*, the physical address of the source variable is stored. The field size contains the size of the variable in number of bytes followed by the *data* itself. For fixed-size variables such as basic types or statically allocated arrays, the data size can be derived at compile time.

For storing these, one could actually omit the field size and store the data immediately after the address. However, it is not possible to derive the size of objects, or records at compile time, as also derived elements could be passed as an

Table 8.1 Recovery block frame

Address	Size	Data
4 bytes	4 bytes	n bytes
…	…	…

argument in procedures. Therefore, the size of these variables must be derived at runtime.

It is interesting to see that only structured languages allow the efficient creation of recovery points, as the scope of a procedure or in other words the visible and accessed variables can be clearly derived. Strong-type safety that was introduced for reliability of programming might be helpful here as in supporting the scheme of deriving the size of all variables. For example, pointer arithmetic in C prohibits the efficient creation of recovery points, as the whole memory can be accessed and modified and thus the entire memory must be saved. Pointer-free languages such as Composita of Prof. Blaeser, briefly introduced in [55], might be immediate next step in making new language for real-time, reconfigurable, and PRE-smart systems.

Various strategies [51, 54, 56] exist how to insert recovery points in a program. This can either be done automatically by a compiler or manually by the programmer. It is quickly obvious that a compiler should do this, as this task is not trivial for a programmer.

The insertion strategy is chosen according to the used system and applications and is always a trade-off between the overhead in terms of time and space and the anticipated recovery time. Frequently created recovery points typically need more time and space to be created, but on the other hand in case of a failure the required recovery time overhead is small. There are no doubts that hardware support might reduce regular overheads of recovery point formation. One example of hardware support is discussed in the next section.

In turn, using sparsely created recovery points typically leads to a smaller overhead but requires more time to recover as more statements have to be executed until the next recovery point is reached [3].

8.4 Hardware Support for Recovery Point Creation

In order to keep the system overhead low, it is proposed here to implement the actual recovery storage procedure in hardware. We introduce therefore the recovery point unit (RPU), which represents a hardware component that is capable to create a recovery point on a nonvolatile medium.

It should thus be able to interpret the above introduced format of a recovery point in memory and store it. To speed up the performance and to have a measure to detect corrupt recovery points, the RPU should be able to calculate a checksum on the fly of the just created recovery point and store it as well together with the recovery point. This checksum helps to identify corrupt recovery points, which cannot be used to recover a system.

The RPU should be placed directly on the CPU bus (Fig. 8.3) and should have direct access to the memory in order to get maximum speed for the generation of recovery points as well as during recovery.

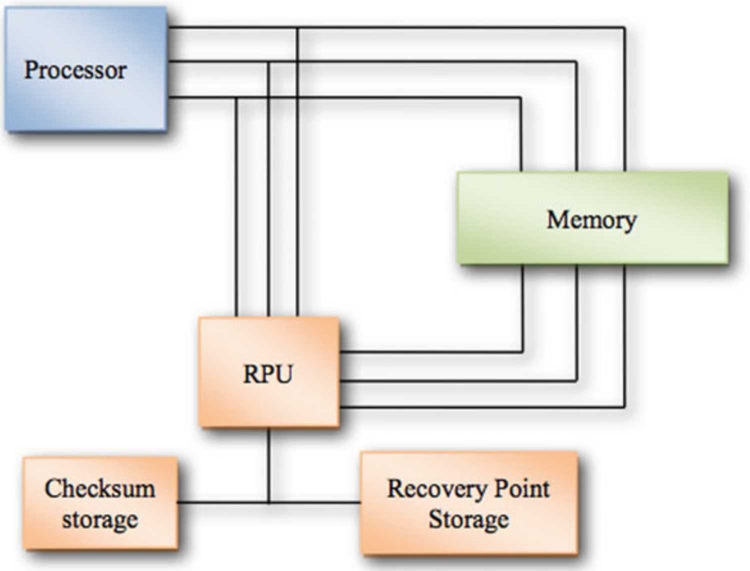


Fig. 8.3 Hardware design with RPU

If dual-port memory is used as system memory, the RPU can even have its own connection to the memory and create a recovery point concurrently to processing without performance impact.

On the other hand, the processor should also have accessed to this unit to perform tests and instruct the unit to generate a recovery point. Direct access for the RPU to the processor could also be allowed in order to store the full processor state as well. To simplify the implementation, the software can also store the current processor state (registers, program counter, etc.) to the memory and instruct the RPU to include this memory area in the RP generation.

The RPU has two operation modes, one that generates the recovery point and the checksum and one that only generates the checksum. The second mode is required for diagnostic purposes and serves well for accelerating the search of a correct RP (see Sect. 9.2.1 for details).

Of course, the checksum should be generated during the actual recovery point creation. If required, the checksums could also be stored distinct from the recovery point to prevent a fault from affecting the checksum and the recovery point at the same time.

8.5 Variable Recovery Point Creation

The principle of recovery point (RP) *tuning* [54] might be best explained with an example. Imagine a program Q with execution time T_q in which during compile and link time n recovery points are inserted. Please note that recovery points are only placed in specific locations, for example, procedure entrance.

In loops such as while, repeat until and for, no recovery points are placed, as it is in general not statically known how many times such a loop will be executed. Therefore, a consistent numbering of the recovery points as it is required as shown later is difficult.

Creating recovery points is expensive; we propose therefore to dynamically adjust the frequency of the recovery point creation. The basic principle is simple. The normal malfunction rate $\Delta\lambda_{\mathrm{est}}$ (faults/time) of the system can be calculated or estimated in advance as well as the maximum tolerable malfunction rate $\Delta\lambda_{\mathrm{max}}$.

If the current malfunction rate c is higher than the maximum rate, the system is considered as faulty. We now split the possible malfunction intensities into n equal intervals (Eq. 8.1).

$$\Delta\lambda = \frac{\Delta\lambda_{max} - \Delta\lambda_{est}}{n} \tag{8.1}$$

The theoretical minimum of $\Delta\lambda_{\mathrm{est}}$ is of course 0. Whenever a request for an RP creation is sent to the operating system, it can decide whether a real recovery point (RP) is created or just a fake one (FRP).

In case of a fake (not-created) recovery point, the call is simply returned to the application that results in a minimal performance impact. To implement this feature, the runtime keeps a table containing a mapping between the reliability modes and the indexes of the recovery points that should start a real recovery point creation.

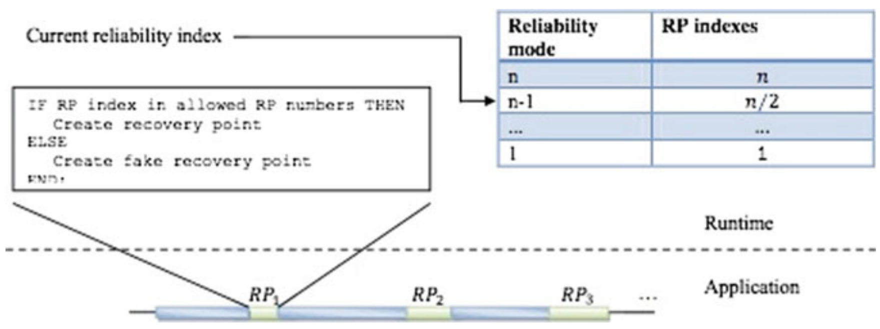
In other words, if the index i of the current recovery point does match the rule given by the current reliability mode introduced by runtime system, the recovery point is created; otherwise, a fake recovery point is created.

Figure 8.4 shows an example of such a rule, where the recovery point is only created if $i \bmod n = 0$, i.e., every nth recovery point. This rule can of course be adapted to the needs of the system and application.

The reliability mode is changed dynamically by the runtime system if a change in the current fault rate is detected or if an additional safety-critical application is started. The simplest scheme to adjust the current reliability index is by changing it linearly depending on the current malfunction rate of the system.

Every fault is logged by the runtime and can therefore be used for fine-tuning. It is assumed that the fault rate does not change in extremely short time, and that the index is only changed after program termination, which eases recovery.

The current index is therefore calculated as follows:



```
IF RP index in allowed RP numbers THEN
   Create recovery point
ELSE
   Create fake recovery point
END;
```

Fig. 8.4 Recovery point reliability mode

$$index = \frac{(\Delta\lambda_{max} - \Delta\lambda_{est})\lambda_c}{n} \tag{8.2}$$

The constraints of this formula must of course also be taken into account. For example, if the current malfunction rate $\Delta\lambda_c$ is below $\Delta\lambda_{est}$, the lowest reliability mode be chosen.

The formula above can be easily replaced by other functions, as needed.

However, to support this scheme of generating recovery points, the recovery point mechanism itself must provide one important property: Even by omitting one recovery point, the system must be able to fully restore the system state.

In other words, a recovery point must cover enough of the state space to cover the omitted recovery point as well. For recovery points on the level of the system, this property is automatically given and this approach is directly applicable. For recovery on the level of tasks (uncoordinated and coordinated), this approach is also applicable. For recovery on the level of procedures, this optimization cannot be used as the individual recovery points vary in their data coverage.

8.5.1 *Efficiency Analysis*

The redundancy used in the creation of recovery points is threefold: Time redundancy *HW (T), SW (T)* as the processor must prepare the recovery point frame and the recovery point unit both need time to perform their tasks.

Structural redundancy *HW(S), SW(S)* for the RPU, the storage and the RP management, and finally information redundancy SW(I), namely, the stored recovery point data as the main used redundancy. The checksum calculation unit in HW could also be added to *HW(S).*

Consider a runtime that supports real and fake recovery points. Consider now a hypothetical program P with two cases: (a) where all recovery points are created independent of the RP index and (b) where the creation of the recovery points is dependent on λ_c. We assume that c changes slowly over time and is therefore considered here as constant during one program run.

In terms of used memory, cases (a) and (b) are equal as the program itself does not know whether a recovery point is real or not. Thus the program prepares every recovery point. The needed memory for the recovery points is allocated in the user programs' memory space.

However, the allocation process itself is done by the runtime and not by the application, as the runtime knows the system topology and can therefore better place the memory area.

In terms of processor overhead, the time required for storing the data is omitted in the case of fake recovery points. The processor for further program execution can then use this time. Therefore, we are interested in the time saved by not storing the recovery point.

In case (a), the maximum time overheads W' given by the maximum malfunction rate and execution time of the FT system:

$$W_t^{'} = W_{\lambda_{max}} \tag{8.3}$$

where $W_{\lambda max}$ is the overhead of creating and storing recovery points during program execution. Currently, we only consider one program execution. $W_{\lambda max}$ consists of the time required to generate the recovery point in software (W_c) and the time to store it on stable storage (W_s) for all recovery points that are created during one program run.

$$W_{\lambda_{max}} = W_c + W_s \tag{8.4}$$

For case (b), the overhead for the recovery points W" is proportional to the malfunction frequency taken from the system log and the resulting number of real recovery points.

We assume that the fault rate does not significantly change during one program run and that the rate of fake recovery points is only adapted between program runs.

$$W_{\lambda_i} = W_c + \mu_i W_s \tag{8.5}$$

Here,

μ_i defines the percentage of the recovery points that are actually stored on stable storage in case of active reliability mode i.

Therefore, $(1-\mu_i)W_s$ defines the time that was saved during recovery. For n program runs, we get

$$W_t'' = \sum_{t=0}^{n} W_c + \sum_{t=0}^{n} \mu_i W_s \tag{8.6}$$

Here, (8.6) defines the total overhead to create the RPs during n program runs. In fact, the time we can save in n program iterations is

$$W_t'' = \sum_{t=0}^{n} (1 - \mu_i) W_s \tag{8.7}$$

The saved time is thus linearly depending on the recovery mode and the number and size of the generated but not stored recovery points.

8.6 Implementation Aspects

In this section, we focus on some aspects that are important for the implementation of recovery points. As already mentioned, the use of structured languages such as Pascal or Oberon improves the efficiency as the strong-type safety of the system allows to statically extract much more information than in other languages such as C or even assembler.

We present here for every program language structure a possible recovery point implementation strategy for recovery on the level of procedures and show the limitations. In Oberon, modules act as formal entities to separate concerns.

Figure 8.5 shows two modules, M1 and M2, where M2 imports the module M1, i.e., M2 gets access to the exported interface of M1.

As the basis for the discussion, we use Oberon-07 [57] as programming language but remove two language features that simplify the implementation considerably.

First feature removed is nested procedures as it is difficult to find the accessible state space of a nested procedure at compile time. Nested procedures do not add additional functionality to the language but simply provide a nonessential structuring mechanism.

The second feature we remove is the export of read-only module variables. Again, this helps the compiler to limit the state space and a procedure in a module can access. Read-only variables can be simulated by wrapping them into exported procedures.

Under these constraints, the only way of interaction between modules is by accessing exported procedures. Generating a recovery point at cross-module

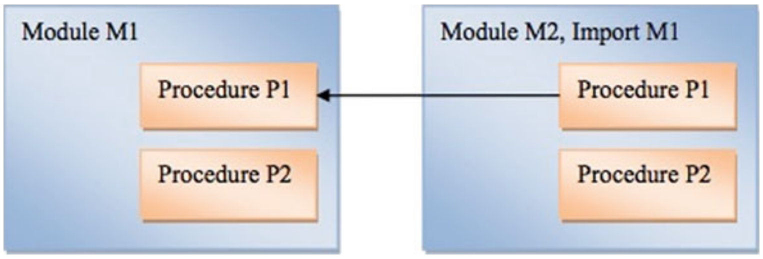


Fig. 8.5 Module imports

procedure calls seems to be a natural implementation approach. If this approach is not fine-grained enough, one could consider generating recovery points at every procedure entrance.

```
Rec1 = POINTER TO Rec1Desc;
Rec1Desc = RECORD
    i: INTEGER;
END;

Rec2Desc = RECORD (Rec1)
    i: INTEGER;
END;

PROCEDURE Foo ( i  : INTEGER; VAR a : Rec1Desc ; b : Rec1) : INTEGER
VAR
    int : INTEGER;
    c: Rec2;

BEGIN
            ….
END;
```

In the above example, all parameters *i*, *a,* and *b* of procedure *Foo* must be included in the recovery point. The size of all basic types as well as records passed as VAR parameters is already known at compile time, and thus the compiler can directly emit the code for the recovery frame preparation.

The more complicated case that needs discussion here is pointers. First of all, a pointer has a *type*, which means that this pointer can only point to objects of a certain type or an extension of the pointer type. The actual size of an object is accessible by its type tag.

However, the tricky part with pointers is that they can point to different objects over the course of time or even to invalid locations such as NIL.

In addition, a pointer could point to a linked list of objects that are manipulated in a procedure. The question that arises in this case is how to take these into account.

Two strategies could be used:

First, for every assignment to a pointer type (procedure parameters of pointer type are here also included), the assigned object must be included in a recovery procedure frame. On the implementation side, the compiler could add code to the type tag that creates a new recovery point covering this object.

Whenever an object is assigned to a pointer, the compiler emits code that triggers this code. Using flow analysis, it is also possible to find out if an object is write-accessed or only read-accessed in the context of a procedure. In the latter case, the object need not be stored. Obviously, this approach possibly results in a huge number of recovery points.

Second, saving the whole accessible memory space could lower the number of recovery points. However, this approach is very time-consuming and therefore only suitable if the recovery point interval is large.

Until now, module variables were not covered. However, all write-accessed module variables must be stored as well and thus included in a recovery procedure frame. Again, the accessed variables are marked and included by compiler control flow analysis.

Using resource contexts (see Sect. 11.5), another way of creating recovery points can be considered. As tasks in a resource group only have access to data in the same resource group, a recovery point could be created when a message to another task is sent or a message from another task is received. In this case, the whole memory area occupied by the context as well as all message parameters (which may not be of pointer type) is included in the recovery point. This results in a very simple recovery point implementation with a good trade-off between recovery point size and frequency.

In the case of working tasks that perform long calculations without interaction with other tasks, a system-generated timer could interrupt the currently ongoing calculation and create an RP of the currently active context.

If an RPU (see Sect. 8.4) is present in the system, the recovery point storage could even be done concurrently to the processing, as the language partitions the memory space and ensures isolation.

One could argue that software isolation is not sufficient, as memory faults could corrupt memory addresses and thus allow the software to access memory areas outside the current resource context. A simple range check in the hardware memory controller could solve this. To the application, only two registers are necessary, one containing the current lower access boundary, and one the upper access boundary. Memory accesses outside these boundaries trigger an exception.

Another key point in generating recovery points is the storage of input/output data. Specifically, drivers that read or write from devices must store the data, so that it can be replayed accurately. To support this feature, every driver can optionally provide such a buffer. It is up to the programmer to decide whether it is necessary and useful to provide these rollback buffers, as they might not be necessary depending on the use of the device (e.g., logging) or the size of the data that is required to be stored (e.g., disk accesses).

As an example, incoming sensor data might be important enough and small enough to be stored for use after a recovery. Of course, the location of this sensor data must be specially marked and protected, for example, by error detection and correction codes, so that it is not erased during a recovery.

As accessing a hardware device is usually done via memory-mapped I/O, i.e., direct memory access, the compiler cannot automatically extract necessary recovery means from the language and has thus to completely rely on the programmer.

Utmost care is required during the implementation of such memory-mapped I/O to guarantee first the correct behavior of the driver and second the correct handling of recovery point data.

8.7 Ultrareliable Storage

If no additional hardware support for storing the recovery point data is available on the target platform and cannot be added, the system has to fall back to a software-implemented stable storage, i.e., the data are not copied by the hardware to the stable storage, but the process is implemented in software and the data copied by the processor.

The stable storage is also exposed to external radiation and must therefore feature-built fault detection and correction mechanism.

A software implementation needs state variables to track the progress of the operation. In case of a full system backup, the state of the software recovery point mechanism is also stored on stable storage. In case of a recovery, the full state is overwritten; therefore, it's necessary to keep an inventory of the stored recovery points on stable storage.

If only part of the system is included in the recovery point, the state might also be kept in memory in addition to the inventory on stable storage.

However, code and data must be protected by some validation mechanism (checksum) to guarantee the correctness of the code and data in memory. Corrupted code can be reloaded from ROM, corrupted data restored from the stable storage inventory.

For more details about the recovery process, refer to Chap. 9.

Here, we present the following requirements for software-based stable storage:

- Simple design,
- Very fast data storage and retrieval,
- Optimized for linear access,
- Transparent device recovery in case of faulty read or write operation,
- Small memory footprint, and
- Low complexity controller.

8.7.1 Physical Storage Media

We consider five options for the storage of recovery point information and discuss shortly their advantages and disadvantages.

Hard disks are often used for stable storage in commercial systems. They are cheap, provide huge capacity, and their linear read and write performance is relatively fast. Their disadvantage is, however, that they are quite big, power hungry, need a complex and fast controller on the host side (S-ATA), and are prone to vibration and shock which disqualifies them for use in an embedded system.

Solid-State Disk Solid-state disks are designed as plug-in replacements for hard disks and use the (S-ATA) protocol with its already explained disadvantage. In contrast to hard disks, solid-state disks use an array of flash chips to store that actual data, resulting in extremely high read and write performance. In addition, they have built-in fault handling of failing chips and automatic wear leveling (see next section), which is welcome in this context.

However, the high implementation complexity of SSD disks (complex controller and firmware) which caused several manufacturers to release new firmware versions to eliminate bugs, sets a question mark behind the reliability of these disks.

Compact Flash Compact flashcards [58] are also flash based, exist for a long time, and are often used in industry as direct replacement for hard disks as they use the P-ATA protocol which is a predecessor of S-ATA. They are physically quite small and they have built-in support for wear leveling.

SD Card Secure Digital Cards (SD Cards) are widely used in consumer electronics such as photo cameras, smartphones, etc. They are available in different sizes, some of them very small. Transfer rates, however, are quite low, currently available SD card do not exceed write speeds of 20 Mbytes/second.

Flash Chips Flash chips are available in different technologies, sizes, and speeds. Solid-state disks use flash chips that support the Open NAND Flash Interface [59, 60] (ONFI) specification. These are the currently fastest chips on the markets featuring write speeds up to 300 Mbytes per flash chip. Recently, Intel flash chips were measured with a write speed of 156 Mbytes/s, which is faster than a hard disk. In contrast to the other options, flash chips are not removable. In the next section, we discuss the properties and limitations of flash chip in more detail.

Table 8.2 shows a more detailed comparison of the different storage options we consider. All options except hard disks use flash chips to store the content. The S-ATA-based options, i.e., hard disks and S-ATA, require a high complexity, high-speed controller (1.5 Ghz) on the host side which is not desirable in this context. SD cards and compact flash have too low write speed as storing the recovery points must be performed as fast as possible.

ONFI-based flash disks, which are also used in solid-state disks seem to be the currently best option to store recovery points as they have large capacity, high write speed, short access time, and a low complexity controller (100 MHz). In addition, they need low power, i.e., the max power consumption is according to the ONFI specification only 0.165 W. The downside, however, is that flash has write limitations that have to be taken into account when designing the stable storage architecture.

8.7.2 Flash Properties and Limitations

Flash memory is organized in blocks (usually 512 bytes) and in erase units consisting of a number of adjacent blocks (usually 32 or 64) [61]. Reads and writes are performed block-wise, and writes must be made only to previously erased blocks. Blocks can either be erased automatically or manually, where the former option is more comfortable while the latter is faster.

Table 8.2 Properties of stable storage

	Hard disk	Solid-state disk	Compact flash	SD card	Flash chip
Technology	Magnetic disk	Flash	Flash	Flash	Flash
Capacity	2 TB	512 GB	64 GB	32 GB	16 GB
Pin count	7	7	50	9	52–100
Interface	S-ATA	S-ATA	P-ATA	SD Card	ONFi
Access time	10 ms (R/W)	0.1 ms (R/W)	0.1 ms	1 ms read, 20–500 ms write	50 us
Transfer rate (MB/s)	130	500	70	20	160
Removable	Yes	Yes	Yes	Yes	No
Host controller complexity	High	High	Moderate	Low	Low
Device implementation complexity	High	Very high	Moderate	Low	Low
Physical robustness	Low	High	High	High	High

The number of erases per unit before “wearing out” is limited, typically to some number between 10000 and 100000, in newer technologies even lower. In the interest of longevity of the flash chip, the use of a “wear leveling” strategy is highly advisable. Wear leveling means that erase cycles and write cycles are evenly distributed across the memory chip. Traditional file systems are unsuitable for flash chips because they exhibit hot spots such as metadata fields that are frequently updated.

8.7.3 Analysis and Design Considerations

As mentioned above, the use of a standard file system is unsuitable as it typically exhibits hot spots. Much of the research [62] into overcoming this problem introduces some virtual-to-physical block number mapping.

Two kinds of data structures are typically suggested for this purpose. Direct maps map a logical block number (index i) to its physical sector number. Unfortunately, such data structures have typically a footprint in the order of several megabytes [63].

Inverse maps store in location i the virtual block number corresponding to sector i. These maps are usually stored on the flash disk itself and are mainly used for regenerating the direct map at boot time.

However, after an unexpected reboot, it would take considerable time to rebuild the direct and indirect maps, which are incompatible with the request of a fast reboot time. Also, many of these algorithms are patented.

While algorithms based on virtual block mapping can greatly extend the lifetime of flash memory, this comes at the price of increased complexity, of a large memory footprint, and of a garbage collection mechanism for reclaiming invalid sectors. As this is again incompatible with real-time constraints, it is not an option in our case.

Another approach is the use of a log-structured file system such as JFFS [64]. Log-structured file systems do not structurally separate metadata and payload data but instead maintain a comprehensive log of all performed operations in chronological order.

While wear leveling is implicit in such systems, they still suffer from the garbage collection problem, which again disqualifies them for use.

Therefore, we refrain from using such advanced storing schemata and propose a simple circular buffer structure, where recovery points are stored linear on the flash, aligned to flash blocks. Whenever a new erase unit is entered, an erase operation is performed as a preparation for subsequent writing.

The obvious drawback of this scheme is internal fragmentation if the size of a recovery point is not an exact multiple of the elementary block size. We consider this as acceptable in particular because recovery points usually occupy a high number of blocks, which results in a very low percentage of lost space.

8.7.4 Implementation Aspects

In addition to the recovery points, some metadata is recorded on the stable storage: The Stable Storage (SS) header occupies n erase blocks and describes the current contents of the stable storage. If only full system snapshots are supported, no header at all is required, as the recovery points have fixed size.

The stable storage is partitioned in m buckets, aligned to erase blocks and the recovery points are stored in increasing order in the buckets. The latest used bucket is kept in memory to ensure fast access to the last stored recovery point and a sequence number stored together with the recovery point ensures that the recovery point history is recoverable if the memory is corrupted due to a fault.

If recovery points do not cover the whole system, i.e., task-based, the size of each recovery point varies and the stable storage cannot be partitioned in fixed-size buckets.

Instead, a header is introduced that contains a chronological list of stored recovery points. Such a header entry contains a global sequence number (analog to a timestamp), and the start and end block where the recovery point is stored.

The same sequence number, a unique fingerprint, a state variable, and a checksum calculated over the whole recovery point are stored with the actual recovery point data. The recovery points are again aligned to erase blocks, and the index of the latest stored recovery point is kept in memory for fast access.

In both variants, the software must check whether there is enough space free on the stable storage. If not, additional space is freed by deleting (erasing) the oldest recovery point entries on the stable storage. The same is done for the recovery point index entries if no free space is left for an index.

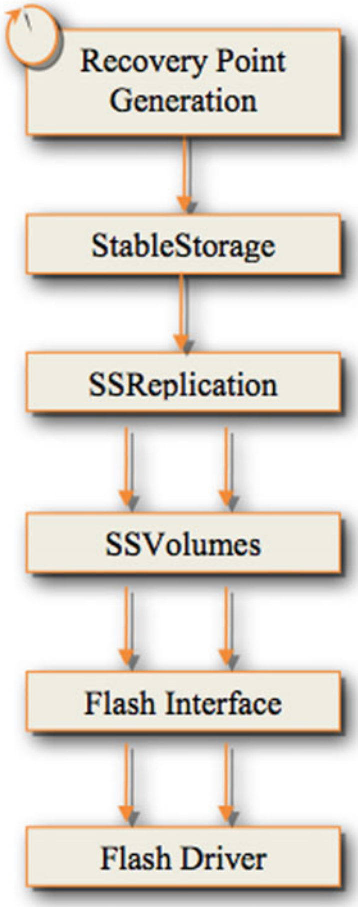
Note that at least two erase pages are needed for the index, as the smallest entity that can be erased is an erase block. The index size must be chosen according to the recovery strategy, based on the expected recovery point size.

The recovery point entries are stored in a log-based way, and thus it is likely that index entries pointing to already overwritten recovery points are present.

Whenever a recovery point is read, the software therefore validates the recovery point by checking the fingerprint, the sequence number and the checksum over the whole recovery point. This simple design guarantees equal distribution of writes across the whole flash and therefore maximum mean time to failure.

Figure 8.6 shows the implementation of stable storage as a layered modular system.

Fig. 8.6 Stable storage implementation



Recovery Point Generation This module implements a recovery point algorithm.

Stable Storage This module provides an API for accessing the stable storage, i.e., reading and writing recovery points, but also administrative tasks such as clearing and initializing stable storage, get the list of stored recovery points, etc. Usually, the recovery points are initialized at system boot up, not by clearing the stable storage, but by finding the last used recovery point and continuing with the next sequence number. In the state variable of the first recovery point, a flag is set to mark this recovery point as the first after boot up, so that the recovery process can recognize this recovery point as the first valid one.

SSReplication This layer partly implements the error detection and correction algorithms. In detail, the module is responsible for computing the CRC-32 for each block. The CRC-32 is automatically generated during write operations and automatically checked during read operations. All read/write operations are performed concurrently on two distinct flash chips.

At each read operation, data integrity is checked automatically. If a faulty CRC-32 is detected, the healthy copy is used to fix the data by merely rewriting the faulty block and an error indication is returned. If one of the flash chips fails permanently, the system still continues to record data on the health card, and a log message plus an appropriate status code are generated to indicate the failure.

SSVolumes This module implements a disk volume object that represents a logical volume, in our case the whole flash chip, and extends its functionality with the ability of erasing erase units on the flash disk. In case of a malfunctioning flash chip, both the controller and the chip are automatically reset and the failed operation is retried.

If it fails again, an error code is returned to indicate the failure. The abstraction as a volume allows to transparently replace the flash chips with another device.

Flash Interface/Flash Driver These two modules implement the ONFI flash standard, an interface for reading, writing, and erasing blocks, and some administrative support such as reading chip information, etc. For ease of implementation, the block-abstracted mode of ONFI [60] is used.

To summarize, this simple implementation of a possible stable storage mechanism optimally deals with the limitations of current flash cards and provides reliable and fast data access.

8.8 Summary

- We revised existing methods of system preparation for recovery;
- These methods can be considered pursuing two distinct approaches: forward error recovery and backward error recovery.
- The first approach resets software to a clearly defined state in the future.

- The processing continues then from this state. We showed that this approach is not generic and highly application-dependent.
- Backward error recovery, however, is generic as its main principle is to create recovery points containing the state of the system or at least parts of it which software can restore to eliminate the effects of the faults.
- For recovery preparation, we revised and compared existing approaches, introduced recovery on the level of procedures and analyzed the advantages and disadvantages of this approach. We then showed a generic approach of hardware support for creating recovery points and explained a method of adapting the current recovery point creation rate to the current fault rate.
- Schemes of possible "embedding" of recovery procedures into runtime system were also described, showing flexibility of system software support of recoverability.
- We also analyzed hardware system and technological supports of recovery processes that have to be implemented for safety-critical systems. We presented an approach of a highly reliable software-based stable storage together with a possible implementation.
- We distinct commercial and real-time systems in terms of requirements and implementation of reliable functioning at the level of architecture, system software and hardware elements involved.

References

1. Liedtke J (1995) On micro-kernel construction. In: Proceedings of the fifteenth ACM symposium on operating systems principles, SOSP '95. ACM, New York, NY, USA, pp 237–250
2. Monkman S, Schagaev I (2013) Redundancy + reconfigurability = recoverability. Electronics 2:212–233. ISSN 2079-9292, https://doi.org/10.3390/electronics2030212
3. Haeberlen A et al (2000) Stub-code performance is becoming important. In: Proceedings of 1st conference on industrial experiences with systems software, vol 1. USENIX Association, Berkeley, CA, USA, p 4
4. Wirth N, Gutknecht J (1992) Project Oberon: the design of an operating system and compiler. Addison-Wesley, Wokingham
5. Шагаев И., Берштейн А. Исследования систем команд их влияние на архитектуру современных ЭВМ. Зарубежная радиоэлектроника, 1989 N7, 8
6. Johannes M (2002) The active object system—design and multiprocessor implementation. ETH Zurich, Zurich
7. Mossenbock H, Wirth N (1991) The programming language oberon-2. Technical report, Johannes Kepler Universitat Linz
8. Martin R, Wirth N (1992) Programming in Oberon: steps beyond Pascal and Modula. Addison-Wesley, Wokingham
9. Wirth N (1977) Modula: a language for modular multiprogramming. Softw: Pract Experience 7(1):1–35
10. Wirth N (1985) Programming in Modula-2. Springer, New York
11. Wirth N (1971) The programming language Pascal. Acta Informatica 35–63
12. Wirth N (1977) The use of Modula. Softw—Pract Experience 7

13. Kaegi-Trachsel T, Gutknecht J (2008) Minos—the design and implementation of an embedded real-time operating system with a perspective of fault tolerance. International Multiconference on IMCSIT 2008, 20–22 October 2008, pp 649–656
14. Fabry RS (1974) Capability-based addressing. Commun ACM 17:403–412
15. Schagaev I (1990) Using software recovery facilities for determining the type of hardware faults. Autom and Remote Control 51(3)
16. McCluskey E et al (2002) Control-flow checking by software signatures. IEEE Trans Reliab 51(1):111–122
17. Schagaev I (1989) Computing process recovery algorithms. Avtomat Telemekh 4
18. Oh N, Mitra S, McCluskey (2002) Error detection by diverse data and duplicated instructions. IEEE Trans Comput 51(2):180–199
19. McCluskey E et al (2002) Error detection by duplicated instructions in superscalarprocessors. IEEE Trans Reliab 51(1):63–75
20. Sogomonyan E, Schagaev I (1988) Hardware and software for fault-tolerant computing systems. Autom Remote Control 49:129–151
21. McCluskey E et al (2000) Dependable computing and online testing in adaptive and configurable systems. IEEE Des Test Comput 17(1):29–41
22. Mukherjee S et al (2002) Detailed design and evaluation of redundant multi-threading alternatives. In: 29th annual international symposium on computer architecture, pp 99–110
23. Dal Cin M et al (1993) Fault tolerance in distributed shared memory multiprocessors. In: Parallel computer architectures: theory, hardware, software, applications. Springer, London, pp 31–48
24. Candea G, Kawamoto S, Fujiki Y, Greg Friedman G, Fox A (2004) Microreboot: a technique for cheap recovery. In: Proceedings of the 6th conference on symposium on operating systems design & implementation, vol 6. USENIX Association, Berkeley, CA, USA, p 6
25. Deconinck G et al (1993) Survey of backward error recovery techniques for multicomputers based on checkpointing and rollback. Int J Model Simul 18:262–265
26. Elnozahy E et al (2002) A survey of rollback-recovery protocols in message-passing systems
27. Lampson BW (1981) Atomic transactions. In: Distributed systems—architecture and implementation, an advanced course. Springer, London, pp 246–265
28. Randell B (1975) System structure for software fault tolerance. IEEE Trans Softw Eng 1: 220–232
29. Lamport L et al (1985) Distributed snapshots: determining global states of distributed systems. ACM Trans Comput Syst 3:63–75
30. Attig N, Sander V (1993) Automatic checkpointing of NQS batch jobs on CRAY unicos systems. In: Proceedings of the cray user group meeting, pp 250–255
31. Strom R, Yemini S (1985) Optimistic recovery in distributed systems. ACM Trans Comput Syst 3:204–226
32. Lorenzo A, Keith M (1996) Trade-offs in implementing causal message logging protocols. In: 15th ACM symposium on principles of distributed computing, PODC '96. ACM, New York, NY, USA, pp 58–67
33. Borg A, Baumbach J, Glazer S (1983) A message system supporting fault tolerance. In: Proceedings of the ninth ACM symposium on operating systems principles, SOSP '83. ACM, New York, NY, USA, pp 90–99
34. Strom R, Bacon D, Yemini S (1988) Volatile logging in n-fault-tolerant distributed systems. In: Digest of papers eighteenth international symposium on fault-tolerant computing, FTCS-18, pp 44–49
35. Elnozahy E, Zwaenepoel W (1992) Manetho: transparent roll back-recovery with low overhead, limited rollback, and fast output commit. IEEE Trans Comput 41(5):526–531
36. Johnson D, Zwaenepoel W (1987) Sender-based message logging. In: Digest of papers: 17 annual international symposium on fault-tolerant computing. IEEE Computer Society, pp 14–19

37. Smith S, Johnson D (1996) Minimizing time stamp size for completely asynchronous optimistic recovery with minimal rollback. In: Proceedings of the 15th symposium on reliable distributed systems, SRDS '96. IEEE Computer Society, Washington, DC, USA, p 66
38. Bhargava B, Lian S, Leu P (1990) Experimental evaluation of concurrent checkpointing and rollback-recovery algorithms. In: Proceedings of sixth international conference on data engineering, pp 182–189
39. Tamir Y, Squin C (1984) Error recovery in multicomputers using global checkpoints. In: International conference on parallel processing, pp 32–41
40. Tong Z, Kain R, Tsai W (1992) Rollback recovery in distributed systems using loosely synchronized clocks. IEEE Trans Parallel Distrib Syst 3:246–251
41. Koo R, Toueg S (1987) Checkpointing and rollback-recovery for distributed systems. IEEE Trans Softw Eng 23–31
42. Janakiraman G, Tamir Y (1994) Coordinated checkpointing-rollback error recovery for distributed shared memory multicomputers. In: Proceedings 13th symposium on reliable distributed systems, pp 42–51
43. Janssens B, Fuchs WK (1994) Reducing inter-processor dependence in recoverable distributed shared memory. In: Proceedings of reliable distributed systems, pp 34–41
44. Li K (1986) Shared virtual memory on loosely coupled multiprocessors. PhD thesis, New Haven, CT, USA. AAI8728365
45. Bershad B, Zekauskas M (1991) Midway: shared memory parallel programming with entry consistency for distributed memory multiprocessors. Technical report
46. Huang Y, Wang Y (1995) Why optimistic message logging has not been used in telecommunications systems. In: FTCS-25, pp 459–463
47. Johnson BD (1990) Distributed system fault tolerance using message logging and checkpointing. PhD thesis, Houston, TX, USA. AAI9110983
48. Lamport L (1978) Time, clocks, and the ordering of events in a distributed system. Commun ACM 21:558–565
49. Brown L, Wu J (1995) Snooping fault-tolerant distributed shared memories. J Syst Softw 29:149–165
50. Plank J (1993) Efficient checkpointing on MIMD architectures. PhD thesis, Princeton, NJ, USA, 1993. UMI Order No. GAX93-16087
51. Schagaev I. Algorithms of computation recovery. Automation and Remote Control, 7, 1986. 26, 36, 65, 122
52. Schagaev I (1987) Algorithms for restoring a computing process. Autom Remote Control 48 (4). 26, 65, 122, 141, 149
53. Schagaev I (1989) Instructions retry in microprocessor recovery algorithms. In: IMEKO—FTSD symposium. 2
54. Schagaev I (1986) Relationship between the formation of program recovery points and equipment reliability indices. Autom Remote Control 47:124
55. Blaeser L, Monkman S, Schagaev I (2014) Evolving systems Worldcomp 2014. In: Proceedings of the international conference on foundations of computer science FCS'14. CSREA Press, ISBN: 1-60132-270-4
56. O'Brian F (1976) Rollback point insertion strategies. In: Digest of papers 6th international symposium fault-tolerant computing, FTCS-6
57. Wirth N (2008) Oberon-07 language report. Technical report, ETH Zurich
58. Compact Flash Association (2007) Cf+ and compact flash specification revision 4.1. Technical report
59. ONFi Workgroup (2011) Open NAND flash interface specification 3.0. Technical report, ONFI Workgroup
60. ONFi Workgroup (2009) Open NAND flash interface specification: block abstracted NAND. Technical report, ONFi Workgroup
61. SanDisk Corporation (2002) Host design considerations: NAND MMC and SD-based products. Technical report, SanDisk Corporation

62. Gal E, Toledo S (2005) Algorithms and data structures for flash memories. ACM Comput Surv 37(2):138–163
63. Chang L, Kuo T (2004) An efficient management scheme for large-scale flash memory storage systems. Technical report
64. Woodhouse D (2001) JOFFs: the journaling flash file system. Technical report, Red Hat, Inc

Chapter 9
Recovery: Searching and Monitoring of Correct Software States

Abstract The last of the three GAFT processes is called recovery and recovery monitoring. After the detection of an error and possible reconfiguration, the last step is recovering the software, which means that the effect of the error on the software must be eliminated. In line with the previous chapters and [1–6], the recovery consists of restoring the last recovery point and continuing the processing. But is this really sufficient? What if latent faults exist in the system and manifest themselves in the system but trigger some detection schemes an arbitrary time later? Assuming this reasonable and unpleasant sequence of events, it becomes clear that just restoring data and program from the last stored recovery point is not enough. We have to admit that we do not have any guarantee that fault is now eliminated: even when hardware is restored or even reconfigured—we have erroneous states of software recorded in recovery points. Thus, we have to consider the recovery process itself and analyze which classic algorithms are applicable and fit the purpose of efficient recovery. We introduce and analyze three recovery algorithms that are able to ensure successful recovery by iteratively go through all stored recovery points.

9.1 Recovery as a Process

If a fault is detected in the system, appropriate recovery actions must be executed to eliminate the fault and to recover the software.

The software recovery process consists of the following actions:

- interrupt processing,
- identify the found fault,
- eliminate the found fault,
- check the integrity of the system, and
- recover software.

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_9

First, a diagnosis procedure signals a fault via an interrupt or a system call. The system must immediately stop processing data and handle the fault to prevent the fault from spreading in the system.

If the fault identification mechanism that found the fault is not powerful enough to diagnose the fault type, it is necessary to run further diagnostic routines, starting from repeating of instruction [4, 5]. In specific, every hardware driver must provide a diagnosis procedure that identifies the fault type. If the checking is fully hardware-based, this software procedure might only read the fault registers (Sect. 7.3), if it is software-based, it might run some more sophisticated tests. Based on the found result, the checking procedure eliminates the fault. The assumption here is that every device driver provides a catalogue of fault elimination mechanisms, for example, "power off" and "power on" the faulty device and rerun the test to show the absence of faults.

If the device is still faulty, it should be excluded from the current working set and turned off (Sect. 7.4). The hardware itself should now be fault-free again; however, software might have been affected by the fault. Dependent on which level the checking is performed (instruction, procedure, task, etc.) the software is recovered differently.

Recovery on the instruction level As the fault was detected during the instruction execution, no software recovery is necessary. However, in the case of permanent faults, a hardware reconfiguration is still required to eliminate the fault.

Recovery on the procedure level and above In this case, it is assumed that faults are detected in the time frame of procedure execution. It is therefore sufficient to perform a rollback based on the last stored recovery point. In fact, a rollback is the inverse function of creating a recovery point, with the difference that the data is not copied from the memory and stored on the recovery point storage, but the data is read from this storage and copied back to the memory.

Prior to that, recalculating the checksum and comparing it to the stored one must ensure the consistency of the recovery point. If the comparison succeeds, the recovery point is considered as valid; otherwise, it is discarded, and the previous recovery point is used for recovery.

In fact, recovery on the level of procedures does not differ from recovery on a higher level. First, the content of the recovery point is restored, then the stack and base pointer, and as the last step, the processing is resumed by jumping to the restored instruction pointer.

From an implementation perspective, the code, which is executed to restore the recovery point, should be executed from ROM, using a reserved memory area for temporary data so that recovered data cannot overwrite data of the recovery process itself. In case of Oberon using module variables only (no heap data structures) would be sufficient.

However, the recovery process still needs its own private stack. Simulating a private heap in a statically allocated buffer would be of course also possible, but not elegant and unnecessary complicated.

The procedure described here reflects only the general part of how to restore a recovery point. In case of coordinated recovery points or message passing systems, additional steps might be necessary.

In the next chapters, we give more details about one specific approach that includes new language extensions.

9.2 Modified Linear Algorithm

The system model used in the previous chapter was based on the assumption of having a fail-stop system. Recovery procedure includes two phases: (a) restoring of system state from the last consistent recovery line and (b) resuming processing.

However, in real life, faults might stay latent in the system for a long time until they trigger an error, which also implies that the last recovery line still contains the latent fault. Optimizations such as reducing the required storage by only keeping the last recovery line can thus lead to a non-recoverable system that must be restarted.

What happens if we change the system model from a fail-stop system to a system where faults can stay undetected for a long time in the system? In this case, a method is required to find the exact appearance of the fault in the system, i.e., the last recovery point that was not affected by the fault.

We present here an approach of how to cope with faults existed in the system undetected an arbitrary period of time. Our approach and implementation algorithm called Modified Linear Recovery (MLR). This section is based on the papers [7–10] and adapted to our needs.

Using an ordered set of recovery points (Fig. 9.1), we split the program execution into pieces of exactly the same execution length.

Let RP be the set of recovery points that is generated during the program execution. By ordering the set RP, we achieve sequential consistency, which means that if all non-determinants are equal, for any RPi and RPj ($i < j$) the computing process passes after the recovery of RPi through all subsequent states RPi + 1, RPi + 2, ..., up to RPj and on.

As in Sect. 8.4, a checksum CSi is calculated for every RPi analog to the probabilistic approach introduced in the previous chapter.

Generating the set of checksums CS concurrently with RP effectively saves time for generation of the recovery point and also for looking up a correct RPk from which the task can be continued. Using the method of Modified Linear Recovery (MLR) when the type of hardware faults is known is possible to even recover from multiple sequential faults. To solve this problem we must answer the following questions:

- Does redundancy used for recovery (recovery points) is sufficient to determine the type of a fault and therefore be involved in checking and recovery?

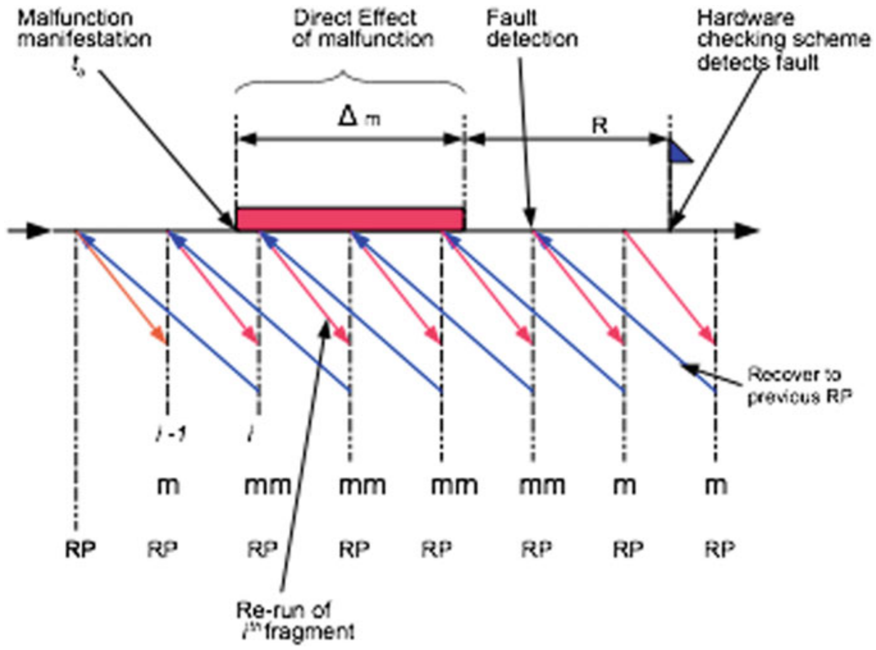

Malfunction manifestation
t_0
Direct Effect of malfunction
Δm
Fault detection
R
Hardware checking scheme detects fault
Recover to previous RP
i-1
i
m
mm
mm
mm
mm
m
m
RP
RP
RP
RP
RP
RP
RP
RP
Re-run of i^{th} fragment

Fig. 9.1 Ordered steps of recovery points and iteration of recovery

- Is it possible to determine a correct recovery point when a permanent fault (failure) has occurred?
- Does the MLR algorithm remain valid for the case of successively occurring faults?

9.2.1 Characteristics of Modified Linear Recovery Algorithm

The purpose of the modified linear recovery algorithm is the detection and elimination of hardware faults during operation, followed by a search for the last correct program state to recover the system. The procedure for the search is explained further down.

Concurrent to ongoing tasks, recovery points are generated together with their fingerprints, the checksums. Recovery is in essence done in a two-step approach. First, the system is recovered to the state of the last recovery point RP_i, and the processing is resumed. If the test procedures that identified the error in the first place do no longer detect the error, the system is considered as restored.

If, however, the testing procedures still detect the error, the fault in the system is a long-lasting fault. Thus, the system is recovered to RP_i' and the processing is resumed. When the program reaches RP_i, i.e., the point in the program execution when RP_i was created, the recovery point mechanism creates a new recovery point RP_i' and the checksum CS_i'. The checksums CS_i and CS_i' are now compared. In case of $CS_i = CS_i'$, the fault is still present, in case of $CS_i \neq CS_i'$ the program calculations changed, i.e., the fault is eliminated.

This procedure is executed recursively until the system is recovered or restarted if recovery completely failed. During recovery, no new recovery points RPs are created, of course.

Let us assume that a malfunction occurred at time t_a and that the duration (m) of its immediate effect on the computing process exceeds the execution time of one or more program segments between the recovery points:

$$\Delta m > xt_f \tag{9.1}$$

Here x (x > 1) is the number of program fragments, and t_f is the execution time of each fragment. In case of timely even distributed recovery points, t_f is constant. During the interval Δm (Fig. 9.1), both the task and the generation of the RPs are executed by faulty hardware.

During interval R, starting at the point in time the effect of the malfunction ends, to the time the malfunction is detected by some check or test facilities, the program is executed on correctly working hardware but with incorrect data that was modified by the malfunction. Obviously, the recovery points that are generated during this time contain also incorrect data.

Is it possible now to find the last correct *RP*? There is an answer actually. At every iteration of the recovery procedure, the recovery depth is increased by one step or segment, the program executed and the resulting checksums *CS* and *CS'* compared. Obviously, *CS* and *CS'* match in the interval R—it is indicated by the letters m (match) in Fig. 9.1 on the right. Of special interest is the interval Δm.

During the generation of the recovery points, the content of some of the RPs might have been influenced directly by the effect of the malfunction on the program execution and by the use of incorrect data affected by the malfunction.

During recovery, the program, which was affected by the malfunction (period Δm), is now run on properly working hardware (either tested or checked by hardware schemes). Therefore, the program is repeated under different conditions with incorrect data on correctly operating hardware. Consequently, the pairs *CS* and *CS'* will not match in the interval Δm, as indicated in Fig. 9.1 by the letters *mm* (mismatch).

When the system is recovered to the last recovery point before the malfunction occurred and the program segment executed, the resulting checksums *CS* and *CS' match*, since the hardware and software that was used during the real data processing and during the recovery were in the same state. The task has thus been recovered and can be continued [3, 7, 8, 11].

In the optimal case, an *RP* includes the CPU registers and the affected memory areas such as the program code and data sections, the stack, the heap, and the relevant operating system data structures. If operating system structures and CPU registers are not included in the *RP*, the *RP* and the respective *CS* cover exactly the application, which might lead to unsuccessful recovery as the fault affected kernel data structures (i.e., I/O data).

Special care must be taken when choosing the checksum generator. A simple modulo 2 adder, for example, can detect any single fault in the program or data and will result in different *CS*.

However, for multiple faults, this is no longer guaranteed. CRCs or Hamming codes provide a better fault coverage and easy hardware implementation. Details about CRC analysis can be found in [12], for Hamming codes in [13, 14].

Cryptographic hash values are the alternative, which are specifically designed for collision avoidance and do not rely on specific error patterns. The downside, however, is that it's not possible to make any guarantees about fault coverage, and their implementation in hardware is complex and expensive.

If, as proposed in [15], analysed in [16] and developed in [7, 8], the *RPs* cover only current program data, a malfunction corrupting the program code or execution might not result in different *RP, CS* pairs. Such *CS* generation impairs the diagnostic power properties of checksum comparison, i.e., it may not be possible to recognize the type of the fault and to determine the correct *RP*.

If a fault already exists at boot up time, the *RP* and *CS* and the comparison of the *CSs* of the first and following runs allow to identify such a fault as a permanent fault since the fault will never disappear. When fault exists ate the boot up time the RP and CS identify such fault as permanent.

9.2.2 MLR Execution in Case of Permanent Faults

The detection of malfunctions and permanent faults uses the same techniques. In both cases, the checking circuit or checking process identifies the fault and the effect of the fault is seen in corrupted data or program code and also misbehaving program execution. This results eventually in the generation of one or more RPs that include corrupt data and program state. However, recovery depends on the fault type as explained Sect. 4.4.

Thus, the MLR algorithm can in the case of permanent faults not directly be applied. The system must eliminate the permanent fault first or go into a degraded state, by excluding the faulty component. The MLR algorithm can therefore only correctly restore the system state if the fault type is known.

If we assume that in case of a permanent fault the faulty unit is replaced with a fault-free one, resulting in the elimination of the permanent fault, then the MLR algorithm successfully recovers the system.

However, if the fault detection mechanism identified the wrong unit as the faulty one, then the MLR algorithm results in a loop as the still present faulty unit in the system triggers the fault again after recovery.

Algorithms that eliminate permanent hardware faults usually consist of three successively executed phases (denoted by *A*, *B*, and *C*, respectively): A: determination of the fault type, B: elimination of permanent hardware faults (hardware reconfiguration), and C: recovery of the correct program state and continuation of the computing process. These three phases are of course a subset of GAFT.

From GAFT, we know that the permanent hardware fault elimination depends on the results of the fault type detection. During fault type detection, the program segment or instruction is rerun, during which the checking logic or test program identified the error.

For the recovery of the correct program state, we recommend to use the MLR algorithm which requires less time to establish the last correct program state than other known algorithms of the same family. The time required to execute a full recovery is the sum of the time required to perform the three steps fault type detection, permanent hardware fault elimination, and recovery of the correct program state. As no computation is performed while the system recovers, the minimization of the time required for full recovery is very important to reach maximum availability and to keep the system response time as short as possible.

Let us now examine the recovery process when the MLR algorithm is used for the fault type determination. Assume that a given system has a hardware failure which has been identified by the checking logic or checking procedures in the program segment j.

As long as the permanent fault is not eliminated, the computing process even if recovered by the current correct RP will remain corrupted. At the same time, if the fault affected the system operation during the first generation of an RP, then, *CS* and *CS'* will match in reruns of the next program segment due to the fault being permanent. If a program fragment preceding the occurrence of a fault is rerun, a non-eliminated fault will corrupt the result of the computation and *CS* and *CS'* will not match. The same checksum mismatch will occur when the recovery depth is further incremented.

Let us now consider a program execution with *n* program segments. Depending on the time of the failure manifestation, the match–mismatch sequences have the form, shown in Figs. 9.2, 9.3 and 9.4.

By examining the sequence patterns, it is possible to decide whether the fault is a malfunction or a permanent fault. The first (rightmost) mm in the sequences of Figs. 9.2 and 9.3, which may be preceded by one or more m's, corresponds to an RP generated before the failure has been manifested.

This RP contains, therefore, the last correct state of the process. For the cases shown in Figs. 9.2 and 9.3, the process can be recovered and continued from this RP after the hardware failure is fixed. If no mismatch is observed during all n recovery steps (Fig. 9.4), the failure occurred before the leftmost RP was created.

If earlier recovery points exist, the recovery process simply continues, if after all *n* recovery steps the first RP is reached, then the only way to recover is to reboot the

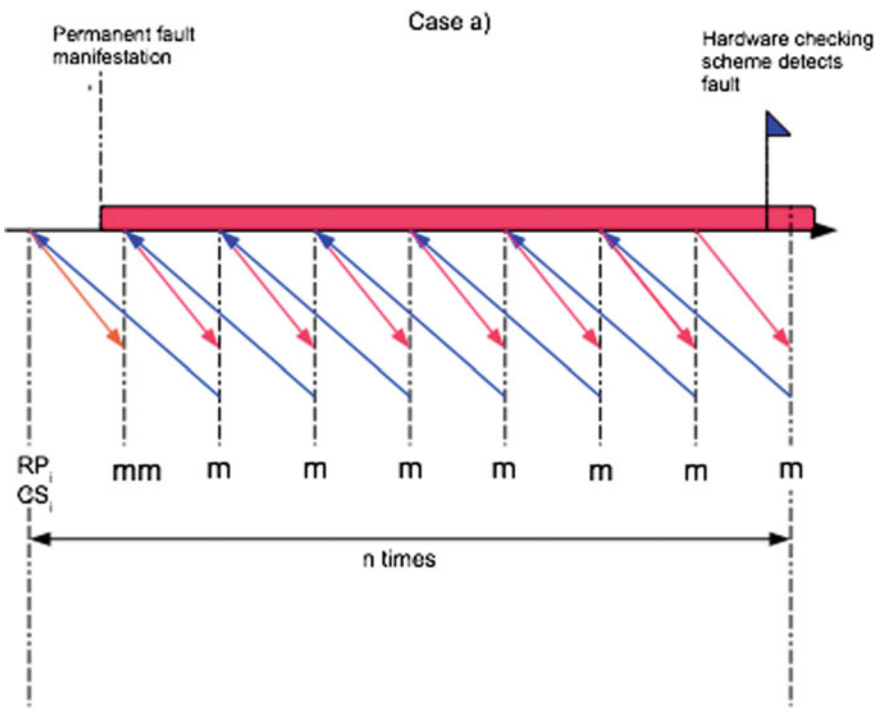

Case a)
Permanent fault manifestation
Hardware checking scheme detects fault
RP$_i$
CS$_i$
mm
m
m
m
m
m
m
m
n times

Fig. 9.2 Permanent HW fault elimination, case (a)

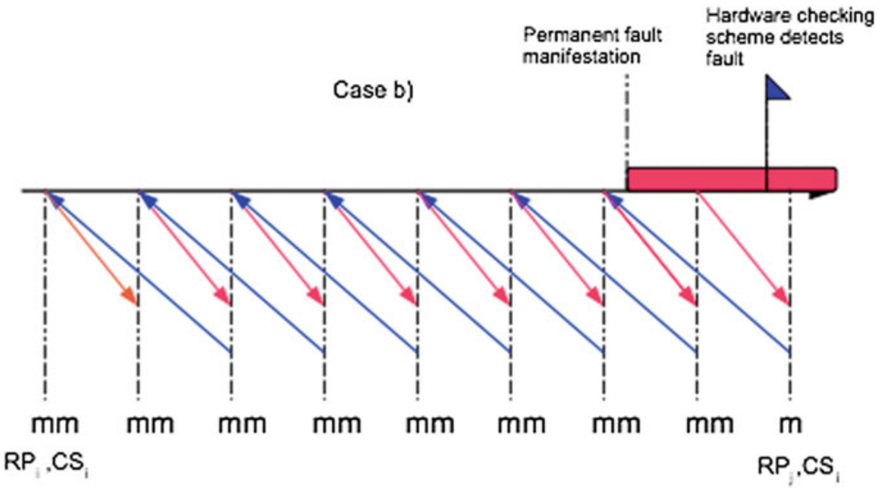

Case b)
Permanent fault manifestation
Hardware checking scheme detects fault
mm
mm
mm
mm
mm
mm
mm
mm
m
RP$_i$,CS$_i$
RP$_j$,CS$_i$

Fig. 9.3 Permanent HW fault elimination, case (b)

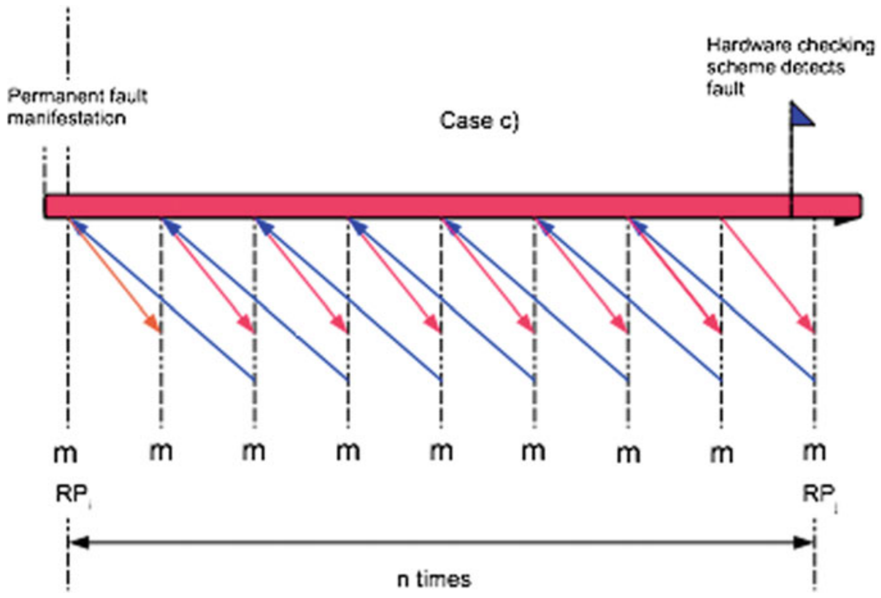

Permanent fault manifestation
Case c)
Hardware checking scheme detects fault
m
m
m
m
m
m
m
m
m
RP
RP
n times

Fig. 9.4 Permanent HW fault elimination, case (c)

system and use the BIST (Built-In Self-Test) procedures to thoroughly analyze the system at boot up time.

The different match/mismatch sequences lead to the following conclusion:

1. The type of fault (malfunction or permanent fault) can be determined from the form of the CS match/mismatch sequence.
2. If during recovery in the first step, a match is detected, then the fault still exists and the fault type is a permanent fault otherwise a malfunction has the computation changed.
3. The execution time of A and B is minimal since phase A is used simultaneously for finding the correct state of the computing process and for the elimination of any corruptions due to the fault.

9.2.3 The MLR Algorithm in Case of Several Successive Faults in Case of Permanent Faults

The MLR algorithm is not restricted to single faults and allows to find the correct state of a process also in case of several successive faults. Consider the example shown in Fig. 9.5 which represents a case of two successive faults (malfunctions) ε_1 and ε_2.

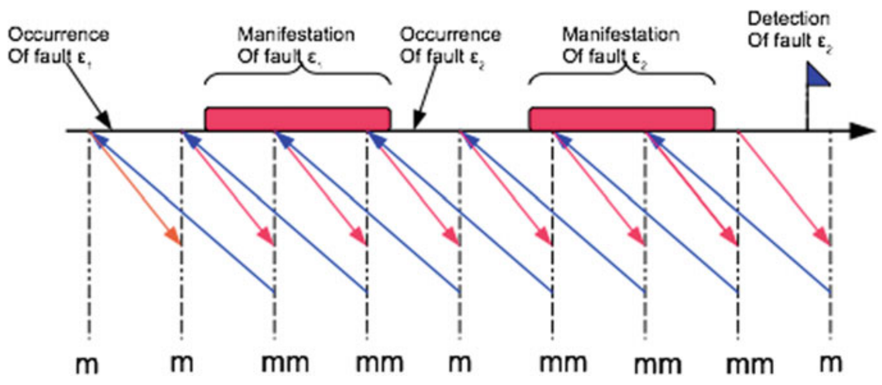


Fig. 9.5 Permanent HW fault elimination, case (c)

When the MLR algorithm is applied to this scenario, we get the sequence *m... mm...m...mm...*, again by comparing the respective *CSi* and *CSi'*.

By analyzing the *CS* and *CS'* match and mismatch sequences, we can identify the different stages of the MLR algorithm.

To eliminate the detected fault ε_2 the recovery is recursively done, starting from the last recovery point until the first ... *m mm* ... transition is found. The program execution is then resumed, as the fault ε_2 is now eliminated.

The detection of ε_1 can happen anytime, even after the detection of ε_2. In the given example the detection of ε_1 occurs after the elimination of ε_2 and triggers a new recovery process.

As ε_2 is already completely eliminated, the *CSi* and *CSi'* second match–mismatch sequence does no longer exist and therefore the *CSi* and *CSi'* comparisons match. The MLR process can thus successfully recover from ε_1.

In other words, several successively occurring faults do not affect the correctness of the MLR algorithm as long as the fault manifestation do not overlap in time, i.e., at least one recovery point exists between the fault manifestations.

If no recovery point was generated between the two fault manifestations, recovery is also successful, but the two faults are no longer distinguishable. From the system point of view, it just recovered from one fault.

9.2.4 Hardware Support for the MLR Algorithm

A possible implementation with hardware support of the MLR algorithm must fulfill the following requirements:

– Any hardware-based checking means must be maskable. This is needed for the generation of an RP and its CS in the program segment where the fault is detected.

- The software needs direct access to the hardware checksum register to read the current checksum value.
- The checksum and recovery point device must provide two operating modes: First, the standard mode, which creates the recovery points and its checksum and seconds a recovery mode, which generates only the checksum. The operating mode is configurable by the software.
- The checksum generator should have its own dedicated connection to the stable storage.
- For performance reasons, the stable storage should be directly attached to the computer so that the processor has direct access for maximum speed during RP generation and recovery.
- The RP and its associated CS should be generated concurrently to speed up the RP generation.

9.3 Summary

We presented a short conceptual description of the general recovery procedure required to achieve resilience, fault tolerance, and reliability.

We then introduced the modified linear recovery algorithm that as we showed is *able to decide whether the recovery was successful or whether the error is still present in the system by comparing the content of recovery points or their checksum.*

We also showed that the modified linear recovery algorithm could even be used to distinguish the fault type, i.e., permanent fault or malfunction. Thus, it covers in fact part of the testing and checking process as well.

Multiple faults do not interfere the modified linear recovery algorithm during recovery, and the MLR algorithm is able to detect them as separate faults.

The MLR algorithm has one strict requirement, which is *repeatability* of program on hardware.

References

1. Sogomonian E, Schagaev I (1988) Hardware and software fault tolerance of computer systems. Avtom I Telemekhanika, 3–39
2. Schagaev I (1989) Computing process recovery algorithms. Avtomat Telemekh (4)
3. Schagaev I (1990) Using software recovery methods for determining the type of hardware faults. Autom Remote Control 51(3)
4. Schagaev I (2008) Reliability of malfunction tolerance. In: International multi-conference on computer science and information technology, 2008. IMCSIT 2008, October 2008, pp 733–737
5. Schagaev I et al (2010) ERA: evolving reconfigurable architecture. In: 11th ACIS International Conference, June 2010, pp 215–220

6. Castano V, Schagaev I (2015) Resilient computer system design. Springer. ISBN 978-3-319150-68-0
7. Schagaev I (1986) Algorithms of computation recovery. Autom Remote Control 7:26, 36, 65, 122
8. Schagaev I (1987) Algorithms for restoring a computing process. Autom Remote Control 48 (4):26, 65, 122, 141, 149
9. Schagaev I (1989) Instructions retry in microprocessor recovery algorithms. In: IMEKO—FTSD symposium
10. Schagaev I (1990) Yet another approach to classification of redundancy. In: IBID
11. Schagaev I (1986) Relationship between the formation of program recovery points and equipment reliability indices. Autom Remote Control 47
12. Kowalk W (2006) CRC cyclic redundancy check. Technical report. Universität Oldenburg Fachbereich Informatik 05.09.06
13. Hamming R (1950) Error detection and error correction codes. Bell Syst Tech J XXVI:147–160
14. Moon T (2005) Error correction coding. Wiley, New Jersey
15. Schagaev I (1986, December) Using data redundancy for program rollback. Autom Remote Control 47(7), Part 2:1009–1016
16. Schagaev I., Viktorova V., Comparative analysis of the efficiency of computation-process recovery algorithms. Automation and Remote Control, 51(1), 1990

Chapter 10
Recovery Algorithms: An Analysis

Abstract Discovered algorithm of modified linear recovery seems to be effective in terms of power of detection of fault and correct state of the system. At the same time, classic algorithms well presented in literature binary search and linear search can also be applied for the same purpose. Thus, we have to consider the recovery process itself and analyze which classic algorithms are applicable and fit the purpose of efficient recovery. We introduce and analyze three recovery algorithms that are able to ensure successful recovery by iteratively go through all stored recovery points.

10.1 Comparison of MLR and Two Other Recovery Algorithms of the Same Family

Two other algorithms, namely, the linear recovery algorithm and the dichotomous recovery algorithm [1–4], are part of the same family of the recovery algorithms as the MLR algorithm.

We quickly introduce these algorithms and then compare their efficiency to the MLR (based on [3–5] and adapted for our needs). We do this by assuming a Poisson error rate and compare the three algorithms with increasing recovery depth.

For all three algorithms, we partition the program execution in segments and create a recovery point after each segment. The difference between the three algorithms lies in fact in the way the recovery is performed. We give more details during the analysis.

10.2 Computational Model

The execution model for the three algorithms is basically the same we used in the analysis of the MLR algorithm.

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_10

The computation process consists of the sequential execution of program segments where a recovery point is made after the execution of every segment. The actual successful segment execution time is considered as a random process.

Even though a program segment in case of no failure is executed in time t with T as the mean segment execution time, the randomness is required as occurring malfunctions or permanent faults and resulting recovery operations need additional time, depending on the executed recovery steps.

We assume that the checking intervals and the intervals between two recovery points are the same, as it is possible to combine hardware testing and recovery point creation into one step [2, 4]. Despite this change, the computation model is the same as the one used in the MLR approach explained above.

We now want to compare the three recovery algorithms (linear, dichotomous, and our modified linear) by their mean execution time to execute program segment *k*. We also want to take into account the possible latencies of the hardware malfunctions.

In the analysis of the MLR algorithm, we included malfunctions and permanent faults. In this chapter, we assume that only malfunctions can be recovered successfully, and these also only if the influence of any occurring malfunction has stopped before any recovery actions are initiated.

Permanent faults always lead to unsuccessful recovery. We also assume that no additional errors occur during recovery. The following events might happen during the execution of a program segment:

- H_c: No malfunction occurred during the execution of the segment and it has been successfully executed in the first run.
- H_m: A malfunction occurred during the execution of a program segment, the segment has been correctly identified, and is successfully finished after recovery step m. The number of total recovery steps is M.
- H_{pf}: A malfunction occurred during the execution of a program segment, the segment has been correctly identified but all K recovery steps did not result in successful recovery. Recovery is achieved after reconfiguration and program restart.
- $H_m(L)$: A malfunction occurred during the execution of a segment but the latent malfunction caused the failed segment to be incorrectly identified.
- $\mathrm{H_{pf}(L)}$: A malfunction occurred during the execution of a program segment, but the latent fault caused the failed element to be incorrectly identified and all K iterations did not result in successful recovery. Recovery is only achieved after reconfiguration and program restart.

The fault distribution during the execution of a segment is assumed to have a Poisson distribution with the parameter $\lambda = \lambda_1 + \lambda_2$, with λ_1 as the rate of malfunctions and λ_2 as the rate of permanent faults [6, 7].

10.3 Linear Recovery Algorithm

The linear recovery method is based on linear search and is described in detail in [2–7]. We give here just a short summary of the algorithm. If the checking logic detects a malfunction at the end of segment *J*, the computation is interrupted, RP ($j - 1$) is recovered and the processing resumed.

If during the repeated execution the checking logic does not detect the malfunction again, the recovery is assumed to be successful and the system continues by executing segment ($J + 1$).

If the malfunction is detected again, the interrupted process is resumed from RP ($J - 2$), etc., until a correct RP is reached which has been established without including information corrupted by the malfunction.

The difference to the MLR lies in the way recovery is performed. MLR uses the checksums to identify changes in the execution, whereas the linear approach always re-executes all code fragments until the end of execution when the checking logic signaled the fault.

The probability that one of the events H_c, H_m, $H_m(L)$, H_{pf}, or $H_{pf}(L)$ occurs is estimated with the aid of Sect. 3.1 and adopted from [7, 8].

The probability of having no malfunction in the interval ($0, t$) is

$$P_0(\lambda, t) = e^{-\lambda t}$$

The probability of having no malfunction at time *t* provided no malfunction was present at $t = 0$ is

$$P_{00}(\lambda, \mu, t) = \frac{\mu}{\lambda + \mu} + \frac{\lambda}{\lambda + \mu} e^{-(\lambda + \mu)t}$$

We use this model to define how malfunctions occur and stay in the system. The random times at which malfunctions occur and random times during which malfunctions stay in the system as latent faults are assumed to have an exponential distribution with the parameters λ and μ, respectively.

The probability of having no malfunction at time t but at least one malfunction before time t is

$$\overline{P}_{00}(\lambda, \mu, t_1, t) = \overline{P}_{00}(\lambda, \mu, t) - e^{-\lambda t_1} P_{00}(\lambda, \mu, t - t_1)$$

By using these equations, we can derive the following expressions for the probability of the above introduced events H_c, H_m, H_{pf}, $H_m(L)$, and $H_{pf}(L)$. We name these probabilities P_c, P_m, P_{pf}, $P_m(L)$, and $P_{pf}(L)$, respectively.

Since the above events cover all events of our model, we have

$$P_c + \sum_{m=1}^{M} P_m + P_{pf} + \sum_{m=1}^{M} P_m(L) + P_{pl}(L) = 1$$

We define now P_{km}, P_{kpf}, $P_{km}(L)$, and $P_{kpf}(L)$ as the conditional probabilities of the above events when a malfunction occurs in segment k.

We do not cover all scenarios with $k \leq 3$, since for $k = 1, 2, 3$, the recovery times are in a similar range for all three algorithms.

By analog to the above equation, we derive

$$P_c + \sum_{m=1}^{M} P_{k_m} + P_{k_{pf}} + \sum_{m=1}^{M} P_{k_m}(L) + P_{k_{pf}}(L) = 1$$

The probability of successful program execution in the first run (event H_c) is

$$P_c = e^{-\lambda T} \tag{10.1}$$

We individually calculate the probabilities for successful recovery execution for latent and non-latent malfunctions, first without latency, and failure detection at the end of the segment where the fault appeared ($k = J$).

The probability of successful recovery of the ongoing task in the case of non-latent malfunctions and recovery depth m (event H_{km}) is

$$\begin{aligned} P_{k_m} = \Big[& \alpha_k \overline{P}_{00}\big(\lambda_1, \mu, T, t_{beg_m}\big) P_0\big(\lambda_2, t_{beg_m}\big) \\ & - \sum_{i=1}^{m-1} P_{k_i} P_{00}\left(\lambda_1, \mu, \Delta t\binom{m}{i}\right) P_0\left(\lambda_2, \Delta t\binom{m}{i}\right)\Big] P_0(\lambda, t_{rb_m}) \end{aligned} \tag{10.2}$$

In case of recovery depth m, the task is repeated m times, assuming the last time starting from RP(k-m). Here α_k is the probability of detecting the malfunction at the end of segment k, $m = 1, 2, \ldots, k$.

The required time for recovery is found as follows: the time when recovery step m starts,

$$t_{beg_m} = T_{\sum m-1} + \delta_i - \delta$$

the time required by recovery step m,

$$t_{rb_m} = m(T + \delta)$$

the time difference between start and stop of step i

$$\Delta t\binom{m}{i} = t_{beg_m} - T_{\sum i} - \delta_i + \delta$$

where T is the program segment execution time, δ is the time when an RP was created, δ_I is the recovery start time, $\mathrm{T}_{\sum \mathrm{m}}$ is the total recovery time with depth m, derived as

$$T_{\sum m} = \left(m\frac{m+1}{2} + 1\right)(T+\delta)(T+\delta) + m\delta_i - m\delta \tag{10.3}$$

The first product in Eq. (10.2) defines the probability that no malfunction occurred at the beginning of recovery step m. From that, we subtract the probability of successful recovery during the previous steps. The coefficient of the expression in square brackets represents the probability that no malfunction occurs during the execution of recovery step m.

The probability of event H_{kpf} (successful or all k steps and successful recovery only after reboot) is given by

$$P_{k_{pf}} = \alpha_k\left(1 - e^{-i\lambda T}\right) - \sum_{m=1}^{nk} P_{k_m} \tag{10.4}$$

For the recovery of latent malfunctions, the latency period of the malfunction must be shorter than the recovery depth m. If the *RP* with depth m is recovered, the recovery is only successful if the latent malfunction has latency periods in the range of 1, 2, …, $m - 1$. The latency period l determines the additional segments that are executed before the malfunction is identified ($J = k + l$).

The conditional probability of $P_{km}(l)$ is defined by

$$\begin{aligned} P_{k_m}(l) &= P[H_{k_m}, latency\,period = l) \\ &= \Big[(1 - \alpha_k\overline{P}_{00}\left(\lambda_1, \mu, T, t_{beg_m}(l)\right)P_0\left(\lambda_2, t_{beg_m}(l)\right) \\ &\quad - \sum_{i=1}^{m-1} P_{k_i}(l)P_{00}\left(\lambda_1, \mu, \Delta t\begin{pmatrix} m \\ i \end{pmatrix}\right)P_0\left(\lambda_2, \Delta t\begin{pmatrix} m \\ i \end{pmatrix}\right]P_0(\lambda, t_{rb_m}) \end{aligned} \tag{10.5}$$

with m = 2, 3, ..., N, where N is the total number of segments.

In case of latent malfunctions, the recovery times ($t_{begm}(l)$) the time when recovery step m starts, $T_{\sum m}(l)$ the total recovery time with m steps) are defined by

$$t_{beg_m}(l) = T_{\sum m-1}(l) + \delta_i - \delta$$

$$T_{\sum m}(l) = T_{\sum m} l(T_i + \delta)$$

We assume now that the latency period of a malfunction has a geometric distribution, which is the usual assumption in reliability theory. In this case, the probability of successful recovery (with depth m) of a task when a latent malfunction with latency l occurs in segment k is given by Equation

$$\overline{P}_{k_m}(l) = \bar{\alpha}(1-\bar{\alpha})^{l-i}P_{k_m}(l) \tag{10.6}$$

Coefficient $\bar{\alpha}$ is the mean probability of detecting the malfunction during the segment execution.

$$\bar{\alpha} = \frac{1}{N}\sum_{i=1}^{N}\alpha_i$$

The probability of event H_{km} *(L)*, i.e., the successful recovery of a program affected by a latent malfunction in segment k after m recovery steps, is

$$P_{k_m}(l) = \sum_{i=1}^{m-1}\alpha(1-\alpha)^{l-i}P_{k_m}(l) \tag{10.7}$$

In case of latent malfunction, the mean total recovery time for depth m is

$$T_{\sum m}(l) = \sum_{i=1}^{m-1}\alpha(1-\alpha)^{l-i}T_{\sum m}(l)$$

The probability of system restart and task reload (event H_{kpf} *(L)*) is

$$P_{k_{pf}}(L) = (1-\alpha_k)(1-e^{\lambda T})\sum_{m=2}^{k+\bar{l}}P_{k_m}(L) \tag{10.8}$$

where the mean length of the latency period is

$$\bar{l} = \sum_{l=1}^{N-K-1}\bar{\alpha}(1-\bar{\alpha})^{l-i}l$$

If we combine all these equations into one, we can derive the mean time $\overline{T}$ of the successful execution of segment *k:*

$$\overline{T} = P_cT + \sum_{m=1}^{k}P_{km}T_{\sum m} + \sum_{m=2}^{k+l}P_{km}(L)T_{\sum m}(L) \tag{10.9}$$

We will use $\overline{T}$ throughout the analysis and comparison of the three recovery algorithms [2–7].

10.4 Dichotomous Recovery Algorithm

The dichotomous recovery algorithm is the second algorithm we consider in this analysis. In fact, the only difference to the linear algorithm is the way the recovery steps are selected. The recovery steps are no longer linearly executed, but the first recovery step is the one with number *i,* and

$$i = \left\lceil \frac{J}{2} \right\rceil$$

If the system could not be recovered (the checking logic still detects the fault after the execution of segment *J*), it continues with RP_i and

$$i = \left\lceil \frac{J}{2} + \frac{J}{4} \right\rceil$$

etc., until a correct RP is found.

This algorithm applies therefore the pattern of a binary search to find the appropriate recovery step.

We derive now the probabilities of the events H_c, H_{km}, H_{kpf}, $H_{km}(L)$, and $H_{kpf}(L)$ for the dichotomous recovery. We derive P_c (the probability of event H_c) as in the case of linear recovery from Eq. 10.2. P_{km} (the probability of event H_{km}) is calculated by using Eq. 10.3 but by deriving new times for the execution of recovery step m and the total recovery time for depth *m*:

$$t_{rb_m} = \left\lceil k\left(1 - \frac{1}{2^m}\right)\right\rceil (T + \delta) \qquad (10.10)$$

$$T_{\sum m} = \sum_{i=1}^{m} \left\lceil K\left(1 - \frac{1}{2^i}\right)\right\rceil (T + \delta) + m\delta_1 - m\delta \qquad (10.11)$$

where $m = 1, 2, \ldots, \lceil log_2 k \rceil$.

The latency period coverage must be adapted for the dichotomous case. A latent malfunction with a latency period *l* can only be recovered by a rerun procedure if the following equation holds for *m*:

$$\mathrm{J}\left(1 - \frac{1}{2^m}\right) \qquad (10.12)$$

By rearranging the equation above, we get minimum depth *m* which is required for a malfunction with latency l

$$m = \left\lceil log_2\left(\frac{J}{J-1}\right)\right\rceil + 1 \tag{10.13}$$

where m = 1, 2, 3, …, $\lceil Log_2 N \rceil$.

The probability P_{km} *(l)* can be derived from Eqs. 9.5, 9.10 and 9.11.

The probability of recovery in step m and the mean total recovery time for depth m in the case of a latent malfunction are

$$P_{k_m}(L) = \sum_{i=1}^{l_{max}} \bar{\alpha}(1-\bar{\alpha})^{l-1} P_{k_m}(l) \tag{10.14}$$

$$T_{\sum m}(L) = \sum_{i=1}^{l_{max}} \bar{\alpha}(1-\bar{\alpha})^{l-1} T_{\sum m}(l) \tag{10.15}$$

where $l_{max} = \left\lceil J\left(1 - \frac{1}{2^m}\right) - 1\right\rceil$.

The probability of unsuccessful recovery with resulting restart for the two fault types and the mean time of successful termination of the program segment k are found analog to Eqs. 10.4, 10.8 and 10.9.

If a malfunction occurs in segment k, the maximum possible recovery depth is $\lceil log_2 k \rceil$ without latency and $\lceil log_2(k + \bar{l}) \rceil$, otherwise [7, 8].

10.5 Modified Linear Recovery

We already described the modified linear recovery algorithm in the previous chapter. The difference of the MLR to the other two described algorithms lies in the condition used to indicate successful recovery.

Unlike the other two cases, in which recovery is considered to be successful if the checking logic does after recovery no longer detect the fault, success in case of MLR is determined by comparing the original and the new RP and the respective checksum.

The probability of successful recovery with recovery depth m and non-latent malfunction (event H_{km}) is given by

$$P_{k_m} = \alpha_k \overline{P}_{00}\left(\lambda_1, \mu, T, t_{beg_m}\right) P_0(\lambda, t_{rb_m}) P_0(\lambda, t_{rb_m}) - \sum_{i=1}^{m-1} P_{k_i} G \tag{10.16}$$

with

$$G = \begin{cases} P_{00}\left(\lambda_1, \mu, \Delta t\begin{pmatrix} m \\ i \end{pmatrix}\right) P_0\left(\lambda_2, \Delta t\begin{pmatrix} m \\ i \end{pmatrix}\right) P_0(\lambda, t_{rb_m}) & \textit{if } T_{\sum i} < t_{beg_m} \\ P_0(\lambda, \Delta t'\begin{pmatrix} m \\ i \end{pmatrix}) & \textit{if } T_{\sum i} \geq t_{beg_m} \end{cases}$$

The total time needed to perform a recovery with depth m is

$$T_{\sum m} = T + \delta_1 + (m+1)(T + \delta') + m(T + \delta) \tag{10.17}$$

where δ' is the time needed to create a new RP and to compare the content of the new and former RP ($\delta' \ll \delta$ if checksums are used).

The recovery times (Eq. 10.3) are determined as follows:

$$T_{beg_m} = T + \delta_i + m(T + \delta') \tag{10.18}$$

$$T_{rb_m} = T + \delta' + m(T + \delta)$$

Denote $\Delta t'\begin{pmatrix} m \\ n \end{pmatrix}$ the time difference between the termination of recovery step m and i:

$$\Delta t'\begin{pmatrix} m \\ n \end{pmatrix} = T_{\sum m} - T_{\sum i}$$

The probability of successful recovery with depth m in the case of a latent malfunction with latency period l is

$$P_{k_m}(l) = (1 - \alpha_k)\overline{P}_{00}\left(\lambda_1, \mu, T, t_{beg_m}(l)\right) P_0\left(\lambda_2, t_{beg_m}(l)\right) P_0(\lambda, t_{rb_m}) - \sum_{i=l+2}^{m-1} P_{k_i}(l) G \tag{10.19}$$

with $m = 2, \ldots, N$ and

$$t_{begm(l)} = t_{begm} + l(T + \delta) \tag{10.20}$$

The probability of successful recovery with depth m is determined in case of a latent malfunction from Eq. 10.7.

The probability of a system restart in case of non-latent and latent malfunctions and the mean time of successful termination are found as in the case of linear recovery from Eqs. 10.4, 10.8, 10.9, respectively.

10.6 Conclusion on Comparison of Recovery Algorithms

We compared the *linear*, *dichotomous*, and *modified linear recovery* algorithms using the mean time T that is required to successfully execute the segment in which the malfunction occurred.

We have analyzed the mean time as a function of the time until the malfunction occurred, beginning at program start and the part of non-latent malfunctions occurring in a segment and the distance, in segments, from the program start to the malfunction position.

The success and time of successful recovery depend on the available checking facilities but mainly on its fault coverage, which indirectly determines the parameter α_k. The higher the fault coverage, the greater the α_k, and the smaller the number of steps (segments) needed to recover the affected task.

Figure 10.1 shows how the time of successful execution of segment *k* is affected by the fault coverage. The MLR algorithm has least mean time *T*, independent of the fault coverage.

If we compare the linear and the dichotomous algorithm, we see that in case of low fault coverage, the dichotomous algorithm is better, whereas the linear algorithm is better in case of high fault coverage. This is easily justifiable by the fact that in case of low fault coverage the required recovery depth is higher and the dichotomous algorithm basically skips the first unsuccessful recovery steps of the linear algorithm. The opposite is true for high fault coverage.

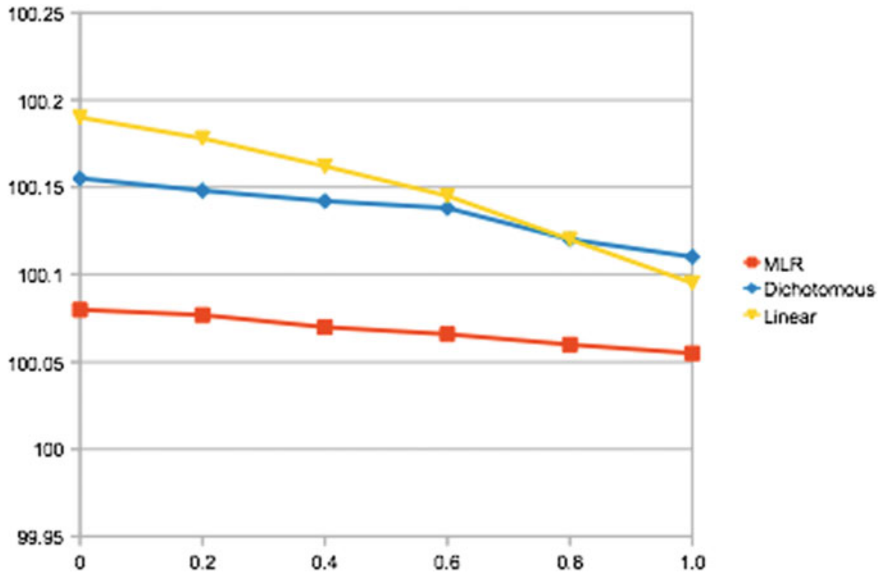


Fig. 10.1 Recovery time as a function of α_k

The recovery depth is small, and thus, the linear algorithm is better. For Fig. 10.1, we used the following values:

malfunction rate [1/sec] $\lambda_1 = 0.0000027$, $\lambda_2 = 0.0000003$, $\lambda = 0.000003$; permanent fault/malfunction ratio $S = 0.1$; segment execution time [sec] $T = 3$; *Time parameters* [sec] $\delta = 0.5$; $\delta_1 = 1.0$, $\delta' = 0.5$; *number of program segments from the start [segments]* $k = 10$.

We analyzed the recovery times of all three algorithms as a function *k,* as shown in Fig. 10.2. It is clear from this graph that the MLR algorithm has the shortest recovery time, independent of *k*.

Figure 10.2 was drawn using the following values: malfunction rate [1/sec] $\lambda = 0.000036$, $\lambda_1 = 0.000004$, $\lambda_2 = 0.00004$; *fault/malfunction ration* $S = 0.1$; *segment execution time* [sec] $T = 3$; time parameters [sec] $\delta = 0.5$, $\delta_1 = 0.2$, $\delta' = 0.5$; *part of non-latent faults* $\alpha_k = 0.8$.

This short analysis clearly shows that the modified linear recovery algorithm is the most efficient of the three compared algorithms, as it requires the least mean time to successfully recover a process.

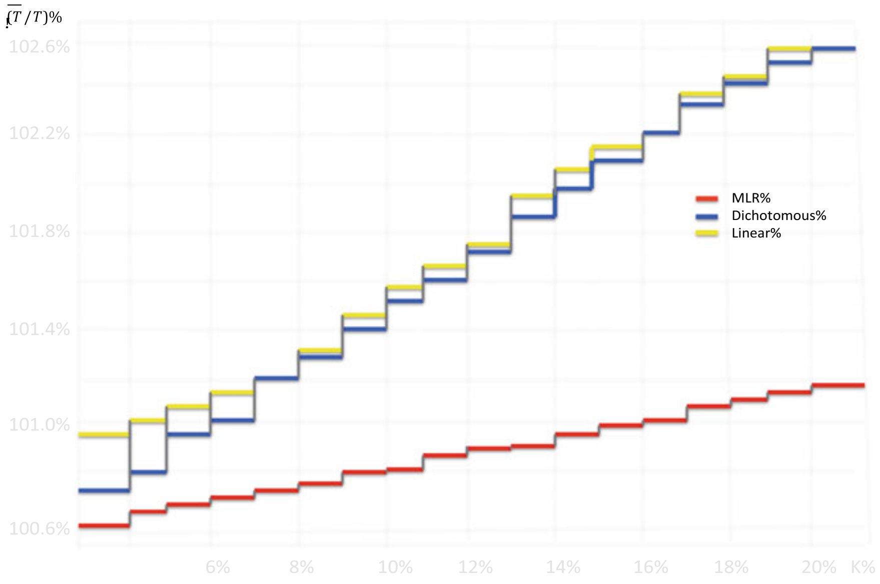


Fig. 10.2 T as a function of *k*

10.7 Summary

- For recovery, we presented the *modified linear recovery* algorithm, an algorithm that allows the determination of the fault type and is able to find the last correct recovery point, even in the case of latent faults. If two subsequent faults occur, the modified linear algorithm does still work.
- We compared the linear modified algorithm with two other algorithms (linear and dichotomous) of the same family. All mentioned algorithms differ in the way the recursive recovery steps are performed.
- It was shown that the modified linear recovery algorithm has fastest recovery time.
- Above all, it was shown that the modified linear recovery efficiency is substantially higher than two classic algorithms mentioned; modified linear recovery algorithm does not use any model or knowledge about the type of hardware fault, making it unique in terms of applicability.

References

1. Sogomonian E, Schagaev I (1988) Hardware and software fault tolerance of computer systems. Avtomatika i Telemekhanika 3–39
2. Schagaev I. Computing process recovery algorithms. Avtomat. Telemekh., (No. 4), 1989
3. Schagaev I (1990) Using software recovery methods for determining the type of hardware faults. Automat Remote Control 51(3)
4. Schagaev I. Algorithms of computation recovery. Automation and Remote Control, 7, 1986. 26, 36, 65, 122
5. Schagaev I (1987) Algorithms for restoring a computing process. Automat Remote Control 48 (4), 26, 65, 122, 141, 149
6. Schagaev I (1986) Using data redundancy for program rollback. Automat Remote Control 47 (7) Part 2, 1009–1016
7. Schagaev I, Viktorova V (1990) Comparative analysis of the efficiency of computation-process recovery algorithms. Automat Remote Control 51(1)
8. Koren I, Su SYH (1986) Analysis of a class of recovery procedures IEEE Trans Comput C-35 (8) 703–712

Chapter 11
Programming Languages for Safety-Critical Systems

Eugene Zouev

Abstract In previous chapters, we introduced the three main processes required to implement generalized algorithm of fault tolerance (GAFT), namely—testing and checking, second recovery preparation, and third and finally recovery and recovery monitoring. We described what every of these steps incorporates, gave possible solutions, and analyzed them. In the Chap. 7, we introduced syndrome for testing and checking; here we introduce programming language models for the two other mentioned processes. What we now want to do is to synthesize the introduced concepts into system software tools—programming languages and their compilers. We will discuss possible project solutions related to the overall architecture of software tools and introduce the major components of the architecture.

11.1 Oberon-07 as the Single Development Tool

Initially, the Oberon-07 programming language [1] was considered as the basis for the programming language extensions, and embedded reliable reduced instruction computer (ERRIC) [2, 3] serves as the target hardware platform. The strong type safety of Oberon together with the simplicity of the language seemed to be quite suitable for safety-critical systems.

Oberon-07 was developed during the ONBASS project [4] and successfully used for the implementation of the MINOS [5] operating system. Its goal of simplicity of the language together with strong type safety and built-in safety features such as automatic array range checks are in line with the requirements of a safety-critical system.

However, during the implementation in a slightly more complex, more flexible setup than ONBASS, it became more and more obvious that the programming model of Oberon-07 does not provide all required functionalities.

First of all, MINOS is heavily based on a plugin concept, which means that all required functionalities such as drivers or application components can be plugged into the system as required. The only possible way to implement plug-ins in Oberon-07 is by using records containing procedure variables that pass the object "SELF" as the first parameter.

One can consider this as a simple way of object orientation. Unfortunately, this proved to be clumsy and error prone as an implementation of such a plugin does not have to necessarily implement all procedures or can even assign NIL to procedure variables, which results in NIL pointer exceptions during runtime.

This "manual" way of assembling objects also prohibits a proper analysis of the program by the compiler and the runtime system, as the concrete instance of such an object is assembled at runtime by the software.

A second downside of MINOS is the lack of efficient multitasking. Although the implemented tasking scheme based on periodic tasks allows the execution of multiple tasks quasi-simultaneously, the processor usage is suboptimal, as there is no way to wait for asynchronous events, such as interrupts without using time-consuming polling. This is mainly imposed by the simple scheme of using just one stack and the necessity of a task to run to completion.

Support for reconfiguration of hardware and efficient recovery points as described in the previous chapters are not given. Reconfigurability is only supported on the application level and recovery point is supported only on the system level. Recovery points on the level of procedures would also be possible, as it was already shown earlier it is a very efficient approach [6–9].

11.2 System Software for ERRIC: The Stack of Development Tools

The obvious conclusion from the considerations presented above could be reflecting within the language features that would meet requirements typical for developing safety-critical systems like reliability, reconfigurability, multitasking, etc. However, the set of requirements that a single language should satisfy seems to be too big and even contradictory.

Also, the variety of tasks and problems that the ERRIC software should support is quite wide: from the very basic low-level software like memory management or concurrency support up to high-level application systems. Therefore, we need separate tools for implementing different kinds of software.

Thus, instead of having a "single monster language for everything" with a big number of different and contradictory features, we came to the solution of a set of different languages each of which is suitable for a particular kind of ERRIC software.

The picture below illustrates the conceptual solution taken for the ERRIC development tools (all language names are still codenames) (Fig. 11.1).

Each language from the stack implements a limited set of features that are critically important just to the corresponding domain—therefore, all of them were

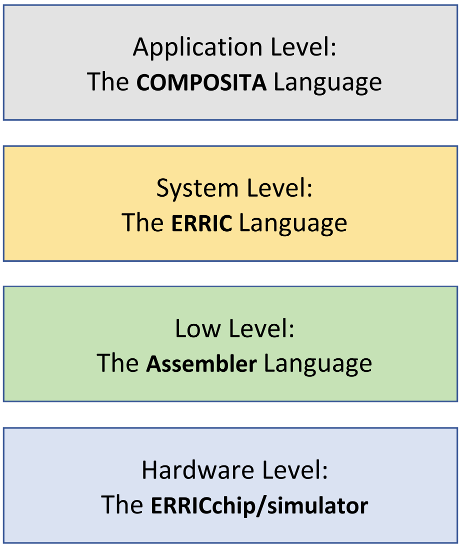


Fig. 11.1 The stack of development tools for ERRIC

made quite compact, concise, and easy to use. Together, they cover all needs for implementing a wide spectrum of software for ERRIC-based machines.

The following sections provide a quick description of each language from the picture above.

11.3 Low Level: The Assembly Language

Conventionally, the purpose of the assembly language is to provide full access to all program-controlled hardware parts of a processor. The ERRIC assembly language supports direct access to registers and allows constructing sequences of instructions from the instruction set.

The language introduces simple mnemonics for instructions that typically has the form of assignment and looks familiar to developers. The set of assembly instructions directly correspond to ERRIC machine instructions. The overall syntax of assembly programs is quite simple.

In general, an assembly program can represent any kind of actions that are possible for ERRIC processors. At the same time, the assembly language does not assume any checks for correctness of the instruction sequence. The developer is fully responsible for code correctness and reliability.

The primary aim of the assembly language is to support developing system-level software components that are critical by some important characteristics (efficiency, memory consumption, etc.), for example, runtime support, device drivers, etc.

The key part of the assembly language syntax is presented below in the form of BNF productions:

Listing 11.1 Assembly language syntax

```
AssemblyProgram
 : { { Label } (Code | Data) }

Code
 : { AssemblyStatement }

Label
 : < Identifier >

Data
 : data Literal { , Literal } end

AssemblyStatement
// statement                        Example        Mnemnonics
 : skip                       //                   NOP
 | stop                       //                   STOP
 | trace                      //                   TRACE
 | format ( 8 | 16 | 32 )     //  format of next instructions
 | Register := *Register      //  Rj := *Ri        LD
 | Register := Literal        //  Rj := Const      LDA
 | *Register := Register      // *Rj := Ri         ST
 | Register := Register       //  Rj := Ri         MOV
 | Register += Register       //  Rj += Ri         ADD
 | Register -= Register       //  Rj -= Ri         SUB
 | Register >>= Register      //  Rj >>= Ri        ASR
 | Register <<= Register      //  Rj <<= Ri        ASL
 | Register |= Register       //  Rj |= Ri         OR
 | Register &= Register       //  Rj &= Ri         AND
 | Register ^= Register       //  Rj ^= Ri         XOR
 | Register <= Register       //  Rj <= Ri         LSL
 | Register >= Register       //  Rj >= Ri         LSR
 | Register ?= Register       //  Rj ?= Ri         CND
 | if Register goto Register // if Ri goto Rj     CBR

Register : R0 | R1 | … | R30 | R31
```

As an example of the assembly language code, let's consider the following simple sorting routine that accepts an array of integers together with its size, and sorts the array elements in ascending order. The routine (written in the ERRIC system language, see Sect. 11.4 below) is given just as an illustration: the next listing demonstrates the equivalent program written in the assembly language.

Listing 11.2 A sorting routine

```
routine Sort(a: [int], size: int) do
    for i in 1..size loop
       for j in 0..i reverse loop
          if a[j-1] < a[j] then
             int w := a[j-1]
             a[j-1] := a[j]
             a[j] := w
          end
       end
    end
 end Sort
```

Listing 11.3 The sorting program written in the assembly language

```
    // i: R1;  j: R2;  intermediate values: R3, R4, R5
    R15 := Size;  // 'Size' labels the data block below
    R15 := *R15;
    R16 := Array; // 'Array' labels the data block below
    trace R15,R16;
    R1 := 1;        // i := 1

<LoopOuter>
    R3 := Size;
    R3 := *R3;      // R3: Size
    R4 := 1;
    R3 -= R4;
    R3 ?= R1;       // Compare Size-1 and i
    R4 &= R3;       // Extract > sign using R4=1 as mask for >
    R3 := OutOuter;
    if R4 goto R3;  // if i>Size goto OutOuter

    // Organize inner loop
    R2 := R1; NOP;  // j := i

<LoopInner>
    R3 := 0;        // w := 0
    R3 ?= R2;       // Compare j with 0
    R4 := 4;        // Mask for equality
    R3 &= R4;       // Extract equality sign
    R4 := OutInner;
    if R3 goto R4;  // if j=0 exit the inner loop
    // Otherwise, compare two array elements
    // to decide if we need to exchange them.
    // R10: address of j-th element
    // R11: a[j]
    // R12: a[j-1]
    R10 := Array;
    R10 += R2;      // array base address+j
    R11 := *R10;    // R11 := a[j]
    R12 := R10;
    R13 := 1;
    R12 -= R13;     // a+j-1
    R12 := *R12;    // R12 := a[j-1]

    R3  := R12;     // w := a[j-1]
    R3  ?= R11;     // Compare a[j-1] and a[j]
    R4  := 5;       // Mask for >=
    R4  &= R3;      // Extract > and = signs
    R3  := OutExchange;
    if R4 goto R3;  // if a[j-1] >= a[j] do not perform exchange

    // Otherwise, perform exchange
    R3 := R10;
    R4 := 1;
    R3 -= R4;       // R5: address of (j-1)th element
   *R3 := R11;      // a[j-1] := a[j]
   *R10:= R12;      // a[j] := a[j-1]

<OutExchange>
    // Decreasing j (inner loop)
    R3 := 1;
    R2 -= R3;       // j := j-1
    R4 := LoopInner;
    if R3 goto R4;  // goto LoopInner

<OutInner>
    // Increasing i (outer loop)
    R3 := 1;
    R1 += R3;       // i := i+1
    R4 := LoopOuter;
    if R3 goto R4;  // goto LoopOuter
    NOP;
```

```
<OutOuter>
    R15 := Size;
    R15 := *R15;
    R16 := Array;
    trace R15,R16;
    stop; nop;
<Size>
    data 20 end
<Array>
    data
       537, 242, 114, 436, 337, 296, 285, 655, 639, 436,
       912, 520, 624, 551, 600, 741, 612, 943, 871, 735
    end
```

11.4 System Level: The ERRIC Language

The ERRIC language (the name is not fixed yet) is designed and implemented as a kind of intermediate programming level between low-level hardware-oriented assembly language and the higher level machine-neutral application language.

The language includes some low-level ERRIC specifics (it even has the full assembly language as its subset).

At the same time, the language supports a set of usual higher level constructs like the notion of type (with compile-time type checks), procedural mechanism, control flow statements (conditionals and loops), etc.

This allows developers to write fast, efficient, highly optimized programs using familiar higher level programming paradigms. In this sense, the ERRIC language occupies the niche similar to Oberon and C languages.

Major features of the ERRIC language are as follows:

- Simple, clear, and unambiguous syntax that looks familiar to developers.
- Simple semantics; possibility to develop extremely efficient software.
- Flexible program structure with a variety of building blocks: modules, routines, data, and standalone code snippets.
- Simple, compact but powerful type system.

The language type system includes hardware-supported types (byte, short, and integer), enumeration types (as an extension of byte/short/integer), arrays, and structures (records).

Also, the special routine type is included as a safe way to add flexibility to the procedural mechanism.

The language supports strong typing paradigm, where type consistency of program entities is checked on compile-type. This enables much higher program reliability comparatively with dynamically typed languages.

Strong typing in the ERRIC language is a bit harder than in many other languages: it prohibits any kinds of implicit-type conversions; only a limited set of explicitly specified-type conversions are allowed. Such a restriction also improves program reliability.

- No error-prone "pointer" types still with flexible and safe access to data and code (routines). Null-safety design: uninitialized variables cannot appear in ERRIC programs.
  Both features actually serve the same purpose as strong typing: they make programs much more reliable preventing unexpected program behavior on runtime.
- Support for direct access to some hardware components.
  In general, the entire assembly language as a subset of the ERRIC language. That is, the construct "assembly statement" is the legal kind of the ERRIC's "statement" notion. Also, there are means for specifying special lower level conditions for some program components. In particular, it is possible to specify registers as return values.

The features mentioned above can be illustrated by the following schematic example:

```
routine Example(a, b, c: integer): R9, R10 do
    // Routine local declarations and statements
    // performing some calculations
    ...
    asm
        R9 := someResult1
        R10 := someResult2
    end
end
```

The routine from this code snippet states that the call to it will (or can) update the registers, whose names were mentioned after the colon delimiter. The body of the routine contains two assembly statements explicitly accessing registers (assigning some values to them).

- The language introduces a limited number of statements: assignments, routine calls, control statements (conditionals and loops), and simple expression syntax supporting all kinds of machine instructions.
- Basic support for multitasking and interprocessor communication with the synchronization mechanism.

Some features either are being implemented as optional ones or are still under discussion:

- Lightweight version of exceptions and exception handling.
- Dynamic memory allocation and automatic memory management for dynamically allocated program entities.
- Object orientation with single inheritance.

The point of concern related to the features mentioned above is that the full implementation of them can slowdown programs significantly.

Support of the notion of exception allows flexible reactions on various kinds of unexpected program behavior, which is quite useful in case of systems that communicate with an outer world actively.

However, typical semantics of exception handling assumes significant overhead that might be not acceptable. Similar reasons apply to automatic memory management (garbage collecting).

Proper object orientation helps developers to clearly structure their programs and implement them in a modern way.

However, from a safety-critical point of view, this feature is not essential and can also be omitted if the additional overhead of an object-oriented language is not acceptable.

The ERRIC compiler translates the source code into semantically equivalent assembly code (see Sects. 11.3 and 11.4), that is, to be further processed by the assembler getting the executable binary code.

The resulting binary code can be loaded to memory and executed directly on the ERRIC processor. Another option is to pass the binary to the ERRIC Simulator (see Chap. 20) to execute it in a model environment.

The ERRIC compiler now (Spring 2019) is being developed by S. Malyutkin and E. Zouev (Innopolis University, Russia).

11.5 Application Level: The COMPOSITA Language

The specification of the original COMPOSITA language together with rationale and discussion about its key concepts and features is given in [10]. The language supports a rich and flexible program structure that allows building complex application software. Following [11] and [4], the language model can be characterized as follows:

- Hierarchical composition;
  Each component can be hierarchically composed, by containing an arbitrary assembly of component instances. The contained sub-components are fully encapsulated by the surrounding super-component.

- Interface connections;
  An arbitrary network of components can be built by connecting the required interfaces of components to corresponding offered interfaces of other components. A component only constructs the network of its sub-components.
- Communication-based interactions;
  Components can interact via interfaces by message communications. An individual communication channel is maintained between a component, which offers an interface, and each component, which uses the interface. The language establishes an agent-based model with self-active objects (components), and protocol-structured communications.

The language semantics supports powerful means and constructs for organizing concurrent execution.

The version of COMPOSITA for ERRIC has some extensions and modifications that serve for better use of the processor's architecture specifics.

11.6 Summary

In this Chapter, we discussed the system tools for creating software for ERRIC-based computers. The major architectural idea is to design a composition of three programming languages supporting the development of various kinds of software—from system low-level code up to complex applications.

Together, these languages (and corresponding system tools like compilers) provide support for the three main GAFT processes (*P1* testing and checking, *P2*, recovery preparation, *P3* recovery). We resume here what features we introduced and to which step of GAFT they belong.

Each language is introduced by discussing its major concepts and features; a few examples illustrating some key aspects of their design are provided.

An obvious next step of the entire project evolution would be a prototype implementation of the compilers. Hopefully, they can be checked as a proof-of-concept even before a real ERRIC processor appears: the ERRIC instruction code software simulator can run ERRIC programs in the interpretation mode. Refer to Chap. 20 for more detailed information about the simulator.

References

1. Wirth N (2008) Oberon-07 language report. Technical report, ETH Zurich
2. Schagaev I (2008) Reliability of malfunction tolerance. In: International multi-conference on computer science and information technology, IMCSIT 2008, pp 733–737
3. Castano V, Schagaev I (2015) Resilient computer system design. Springer, New York. ISBN 978-3-319-15069-7
4. https://cordis.europa.eu/project/rcn/72840/reporting/en

5. Kaegi-Trachsel T, Gutknecht J (2008) Minos—the design and implementation of an embedded real-time operating system with a perspective of fault tolerance. In: International multi-conference on IMCSIT 2008, pp 649–656, 20–22
6. Schagaev I (1986) Algorithms of computation recovery. Autom Remote Control 7:26–36
7. Schagaev I (1987) Algorithms for restoring a computing process. Autom Remote Control 48:530–538
8. Schagaev I (1989) Instructions retry in microprocessor recovery algorithms. In: IMEKO FTSD symposium
9. Schagaev I (1989) Yet another approach to classification of redundancy. In: Proceedings of FTSD Prague, Czeschoslovakia, pp 485–490
10. Bläser L (2007) A component language for pointer-free concurrent programming and its application to simulation. Dissertattion, ETH No. 17480
11. Bläser L (2006) A component language for structured parallel programming. In: Joint modular languages conference (JMLC) 2006, Oxford, UK. Lecture Notes in Computer Science 4228, September 2006. Springer
12. Schagaev I, Sogomonyan E (1988) Hardware and software for fault-tolerant computing systems. Automation remote control 49:129–151
13. Blaeser L, Minkman S, Schagaev I Evolving systems 12. http://worldcomp-proceedings.com/proc/proc2014/fcs/FCS_Papers.pdf
14. Reali P (2004) Active Oberon language report. Technical report, ETH Zurich
15. Mossenbock Wirth N (1991) The programming language Oberon-2. Technical report, Johannes Kepler Universitat Linz
16. Wirth N, Gutknecht J (1992) Project Oberon: the design of an operating system and compiler. Wesley, New York
17. Johannes M (2002) The active object system—design and multiprocessor implementation. ETH Zurich, Zurich

Chapter 12
Proposed Runtime System Structure with Support of Resilient Concurrency

Abstract This chapter is some kind of summary of Chaps. 6, 7 in terms of implementation of principles described there at the level of the runtime system. Here, we summarize new functions and features of runtime system that simply must be implemented during the design of fault-tolerant and resilient systems. Special attention was given to concurrency monitoring in the presence of potential threats and degradation. While it is clear which part of system software has to deal with this, the whole problem of concurrency never was addressed proper—here we present our attempt.

12.1 Runtime System Structure

The runtime itself is built in a strictly hierarchical manner and runs completely in its own resource. Some of them, such as memory management are well known and need no further explanation. Figure 12.1 illustrates the structure of the runtime system, while each box is briefly explained when necessary.

FT Interrupt Handler: The interrupt handler serves as the lowest level component that interacts and handles all hardware interrupt requests including the syndrome interrupt and forwards the request to the appropriate handler. The interrupt handler is fault tolerant in that it is part of the syndrome management.

Main Monitor: The lowest level module has no dependencies at all and consists of the main system monitor, which is responsible for the coordination of all other modules, such as the initialization of resources, timer services (not shown), interrupt handling, and all the remaining depicted functions.

Resource Management: This module is responsible for resource management, i.e., memory allocation, device driver management, and their associated resources, etc.

Reconfiguration Monitor: This module keeps track of the current software configuration, i.e., the current resources configuration and can in case of faults initiate and perform the software configuration. For each of the available resources, it also keeps track of its state, according to Sect. 7.4.3.

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_12

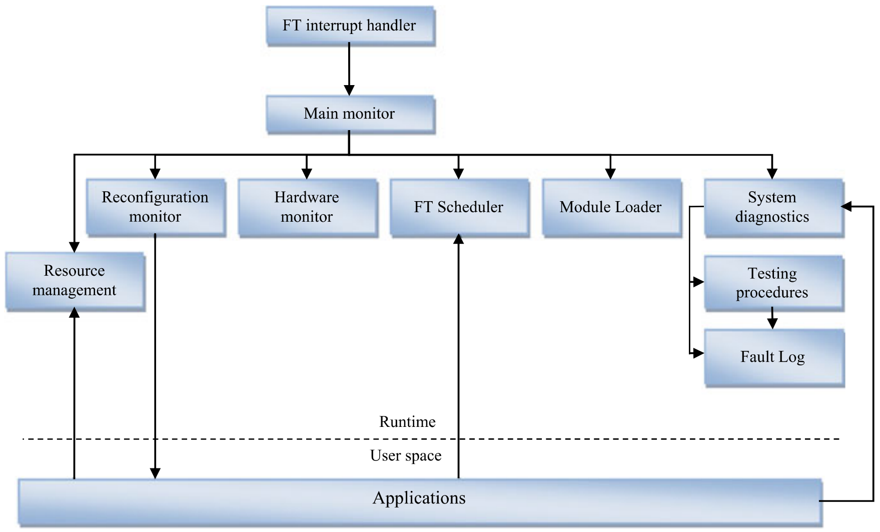


Fig. 12.1 Proposed structure of runtime system

Hardware Monitor: The hardware monitor tracks the current state of each hardware component (see Sect. 7.4.2) and is also responsible for managing the syndrome.

In case of a permanent fault, this module reconfigures the hardware to a new fault-free state and informs the reconfiguration monitor about the new hardware topology.

FT Scheduler: The FT scheduler is responsible for scheduling tasks and tests as described in Sect. 7.2. It manages all activities and is responsible for scheduling hardware tests according to the SSD policy.

Module loader: The module loader is optional and is responsible for allocating and linking the modules at runtime. For a production system, depending on the linking strategy, the module loader might not be necessary.

System Diagnostics: This module manages all hardware and software checks that are currently present in the system and interacts with the FT scheduler. In case of a fault, this module is responsible for selecting the appropriate tests for diagnosing the fault (software failure, malfunction, permanent fault, etc.) and notifies the software and hardware monitoring services of any required changes.

Testing Procedure: The runtime system itself also provides a built-in testing mechanism to ensure the integrity of the runtime itself and manages also the tests of the other

Fault Log: All faults are logged in the fault log for traceability and to support the hardware monitor. High fault frequency in a specific hardware component might indicate a faulty component or a near future failure of a specific component.

Applications: All application run in their own respective activity and communicate with the runtime system module over messages.

12.2 Concurrency Support for Resilient Computing

Two monitors as above **FT Interrupt Handler** and **Main Monitor** deal with interruptions and resource handling, including reconfiguration of architecture and software structure when required.

12.2.1 Problem of Concurrency

Real world is represented in our imagination and later developments either algorithmically or heuristically. The algorithmic representation used to use one or several logics of Aristotle, Boolean, sequential, concurrent, k-logic.

Technological support of the one that was chosen manifests itself via hardware implementations.

From the other side, flexibility implementation by the software of world problems is required. These requirements define the design of system software: languages and operating systems.

Elements of hardware might be specified in terms of required performance and reliability. At the same time, every hardware technology has its own limits: either maximum possible frequency is approaching or, actually, implementation problems for brand new approaches such as quantum—computing trick with Shreddinger' paradox. Here, we have to be very careful what is achievable and what is for funding and hardly possible: nature does not accept cheating;

Thus one can see a semantic gap between physical possibility and limitations of hardware and required flexibility of software to make problems represented by algorithms doable.

Clear, someone has to pay for this gap, and pay dearly: physical limits of calculations are not even discussed seriously, our personal observations show that modern home computers easily take 0.5 KW to do the simplest work such as typing!

Current situation looks even more surprising and confusing: 1–5 GHz hardware nowadays is required to cope with simple problems and demands from concurrency approach is flourishing by reinventing the wheel: concurrent programming and concurrent architectures are "blue chips" now in research domain for scientists and engineers: more than 10 Mln entries one might find using simple Google search.

We lost the sense of reality and historical perspective: Cray has never achieved any industrially sensitive results, but it was one of the most expensive and one of the best among supercomputer systems for several decades… If best engineers failed then, why we naively believe that the new generation of scientists and engineers can reincarnate an idea and concept?

One of the reasons for lack of progress in multiprocessor systems was the inefficiency of parallelism of the programs—see Chap. 17 of this book and ERA paper [1, 2].

Another reason was the inefficiency of synchronization of processes within the system. Hopefully, this chapter will not cause extra skepticism from the past but some observation of the known drawbacks in concurrency and attract researchers to develop along proposed concept for possible improvement in the future resilient systems.

12.2.2 Why It Is Needed

Concurrency of the computer system is required due to several hardware subsystems operation at the same time and at some points compete for resources and sooner or later they compete for any of resources.

Resources are defined in [1, 2] and, namely, for computer systems they are *structure*, *information,* and *time*.

Some examples of concurrency presence are

- multiprocessor system (processes within various parts of hardware);
- databases, especially distributed ones;
- I/O devices interactions;
- router servicing of packets system.

In terms of computer hierarchy, concurrency of computing might occur and be supported at levels of

- machine instruction,
- high-level language statement,
- module, and
- program.

So far, there is no introduction at the language level of special operators for concurrency of processes, while in the structure of runtime system two main monitors as above are dealing with a concurrency of processes.

At the instruction of hardware, there are known concurrency instructions that were implemented since the late 50s of the past century. Two further figures—taken from [3] describe concurrency levels of presentation and appearance in the system and system main elements and layers (Figs. 12.2 and 12.3).

There is no doubt that amount of entries related to concurrency in the web and intensity of research and development is high. The point is through—quite a few—are dealing with algebra representations, and very few, if any, address the real problems or requirements—PRE: Performance-, Reliability-, Energy–efficiency of concurrency handling.

- “Low-level” subsystems
  - DMA devices
  - pipelined multipliers
  - carry-lookahead adders
  - signed-digit ALUs (carry-free adders)
  - race conditions in sequential logic
- “High-level” subsystems
  - special-purpose coprocessors
  - instruction-level parallelism
  - communication, especially in networks
  - multithreaded computations
  - VLIW architectures
- Multiprocessing
  - load-balancing and scheduling
  - asynchronous communicating processes
  - coarse-grain vs. fine-grain parallelism
  - modalities of multiprocessing
    * shared memory or address space
    * processor arrays
    * workstation farms
    * heterogeneous systems
  - data- vs. control-parallelism
  - communication between control threads
  - the data-flow paradigm

Fig. 12.2 Hardware concurrency [3]

- Explicit parallelism
  task spawning
  - message passing
  synchronization primitives
  - shared vs. local address spaces
- Implicit parallelism
  nonprocedural programming
  - “big” data types (as in APL)
    * functional programming
    * “high-level” database query languages
    * data flow languages
    * logic programming
    * blackboard systems

- Communication issues
  - communication primitives (broadcasting, gossiping, etc.)
  - message-passing modalities (store-and-foreward, hot-potato, wormhole)
  - latency hiding (pipelining, multithreading)
- Algorithmic paradigms
  - parallel-prefix (a/k/a scan)
  - symmetry breaking
  - permutation routing
  - the many roles of randomization

Fig. 12.3 System software concurrency in languages compilers and algorithms [3]

12.2.3 The Matter of Concurrency

Let us have a look at what is a matter of concurrency for processes that have the duration and mostly independent—Dijkstra called them “cooperating sequential processes” [4, 5].

Where individual processes appear that will compete (concur) in the future for any of the concurrency problem to solve. As Dijkstra said:

Two processes are said to be loosely connected if they are completely independent of each other, except for periods of communication.

Essentially, this specification disallows assumptions based on relative speeds of processes.

Imagine we have two processes run off of the same clock; this is known and it works. one can exploit the fact that the clock is the same and frequency is set.

It is possible to use it efficiently—as it works with SIMD structure. At the same time, assuming changing the clock frequency for one or both processes might ruin the synchronization completely.

Processes that run off of the same clock can actually be viewed as having implicit communication via the clock, and instead, a solution is required that allows a combination of processes to work together regardless of how they are implemented.

This alienation of the synchronization technology from synchronization concepts is crucial for the success of any concurrency handling and has to be analyzed in full details.

Thus, the next steps are about known solutions and extracting the set of principles that must be pursued in the system design when concurrency appears to be solved.

Without synchronization of loosely tied processes, first, computers were waiting for the completion of printer task, then perform memory reading (from a magnetic device called DRAM), and only then perform calculations.

Thus, the slowest device was defining the overall performance of the whole system. Well, this is not right as we described above—time is not a resource to play with making concurrency working.

12.2.4 Known Solutions and Derived Principles to Follow

Well-known scheme of round-robin resource sharing as above was one of the first implemented.

It is still in action on small local area networks and distributed computer systems (including multiprocessor systems).

In this solution time is used as a shareable resource, time allocated for synchronization/concurrency resolution is divided into slots, the number of equal slots are defined by the number of processes requested the slot of communication.

This scheme is "equal rights" scheme, where all processes are allowed the same amount of time and wait their turn next time around when all slots were given.

Interestingly, concurrency is a drawback of parallelization—(explanations further follow) and both were designed to maximize the performance of the system.

Thus, the major drawback of round-robin synchronization is waiting time for each process—the whole round to be wasted.

Another drawback of round-robin synchronization is wastage of critical resource: no matter whether the process is in need or not the resource (slot of time) is allocated and therefore ALL processes that requested synchronization will wait. Timing of executing the right to have a slot of communication becomes costly for all the rest. It is clear that it is worth trying to separate concerns—personal interests and system interests are not always matching each other.

This round-robin approach was criticized, since Dijkstra as if time as a resource is in shortage one SHOULD NOT USE time as a resource to arrange coexistence of processes, their communication, and completion.

It means that WHILE and similar waiting constructions should be used very carefully when it concerns a synchronization. Our personal opinion is that these types of construction should be prohibited from concurrency management schemes.

Below some principles of concurrency handling are presented as they were developed by computing society to achieve some decency in solving a synchronization problem.

12.3 Communication of Sequential Processes: Principles of Synchronization

For successful communication (in philosophy—an *interaction*) of processes, the system must provide race free condition; in other words, no matter which process comes first this time rule of entering into interaction session must not depend upon ordering of execution;

Processes interests are different in their own time and interacting time, thus there is a reason to separate behavior of the processes as a *thing for itself* and *thing for us* (more detail see E. Kant, Critic of pure reason) [6]. A section of the interaction of a process with others using resources of the system was called a critical section.

Critical section, therefore, had to have the following required properties, Fig. 12.4:

Figure 12.4 presents only obligatory conditions. Above them, one might introduce desirable properties of synchronization system as we have mentioned PRE-smartness, for example. Thus, good concurrency system solution should be (Fig. 12.5):

Let us consider several known solutions—well, fundamental ones, and check how these rules—obligatory and desirable are implemented.

The first known solution of a problem of mutual exclusion for processes that match the requirements above was done—to my best knowledge by Dekker [3, 6], Fig. 12.6.

- ***mutual exclusion*** *must be implemented - only one process must be permitted to be within critical section;*
- ***conflict resolution for simultaneous requests****, with one process to go through into the critical section; also called* ***liveliness*** *- if several processes request and entry one of them get access*
- ***independence*** *from processes demand from outside critical section*
- ***deadlock free*** *- no process to block each other*
- ***starvation free - bounding*** *- system does not let a process to wait forever; each process will get its time in critical section*

Fig. 12.4 Synchronization rules

- ***efficient***; in other words do not consume substantial resources from the system or processes to arrange a process of waiting
- ***progress:*** *no process is forced to wait for an available resource -- otherwise very wasteful at a time;*
- ***do not implement busy wait*** - i.e. endless knocking the closed door is forbidden;

Fig. 12.5 Synchronization rules—desirable

The basic idea behind the implementation of Dekker's algorithm is that a process enters the critical section—further (CS) straight away when the other process is not competing for CS.

When both processes are simultaneously interested in accessing CS, the tie is broken by allowing the process which accessed CS least recently to succeed.

The algorithm uses three binary variables, cl, c2, and turn.

For better readability, we rename the first two variables, respectively, as state [1] and state [2].

Also, we define turn as "other", that may take value either 1 or 2, while *status* out as 0, and status of competing as 1.

The status bits are initialized to "out". The formal, pseudocode of mutual exclusion is presented in Fig. 12.7. Another few hundreds of versions of the similar code are everywhere starting from Wikipedia and ending every second Ph.D. on the subject in the world and, therefore there is no need to refer to any authorship here—this is public domain knowledge.

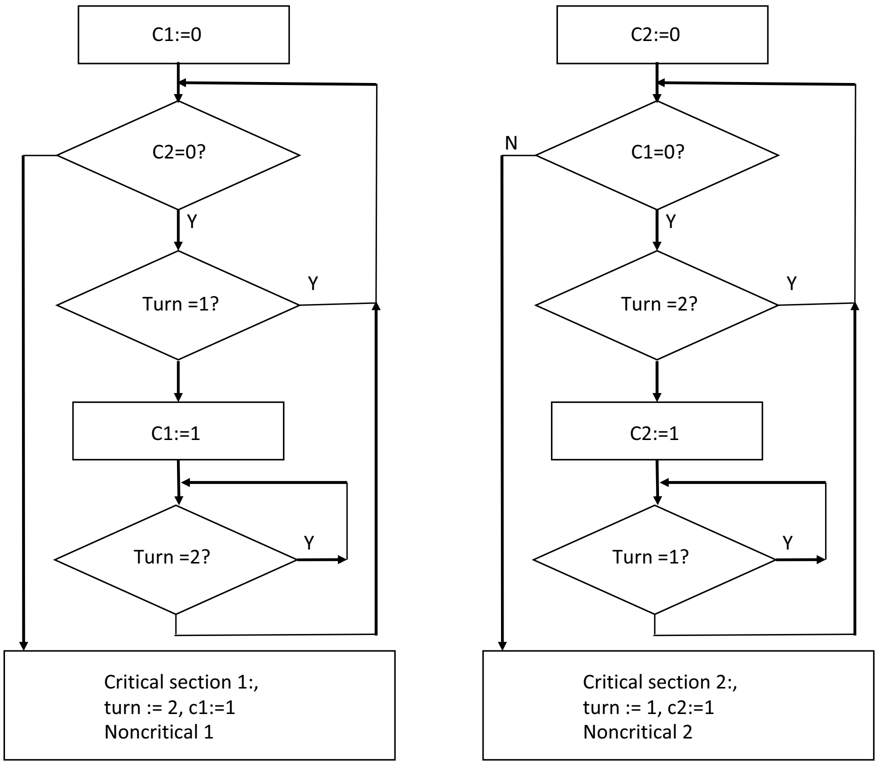


Fig. 12.6 Synchronization by Dekker for two processes

Please note that in both variants of mutual exclusion: Figs. 12.6 and 12.7 we have to assume un-interrupting execution of tests of global variable and setting of own variables.

This is supported by hardware instruction usually called "test and set" introduced in early 50s hardware architectures.

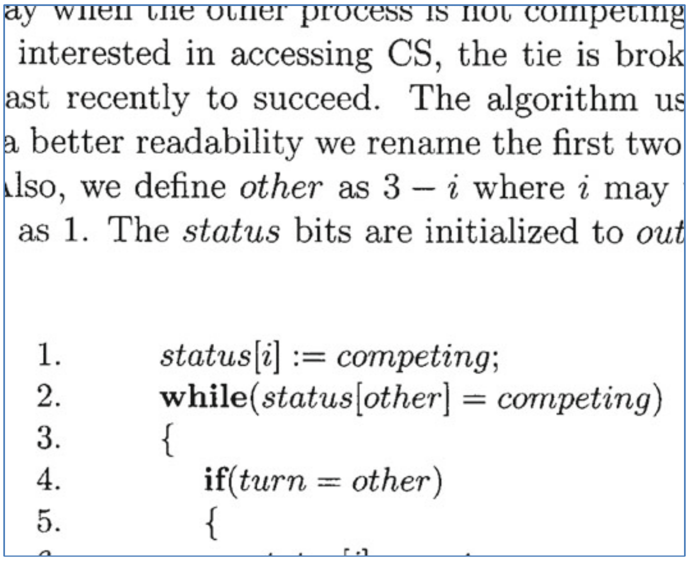
Mutual exclusion is implemented here because entering of the critical section is taking place if and only if:

- either other process does not want to enter, or
- other process wants to enter, but it is not its turn.

A property called ***progress*** is achieved: both processes cannot wait forever at a while loop.

The process gets access to the critical section immediately if other processes do not want to enter, or other processes (matching turn) will eventually finish.

Starvation free property is also fulfilled as one process after entering the waiting phase will wait at most one critical section.



ay when the other process is not competing
interested in accessing CS, the tie is brok
ast recently to succeed. The algorithm us
a better readability we rename the first two
lso, we define *other* as 3 − *i* where *i* may
as 1. The *status* bits are initialized to *out*

```
status[i] := competing;
while(status[other] = competing)
{
    if(turn = other)
    {
```

Fig. 12.7 Graph-logic model, helps to separate concurrency and parallelism

12.4 Parallelism and Concurrency Versus GLM

Using Arnold Rosenberg paper [3] and ERA paper [2], one might find that quite a lot of mess we have found in discussing of concurrency and parallelism. Thus, if one will use Intel official link:

http://software.intel.com/en-us/articles/how-to-sound-like-a-parallel-programming-expert-part-1-introducing-concurrency-and-parallelism/

he might end up that concurrency and parallelism are synonyms. To clarify this situation and preserve others from confusion, our model presented below called the graph-logic model (Fig. 12.8) can be used clarifying this matter.

Every meta-program structure might be described using GLM (Graph Logic Model) that provides a scheme to redevelop existing programs into their maximum parallel and minimum concurrent form, limited only by available hardware resources.

Note that GLM might be applied for any graph of the program or process model.

GLM uses logical operators from the set {AND, OR, XOR} for every program or hardware scheme that it describes.

These operators are allocated for the input and output of each vertex.

A vertex might be an operator, instruction, or state. Vertex a in the example of Fig. 12.8 may be described thus as

$$\boldsymbol{a} : \boldsymbol{OR} - (\boldsymbol{\alpha b}, \boldsymbol{\gamma d}), \boldsymbol{AND} + (\boldsymbol{\beta b}, \boldsymbol{\delta c}) \tag{12.1}$$

here "−" stands for every logical operator of an output link and "+" for every input link, while α, γ, β, δ are weights or priorities assigned for the link.

Until now research in parallelism was mostly targeted at finding parallel branches of programs and independent data elements.

However, expecting pure parallelism is hardly feasible: what is initiated as parallel segments end up ultimately in concurrent mode, competing for a resource such as a socket, printer, data concentrator, etc.

The rare exception, such as graphic processors with high numbers of SIMD like processors just proves the rule.

The simple notation as in Eq. 12.1 of Fig. 12.8 can be used as a first step in the formation of the graph logic language to describe program structures and hardware structures consistently in terms of coexisting concurrency and parallelism.

GLM explicitly separates parallel and concurrent elements in the system description by introducing logic operators in the program graph for incoming and outgoing ends of edges.

Thus, the application of the logic operator XOR (exclusive OR) on an input or output of an edge defines ALL possible concurrencies in the program graphs.

In turn, all possible parallelism in the control graph are defined by finding all outgoing or incoming edges explicitly described by the AND.

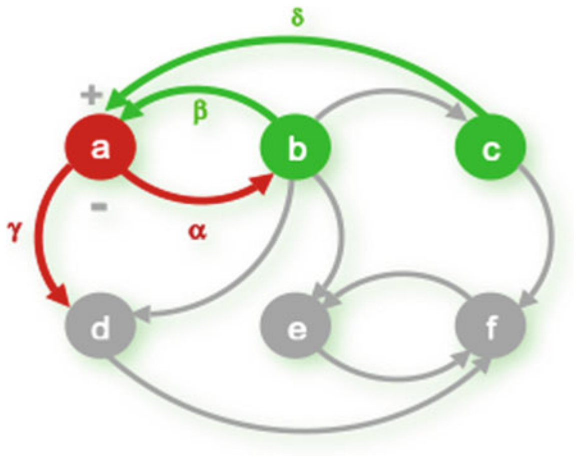


Fig. 12.8 Graph-logic model, helps to separate concurrency and parallelism

12.5 Parallelism and Concurrency: Further Models

Let us start from simple structure—program as Janov's schemata and analyze not want we want or dream but what we actually have algorithmically and how can we implement these algorithmic solutions concurrently, if we can and… what is the gain?

In other words, do we have any performance improvement, at least theoretically?

The simplest case here is when we are having a sequence of independent problems that we are able to rearrange into parallel calculation process:

$$\{A1,\ldots,An\}$$

and each of them is, say, has time to complete Ta.

Suppose we are able to split them to run in parallel. Sooner or later, they will compete for resource whatever it is and step into the phase when the resource is not accessible enough for everybody.

Then "negotiation phase" is required to provide a resource for everybody by following the principles presented above including recommended principles as well.

When monitoring of resource management and sharing is using the time to split processes we afraid we are losing the plot:

concurrency was invented to save system time not to waste it

Here there is a question no 1:

Do we have (if we have any) a concurrency monitoring algorithm that is shortest in terms of time? how to find it? how to prove it?

The eventuality of reaching a critical section for each process declared above (see conditions of concurrency solutions) is not an option worth to consider:

- it means that concurrent processes might be almost prohibitively delayed by… scheme of concurrency!
- it does not mean ith process will starve to death as it is written in the most papers about solutions of concurrent processes.
- it does mean that a goal to achieve any reasonable growth of performance by parallelization of programs and their further execution (in our case, it is processes A1–An) might be deadly ruined by the scheme of concurrency!
- what is actually possible to do? what kind of resource one needs to be able to create for effective concurrence management schemes? good questions…

12.6 Concurrency Handling: Language and Runtime System Aspects

At the same time, there is no great breakthrough in use of all this hardware and software to cope with the problems of concurrency and parallelization in the most effective way.

Each structure of the computer system can be considered from the three key redundancies point of view—structure, time, and information. Algorithm and its meta-representation such as this one on Fig. 12.9 is representing predicates, control, and data dependencies.

All of them are mapped on hardware and all of them are required to be (a) parallelized as much as possible and (b) made concurrent as less as possible.

Thus, the blue star of ith layer and green box of the same layer are potential concurrency-handling-need points. Making them efficiently following through further is a direct impact on the performance and reliability of the whole system (Fig. 12.9).

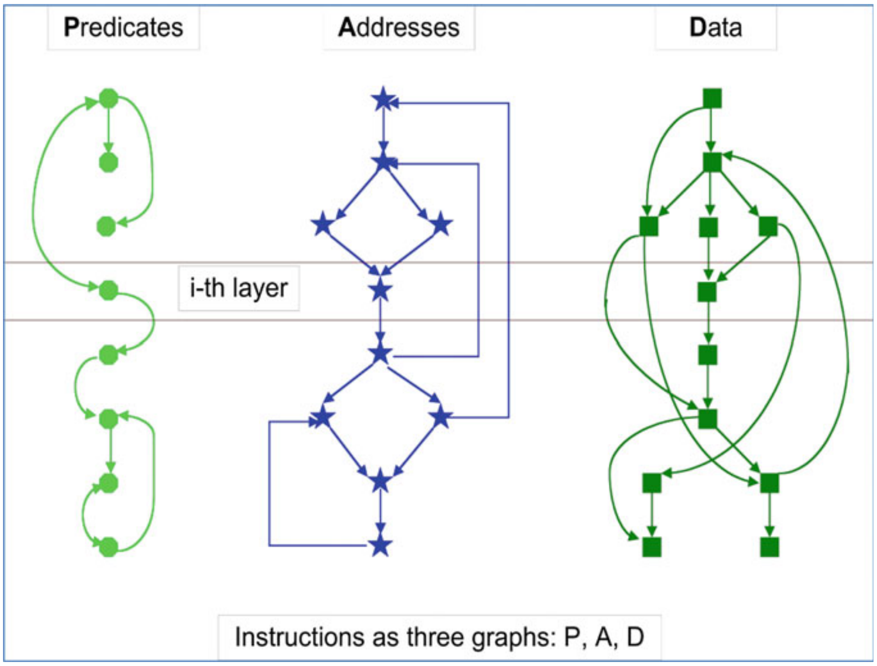


Fig. 12.9 CDP: all three graphs are sources of concurrency implications

12.7 Concurrency: Further Steps: Fault-Tolerant Interactors

This section is dealing with the description of principle of fault tolerance for semaphores that extend a condition proposed by Dijkstra during NATO school presentations about losing process out of the critical section.

Real problem of safety-critical systems is exactly opposite—dealing with process behavior when this process is in its critical section.

An approach is obvious—integer semaphore (not Boolean) must be applied and recovery points such as in 1987 organized for each process entering into the critical section.

Also—if you read carefully Dijkstra assumptions, his synchronization assumes that noncritical section behavior of a process is not important. And this is a pretty naïve assumption, but reconfigurability of hardware was reasonably described in [7, 8] and in runtime system description section there is no room for further analysis of this statement.

But! and this but is big: the behavior of a process in critical section CANNOT be ignored. The assumption that eventually process will leave its critical section is not strong enough: synchronization is about to maximize performance, to kick out a process out of CS by a timer—10 million years later—is not a solution—see the statement about time is not an option.

The question if system resilience and functions of runtime systems monitors come unanswered:

What if a process that own critical section at the moment "died" or "hang up" in there for an unlimited time? All other processes will wait and die eventually. What can we do?

That is how an idea of *faul-tolerant semaphore* came to life.

Thus, *the fault-tolerant semaphore* means that any change of process condition due to internal fault must automatically lead to

- **release a critical section and**
- **release all system resources that were used.**

Further, it implements fault-tolerant semaphore—if we are using the algorithm of Fig. 12.7 we will require a change of variable "turn" by reducing the number of processes that will be permitted to apply for access to the critical section after the fault of a process has happened. Thus, for N scheduled processes monitor of interruptions should reduce the number of "applicants" to N–1.

To avoid fast degradation of the system due to reconfigurability of hardware in this case one has to (eventually, when the workload is not critical) to check the health of a process that was self-declared dead. This is because, for example of natural radiation impact after which malfunctions in the real-time systems now lasts up to for 200–350 ms. Book [7], papers [2, 3, 8] and some chapters of this book might explain details of reconfiguration handling.

This is known the duty of run time system—to deals with synchronization. Implementation of the proposed approach and, specifically *fault-tolerant semaphores* will improve the performance of synchronization of the system and, at the same time, make system resilient—no matter which process fails.

To our best knowledge, nobody neither proposed or implemented anything similar and this is a subject of further modernization of any runtime system for resilient computer systems.

12.8 Conclusion

- Structure of the runtime system for resilient compute is presented and explained.
- The concurrency problems of computing are outlined.
- Requirement for efficient concurrency handling are clarified.
- Shown similarities of concurrency problems for hardware and system software.
- A graph-logic model introduced explicitly separating concurrency and parallelism.
- Control–data–predicates scheme explicitly shows a concurrency points in a program.
- A concept of *faul-tolerant semaphores* for resilient computer systems is presented.
- Tuning of known concurrency algorithms for new semaphores are explained.

Appendix: The Main Algebraic Books on Process Algebra Are

Milner R. (1989): Communication and Concurrency, Prentice Hall
Hoare C.A.R. (1985): Communicating Sequential Processes, Prentice Hall
Hennessy M (1988): Algebraic Theory of Processes, MIT Press
Baeten J.C.M. & Weijland W.P. (1990): Process Algebra, Cambridge University Press.
In turn, applications of the ACP approach explained and well-presented in
R. Milner R. (1990): Operational and algebraic semantics of concurrent processes, in J. van Leeuwen, editor: Handbook of Theoretical Computer Science, Chapter 19, Elsevier Science Publishers B.V. (North-Holland), pp. 1201–1242.

References

1. Schagaev I (2008) Reliability of malfunction tolerance. In: Proceedings of 2008 IMCSIT. https://doi.org/10.1109/imcsit.2008.4747323
2. Schagaev I, Kaegi-Trachsel T, Gutknecht J (2010) ERA: Evolving Reconfigurable Architecture. In: 2010 11th ACIS international conference. https://doi.org/10.1109/snpd.2010.40
3. Rosenberg AL. Thoughts on parallelism and concurrency in compiling curricula. J ACM Comput Surv (CSUR), NY, USA. https://doi.org/10.1145/210376.210399
4. Dijkstra E (1965) Solution of a problem in concurrent programming control. Commun ACM 8 (9):569
5. Dijkstra E (1965) Cooperating sequential processes. Techniche Hogeschool, Eindhoven. Reprinted in: Genuys F (ed) (1968) Programming languages. Academic Press, pp 43–112
6. Kant E (2001) Critic of pure reason. Everyman. ISBN 0 460 87358 X
7. Castano V, Schagaev I. Resilient computer system design. Springer, Berlin. https://doi.org/10.1007/978-3-319-15069-7
8. Monkman S, Schagaev I (2013) Redundancy+reconfigurability=recoverability. Electronics 2 (3), 212–233. https://doi.org/10.3390/electronics2030212

Chapter 13
Proposed Runtime System Versus Existing Approaches

Abstract In this chapter, we briefly compare our approach of a fault-tolerant operating system with existing approaches. We use our own definition of fault tolerance as a process that is required to support the implementation of all steps of GAFT.

13.1 What Is Available?

Integrity OS: Integrity OS [1] is a commercial product by Green Hills Software. It is the only Operating System available on the market, which is certified as Criteria Evaluation Assurance Level (EAL) 6+. However, as the name of the OS indicates, the focus lies on the integrity of the OS and the applications running on the OS, i.e., the tamper proof of the applications running on that platform.

However, as high as its certification level is, there is no fault tolerance in case of malfunctions or permanent errors built in except for restart in case of a failed application.

L4 Se: L4 Se [2] is a member of the L4 microkernel family and is the only Operating System known to be formally verified and tested and thus certified by EAL 7. The focus of this operating system lies on the correctness of the Operating System implementation and the used compiler.

Although its certification level allows it to be used in safety-critical applications, it has no built in means to deal with external radiation induces faults except for restarting processes in case of failures. The microkernel model supports fault containment as all processes and even drivers run in their own address space. The support of a complex MMU is, of course, a strict requirement for these systems.

L4ReAnimator: L4ReAnimator [3] is a framework on the basis of L4 Re that uses the Fiasco.OC [4] microkernel implementation as the basis. Fiasco.OC supports so-called capabilities, which indirectly reference objects and act as communication channels. As no single component in the system knows all capabilities, only the application that uses the capability can detect failures.

Thus, the applications are responsible for reobtaining the capabilities they need. A capability fault handler, which is raised in case of a failed capability can restore the state of a capability, e.g., based on a stored checkpoint.

However, recovery is application specific and is therefore semi-transparent. In case of used resources such as frame buffers, an additional cleanup handler is required.

Minix 3: Minix 3 [5] is another microkernel operation system that claims to be fault tolerant. However, the fault tolerance in this operating system is limited to fault containment by isolated processes and component restart in case of failure. A reincarnation server restarts the failed component and notifies applications about the restart. The state of the component is lost after the restart, thus the application programmer interaction is required to resume processing.

CuriOS: CuriOS [6] is a capability-based client–server operating system that stores client-related state space, so-called Server State Regions, on the server in distinct memory areas which are only accessible by the server if it serves a request by a client. In case of an error on the server (C++ exception), the client-related state space is still available an thus the server can transparently resume operation. This model is designed for misbehaving drivers or other software related errors, but cannot help in case of hardware malfunctions or permanent error that corrupt the state space.

EROS: EROS [7] is also a capability-based Operating System that employs checked recovery points of the whole system to recover from faults. Fault detection is based on failures of tasks, and recovery is done by reloading the last global snapshot.

The assumption, in this case, is that the last recovery point is consistent. Recovery from permanent faults is not supported, nor reconfiguration of the hardware. The recovery point consistency checks, as we see it, just ensure the consistency of the recovery point but not of the system at the point of the recovery point creation.

Table 13.1 lists the main steps of GAFT and states which of the steps are implemented by the above introduced operating systems. This table clearly shows that from a GAFT point of view no OS except for ERA implements all steps.

Especially, the steps *Fault-type* determination and *Hardware reconfiguration* are not natively supported by any OS except ERA. For *recovery*, most of the OS just restart a failed component and component failure is usually detected by assertions, exceptions or watchdogs.

Obviously, masked faults or malfunctions that do not trigger one of the before mentioned schemes are not detected. Only L4ReAnimator supports a checkpointing scheme, which can recover the system to an earlier state. Overall, the ERA concept seems superior in terms of fault tolerance to the other approaches.

The other operating systems, especially the microkernel-based ones have better fault containment than ERA and can also deal with malicious programs.

Table 13.1 Comparison of existing FT approaches to ours

FT OS feature	ERA	Minix 3	CuriOS	Integrity OS	L4ReAnimator
Error detection	Yes	Yes (component failure)	Yes (component failure)	Yes, task failure	Yes, capability and task failure
Fault-type determination	Yes	No	No	No	No
Hardware reconfiguration	Yes	No	No	No	No
Location of faulty software states	Yes	No	No	No	No, last recovery point is assumed to be correct, or application restart
Automatic software reconfiguration	Yes	No	No	No	Semi-transparent, application intervention required
Software recovery	Yes	Yes, restart	Yes, restart	Yes, restart	Yes, restart or recovery point

However, this could be built into ERA by using a simple, memory boundary based memory management unit that would be by far simpler than the fully fledged MMU which is required for the microkernel approach.

References

1. Green Hills Software. Integrity, the most advanced RTOS technology. Technical report, Green Hills Software, 2008. 61, 191
2. Klein G et al (2009) sel4: formal verification of an os kernel. Technical report. 191
3. Dirk Vogt D, Döbel B, Lackorzynski A (2010) Stay strong, stay safe: enhancing reliability of a secure operating system. In: Proceedings of the workshop on isolation and integration for dependable systems (IIDS 2010), Paris, France, April 2010, New York, NY, USA. ACM. 192
4. Lackorzynski A, Warg A, (2009) Taming subsystems: capabilities as universal resource access control in l4. In: Proceedings of the second workshop on isolation and integration in embedded systems, IIES '09, New York, NY, USA. ACM, pp 25–30. 192
5. Tanenbaum A (2006) Reorganizing Unix for reliability. In: Proceedings of 11th Asia-Pacific, pp 81–94. 192
6. David F et al (2008) Curios: improving reliability through operating system structure. In: OSDI'08, Berkeley, CA, USENIX Association, pp 59–72. 192
7. Shapiro J (1999) EROS: a capability system. PhD thesis, University of Pennsylvania. 192

Chapter 14
Hardware: The ERRIC Architecture

Abstract There is no doubt, system software support of computer resilience is incomplete when hardware does not have essential properties required to implement it. Luckily, hardware with required support of recoverability and fault detection is available and described in full details in [1–3]. Here, we briefly describe available hardware with attention to implementation support of PRE properties (performance reliability and energy) requirements.

14.1 Processor Architecture

The main principle used in the design of the ERRIC processor is simplicity. The instruction set as well as the implementation of the processor is reduced to the absolute minimum required to support general-purpose computing.

This allows the careful controllable introduction of redundancy features that allow the processor to detect malfunctions during the execution of the instruction itself, abort the current instruction, and re-execute it transparently to software.

In addition, no pipelining or caches are used which greatly simplifies the processor design which in turn simplifies the implementation of any fault-tolerant features.

Antola [4] proved that the overheads necessary to make a CPU fully fault tolerance might easily exceed 100% which in fact corresponds to at least duplication. Thus, to this huge overhead, we strive to keep the redundancy level needed to achieve fault tolerance as low as possible.

In comparison to other architectures such as CISC processors, ERRIC differs in the following features: one addressing mode, hardwired design (no microcode), regular instruction set, and few and simple instructions.

The difference of ERRIC to RISC processors is not as big as to CISC processors, but, for example, the number of instructions is still smaller and the instruction complexity lower (no multiply, etc.).

Figure 14.1 shows a simplified version of the instruction execution of the ERRIC processor. The execution steps correspond to the ones in other RISC

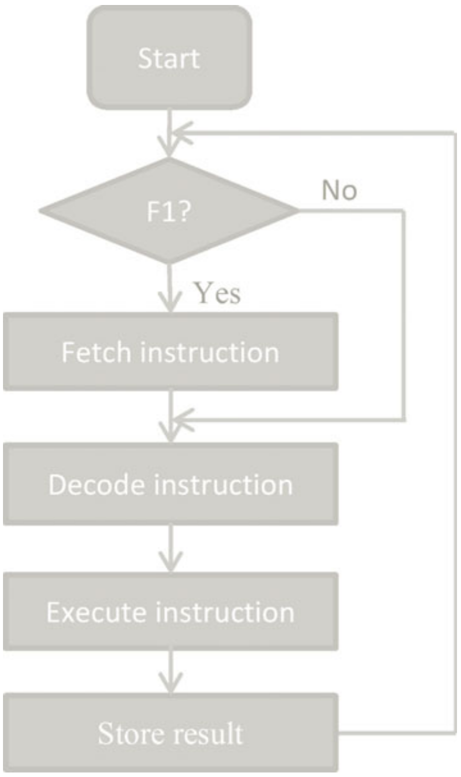


Fig. 14.1 ERRIC instruction execution—simple version

processors, with the exception that no pipelining is used and therefore all steps of one instruction are executed in one processor cycle and specially used memory controller.

We briefly discuss the steps involved in the execution of one instruction.

Fetch Instruction This step loads an instruction from main memory and stores it in the processor internal instruction register (IR). As every instruction is only 16-bit wide (see Sect. 14.2), this step is needed every second instruction, if the instruction is not a jump instruction.

Decode Instruction It decodes the instruction.

Execute Instruction It executes the instruction, i.e., calculates the result of the instruction.

Store Result The instruction execution might have affected register content, processor flags, and other processor state which is updated and written back in this last step.

At first, it could be argued that the proposed ISA does not have enough instructions to perform most of the operations that such a small ISA would result in higher compilation effort/time, and that the resulting programs would need more space.

However, complicated functions are best handled by the compiler instead of having specialized instructions within the ISA dedicated to very particular tasks. The program size of applications compiled for ERRIC is indeed often larger than the same application compiled for other architectures.

The 16-bit instructions help to keep code size small, but the lack of direct immediate support in instructions increases code size significantly. The lack of some often used instructions, such as multiplications, also increases code size, as the corresponding library calls need additional setup code. However, we consider this as acceptable to keep the processor architecture simple.

The processor has a large register file with 32 32-bit wide general-purpose registers. All normal instructions expect exactly two input registers and save the result of the operation in the second register, therefore always overwriting the second operand content. Memory access is only possible for 32 bit at a time and has to be aligned.

The ERRIC processor has an internal state (*F1, F2*), which is required by the special memory controller. The main philosophy of the ERRIC architecture is to keep it simple, which is not only applied to the processor but also to the memory controller. The processor and memory controller are designed not to stall the processor due to pending memory operation.

This is achieved by interleaving instruction fetch and memory operations. All ERRIC instructions are 16-bit wide; an instruction fetch operation therefore always loads two instructions from memory and stores them in a processor internal instruction register (IR).

In sequential instruction execution, the compiler can schedule memory instructions in every second instruction slot where no instruction has to be loaded by the processor. This concept is illustrated in Fig. 14.2, which shows an extended version of Fig. 14.1 that includes memory operations.

In state F1, the processor fetches an instruction from memory, and the memory unit is therefore busy in this cycle and cannot execute a memory operation at the same time. If the processor is in state F2 (not in state F1), the processor can execute a memory instruction. The compiler is responsible to schedule the instruction in proper order.

If the executed instruction is a branch, the processor switches automatically to the F1 state as the memory controller can only load 32-bit aligned addresses. Jump destination locations must therefore also be 32-bit aligned. All these requirements force the compiler to fill gaps with NOP operations.

The main structure of the processor architecture is shown in Fig. 14.3. The instructions are fetched from memory into the instruction register and decoded by the control unit, which also manages the execution of each instruction.

The operands for each executed instruction are fetched from either the register file or memory, multiplexed into the Arithmetic and Logic Unit (ALU) for operation. The output data from the ALU go either to memory through the data bus or is written back to the register file. The current value and type of the data might also indicate an address for branch instructions.

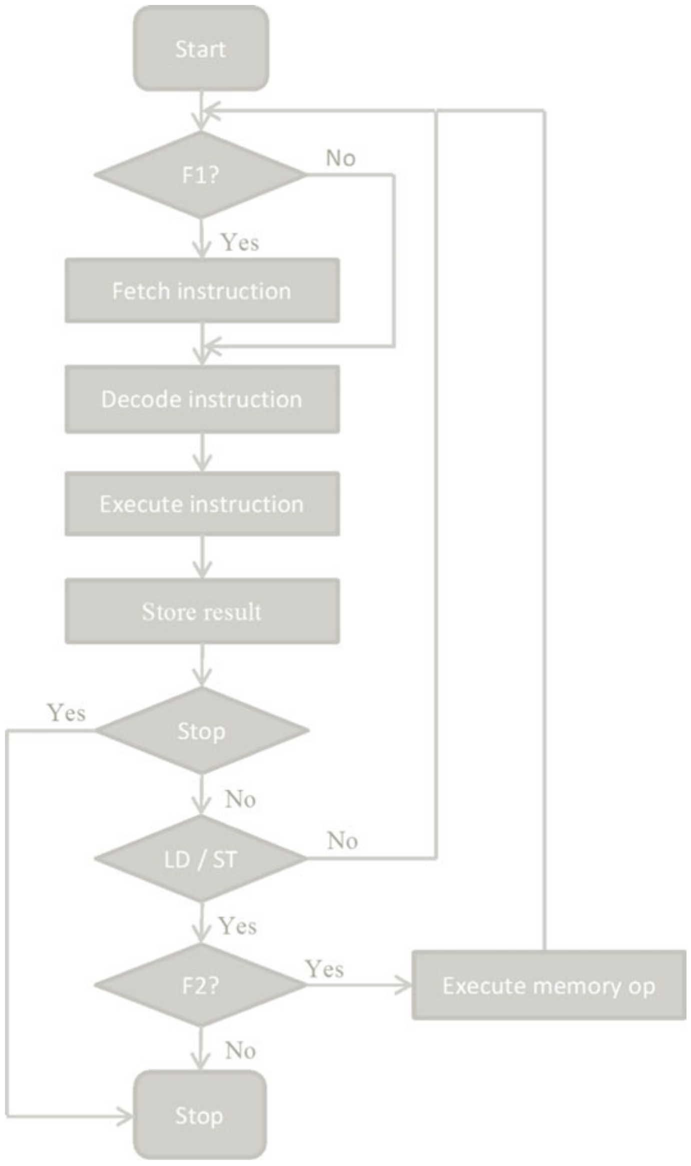


Fig. 14.2 ERRIC instruction execution—extended version

The simplified architecture presented in Fig. 14.3 allows implementation of GAFT at instruction level. As before the three processes are essential for the guaranteed and successful execution of each instruction.

Two processes P1 and P2 (blue and red) cope with fault checking and recovering of malfunctions, respectively.

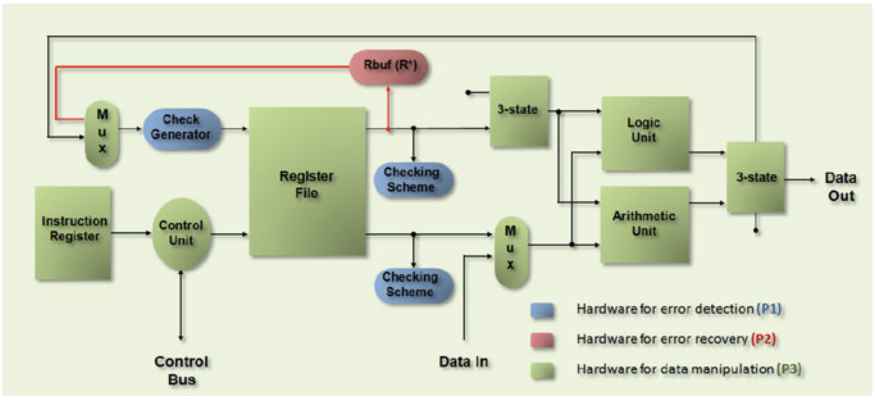


Fig. 14.3 Malfunction tolerance in the active area

P1 is initialized at the start of the instruction execution; P2 is required and initiated when fault has been detected, but it is essential that the pre-modified state is stored at the start of execution of every instruction. P1 and P3 can operate concurrently.

P1 and P2 have an influence on each other: the higher the fault detection coverage achieved by P1, the more successful recovery should be.

When data is written into the register file, the check generator (marked in blue in Figs. 14.3 and 14.4) generates the checking information for the 32-bit data stored into the register file; this information allows the stored data to be verified later on.

The checking schemes (marked in blue in Fig. 14.3) check the data integrity when data are read out from the Register File and when it is possible, correct the data before the ALU operation takes place.

In order to implement the fault recovery process P2, an extra Register Buffer Rbuf(R*) is introduced (marked in red in Figs. 14.3 and 14.4). The register buffer keeps record of the modifying state of the CPU.

When a fault is detected during instruction execution, it allows the processor to restore to the initial state before the execution of the instruction enabling the instruction to be repeated and therefore to tolerate the malfunctions within the instruction level.

The extra Register Rbuf(R*), the checking schemes, and the reverse instruction sequencer combined allow the implementation of the two processes P1 and P2 without any perceptible time overheads. Note that this current version covers SEU only and must be modified in case of MEU.

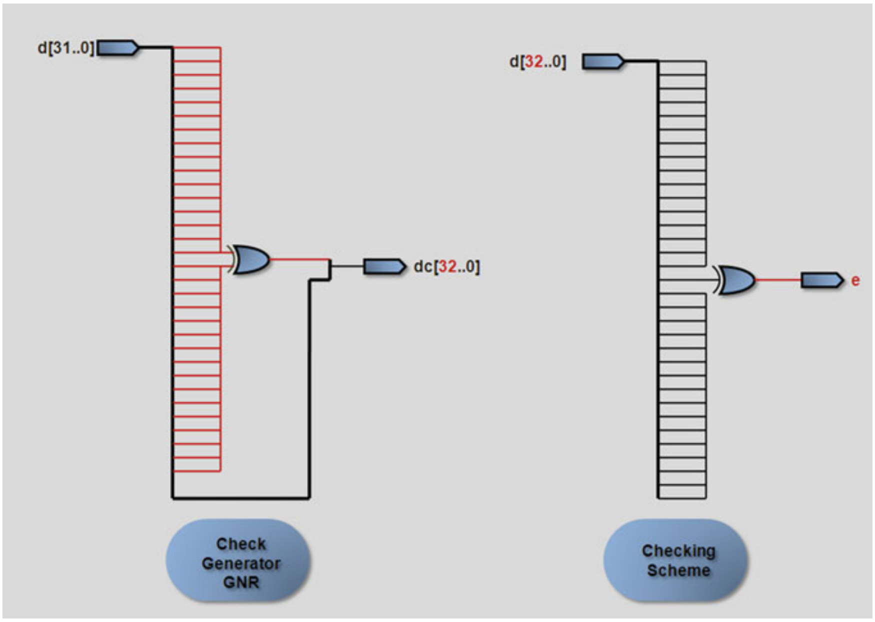


Fig. 14.4 Check generator and checking scheme

14.2 Instruction Set

All the requirements to the processor which are described in (reference) can only be achieved using an ultra-reduced instruction set, consisting of only 16 instructions where only two of them are memory instructions (load/store).

This allows to save the state of the processor before each instruction execution, and, if a fault is detected, to cancel the instruction and re-execute it on the same processor. Furthermore, such an instruction set generally requires less hardware, which increases the reliability of a single processor.

The processor has 32 general-purpose registers. They have a width of 32 bits and are equal, i.e., all registers can be used for every instruction. The instructions are designed as two register instructions, where the second register is used to store the result.

The compiler designer has to keep in mind that the content of the second register is overwritten. Memory access is 32-bit only and has to be aligned to 32-bit.

Every instruction is encoded into 16 bits (see Fig. 14.5). The two highest bits (bits 14, 15) indicate the format of the operation, which can be 8, 16, or 32 bits.

Not all instructions support a format other than 32 bits, but if they do, they only use the specified amount of bits in their operand registers and ignore the others.

Bits 10–13 contain the instruction opcode.

Bits 5–9 identify the first operand (register) and bits 0–4 the second operand.

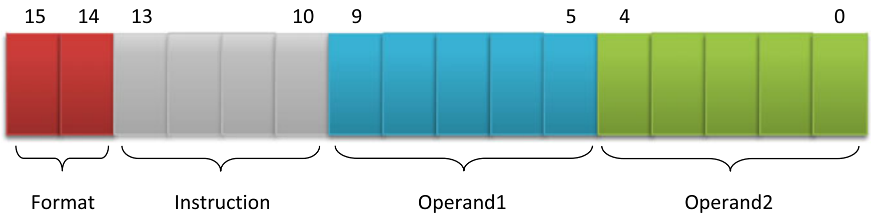


Fig. 14.5 Instruction format

The first and second operands can be any of the 32 general-purpose registers. Constants cannot be directly encoded in the instructions.

Opcode 0 is a special case and is used for four different instructions. Depending on the format, it can indicate a STOP, a NOP, a debug TRACE or RETI, which is used to exit an interrupt handler.

Another special case is the instruction LDA, which loads a constant to the specified register. The constant is loaded from the next aligned 32 bits after the active program counter. The program counter then skips the constant and continues the program execution directly after the constant.

In general, placing two instructions into one 32-bit word increases code density and performance. The processor executes the left instruction first and can then, without accessing the memory, directly execute the second one.

Note that the program counter has the same value for both of these instructions. This can be a problem because it is obviously not possible to jump to an instruction on the right side, as it has no unique address and is not 32-bit aligned.

A compiler has to make sure that NOPs are inserted at the correct places to prevent these cases if instruction reordering fails to fill the gap.

All arithmetic operations treat the value in the operand as signed values. There is no method to execute unsigned operations. Except for the conditional jump instruction, every operation accepts the same register for both arguments.

Table 14.1 lists all available instructions together with a short description of their effect. It also shows the assembler representation used throughout this report.

While most entries are self-explanatory, the result of the compare instruction CND requires more information. Similar to other platforms, it checks for three conditions and saves them as flags in the first three bits of the result register:

$$\begin{array}{lll} \text{Bit} & 0: & \text{R1} > \text{R2} \\ \text{Bit} & 1: & \text{R1} < \text{R2} \\ \text{Bit} & 2: & \text{R1} = \text{R2} \end{array}$$

Using an appropriate constant mask and the bit instructions, the result for every possible comparison operation can be extracted from the flags and saved as a Boolean value, or used as an argument in a conditional jump:

Table 14.1 Available instructions

Instr. name	Format code	Instr. code	Opl	Op2	Description
NOP	01	0000	Ignored	Ignored	Execute no action except increasing the PC
STOP	00	0000	0	0	Stop instruction execution
TRACE	00	0000	Ri > *0*	Rj > *0*	Output *Ri* to debugger. Operandi or Operand2 must be > 0
RETI	11	0000			Return from interrupt (Address in ii31)
LD	11	0001	Ri	Rj	Load 32-bit memory at address *Ri* into register *Rj(Rj: = *Ri)*
LDA	11	0010	ignored	Rj	Load the value from the next 32-bit word (rel. to PC) and store it in *Rj {Rj: = constant).* Operand 1 is ignored
ST	11	0011	Ri	Rj	Store content of register *Ri* to the memory at address *Rj(*Rj: = Ri)*
MOV	XX	0100	Ri	Rj	Move content of register *Ri* to register *Rj(Rj: = Ri)*
ADD	XX	0101	Ri	Rj	Arithmetically add the content of *Ri* to the content of *Rj* and store the result in Rj (Rj: = Rj + Ri)
SUB	XX	0110	Ri	Rj	Arithmetically subtract the content of *Ri* to the content of *Rj* and store the result in Rj (Rj: = Rj - Ri)
ASR	XX	0111	Ri	Rj	Shift the content of register *Ri* arithmetically one bit to the right and store the result in *Rj*
ASL	XX	1000	Ri	Rj	Shift the content of register *Ri* arithmetically one bit to the left and store the result in *Rj*
OR	XX	1001	Ri	Rj	Perform a bitwise logical OR of register *Ri* with register *Rj* and store the result in *Rj*
AND	XX	1010	Ri	Rj	Perform a bitwise logical AND of register *Ri* with register *Rj* and store the result in *Rj*
XOR	XX	1011	Ri	Rj	Perform a bitwise logical XOR of register *Ri* with register *Rj* and store the result in *Rj*
LSL	XX	1100	Ri	Rj	Shift the content of register *Ri* logically one bit to the right and store the result in *Rj*
LSR	XX	1101	Ri	Rj	Shift the content of register *Ri* logically one bit to the left and store the result in *Rj*
CND	XX	1110	Ri	Rj	Arithmetic comparison of *Ri* with *Rj* and store the result in *Rj*
CBR	XX	1111	Ri	Rj	Jump to address in *Rj* if *Ri* is non zero and save PC in *Ri*

Relation	Bit Mask
$<$	010
$\leq$	110
$>$	001
$\geq$	101
$=$	100
$\neq$	011

References

1. Schagaev I (2008) Reliability of malfunction tolerance. In: International multi-conference on computer science and information technology, IMCSIT 2008, Oct 20–22, pp 733–737
2. Blaeser L, Monkman S, Schagaev I (2014) Evolving systems. In: Proceedings of World Comp'14, USA, July 14. https://www.academia.edu/7685575/Evolving_systems_WorldComp_2014
3. Castano V, Schagaev I Resilient computer system design. Springer ISBN 978-3-319-15069-7
4. Antola A, Er´enyi I, Scarabottolo N (1986) Transient fault management in systems based on the AMD2900 microprocessors. Microprocess Microprog 17(4):205–217

Chapter 15 Architecture Comparison and Evaluation

Abstract Currently, there are several processor architectures on the market, dominated by the x86 architecture on the desktop and the ARM in embedded devices and should therefore be included in an architectural comparison. Thus without even brief analysis, how resilient computing "fit" these available architectures and why we need to develop our own ERRIC, our implementation and application would be incomplete. The core of the analysis of instruction set and impact on computer architecture was presented in Schagaev et al. (Instruction set and their impact on modern computer architecture, 1989, [1]). Since then nothing much has been changed in architecture, few system architectures were lost in the market competition (not the worst one, to be honest), leaving SPARC, Intel, and ARM with lion shares of the market. The SPARC processor is included in this comparison, as it is heavily used in space, especially the fault-tolerant version LEON3 (LEON3-FT SPARC V8-RTAX—Data Sheet and User's manual, 2009, [2]). In order to have a consistent comparison, we only compare the 32-bit versions of the respective processors, although 64-bit versions are available for the x86 and SPARC architecture. Both the SPARC and the ARM processors are RISC based and are thus quite similar in the instruction set, whereas the x86 architecture is CISC based and provides a multitude of instructions in comparison to the other architectures. The x86 is also the only architecture that allows using memory locations directly in instructions (register memory architecture), whereas RISC machines (load and store architecture) must first load the argument into a register. This enormous number of instructions leads to the curious situation that the instruction decoder of the Intel Atom CPU needs as much space as a complete ARM Cortex-A5 (Nachwuchs fr die die cortex-a-familie, 2009, [3]).

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_15

15.1 Processor Architecture Overview

Table 15.1 gives an architectural overview of the mentioned architectures. The overview and analysis are inspired and adapted from the architectural comparison in [4]. All information to compile the table were acquired from various documents [1, 4–9].

Table 15.1 Processor architecture comparison

	ERRIC	x86	SPARC v8	ARM7TDMI (ARMv5-TE)	ARM7TDMI thumb
Date announced	2006	1978	1987	1985	1995
Architecture family	Reduced RISC	CISC	RISC	RISC	RISC
Integer registers	32 × 32 bits	8 × 32 bits	31 × 32 bits	15 × 32 bits	8 × 32 bits + SR, LR
Floating-point registers	0	Optional 8 × 32 bits or 8 × 64 bits (8 × 80 bits internal)	32 × 32 bits or 16 × 64 bits or 8 × 128 bits	Optional 32 × 32 bits or 16 × 64 bits	Optional 32 × 32 bits or 16 × 64 bits
Vector registers	0	Optional 8 × 64 bits or 8 × 128 bits	0	Optional 32 × 32 bits or 16 × 64 bits	0
Address space	32 bits, flat	32 bits, flat or segmented	32 bits, flat	32 bits, flat	32 bits, flat
Instruction size (bytes)	2	1–15	4	4	2
Multiprocessor capable	No	Yes	Yes	Yes	No
Processor modes	1	3 operation modes	2	7	7
Data aligned	Yes	No	Yes	Yes	Yes
Memory management unit	No	Yes	Optional	Optional	Optional
Memory addressing modes	1	7	2 (reg ind + disp, reg ind. indexed)	6	6
Memory addressing sizes	32-bit	8, 16, 32 bit	8, 16, 32, 64 bit	8, 16, 32 bit	8, 16, 32 bit
Number of instructions	16	138 Integer and logic 92 floating-point total: 332	72	53	37

(continued)

Table 15.1 (continued)

	ERRIC	x86	SPARC v8	ARM7TDMI (ARMv5-TE)	ARM7TDMI thumb
I/O	Memory mapped	Instructions, memory mapped	Memory mapped	Memory mapped	Memory mapped
Pipeline length	No pipeline	Atom: 16 Core i7/Core 2: 14 Pentium4: 20–31	Leon 3: 7 SPARC64 V: 15 UltraSPARC T2: 8	3	3
Specialities	Very simple instructions, built-in fault tolerance	Instructions of variable size, grown instruction set, memory operands	Register window, delayed control transfer	Conditional instruction execution	32-bit ARM instructions partly required. Solved with Thumb 2 mode

The CPU architecture summary given in Table 15.1 clearly illustrates the simplicity of the ERRIC processor architecture with its 16 instructions, which is by a large margin smaller than the other architectures.

However, the simplicity is not for free. It comes at the cost of less powerful instructions in comparison to the other instruction sets and results thus in longer code. With longer code, we mean that the number of instructions is higher, not necessarily the code size, as the instructions are only 16-bit wide.

The load from memory serves as a typical example, where in case of array accesses the absolute memory address must be explicitly calculated and stored in a register prior to the memory access, where, for example, in the SPARC architecture the offset to the base address can be stored in a distinct register and then added on the fly in the load instruction itself. The ARM even allows encoding an offset to a base address given in a register directly in the instruction itself.

The ARM Thumb instruction set is a subset of the standard 32-bit ARM instruction set. It is intended for resource-constraint environments, where only a 16-bit data bus is available and it increases the code density.

The Thumb mode does not change the ARM architecture at all; in specific, the address space is still 32-bit and all registers are 32-bit wide. However, only registers R0–R7 are directly accessible, and R8–R16 are hidden. The stack register (SR) and the link register (LR) are accessed by some instructions.

Interestingly, exceptions (interrupts) always trigger a processor switch to the 32-bit ARM mode prior to executing the exception handler. This supports the statement by Hennessy [4, 5] that the 16-bit mode is not supposed to be a full architecture, but “they [the instructions] are enough to encode most procedures.”

In contrast to this, we say that the ERRIC instruction set, although it is even more constraint than the Thumb instruction set, is intended to be used as a full instruction set, generic enough to encode all language features.

Table 15.2 Supported addressing modes

Data addressing mode	ERRIC	x86	SPARC v8	ARM	ARM thumb
Register	X	X	(X)	X	X
Register + offset (displacement or based)	–	X	X	X	X
Register + register (indexed)	–	X	X	X	X
Register + scaled register (scaled)	–	X	–	X	–
Register + offset and update register	–	–	–	X	–
Register + register and update register	–	–	–	X	–

Compared to the ARM, the ERRIC has less than half the number of instructions, making it much simpler. On the downside, more elaborate instructions must be emulated such as relative memory accesses or procedure calls.

Especially, the latter is very efficiently implemented on the ARM by the "load multiple" and "store multiple" instructions that allow to efficiently putting all procedure arguments on the stack. ERRIC on the other hand has to store all arguments individually on the stack.

Table 15.2 shows the various data addressing modes supported by the various architectures.

Although some SPARC does not directly provide an absolute addressing mode, the register + register mode with a nullified second register is used to emulate this. The SPARC architecture even provides a register which is always nullified, thus the absolute addressing does not incur any performance penalty.

The x86 addressing modes are more powerful than the RISC ones; thus, instead of extending the above table, the x86 addressing modes are summarized in Fig. 15.1.

The offset part of a memory address can be either a static displacement or through an address computation made up of one or more elements in Fig. 15.1.

$$offset = \begin{Bmatrix} CS: \\ DS: \\ SS: \\ ES: \\ FS: \\ GS: \end{Bmatrix} \begin{bmatrix} EAX \\ EBX \\ ECX \\ EDX \\ ESP \\ EBP \\ ESI \\ EDI \end{bmatrix} + \left[\begin{pmatrix} EAX \\ EBX \\ ECX \\ EDX \\ EBP \\ ESI \\ EDI \end{pmatrix} * \begin{pmatrix} 1 \\ 2 \\ 4 \\ 8 \end{pmatrix} \right] + \begin{bmatrix} None \\ 8-bit \\ 16-bit \\ 32-bit \end{bmatrix}$$

$$offset = Selector : Base + (Index * Scale) + Displacement$$

Fig. 15.1 x86 addressing modes

Table 15.3 Offset size of encoded instructions

Offset size encoded in instructions (in bits)	ERRIC	x86	SPARC v8	ARM	ARM thumb
Unconditional jump/call	0	8–32 signed, rel. or absolute, direct or indirect	30	24	11
Conditional branch	0	8–32 signed	19	24	8

The resulting offset is called effective address and can constitute of either positive or negative values except for the scaling factor.

The ERRIC architecture provides only direct memory addressing which results in longer code as the addresses must be explicitly computed before the data can be loaded. Table 15.3 compares the offsets sizes that are directly encoded in the instruction. If not mentioned differently, the offset is always relative to the current instruction pointer.

The ERRIC architecture does not allow encoding the offset in an instruction but must always be given as an absolute address in a register. Thus, it requires an additional "load constant" instruction, which requires another 8 bytes (Load + NOP + 4 Byte constant). In terms of code safety, absolute jumps are preferable as calculated jumps (relative) are more susceptible to faults than absolute jumps.

Table 15.4 compares the instructions required in each architecture to perform basic RISC operations such as load, stores, etc. A sequence of instructions is given

Table 15.4 Comparison of selected instructions

Instruction	ERRIC	x86	SPARC v8	ARM	ARM thumb
Load word	LD	MOV	LD	LDR	LDR
Load byte signed	–	MOVSX	LDSB	LDRSB	LDRSB
Load byte unsigned	LD, LDA, AND;[a]	MOV	LDUB	LDRB	LDRB
Store word	ST	MOV	ST	STR	STR
Store byte	–[b]	MOV	STB	STRB	STRB
Add	ADD	ADD	ADD	ADD	ADD
Add (trap if overflow)	–	ADD, INTO	ADDcc, TVS	ADDS, SWIVS	ADD, BVC +4 SWI
Sub	SUB	SUB	SUB	SUB	SUB
Sub (trap if overflow)	–	SUB, INTO	SUBcc, TVS	SUBS, SWIVS	SUB, BVC +4 SWI
Multiply	–	MUL; IMUL	MULX	MUL	MUL
Divide	–	DIV; IDIV	DIVX	–	–

(continued)

Table 15.4 (continued)

Instruction	ERRIC	x86	SPARC v8	ARM	ARM thumb
AND	AND	AND	AND	AND	AND
OR	OR	OR	OR	ORR	ORR
XOR	XOR	XOR	XOR	EOR	EOR
NOT		NOT			
Shift logical left	SLL[c]	SHL	SLL	LSL	LSL
Shift logical right	SLR[c]	SHR	SRL	LSR	LSR
Shift arithmetic right	SAR[c]	SAR	SRA		
Compare	CND	CMP	SUBcc rO,	CMP	CMP
Conditional branch	CBR	JG; JNLE; JGE; JNL; JL; JNGE; JLE; JNG; JNO; JNS; JO; JS	BR_Z, BPcc ($<$, $>$, $\leq$, $\geq$, $=$, #)	B	B
Call	CBR	CALL	CALL	BL	BL
Trap	CBR	INT n	TIcc, SIR	SWI	SWI
Return from interrupt	RETI	IRET	DONE; RETRY; RETURN	MOVS pc, rl4	–[d]
NOP	NOP	NOP	SETHI r0, 0	MOV r0, r0	MOV r0, r0

[a]If the 8-bit data is aligned, the sequence LD, LDA, and AND can be used; otherwise, an LD and an appropriate number of shift operations must be used
[b]For storing an 8-bit value, the destination address must be loaded, the appropriate bits cleared using a bit mask, the argument shifted, and then written back to memory. In other words, omit using 8-bit values
[c]Only one bit
[d]Interrupt is always handled in 32-bit mode. Thus, a pure 16-bit Thumb CPU without 32-bit mode is not supported

if an instruction is not available in the instruction set. A "–" means that more than a short instruction sequence is required to simulate the specific functionality, typically a longer sequence of inline code or a library call. As the ERRIC architecture does not include a floating-point unit, we omit here all floating-point instructions, as all of them are emulated in software.

References

1. Schagaev et al (1989) Instruction set and their impact on modern computer architecture. Zarubeznya Electronica, No 7:8 (In Russian)
2. LEON3-FT SPARC V8 - RTAX - Data Sheet and User's manual. Aeroflex Gaisler AB, Goeteborg, Sweden, 1.1.0.8 edition, June 2009
3. Verlag H (2009) ARM: Nachwuchs fr die die cortex-a-familie. World Wide Web electronic publication, October 2009

4. Hennessy J, Patterson D (2002) Computer architecture: a quantitative approach, 3rd edn. Morgan Kaufmann
5. Hennessy J, Patterson D (1996) Computer architecture, a quantitative approach, 2nd edn. Morgan Kaufmann Publishers, Inc.
6. Intel Corporation. Intel 64 and 32 architectures software developers manual,vol.1: Basic architecture. Technical Report 253665–032US, 2009
7. Intel Corporation. Intel pxa255 processor developer's manual. Technical Report. Order number 278693-002, Intel Corporation, January 2004
8. Limited ARM (2005) ARM architecture reference manual. Technical report, ARM Limited
9. SPARC International (1992)The SPARC architecture Manual–Version 8. Prentice Hall

Chapter 16
ERRIC Reliability

Abstract The key property of our design is resilience. So far we mostly covered system software methods and schemes to achieve or support it. Previous Chaps. 14 and 15 explained briefly what hardware (processor) should possess (including reduction on functions and limited in architecture options) to be able to implement resilience in the most efficient way. Here, we intend to analyze what we have achieved in hardware design in terms of malfunction tolerance—attempting to use heavy artillery of system software as less as possible, making dirty work of fault detection and determination (malfunction or permanent for hardware).

16.1 ERRIC Reliability Analysis

Having introduced the reliability analysis of malfunction tolerance in Sect. 4.6.1, we apply it here to the current ERRIC processor prototype. Current estimations show a fault coverage of 100% for single-bit upsets (faults that affect one bit in the current operation).

We do not consider multiple faults although it is expected that with growths of die density, the amount of multiple faults will be higher than the amount of single faults. This is subject to future work.

The actual implementation of the ERRIC processor shows that hardware overheads in the range of 12% for checking and 3% for recovery [1, 2].

Using these two values, we can estimate the Mean Time To Failure (MTTF) of this system and also the reliability as a function of time.

The $MTTF_{nf}$ and reliability over time ($P_{nf}(t)$) of ERRIC without fault tolerance are given by Eq. 4.4 and Eq. 4.3, respectively. For the $MTTF_{ft}$ and $P_{ft}(t)$ of ERRIC, we can apply Eq. 4.8 and Eq. 4.7, respectively. We do not use Eq. 4.11 as this equation is not directly applicable to a concrete system.

We recapitulate the equation pairs here reflecting them with figures:

$$P_{mf} = e^{-(1+k)\lambda_{pf_1} t} \tag{16.1}$$

© Springer Nature Switzerland AG 2020
I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_16

and

$$MTTF_{mf} = \frac{1}{(1+k)\lambda_{pf_1}} \tag{16.2}$$

Figure 16.1 illustrates that impact of malfunctions k assumes 100 here, while in reliability the malfunction rate is in the order of 5 higher than the ratio of permanent faults.

Permanent fault is assumed at rate 10^{-5}, k as above = 100.

Assuming that redundancy in the system is introduced for detection of fault denoted as *d*, and recovery denoted as *r,* we claim that implemented checking process and recovery process reduce impact of malfunction on the processor reliability.

Clear a share of tolerable malfunction (success of detection and recovery) should be taken into account; we denote α and it stands for success of malfunction toleration.

$$P_{mft} = e^{-(1=d+r)(1+\alpha k)\lambda_{pf_1}} \tag{16.3}$$

and

$$MTTF_2 = \frac{1}{(1+d+r)(1+\alpha k)\lambda_{pf_1}} \tag{16.4}$$

Redundancy for checking (detection) d varies in various hardware schemes, and the maximum redundancy to provide checking is 100%, or *d* = 1, when duplication of the hardware is used to compare outputs.

Maximum power of fault detection is a privilege of duplication. Assuming the same coefficient *k* of malfunction/permanent fault ratio, resulting reliability with the implementation of fault tolerance proposed for ERRIC is shown in Fig. 16.2.

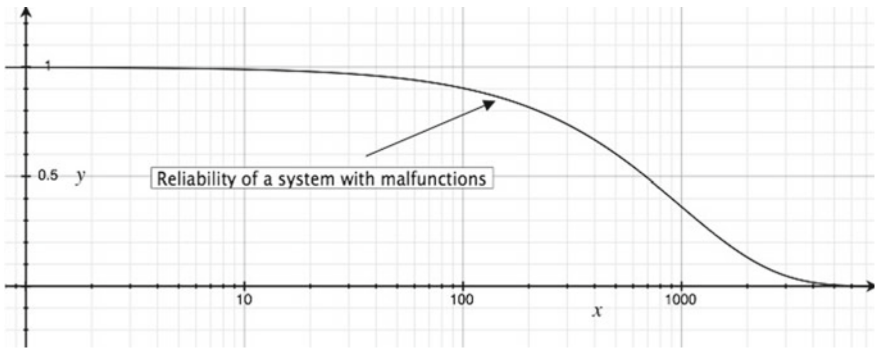


Fig. 16.1 Probability of system operation in the presence of malfunctions

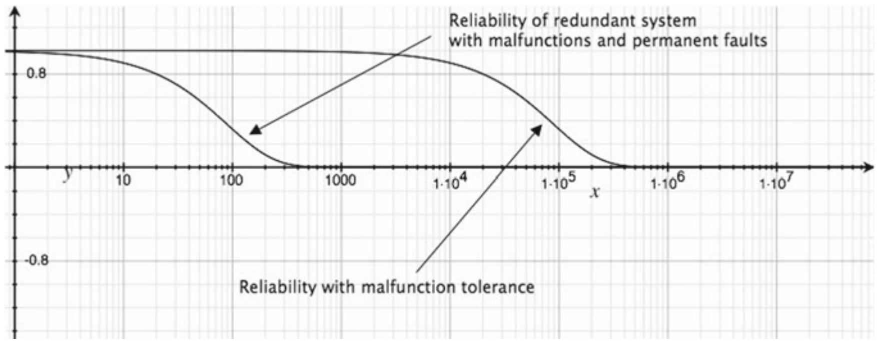


Fig. 16.2 Reliabilitywith and without malfunction

This figure was made with the following parameters and assumptions:

- Redundancy for checking and recovery varies from 0.12 (for ERRIC) up to 1 (i.e., 100% for duplication schemes);
- Coefficient of malfunction to permanent fault varies from 0 to 1000, where 0 means that all malfunctions are tolerated, while 1000 means that malfunctions exist and impact on the hardware.
- Impact of redundancy *d* is obvious and decreases reliability further (see Fig. 16.1).

Note that when redundancy is used to make malfunctions tolerated, we see substantial gain in reliability—right curve in Fig. 16.2. Provided, of cause if we use redundancy wisely.

The resulting efficiency is calculated by dividing the fault-tolerant $\mathrm{MTTF_{ft}}$ by the non-fault-tolerant $\mathrm{MTTF_{nf}}$. For a concrete estimation, we use the values in Table 16.1, comparing ERRIC with ARM and Intel processors [3–5].

Note that the failure rate is only required for the reliability estimation (Fig. 16.1), but not for the resulting efficiency comparison.

The last line of the table above shows a resulting efficiency of roughly 8700, i.e., the fault-tolerant ERRIC is 8700 times more reliable (longer MTTF) than the non-FT version of ARM processor.

Table 16.1 Complexity of redundancy required for fault tolerance

	FT ERRIC	ARM
Complexity overhead d	0	1
Redundancy for checking d_i	12	0
Redundancy for recovery d_r	0.03	0
Malfunction reduction α	0	1
Ratio transient fault—permanent fault k	10^4	10^4
Failure rate λ	10^{-7}	10^{-7}
Resulting efficiency	≈8700	0.5

The reliability of the ARM [5] is calculated under the assumption that twice the number of transistors is used in an ARM processor in comparison to ERRIC which is reflected in the complexity overhead *d,* which is applied to Eqs. 16.3 and 16.4, i.e., using d instead of d_i and d_r.

This estimation is conservative as an ARM processor due to its much more complex instructions set and implementation (pipelines, cache, etc.) needs probably more than twice the number of transistors (latest estimation of ARM and ERRIC complexity enables us to talk about 6:1 [evsy 14]).

It's obvious that ERRIC has by far the highest MTTF due to its ability to recover from malfunctions.

Calculating the efficiency of ERRIC and comparing it to ARM are only half the story. We also should create experiments and provide analytic estimation of the performance of ERRIC over time. There is no doubt that our next and nearest goal, in the analysis of evolving process of reliability and performance of resilient computer system, is to use ERRIC as an example.

16.2 Digging a Bit Deeper in Estimation of Efficiency of Reliability Solutions

Let us add some more realism for the estimation of efficiency as presented below (Eq. 16.5):

$$E = \frac{(1+k)}{(1+ck(1-x^{b(1-x)})(1+x)} \tag{16.5}$$

Comparing the efficiency of solutions for malfunction tolerance of two processor models Eq. 16.5 means that if we "kill" malfunctions with coverage $c = 1$, and with minimum redundancy ($x \rightarrow 0$), then resulting system will be almost k times more reliable as all malfunctions are tolerated, and this is reasonable.

In turn, recoverability of hardware requires redundancy, defined for Eq. 16.5 as *b*. Clearly, when *b* is approximately 10–15%—similar to as ERRIC design—the overall reliability gain is higher, when $b \rightarrow 1, \ldots, 0.95$, the gain decreases (Fig. 16.3).

Comparing the estimated reliability gain, one can see that the potential of this architecture is huge, especially and primarily for resilient computer systems.

Additionally, proposed processor is 5–6 smaller in power consumption than ARM or Intel current models. Therefore, in combination with system software described in previous chapters, we made a substantial progress in this area on resilient computer technology and system software.

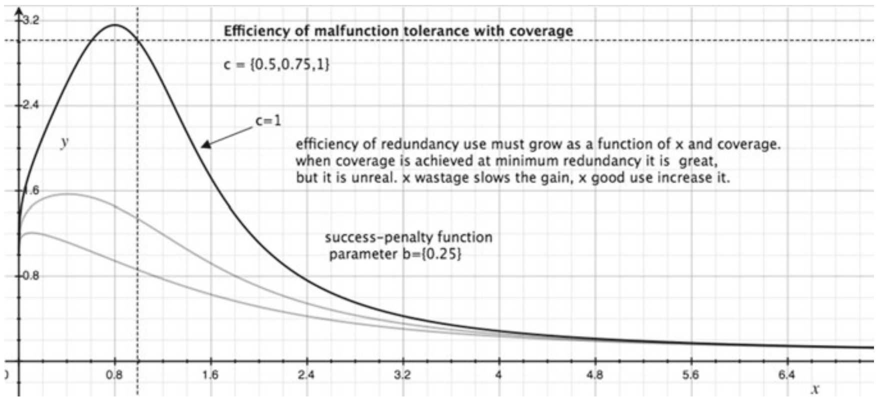


Fig. 16.3 ERRIC comparative reliability with and without fault tolerance

References

1. Blaeser L, Monkman S, Schagaev I (2014) Evolving systems. In: Proceedings of the World Comp'14, USA, July 14. https://www.academia.edu/7685575/Evolving_systems_-_WorldComp_2014
2. Castano V, Schagaev I. Resilient computer system design. Springer. ISBN 978-3-319-15069-7
3. Intel Corporation (2009) Intel 64 and 32 architectures software developers manual, vol 1: basic architecture. Technical report 253665-032US
4. Intel Corporation (2004) Intel pxa255 processor developer's manual. Technical report. Order number 278693-002, Intel Corporation, January 2004
5. ARM Limited (2005) ARM architecture reference manual. Technical report, ARM Limited

Chapter 17
On Performance: From Hardware up to Distributed Systems

Igor Schagaev, Hao Cai and Simon Monkman

Abstract Nothing is easy nowadays: frequency of processors increased thousand times, system performance as a whole sometimes tripled. Complexity of the system became uncontrollable with zillions of processes and elements to juggle increased unconsciously, leaving for us some comfort but at an astronomical cost. What it means? We are doing something seriously wrong and doing it consistently and persistently. Thus, authors of this work have decided to put together our own discussions and estimations we did since 2002 up to now. We show that system performance depends on user, hardware, and software, structure or architecture of a system and its topology. We propose to see performance analysis a bit wider, thinking systematically what various zones of computer or distributed system can bring or contribute, including the role of processor, structure of system software and overvalued parallelization (try to eat and dance at the same time—it might be fun). We have introduced a kind of virtual architecture through which see instruction execution considering what is in there for us and what system requires for itself. The observation is rather pessimistic. We have briefly demonstrated what simplest architecture if carefully designed can give regarding performance, reliability and energy efficiency AT THE SAME TIME! Regarding distributed systems, we show that Amdahl Law is also very overoptimistic mostly serves to promote parallel architectures and distributed systems. Simple model that we have explained for kids from British primary school and even did field study with them so-called "fence model" made clear that the limit of performance or simply overall reasonably good design is unachievable until we start rethinking the whole architecture and its main element interaction—human, hardware and system software together, pursuing three nonfunctional requirements, performance, reliability, and energy efficiency in concert.

As it was presented and argued in [1–6] any system should be considered from the first sketch down to maintenance using the following nonfunctional requirements:

Performance;
Reliability;
Efficiency (cost, energy).

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_17

We call it PRE-requirements. When a system can trade P for R or E and vice versa we call this system PRE-smart system. This chapter is about "P"—performance.

Any system is evaluated in terms of performance, considering performance of elements and system as a whole. Good systems exceed performance of their components, or equal production of component performance; badly design system in terms of overall performance is much less than production or sum of performances of its components. Regretfully, computer systems are poorly designed if we accept this classification. This chapter is about performance and ways to analyze it. We also will model computer system from the position of performance and seeking the ways to improve it, considering use and system performance aspect.

17.1 System Level

Suppose one element has performance Pi; then system of n elements if we can add performance will have maximum performance as n*Pi, i.e., linear growth is assumed. Unfortunately, one has to take into account that external interaction zone (Fig. 17.1) and task structure reduce our expectations about unlimited performance growth.

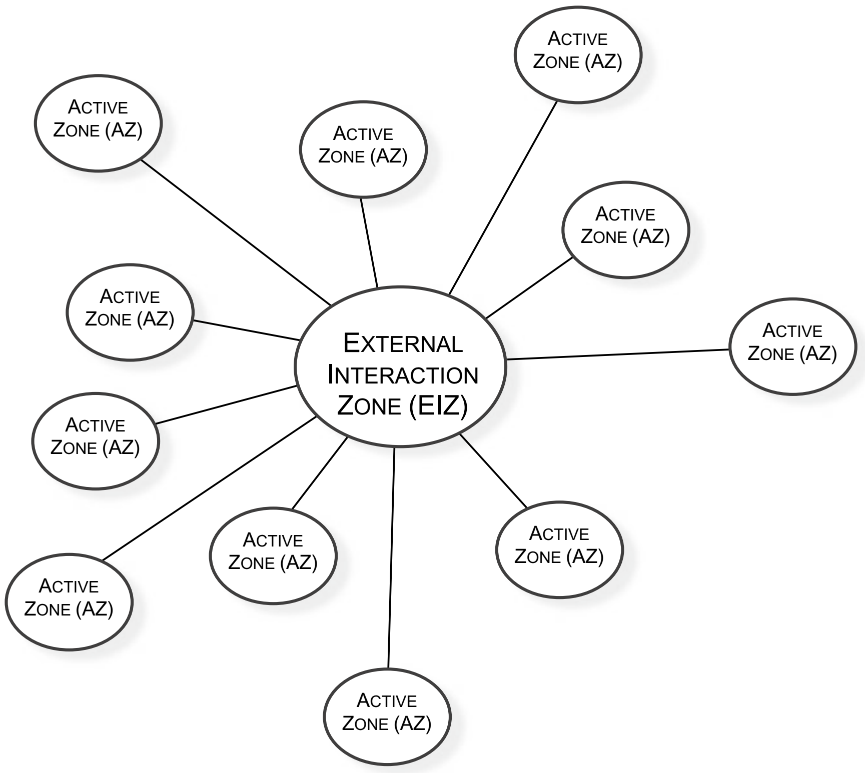


Fig. 17.1 System level of distributed computing

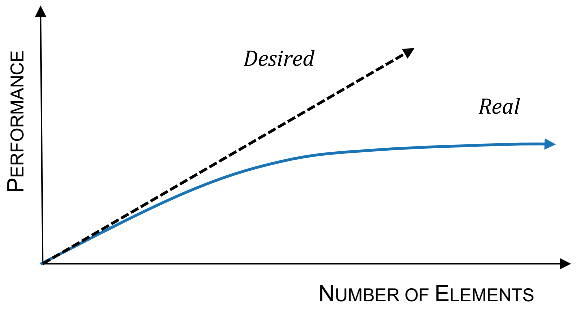


Fig. 17.2 performance versus system structure

Then performance growth is defined as a function of number of elements and EIZ:

$$\mathrm{Ps} = \mathrm{f}(\mathrm{EIZ}, \mathrm{n}) \tag{17.1}$$

where EIZ stands for external interacting zone, n—number of "performers" considered only by performance, not the organization.

Thus, structure of EIZ and its dynamic features (ability to connect transparently arbitrary number of elements with heavy information exchange requests) will impact on system performance of both: system level of performance and element level of performance (Fig. 17.2).

One has to address EIZ features as well as properties of program structure to achieve reasonable gain in performance. System performance-wise program structure itself impact is crucial, as well as ability program to split into independent elements. This ability, in turn, is limited; this causes substantial amount of traffic through EIZ and, therefore, kills performance gain.

17.2 Information Processing Aspect

On the information processing level, consider a system as a black box with input x and output y, with arbitrary function F, Fig. 17.3, top box.

A function or a task of this box in Fig. 17.3 is to get a result y from an input x within allocated time and, if necessary, alter appearance of x in the time slot expected.

Contrasting with mathematical assumptions, for information processing, in principle, input and output timing is loosely dependent, input x might have its own duration while readiness of output y has its own duration, both might overlap, see Fig. 17.4.

A special interest might be in resolving the following cases: how form of x defines (is connected) with function and form of y. Comparative study of durations x and y might be interesting research for embedded systems especially.

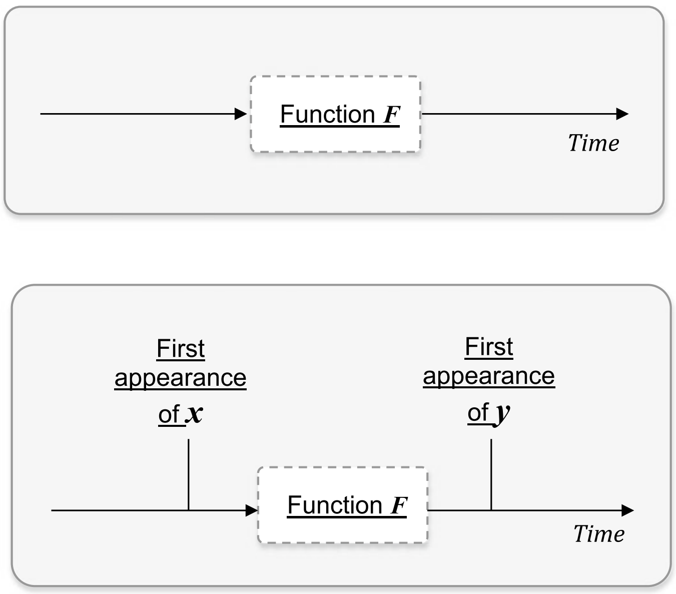


Fig. 17.3 System as a black box, with arbitrary function F

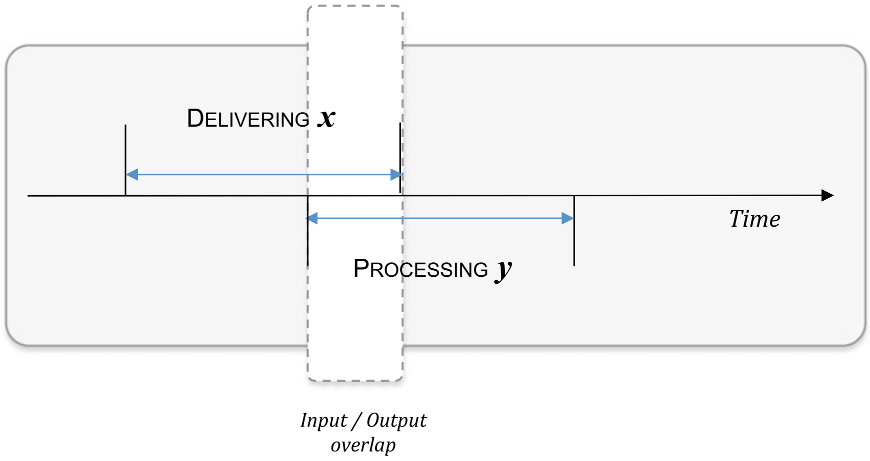


Fig. 17.4 Input appearance might be overlapped with outcome

17.3 Information Systems Task-Wise Hardware Involved Performance

Information system is a combination of three wares: userware, software, and hardware, UW, SW, and HW, respectively (Fig. 17.5).

Thus information processing system might be described in terms of performance of all three: UW, SW, and HW, as they all are involved in information processing. It means that in the long-run performance and efficiency of the system depends on userware, software, and hardware performance.

Their combination might have very peculiar form and mix. In terms of Fig. 17.5, userware, software, and hardware might be connected to perform task as shown in Fig. 17.6.

For real-time systems (and embedded systems especially), or general applications, computer systems over 30 years user features were ignored in terms of overall system performance; it is great regret, we now press much more buttons and click more clicks than 20 years ago achieving very disputable advantage, if any.

This is a subject of special study in UW-SW-HW systems. Here, we spend some more time and analyze details of HW only. However, SW, especially system software (further SSW) has serious impact on performance of hardware, it will be also shown further.

Performance is about task completion in time allocated. The same principle of task allocation and analysis might be applied further down, to the whole system

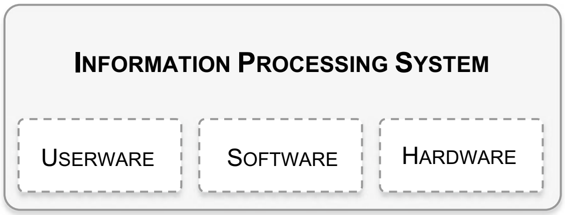


Fig. 17.5 Information system components

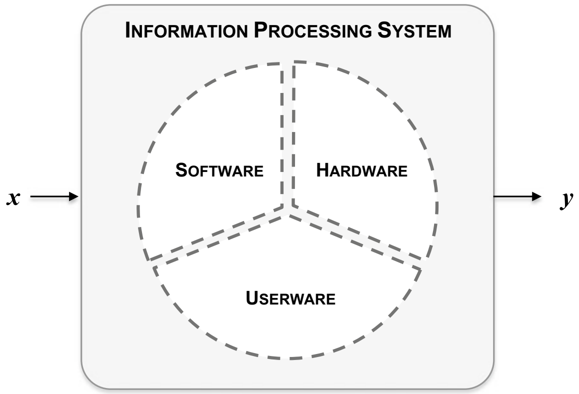


Fig. 17.6 Information system box: x + time = y as three components, S, H, U

hierarchy, for any component—UW, SSW, HW, and business management. Leaving efficiency of management discussion to business schools, we will consider mostly SSW-HW model.

17.4 SSW-HW Performance Model

Task of user (userware tasks, further (UWT)) splits down at the level of HW as a series of micro-tasks (instructions, microinstructions). Program tasks are surrounded by another group of micro-tasks that define the system software (SSW) involvement on execution of the user program, secure a completion, and system resource monitoring.

Thus, userware tasks UWT are accompanied by another group of micro-tasks defined by the system software (system software tasks, further SSWT) (Fig. 17.7).

This total amount of hardware workload in number of instructions W_{ux} to perform user task *x* can be expressed as in Eq. 17.2 where i, j indexes stand for number of hardware instructions required to complete supportive actions (system software need) and user ones (1):

$$W_{ux} = \sum_{j=1}^{m} h_j(sswt) + \sum_{i=1}^{n} h_i(uwt) \tag{17.2}$$

Indexes m and n stand for a system software and user software instructions execution time. Assuming that all hardware instructions have similar execution time (for RISC systems it is essential design condition), one might introduce an efficiency of measure as it is shown below, Eq. 17.3:

$$E_{ux} = \frac{\sum_{i=1}^{n} h_i(uwt)}{\sum_{j=1}^{m} h_j(sswt) + \sum_{i=1}^{n} h_i(uwt)} \tag{17.3}$$

Further, we will dig deeper on performance and efficiency in terms of hardware impact on performance of the system but here note existing relation of efficiency and performance.

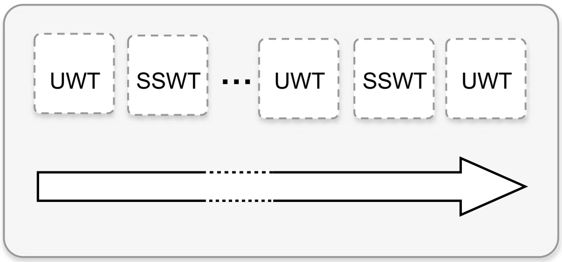


Fig. 17.7 User and system task sequences

Definition 1 Efficiency Eux of computer system is measured by number of instructions required to perform to the total number of instructions performed by computer system.

Naturally, efficiency Eux $\rightarrow$ 1, while m $\rightarrow$ 0, and, no matter what frequency a processor is if m $\rightarrow$ n, Eux $\rightarrow$ 0.

Regretfully, it is a case for current state of the art in computer systems and especially embedded computer systems. What it means for embedded systems especially?

For various systems, it means that

- Application of Java, or use of modified standard operating system, unavoidably reduces efficiency and, above all, runs out our computer batteries for nothing;
- For military systems, an availability and reactiveness is substantially lower than it could be;
- For office systems, nowadays, the employees are sitting and waiting for Windows or Cisco service more than they actually work.

Leaving further comments about efficiency for system software research and PhD projects, let's concentrate on implementation of hardware instructions in terms of time.

17.5 Hardware Performance

One of the simplest information processing systems is a simple turing machine—(http://ideonexus.com/2009/02/05/javascript-turing-machine/); it demonstrates in principle almost minimum of hardware required to perform information processing. Turing machine usually is drawn similar to mine (Fig. 17.8).

Head is moving left and right accordingly instructions stored on the tape and modifies content of the tape. Surprisingly, if an algorithm for problem exists, it is very possible to calculate it and complete.

Turning machine was invented as a model well before Von Neumann's architecture. The latter one assumes that programs and data are placed on the same tape and head Q might perform instructions. More details about this machine might be

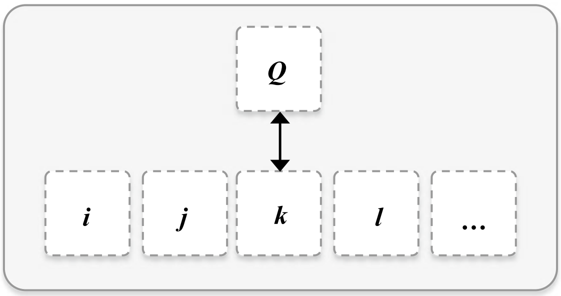


Fig. 17.8 A-la turing for performance evaluation

found in http://plato.stanford.edu/entries/turing-machine/. What is missing in turing or Von Neumann architectures is an answer *how* and *when* algorithm and data were placed into the memory (tape in Turing machine). Besides, arrow on the figure is assumed to connect but not explained how this communication is executed.

17.5.1 Hardware Zones

Thus, from information processing point of view, we have even from this simple picture involvement of three zones (Fig. 17.9):

- active zone,
- interface zone, and
- passive zone.

Active zone (AZ) includes schemes that change information (processors, converters). Interface zone (IZ) includes schemes that transfer information from external source or between AZ and PZ (internal source). Finally, passive zone (PZ) includes schemes that save or store information.

What is missing in the previous picture? To make any information processing model useful, we have to introduce input and output options for information. Each AZ, PZ, and IZ might be active in terms of data flow control (so far not processing) flags—demand of service, interruption request, message writing/sending requests, etc. Let us have a look at Fig. 17.10.

All arrows might be different in speed, and use various frequencies and bandwidths, bit size (8, 12, 24, 36, 48, 64, 128) procedures of control, and data delivery (parallel, sequential). This is still not self-explanatory how we deal with information exchange. More realistic figure is presented in Fig. 17.11.

Level of information interaction AZ, PZ, or IZ actually defines structure of the system similar to Flynn diagram (find what is it, dear reader, it might be useful).

In terms of input and output examples of implementation of each zone, one might suggest the following:

AZ inputs and outputs iAZ and oAZ:

- register content exchange between processors,
- signal write or read to interrupt processor execution,
- mutual exclusion efficiency of multiprocessor communication, and

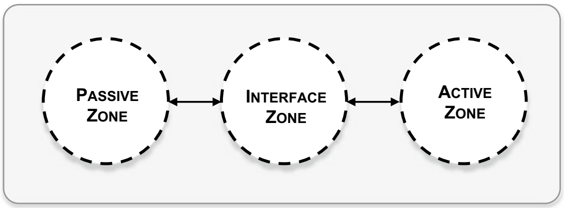


Fig. 17.9 Hardware segments of CA of information processing

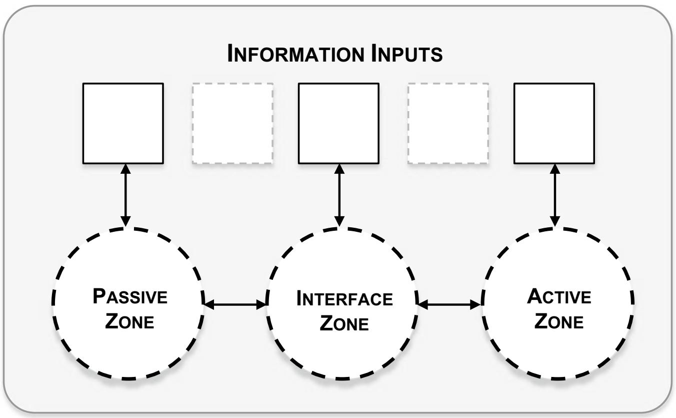


Fig. 17.10 Hardware segments of CA including information exchange options

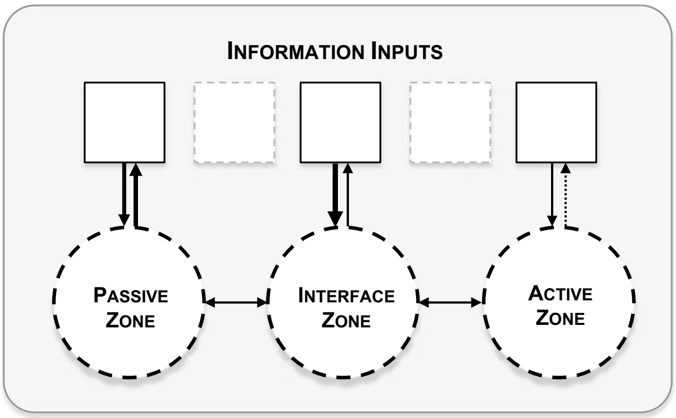


Fig. 17.11 Separation of input and output options for each zone

- Direct writing of processor status word down to syndrome register when syndrome register is external for AZ—flags N, Z, V, and C all might be sent out.

PZ inputs and outputs iPZ oPZ:

- data or control lines from internal bus,
- data path for memory to upload in AZ,
- writing of the result of instruction to output buffers using bus,
- direct access to memory, and
- dual-port memory.

As a small challenge, we invite a reader to present own examples for IZ input and outputs.

17.5.2 Hardware Performance—Instruction Execution

Hardware instruction is implemented using different blocks (zones) that in combination enable to processing.

An example of information tracing through instruction execution is presented in Fig. 17.13 and serves for further understanding of performance evaluation.

Data might be loaded:

– from external source straight to AZ (instruction type A),
– unloaded from AZ to external source (instruction type B),
– loaded directly to the passive zone (instruction type D), and
– unloaded from PZ to the external source (instruction type C).

Information might be

– processed and loaded to passive zone (instruction type H),
– retrieved from PZ into AZ to be processed (instruction type K), or
– processed in AZ and unloaded to PZ.

In fact, interface zone is involved in almost any information transfer internal or external, and in both directions inbound and outbound. It means that the performance of IZ contributes to overall performance of information processing. Special and most promising architecture will be discussed further that limit the role of IZ on overall performance of the computer system.

Other types of instruction not shown might combine the proposed types of instructions.

Let us elaborate a bit what Fig. 17.12 illustrates. Information can come externally through interfacing zone IZ and stored in active zone for further processing—A-type instruction.

From active zone (Arithmetic unit or logic unit or several of them), information through interfacing zone can be transferred out—B-type of instruction.

Well-known good old days direct memory access (DMA)—C-type instructions—are about disabled active zone and read data directly from passive zone (including registers and all types of memory).

The same way information might be loaded into passive zone—D-type instructions—and this is very useful for embedded systems.

In turn, self-isolated instructions, E-type, are about using only active zone and keeping results inside it—sometimes not even addressable at the level of registers—accumulators, signs, or working register.

When instruction processes data, the result can be transferred down to passive zone through interfacing zone—F-type.

Opposite data move: G-instruction are required to deliver data for further processing.

Well away from RISC architectures, instructions can have several cycles of data processing and include result transfer down to passive zone: H-type of instructions.

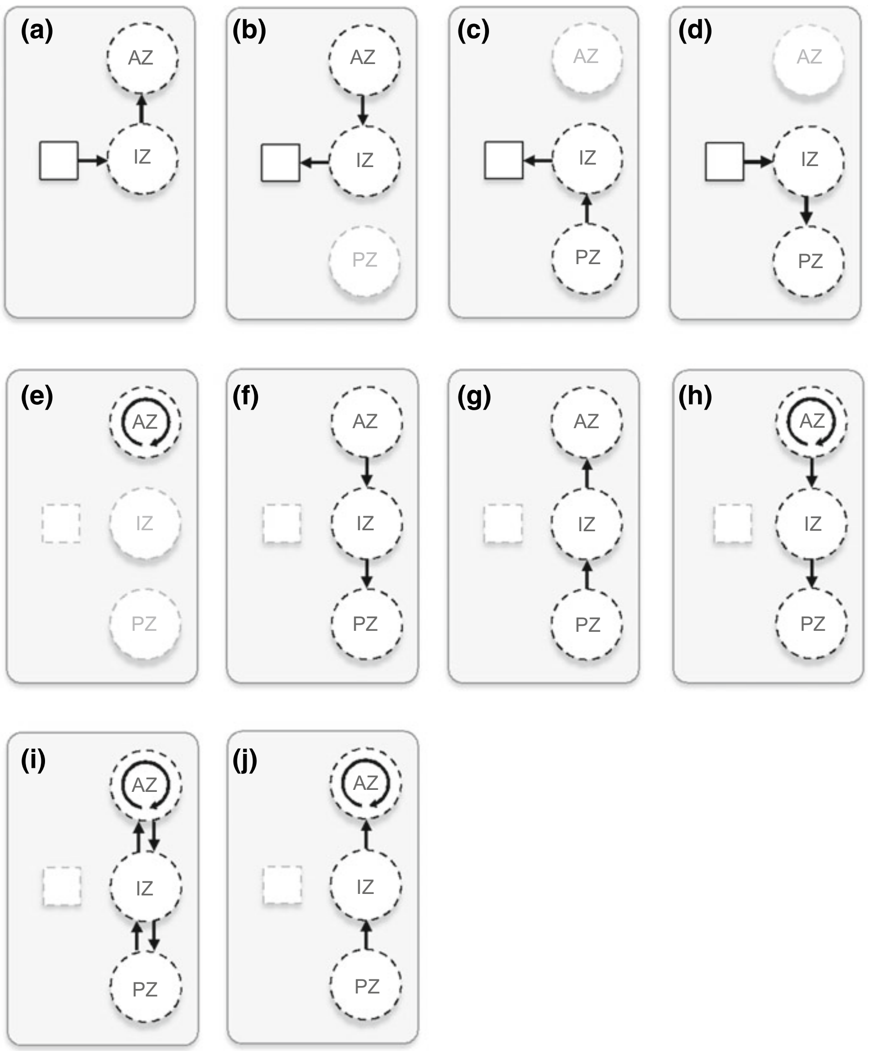


Fig. 17.12 Types of instructions

Last century architectures which are still procrastinated electronic technology by orthodox instruction sets include two types of instructions, I-type and K-type. These two assume that data is taken from passive zone and delivered to active zone, where sometimes after several cycles of processing go back to passive zone (for I-type).

K-type serves the purpose to grab data from passive zone and process, saving results inside active zone for immediate further use.

Please note that passive zone is not starting any instruction; instruction is either initiated by external zone or by active zone. There are some natural questions we might ask here:

Why we need to have a look on these types? why we need to know instruction formats at all?

The answer is simple:

Assuming sequential execution of the instructions, one can estimate an overall performance of hardware for particular tasks and task mix. Each zone has its own specific features in terms of performance and interaction with other zones.

Thus passive zone includes several sources of memory (Flash, SRAM, DRAM) that have different control schemes and timing diagrams, and while involved in instruction execution cause a lot of uncertainties. Interfacing zone might be different in control protocols, width, performance, and reliability.

Two examples of instruction execution using all three zones are shown in Fig. 17.13.

Hiy is loaded from external source, saved in passive zone PZ, and retrieved into AZ to be executed; when results are ready, they are delivered back to IZ_e, where index e stands for external.

Another type of instruction shown describes the load of HIy to AZ, executed in several iterations (or just one) and saved back in PZ.

Data might be processed by instruction when data come from external source directly to the active zone and after processing goes directly to PZ, or when data follow reversed order, from passive zone to active zone for processing and then to external destination.

Data go directly to passive zone—well-known direct access to memory, or when data are dread by external source—direct reading from memory.

Finally, there is a group of instructions in modern hardware that is assigned for reading or writing from or to external source and when data retrieved and processed in active zone.

Passive zone is not starting any instructions either initiated by external zone or by active zone.

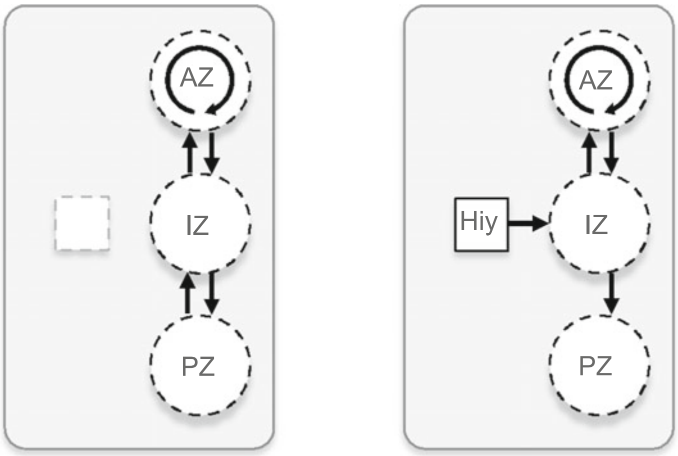


Fig. 17.13 Complex processor instruction from the previous century

In turn, active zone might consist of (even within a mobile telephone) several information processing units for parallel execution of tasks or specific sequence of instructions such as data coding and signal processing. Detailed description of the zones, and their involvement in instruction execution, therefore, might help to estimate the performance of embedded device.

17.5.3 Performance Estimation—Instruction Timing

To do performance estimation of embedded and any other computer system, we have to assume following points:

- Mix of instructions with weight of all types of instructions defining the system performance.
- Time required to perform any instruction—it is worth to separate AZ, IZ, and PZ when instruction is executed.
- Note that instructions that use IZ (external) part are not defined in completion time.

Thus, a mixture of presented types of instructions with taken into account an amount of each of them is a good reflection of overall hardware performance.

In other words, hardware performance might be calculated by summarized length of the instructions required to process user task. Let us consider what is AZ and others in terms of overheads:

$$\mathrm{T}_{\text{instruction execution}} = \mathrm{Tz} + \mathrm{Tiz} + \mathrm{Tpz} \tag{17.4}$$

Obviously, the shorter the cycle of instruction, the better the instructions of A, B, C, D, E, F, and G types that are preferable. It is possible to calculate what difference one might face if use H, I, and K types of instructions, but it is clear that 10^{-9}, 10^{-5}, and 10^{-6} do not fit together well.

It is worth to analyze active zone performance in terms of instruction execution phases:

- Loading,
- Decoding an instruction,
- Preparation of operands,
- Execution of the instruction, and
- Loading back modified operand

Loading means that instruction from memory goes to the load register in parallel with increment of program counter (inside processor) and goes through decoding scheme that activates requires activation operands and execution device ALU.

Decoding means that instruction (loaded from memory) is placed in the internal register (not always addressable) inside processor and by special timing diagram section all hardware links activated: for ALU, operand's addresses and signs (such as N, Z, V, C).

Preparation of operands means that input ports are activated and buffers for instruction data are enabled. For big instructions, it means that instruction execution will be postponed until all memory cycles required to upload operands in processor are complete.

Execution of instruction assumes that ALU (or combination of them) are ready and perform instruction from operation field (ADD, SUB, etc.) in assumption that required operands are ready and delivered from memory (or registers) to internal buffers.

Thus, performance of active zone might be estimated as

$$Perf_{AZ} = \frac{1}{T_{AZ}} \tag{17.5}$$

while

$\mathrm{Taz} = \mathrm{T}_{\mathrm{loading}} + \mathrm{T}_{\mathrm{decoding}} + \mathrm{T}_{\mathrm{preparation\ of\ operands}} + \mathrm{T}_{\mathrm{execution}} + \mathrm{T}_{\mathrm{loading\ back}}$

Pretty much the same one might do with passive and interfacing zones.

Let us have a look for illustrative purposes how instruction is executed by recoverable processor (ReP) made by ITACS Ltd. and prototyped with partial support by grant of FP6 www.onbass.org (Fig. 17.14).

At first from instruction register through control unit via control bus microinstruction goes to either AU or LU. At the same time, microinstruction about

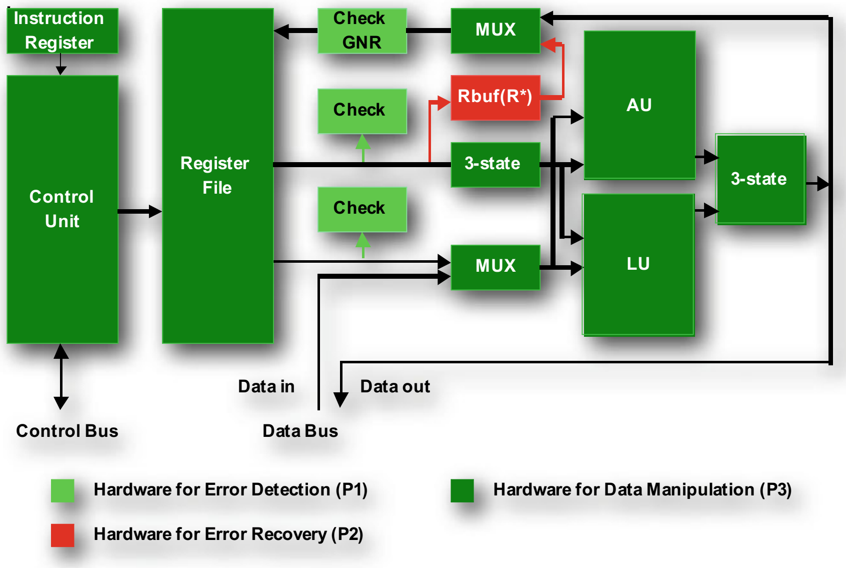


Fig. 17.14 REP structure

operands of instruction goes to register file and chosen operands go via three-state buffers down to either AU or LU. When all signals are arrived and timing diagram cycle is competed, information processed and arrived at the last right three-state buffer.

When confirmation from checking schemes arrives, i.e., no errors are detected, the right three-state buffer will release information to deliver either back to register file or through interfacing zone out of processor to the memory or bus.

The picture of prototype of ReP (Fig. 17.15) might be useful to analyze from the performance point of view and organization of embedded system.

Clear visible active zone and passive zone as well as board traces define performance of ReP. Interesting bit of this design is limitation of instructions complexity; only simple instruction to get or put data from and to outside world, store and read data from active zone, and process data inside active zone are allowed.

Thus, when we need fast calculation, processor is performing it with maximum frequency, without waiting of memory of IZ. In turn, clear read/write instructions limit slow down from interaction with external world.

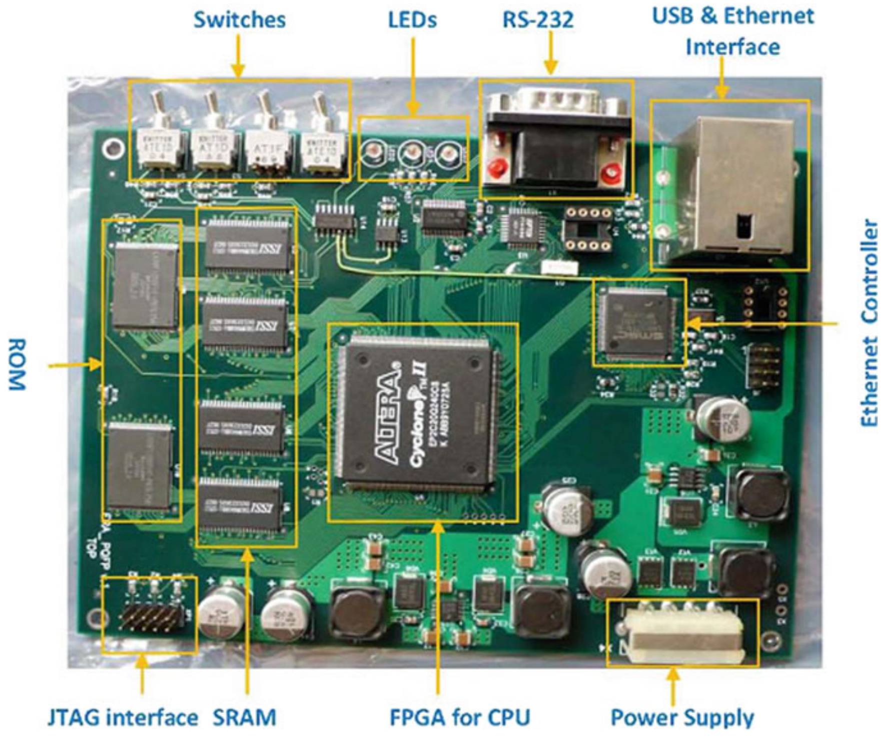


Fig. 17.15 Prototype of ReP

17.5.4 Standard Performance Tests

When we need to have a standard performance estimation, one might use SPECint.

SPECint is designed to measure processor speed and compare various hardware. There is a special company to look after this:

SPEC = Standard Performance Eval. Corp., their website is www.spec.org

What is widely known at the moment is SPECint2006.

SPECint2006 includes 12 tests (all in C/C++)—perlbench, gcc, bzip2, …, xalancbmk.

One can create performance comparison of embedded devices of processor overall 12 tests. Then, designed system might be 12-dimensional evaluation of the known system. Therefore, application segment relative performance might be known. For more details about standard performance, see

http://www.spec.org/cpu2006/results/cint2006.html.

17.5.5 Real Processor Hardware Comparison

The performance of processor is reduced if instruction uses intensively data traffics between memory and active zone. Today, a top-of-the-line x86 microprocessor runs at 3–4 GHz, while the memory runs at 100–500 MHz.

The time to execute an instruction inside a CPU is almost negligible in contrast with access memory time. In other words, the performance of architecture depends on memory traffics.

Proposed recently by ITACS performance of ReP is evaluated by the comparisons with 80×88 architecture and MIPs' instruction architecture.

Simple instructions dominate this list and are responsible for 96% of the instructions executed. These percentages are the average of the five SPECint92 programs [7].

Various mixes of instruction cover different applications. However, we can evaluate the CPU performance based on the average case as illustrated in Table 17.1. Within this evaluation, we assume loading data and storing data into memory takes equal time.

The execution time comparison between 80×86 and ReP is shown in Table 17.2.

MIPs dynamic instruction mix for five SPECint2000 programs (gap, gcc, gzip, mcf, and perl) is presented in [7]. The execution time comparison between MIPs and ReP is shown in Table 17.3.

Table 17.4 compares the performance of MIPs and ReP.

Modern embedded systems need efficient cost–performance ratio and very often minimize power consumption. One of the design options is as it was mentioned above the design as much as possible and therefore reduce the hardware overheads.

Table 17.1 Comparative ratio of memory use for ReP and 80 × 86

No.	80 × 86 instruction		ReP instructions		Integer average (% total executed) (%)
	Instructions	Memory traffic	Instructions	Memory traffic	
1	Load	2L	LD Ri, Rj	2L/2	22
2	Conditional branch	1L	XOR Mask, Rj	1L/2	20
			CBR Ri, Rj	1L/2	
3	Compare	1L	CND, Ri, Rj	1L/2	16
4	Store	1L, 1S	ST Ri, Rj	(1L + 1S)/2	12
5	Add	1L	ADD Ri, Rj	1L/2	8
6	And	1L	AND Ri, Rj	1L/2	6
7	Sub	1L	SUB Ri, Rj	1L/2	5
8	Move	1L	MV Ri, Rj	1L/2	4
9	Call	1L	MV Ri, Rj	1L/2	1
			CBR Ri, Rj	1L/2	
10	Return	1L	MV Ri, Rj	1L/2	1
			CBR Ri, Rj	1L/2	
Total					96

Table 17.2 Comparison of execution time for 80 × 86 and ReP

Processor and instruction mix	Ratio
80 × 86 22%* 2L+20%*1L+16*1L +12*2L +8*1L +6*1L +5*1L +4*1L +2*1L	=1.66
ReP 22%* L+20%*1L+(16*1L +12*2L +8*1L +6*1L +5*1L +4*1L)/2 +2*1L	

Table 17.3 Comparative ratio of memory use for MIPs and ReP

MIPs instruction		ReP instructions		Integer average (%)
Instruction	Memory traffic	Instruction	Memory traffic	
Load	2L	LOAD	2L/2	26
Add	1L	ADD	1L/2	19
Cond branch	1L	Compare	1L/2	12
		XOR	1L/2	
		CBR	1L/2	
Store	1L, 1S	STORE	(1L+1S)/2	10
Or	1L	OR	1L/2	9
Compare	1L	Compare	1L/2	5
And	1L	AND	1L/2	4
Sub	1L	SUB	1L/2	3
Xor	1L	XOR	1L/2	3

(continued)

Table 17.3 (continued)

MIPs instruction		ReP instructions		Integer average (%)
Instruction	Memory traffic	Instruction	Memory traffic	
Load imm	1L	LOAD	2L/2	2
Shift	1L	SHIFT	1L/2	2
Cond move	1L	COMPARE	1L/2	1
		XOR	1L/2	
		CBR	1L/2	
		MOVE	1L/2	
Jump	1L	MV	1L/2	1
		CBR	1L/2	
Call	1L	MV	1L/2	1
		CBR	1L/2	
Return	1L	MV	1L/2	1
		CBR	1L/2	
Total				99

Table 17.4 Comparative performance of MIPs and ReP

Processor and instruction mix	Ratio
MIPs 26%* 2L+19%*1L+12*1L +10*2L +9*1L +5*1L +4*1L +6*1L +8*1L	=1.67
ReP 26%* L +(19*1L +36L +10*2L +18*1L +6*1L)/2 +5*1L	

Thus, complex instructions of most processors for embedded systems are implemented by a sequence of simple instructions. For embedded systems, there is therefore developed set of performance evaluation, so-called embedded benchmarks.

17.5.6 Embedded Benchmarks

Benchmarks for embedded computing systems vary accordingly applications due to RT, HRT, PW, and CW performance requirements where RT stands for real time, HRT means hard real time, PW is power-wise and CW is cost-wise performance requirements, respectively.

That is why material presented above is important. There is one a bit more known benchmark developed by Embedded Microprocessor Benchmark Consortium (EEMBC). It has five categories: automotive/industrial, consumer, networking, office automation, and telecommunications.

Selecting Table 17.5 what this benchmark is (taken from Hennessy book recommended earlier), full site consists of 34 kernels in five classes.

Table 17.5 EEMBC benchmark suite

Benchmark type	This type	Example benchmark
Automotive/ industrial	16	Six microbenchmarks (arithmetic operations, pointer chasing, memory performance, matrix arithmetic, table lookup, bit manipulation), five automobile control benchmarks, and five filters or FFT benchmarks
Consumer	5	Five multimedia benchmarks (JPEG compress/decompress, filtering, and RGB conversions)
Networking	3	Shortest-path calculation, IP routing, and packet flow operations
Office automation	4	Graphics and text benchmarks (Bezier curve calculation, dithering, image rotation, text processing)
Telecommunication	6	Filtering and DSP benchmarks (autocorrelation, FFT, decoder, and encoder)

17.6 Relative Performance Gain—Amdahl's "Law"

Relative gain in performance usually called "Amdahl's law". Well, law in terms of science, not society, is "a regularity in the material world" (Shorter Oxford English Dictionary, 6e, Vol 1). Thus, naming simple proportion of performance after improvement P_{ai} with performance before improvement P_{bi} is, to put it politely, too ambitious.

$$Speedup = \frac{P_{ai}}{P_{bi}} \tag{17.6}$$

But this proportion is useful to evaluate success of the modification of processor structure after re-iterative design. What is interesting here is that arithmetic expectation of linear growth of performance by improving element performance (Figs. 17.1 and 17.2) has nothing near to the real situation.

17.6.1 Distributed Computing

In the late 1960s, an idea for the parallelization of computer program using distributed computing paradigm instead of single-processor scheme was proposed [8].

It was declared that parallelization of tasks and programs and use of available distributed hardware for support of parallel execution is the most feasible way to boost system performance.

Later, Sun [9] introduced "system fallacies" of distributed computing (Table 17.6). Omitting topologic factors and paying attention to Fallacy 2, 3, and 7, we discover that these fallacies fit into the area of parallel, closely connected computers with multiprocessors—in fact, all modern computers.

Table 17.6 Sun fallacies of distributed computing

1	The network (distributed system) is reliable
2	Latency is zero
3	Bandwidth is infinite
4	The network is secure
5	Topology doesn't change
6	There is one administrator
7	Transport cost is zero
8	The network is homogeneous

If we look harder, these fallacies might be not strong enough and some of the declared features described became obsolete.

Besides, again, when definition includes eight other elements that are not connected or have vague relation to each other, it seems odd or at least inconsistent.

If we follow Sun definition, we are not including Internet into the distributed computing even as a supportive hardware infrastructure. Anyway, we've proposed our own définition of distributed computing:

Definition 2 Distributed computing is a paradigm that assumes an execution of functionally connected tasks as a single process over distributed media and resources.

Clearly, a joint collaborative work of thousands of processors at once might bring substantial profit for both loosely connected tasks (when they share HW resources, but not logically connected, such as Google cluster), or closely tight models that include of several thousands of DE.

But in the second case, it is much harder to get the gain from distributed computing, and it is not a surprise.

Amdahl described drawbacks of distributed computing in the late 1960s [8], highlighting that even small parts of a program must be parallelized to reach their full potential. This way linear growth of speedup is not possible at all.

In other words, if 1 is a length of a sequential program and we have managed to parallelize p fraction of it, then sequential part is shrinking down to 1 − p, while parallel part requires p/n time where n stands for number of processors, (3) and Fig. 17.16.

$$S = \frac{1}{1 - p + p/n} \tag{17.7}$$

17.6.2 Real Performance and Amdahl "Law"

The proportion Eq. 17.7 is useful to evaluate a success of the modification of processor structure in re-iterative design. What is interesting here is that the expectation of linear growth of performance by improving element performance (Figs. 17.1 and 17.2) has nothing near to the real situation.

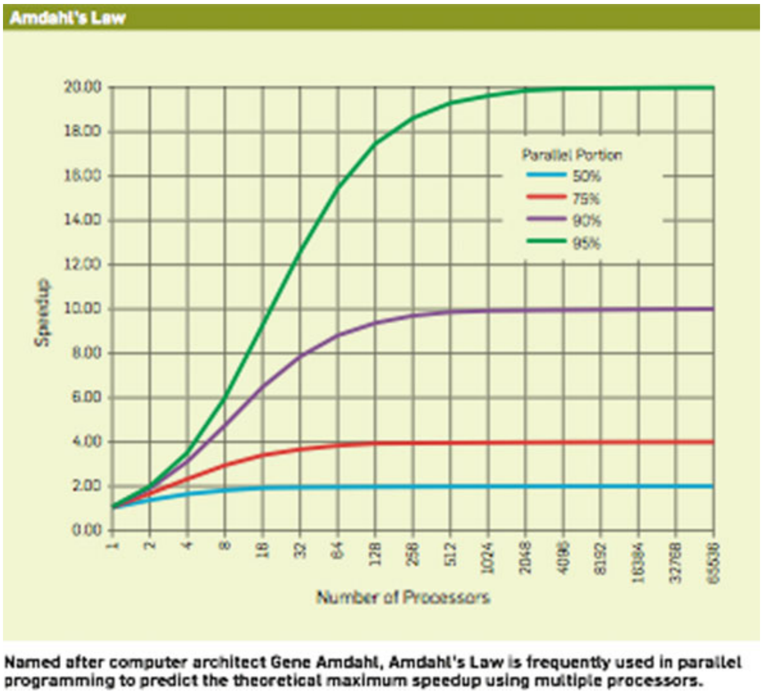


Fig. 17.16 System speedup by Amdahl [8]

It means that if we make super parallel execution of 80% of a program, we still have to complete another 20% sequentially. The number of speedups versus number of processors as a family of functions is presented in Fig. 17.16 taken from [8]

17.6.3 A Fine-Tuning of Parallel Speedup Model

The theory behind computational work in parallel has some limitations that reduce the advantages of parallelization. Usually, the goal in large-scale computation is to get as much work done as possible in the shortest time within the budget.

Furthermore, the system can be considered good and well-designed when it is able to get a big job done in less time, or a bigger job done in the same amount of time without any problem; in other words, a system should be a scalable.

Therefore, the power of a computational system can be represented as the amount of computational work done, divided by the total time it takes to do it. It is important to emphasize that usually the aim is to increase power per unit cost, or more importantly nowadays, cost–benefit, and in this regard physics and economics conspire to limit the raw power of individual single-processor systems available to perform any particular piece.

It is agreed within the research community that the cost–benefit scaling of increasingly power single-processor systems is usually nonlinear and very poor. For instance, one processor that is twice as fast might cost four times as much, yielding only half the cost–benefit per pound.

Physics sets its own limit as well—a so-called "thermal barrier" [5]—an amount of heat that material is capable to dissipate is limited making endless increase of frequency of operation impossible.

These two arguments are usually applied to justify alternative solutions and development of parallel designs. There are some drawbacks though, as Amdahl pointed out, and they are serious.

Let us rewrite Amdahl ratio in terms of time: T(N) will be the time necessary to finish the task on N processors. The speedup S(N) is expressed by the ratio (Eq. 17.8):

$$S(N) = \frac{T(1)}{T(N)} = \frac{Ts + Tp}{Ts + Tp/N} \tag{17.8}$$

In many cases, the time T(1) possesses, as represented above, both the serial part Ts and the parallelable part Tp.

Unfortunately, Amdahl ratio ignores a role of runtime system tasks (see first section of this chapter) that must be considered when a parallel execution is assumed.

A more detailed analysis of parallel speedup would include two more parameters of interest, namely,

- Ts—the original single-processor serial time;
- Tis—the average additional serial time spent performing, for example, inter-processor communication (IPCs), see Fig. 17.1, where it is introduced as EIZ, setup, and so forth in parallelized tasks. It is important to note that this time can depend on N in a variety of ways; nonetheless, the simplest assumption is that each system has to spend this much time one after the other, so that the additional serial time is, for example, N*Tis;
- Tp—the original single-processor parallelable time;
- Tip—the average additional time spent by each processor performing just the setup and work that it does in parallel; this may as well include idle times, which is also very important and should be accounted for separately.

The most important element that contributes to Tis is the time required for communication between the parallel subtasks. This communication time is always there—even in the simplest parallel models where identical jobs are farmed out and run in parallel on a cluster of networked computers, the remote jobs must begin and be controlled with message passing over the system.

In systems with more complex jobs, partial results developed on each CPU may have to be sent to all other CPUs in the distributed computing system for the calculation to proceed, which can be very costly in scaled time. The (average)

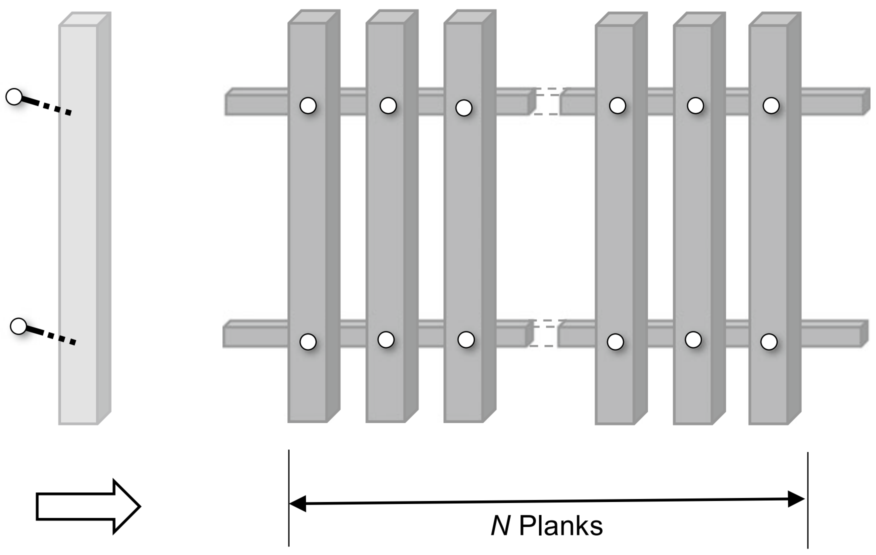


Fig. 17.17 Fence model of processing

additional serial time (Tis) plays an extremely important role in defining the speedup scaling of a given calculation.

Most computer systems process information sequentially. Lines of code in a computer program get translated into assembly language by the compiler, and the latter gets decoded into microcode in the processor. Everything and every step along the way is done sequentially. For example, a flowchart processing usually includes multiplication or comparison of two digits; it starts with the first digit, and then the second digit is introduced and the working register is set to 0.

To explain what is real and what is not and why Amdahl rule is mostly misleading, we have developed a simple model—so-called "fence making model", illustrated in Fig. 17.17 and following expert recommendations [10].

17.6.4 Parallel Versus Sequential: A Fence Model

Our task is to make a fence with N planks and two horizontal rails; each plank needs two nails and has to be "preprocessed". Two rails have to be placed at the assembling site. Each plank needs to be placed at site and finally nailed. We also need hammers and nails and sequence and instruction to operate.

Task requirements: number of planks N; number of rows—2. Each plank needs to be nailed half-way through before placement for final processing and assembling a fence.

There are two principally different options to make this fence:

(A) by distributing tasks;
(B) by making all tasks on site sequentially.

In case (A), distributing task scheme assumes the existence of agents–workers and distributers and their abilities to act:

- *N* workers for plank processing are available and ready;
- a distributor of the nails is in place;
- a distributer of the hammers is in place;
- a distributer of the planks is in place;
- a distributer of rails is in place;
- a collector of the fence segments initially is and placing the planks;
- nailing the planks at two rows are performed by workers;
- collecting the hammers is performed; and
- garbage collector is in place and completes the task execution.

Case (B), in turn, assumes that the same worker is doing all actions, like "a jack for all trade", has one hammer, bucket of nails, and does the following:

- takes nails;
- planks where they are;
- half-nail planks;
- places them on the rails;
- nails them all;
- place fence where necessary, collect garbage.

Let us consider the process of making the fence from *N* planks in more details for both cases, assuming that nails, hammers, planks, and rails are ready and placed in the local warehouse (storage and executed by "a system officer", while workers execute user task). Sequences are presented in Table 17.7.

Our task now is about giving elementary time slot t_e and constant coefficients equal for both variants of fence processing to prepare two variants of the fence completion as a sequence of steps for A and B cases. This will illustrate a gain from distribution of works.

We need to compare these cases as well as explain what is possible to prepare in preprocessing and what is possible only during operation. One might find useful to make a table of all works mentioned and using own experience and case estimate a concrete gain for concrete case.

Now we have to answer the following questions:

When distributed computing is efficient in comparison with sequential;

What impact system software makes on parallelization of task and efficiency of a system.

It is clear that *planks* are data, *nails* and *hammers* are programs to process data on site, *and distributer* is runtime system.

Table 17.7 Parallel versus sequential execution in more details

Parallel operation	Sequential operation
Distributor	*Distributor*
Gets pack of planks	Activate worker
Distribute planks	Check garbage left
Distribute rails	
Distribute nails	
Distribute hammers	
Distribute planks along rails	
Activate *N* workers start	
Collect hammers and left garbage	
Place two rails in assembling area	
Clean garbage	
Worker	*Worker*
Receive planks	Gets packs of planks
Receive nails	Gets buckers of nails
Receive hammer	Gets a hammer
Preprocess plank (two nails nailed half-way through)	Places (distribute) planks to the assembling area
Spread planks along rails (fine-tuning)	Places rails in assembling area
Nail plank (two nails) to the rails at the final assembling	Preprocess *N* planks (two nails per each)
Prepare to final assembling	Places (distribute) planks along the rails
	Nails *N* planks Assemble fence Clean garbage

Let us leave an arithmetic exercise with various values of parameters from job descriptions above to good master students.

Our estimation indicates that overheads of runtime system for distributed execution might achieve almost 60% of user task cost (time). We add in denominator of (5) a coefficient *k*, a relative value of system software overheads per user task (Eq. 17.9):

$$y = \frac{1}{(1-p)+k+\frac{p}{x}}, x = \{1, 2, \ldots, 10\}, p = \{0.85\}, k = \{0, 0.1, 0.4\} \qquad (17.9)$$

Following Eq. 17.9, the graph of Fig. 17.16 presents three curves in three colors: green, blue, and red k = 0, 0.1, 04, respectively. The top one stands for known "pure" Amdahl ratio (k = 0).

Figure 17.18 shows that for extremely good runtime system, one can double performance with 4 cores. It is still too optimistic statement, especially recalling Multics 85% and Window 65% of total workload time.

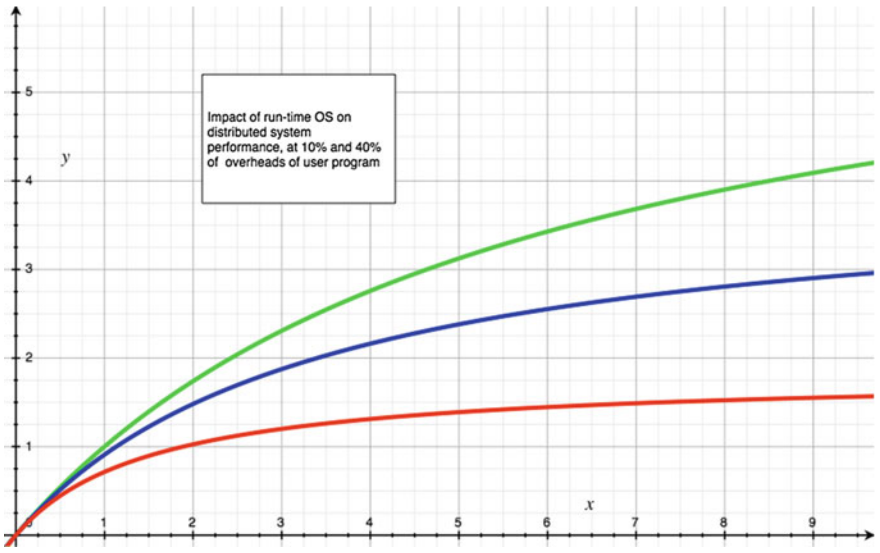


Fig. 17.18 System software role in distributed computing

17.7 Conclusion

- Hierarchy of performance models is proposed from the point of view of information processing.
- Shown that calculation of efficiency of computer system should include a role of user and system software as well as hardware.
- Model of hardware from the point of view of information processing is proposed.
- Calculation of performance of embedded system hardware is presented.
- Comparative study of proposed embedded architecture with most known architectures is presented.
- Role of complexity of instruction set on performance is briefly discussed.
- Brief description of what kind of benchmarks is used for performance evaluation of new and existing systems is presented.
- In details, using fence manufacture model shown that Amdahl Law is overoptimistic at an order of magnitude. It requires to consider the role of system software and algorithms ability of parallel computing.

References

1. Schagaev I (1990) Yet another classification of redundancy. In: IMEKO 7th symposium technical diagnostics, 17–19 Sept 1990, Helsinki, pp 485–491
2. Schagaev I (1990) Instruction sets and their role for computer architectures (in Russian). Electronics Publication
3. Sogomonyan ES, Schagaev IV (1988) Hardware and software of fail-safe computing systems. Automat I Telemech **2**:3–39
4. Schagaev I (2001) CASSA—concept of active system safety for aviation. In: IFAC automatic control in aerospace 2001 a proceedings of the 15th IFACS symposium Bologna/Forli, Italy, 2–7 September 2001
5. Blaeser L, Monkman S, Schagaev I (2014) Evolving systems. In: Resilient computer system design. Springer. ISBN 978-3-319-15069-7
6. Schagaev I. Active system control design of system resilience. https://doi.org/10.1007/978-3-319-46813-6. ISBN 978-3-319-46812-9
7. Hennesy J, Patterson D (2003) Computer architecture: a quantitative approach. Morgan Kaufmann Publishers Inc., San Francisco. ©2003 ISBN:1558607242
8. Amdahl GM (1967) Validity of the single processor approach to achieving large scale computing capabilities. In: Proceedings of the April 18–20, 1967, spring joint computer conference, AFIPS '67 (Spring), pp 483–485. https://doi.org/10.1145/1465482.1465560
9. https://blogs.oracle.com/jag/resource/Fallacies.html
10. www.doityourself.com/stry/buildwoodfences

Chapter 18
Distributed Systems: Maximizing Resilience

Igor Schagaev

Abstract We claim that system redundancy (natural or artificial, deliberately introduced) should be applied for the purposes of Performance, Reliability and Energy efficiency or “PRE-smartness”. A process of an algorithm of application of available redundancy for PRE-smartness is proposed showing that it is further extension generalized algorithm of fault tolerance, when property of fault detection is replaced as property of PRE-smartness. We present a system level implementation steps of PRE-smartness. Using ITACS LTD *forward* and *backward* tracing algorithms applied for distributed computer system we demonstrate that efficiency (reliability as a whole, especially availability; performance of detection and recovery) grow up to the level when real-time applications of distributed system grows substantially. Ans estimation of reliability gain is modeled and demonstrated suggesting a policy of implementation of PRE-smartness as a permanent on-going process required for the system successful functioning.

18.1 Introduction

Desperation management is one of the concepts we use in everyday life for effective handling of our conditions using available resources. Obviously, organization, hierarchy and structure of these resources within any system, not only in our body or brain (knowledge IS the resource!) can either help or complicate their utilization.

For networked computers or distribute computer systems further (DCS) as a whole we can consider as resources the following (we are not pretending to make a federal level of classification here):

– topology of the system
– elements—nodes (routers, switches etc.)
– rules—algorithms of behavior control for of the system as a whole
– routing algorithms
– network protocols

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_18

Using available resources efficiently is not easy: our designs are limited by market domination of existing systems, technological limitations and created sometimes artificially demand of, frankly speaking, completely non-necessary applications. At the same time, we have to ask ourselves:

- *What else is possible to add in our systems? and*
- *How to address real problems to assist human beings and living using technologies?*

Considering that DCS consists of over 3 billion computers and dozens of billions of controllers, all connected, the challenge in handling this world for good is astronomical. Surprisingly, some hints how to address these challenges were already a subject of R&D efforts of scientists from ETH Zurich and IT-ACS Ltd, their papers [1, 2], books [3–5], interviews [6], and keynote speeches (the latest is: [7]). In short, we can summarize challenges as combination of essential non-functional requirements (NFR) for the DCS and stand-alone future CS in terms of as Performance-, Reliability- and Energy-efficiency.

Thus, widening application area for ICT (information and computer technology) force us to think about computers, distributed computer systems and structures from a perspective of resources, available redundancy, then see their applications in terms of improvements and efficiency of a system as declared in [1–7], considering three strongly dependent requirements:

- performance
- reliability
- energy efficiency (power consumption)

When the system is considered to be used by various applications which have various demands and these demands also vary from application to application we simply must think about design of the system with capabilities of trading available resources for different purposes. Trading resources here means that, for example, if applications requested maximum performance then the system should use its resources for that and even borrow from resources previously allocated for reliability.

There is no doubt that implementation of these non-functional requirements needs some redundancy in the software and hardware [3–5]. Note that redundancy is already a property, and its efficient use is a goal to satisfy mentioned NFR of ICT, primarily in DCS domain. It is worth to mention here also, that handling redundancy for various purposes, addressing various and changeable requirements becomes actually a process.

Since late 60's up to early 80's of last century researchers were questioned themselves on how to design computers systems for the future, making them reliable, resilient, power efficient and powerful. 50 years on and we still have the same questions. The discussion now is more focused on properties and features required for next generation of computer systems, especially distributed computer systems (DCS), that includes networking and clustered computers.

Thus: performance, reliability, and low power consumption as NFR become altogether a serious challenge. By applying of DCS redundancy (of both types natural and deliberately introduced), serving a property of reconfigurability these challenges are resolvable.

The Internet expanded enormously and still growing in terms of data transfer and various types of data flows, to name a few: emails, voice and video. The referred applications are not safety critical and not even really hard real-time ones. Nowadays, computer and networks are considered through so called Quality of Service (QoS) requiremens and try to develop mechanisms to provide satisfactory communications for different combination or sub-domains of QoS. This is actually completely and perfectly wrong. According to various sources, more than four billions of IP addresses are assigned permanently, this number which is growing exponentially puts a lot of strain in the internetwork (Internet). These resources simply must be used and exploited for good and be flexible, not fixed. Thus, a *reconfigurability* of DCS should be used with intelligence, smartness, and flexibility for all mentioned PRE-requirements.

18.2 Reconfigurability of DCS

Modern communication networks are formed using wired or wireless media, using distinct signal carriers (fiber, copper, and air). Problems arise due to the complexity of data and its requirements with regards to timely data delivery and integrity over a very complex interconnected networks (Internet) [8, 9].

Computer networking encompasses all zones of information processing. This capability makes them faster and technologically sound. Note that, the zones of any computer architecture or computer system differ semantically in terms of information processing, namely active, passive and interfacing zones. Below is presented a definition for each of these zones [1–5]:

(a) Active zone—this is the zone where the information is transformed and is known at the moment in the form of CISC (complex instruction set computers), RISC (reduced instruction set computers), SIMD (single instruction multiple data), and MIMD (multiple instruction multiple data) [10].
(b) Passive zones—this is zone is known in the form of dynamic and static memory, such as DDR SDR and flash memory.
(c) Interfacing zone—this zone is in charge of transferring data from one zone to another.

Interfacing zone for DCS computers becomes overdeveloped complex and inefficient. Problems we are facing today in terms of networking are caused by the fact that historically computers technology was not considering potential work within the DCS framework.

Therefore, effective ways and procedures to deal with DCS in both levels: at the level of element—all zones introduced, and at the level of system. Solutions should be for sure scale-independent as DCS growth now in fact is uncontrollable. Dealing with all zones and dynamically applied NFR of PRE-smartness might provide a smooth exchange or transfer of information along the DCS as a whole. This is crucial, as our lives became too dependent on the communication devices and networks.

Distributed computing trend is shifting toward distributed databases, this approach which uses distributable large-scale tasks exploiting connected computer systems (networks) is called cloud computing [8, 9].

DCS to be robust, resilient and effective requires tuning or optimization of network topology and its backbones—nodes—read computers as elements of this network. These tuning should be performed at the application software, system software, and topology level. In addition, the computer′s internal structure should be set the most efficient way to execute requested operation and supported dynamically during operation.

There was no substantial progress in this area, instead, there were a considerable investment for "Cloud Computing", "Internet of Things" which do not address the problems of DCS organization, on the contrary.

Real-time, safety critical, active control systems, health monitoring, etc., all require special attention in terms of investment and more research to improve (or change) hardware, system software, application software, system and user data with aim to make billions of connected computer systems applied more efficiently with increased reliability and performance [1–4].

All real-time safety-critical applications (active control systems, military, health monitoring, air-traffic control, banking, etc.) should merge and exploit the advantages offered by DCS.

Therefore, to maximize advantage of DCS on should either to:

- Modify the existing networking schemes to match challenging areas, or
- Build a special purposes safety-critical system for common applications and make them as a key part of a network.

Otherwise, the DCS domain and market will be organized in different clusters and clustering is leading to industry segmentation [8–10].

The most important point here is:

- The network should have properties to absorb and support merging process; this way market clustering and technology segmentation can be avoided.
- Network unique feature is redundancy at every level of implementation. It should be exploited efficiently. Efficiency, as always, requires some analysis considering networks reconfiguration for various purposes, like for instance, performance enhancement, reliability or power efficiency.

18.2.1 Redundancy of Distributed System: A Description

Important categories to describe redundancy are: *structure, information* and *time* [11]. According to [5], a structural redundancy can be measured using graph or graph-logic notation, as represented below:

$$dS : \langle dV, dE \rangle. \tag{18.1}$$

where dS represent structural redundancy, whereas dV and dE represents extra vertices and edges added in the structure to aid in the implementation of reconfigurability. Therefore, PRE-goal can measure quantitatively as redundancy using the following expression:

$$dR = \langle dT, dS, dI \rangle \tag{18.2}$$

The cost associated to each of any redundancy type used and defining the steps of an algorithm implemented with aim to improve performance even in its most simplified version as depicted by Fig. 18.1. Steps of algorithm A-G may be

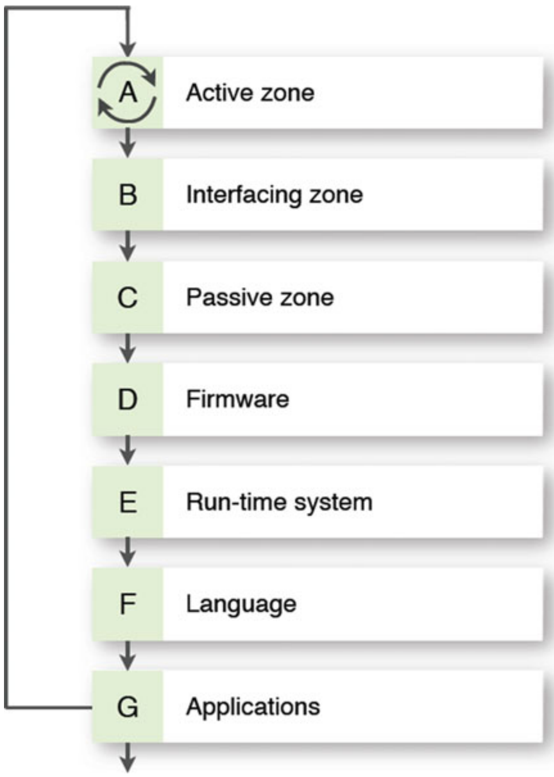


Fig. 18.1 Performance gain steps

associated with application of redundancy types to improve (or achieve) performance of Active zone, Passive zone and Interfacing zone of network computer architecture, using different redundancy types Table 18.1.

Using this framework, one might quantify the impact on each solution along the redundancy type used. The similar scheme might be applied for the improvement efficiency of energy consumption. Note here that when system software based on Java or any other interpretative language will always consume more volume of hardware to store and execute (interpretation of text and execution), and, consume more energy.

Two main reasons for this: it requires more hardware (memory) and more time for processing. It is possible to save more energy by special purpose-built system software including run time systems and language support [1–7]. Also, using voluntarily advances in memory technology—such as flash-based memory adds extra energy waste as activation of one memory cell in flash is equivalent to applying power for bulk of 64 K.

On the top of that, all previously proved solutions for reliability of hardware goes down to drain as Hamming code and models of single bit faults are no longer valid as damaged area of hardware after the same alpha particle impact exceeds 64 K bit [3–5]. Here again, the principles of PRE-smart design must be applied to DCS as a whole, and, as mentioned previously, redundancy and reconfigurability must be implemented wisely [1–4].

Next generation of DCS should be PRE-smart systems with redundancy and reconfiguration features embedded for performance, reliability and energy-wise purposes. This approach would reduce market segmentation for computers drastically and networks as a whole.

Table 18.1 Redundancy types to gain performance

Redundancy: Hardware (HW), Software (SW)					
HW(i)	**HW(s)**	**HW(t)**	**SW(i)**	**SW(s)**	**SW(t)**
	▨				
			▨		
					▨
		▨			
▨					
				▨	
		▨			
					▨

18.3 Resilience and Recoverability in Networked System

Previous section focused on clarification of PRE-smart design for computer systems, the same approach can be applied to networks, and make the latter more suitable for real-time and safety-critical applications. First of all, is important to establish the difference between a computer system and Distributed Computer System (DCS), and this can be summarized in the following [3–5]:

(a) *Redundancy in networks is already a fact*
(b) *Latency of thread impact for network cannot be avoided*
(c) *The propagation of thread impact for network is flooding-like*

Clear that DCS to be PRE-smart and fit new requirements of RT and safety-critical applications and should deal with (b) and point (c) above. Two concept or principles here might be useful, they are: ASAP and ALAP: stop a threat as soon as possible (ASAP), and handle it as local as possible (ALAP). Also, as it is described in full details in [3–5] malfunctions should be handled hierarchically, from hardware up to system level support of software recovery and system, reconfiguration.

In turn, reconfiguration of hardware in DCS is essential when permanent fault took place. Again, ALAP concept should prevail over other design solutions as wasting of hardware resources in volume is not feasible. All these arguments and analysis are published in two books [2–4] and here we just summarize elements that fit the purpose of making DCS fit for new applications.

Thinking about point (a) above we can see that *Recoverability* can be achieved in DCS applying scheme of application redundancy (Fig. 18.1). Let us consider in a bit more details how it can be implemented in DCS. Figure 18.2 presents a hypothetic segment of network topology with incoming and internal connections. Incoming and out-coming edges are represented with arrows.

By looking at Fig. 18.2. one might conclude that structural redundancy of the topology is considerable and the application of the above topology for real-time and safety-critical applications is straightforward.

Note here that any thread appeared inside the segment might cause serious troubles because it might propagate very rapidly and cause irreversible damage to the elements or the structure of DCS as a whole.

Note that here threads signify faults which could be permanent or just malfunctions of software or hardware components [1–5].

Also, a recoverability for networks requires more efforts and extending of GAFT introduced in [12], and further developed and explained in [1–7, 8, 9]. These actions, specifically, are:

(a) *Find where threads propagate*
(b) *Make an estimation of the damages*
(c) *Stop propagation*
(d) *Find the source of the threads*

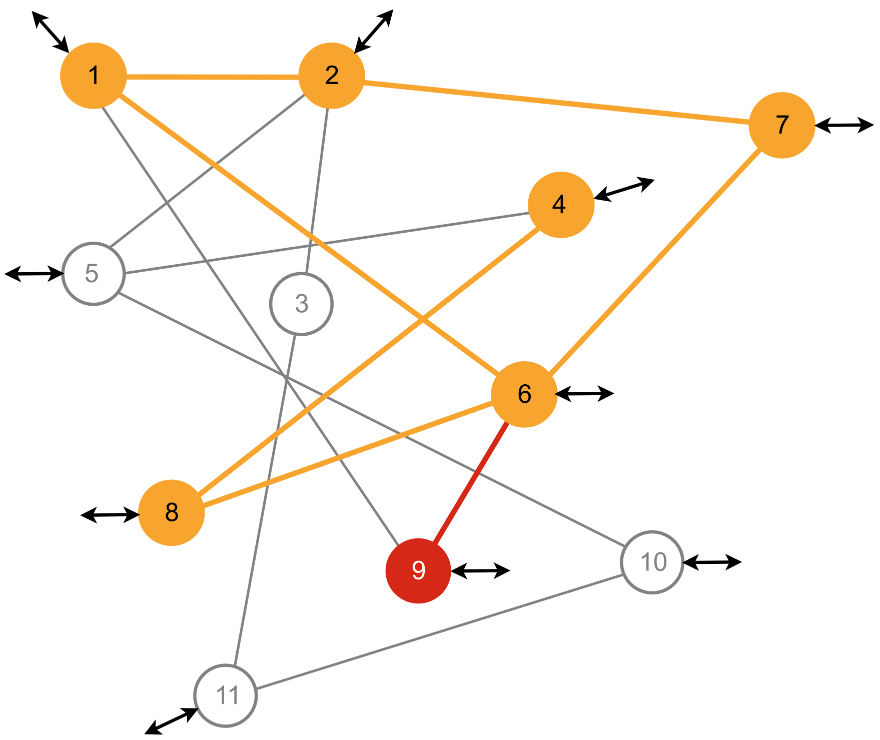


Fig. 18.2 Connected computer topology (a fragment)

(e) *Exclude or block the source of the threads*
(f) *Restore the best-fit configuration of hardware*
(g) *Restore best-fit of configuration of system software*
(h) *Restore best-fit of configuration of application software*

Moreover, making DCS fit for RT and safety-critical applications one should follow the same procedures established in the GAFT framework with a little bit of extra work:

(a) Analyze possible damages along with estimation of potential consequences for network topology as well as elements,
(b) The speed of thread propagation in the network is a factor that delineates the complexity of recovery procedures,
(c) Different segments of the network might have different importance for the network as a whole, for instance a gateway host and a simple internal router have different roles in the network.

Therefore, the consequences of damages resulting from threads might be different, occasionally substantial and exponential dangerous if the problem it is not treated in an effective manner. Restarting and segmental switching are solutions which are being applied so far but they did not comply with requirements for *real-time and safety-critical applications*.

Effective recovery is based on evaluating of the impact on the DCS and its elements, applying a concept ASAP and ALAP (better from the design phase of DCS). For DCS and its model we analyze the interdependencies of elements in terms of thread propagation using a graph Fig. 18.2 where the elements (routers, switches) are the vertices and edges are the links that connect the vertices.

The strength of the dependence between the vertexes is defined by the thickness of the edges. It is also important to stress that dependencies among vertexes are not symmetrical, this means that vertex 10 might have higher impact on vertex 7 than vertex 7 might have on 10. In this case, dependencies in terms of problem propagation might be presented as square non-symmetrical matrix where the indices represent the vertices and the dependencies are represented by contents, this is illustrated [4, 5, 8, 9] in Fig. 18.2.

The propagation of thread in the network can be defined as a vector P of predicates $\{p_i\}$ that represents the condition for each vertex [4]:

$$P = \{p_1(m_1(v_1(d_1(t)))), p_2(m_2(v_2(d_2(t)))), \ldots, p_k(m_k(v_k(d_k(t))))\} \tag{18.3}$$

where $m_1 \ldots m_k$ represents the models of vertexes in terms of vulnerability to thread, $v_1 \ldots v_k$ are vertices and $d_1 \ldots d_k$ are data related to each vertex conditions. This data can be gathered using various methods such as testing, and online-checking.

For networked systems it is important to evaluate flood-like thread propagation, for instance all the vertexes to the initial point, which can be named vertex 1 need to consider adjacency with 2, 6, 9th vertexes, vertex 11 need to consider adjacency to vertex 3 and 10th and so on. The initiation of recovery process might have various reasons, but it becomes an essential part of extended GAFT [1–7].

18.3.1 Implementation Steps

The framework assumes two algorithms involved in the recoverability procedures of connected computer systems (networked systems), namely *Forwarding tracing* and *Backward tracing*. How this works? When a thread is identified, via symptoms noted through changes in the behavior of elements, the tracing algorithms searches the Dependency Matrix related to the threat propagation through the system with the aim to identify the consequences or the problems, starting from the vertex where the thread was primarily identified [1–5, 10].

Example of thread dependency graph in the matrix form is presented by Table 18.2.

Executing the tracing forwarding algorithm involves the calculation of a cumulative probability along all the possible paths until a termination threshold ε is reached. This threshold is defined empirically and should be considered as a constant for a particular network configuration [3, 4, 10].

Table 18.2 Thread dependency for DCS Fig. 18.1

	1	2	3	4	5	6	7	8	9	10	11
1	P_{11}	P_{12}				P_{16}			P_{19}		
2	P_{21}		P_{23}		P_{25}		P_{27}				
3		P_{32}									$P_{3,11}$
4					P_{45}			P_{48}			
5		P_{52}		P_{54}						$P_{5,10}$	
6	P_{61}						P_{67}	P_{68}	P_{69}		
7		P_{72}				P_{76}					
8				P_{84}		P_{86}					
9	P_{91}					P_{96}					
10					$P_{10,5}$						$P_{10,11}$
11			$P_{11,3}$							$P_{11,10}$	

The other termination conditions with regards to search of thread propagation is to check all dependent vertexes. There is no doubt that only tracing of all elements provide a 100% cover for threat checking. Unfortunately, this termination condition is not viable as it becomes network scale-dependent. It is important to note that the probability matrix associated with Fig. 18.2 is not Markovian because the sum of probabilities on the edges at each node may not be equal to 1.

18.3.2 Treat Propagation: Tracing and Handling Algorithms

The threat propagation from one element to another through the edges, according to Tracing algorithm is defined as Π ($p_{i,\ j}$). Here, ($p_{i,\ j}$) represents the probability of threat propagation from node i to node j through the corresponding edge. Note that the treat might spread through a series of other nodes.

Therefore, when there are several paths leading from node i to node j all possible cumulative distributions are ranked and all the nodes along the paths are included in the group of suspected nodes.

The Fig. 18.3 presents Forward Tracing Algorithm where pseudocode demonstrates how it works. It starts from the vertex where the threat manifests, in this case from vertex i, and the impact is analyzed and evaluated by searching from d_1 to all directly or indirectly connect nodes.

The final result of the search is the "consequence" caused by the threat propagation and is presented in the form of a ranked list of the elements suspected to be affected.

As all the paths from one node to another are evaluated only the edge with the highest probability of is tracked at each node. Note that each node is only included once in any path to guarantee termination in a graph which has loops [3, 4].

The Forwarding Tracing algorithm does solve the problem of threat elimination from DCS. It is important to note that the manifestation of a threat in one vertex and the detection of this threat consequences across another vertexes has an arbitrary duration (the duration varies). Also, while consequences are finally spotted, the

```
PROCEDURE Tracing ($s$, $D(N)$ , $D_s$ , $\prod(p_{s,x}), xR_s$ )
Input: Dependency matrix $D(N)$ with $N$ elements of a weighted connected graph $G =< V, E >$
Input: The start node $s$
Output: The set of nodes $D_s$, where $\prod(p_{s,x}) > \epsilon$
Output: The highest probability $\prod(p_{s,j})$ of node $j$ reached by node $s$

$Q$; (* a priority queue sorted in descending order of the probability of nodes reached by $s$ *)
$L$; (* the set of already visited nodes, used to avoid tracing loops *)
Initialize ( $Q$ ); (* initialize nodes priority queue to the empty queue *)
FOR each node $v$ in $V$ DO $p_{s,v} \leftarrow \epsilon$; (* set default probability to $\epsilon$*)
    Insert( $Q$ , $v$ , $p_{s,v}$ ) (* initialize the priority queue *)
END;
$p_{s,s} \leftarrow 1$; Increase( $Q$ , $s$ , $p_{s,s}$ ) (* update priority of $s$ with $p_{s,s}$ *)
$D_s \leftarrow Empty$ (* presume all elements are safe *)
$L \leftarrow Empty$
FOR $i \leftarrow 0$ TO $N-1$ DO
    $a^* \leftarrow GetMax(Q)$ (* get the element with the highest priority *)
    IF $p_{i,a^*} > \epsilon$ THEN
        $D_s \leftarrow D_s \cup a^*$; $L \leftarrow L \cup a^*$; $\prod(p_{s,a^*}) \leftarrow p_{i,a^*}$
        FOR every node $a$ in $V - D_s - L$ that is adjacent to $a^*$ DO
            IF $p_{s,a^*} * D_{a^*,a} > p_{s,a}$ THEN
                $p_{s,a} \leftarrow p_{s,a^*} * D_{a^*,a}$;
                Increase( $Q$ , $a$ , $p_{s,a}$ )
            END
        END
    ELSE (* corresponds to $p_{i,a^*} < \epsilon$*)
        EXIT;
    END
END
```

Fig. 18.3 Forward tracing algorithm to determine area of thread propagation

threat propagation continues. Therefore, the role of forwarding tracing algorithm is to localize damages, and where is possible block the threat propagation.

So far we have not mentioned how to deal with the reason or source of threat the another algorithm was developed to perform Backward Tracing, Fig. 18.4. The latter performs a searching of the reasons (for example, where, when and why a threat appeared in DCS).

When Backward Tracing algorithm has completed the entire area of the damage caused by the threat became known and defined. This eases recovery and optimize the reconfiguration processes. This algorithm detects the source or reasons for the potential order of manifested threats and caused consequences for each vertex included. The threat propagation might be manifesting as inconsistency of data or behavior at another node [4, 5, 10].

In one sentence—It is essential to start the recovery process from the vertex where the threat appeared, so the treatment will start from the reasons not just the symptoms.

Backward Tracing of the dependency matrix enables us to identify a suspected element or elements.

```
PROCEDURE Backward Tracing ( $s$ , $R(N)$ , $S_s$ , $\prod(p_{s,x})$ , $xR_s$ )
Input: Recovery matrix $R(N)$ with $N$ elements of a weighted connected graph $G=<V,E>$
Input: The suspected node $s$
Output: The set of nodes $S_s$, where $\prod(p_{x,s}) > \epsilon$
Output: The highest probability $\prod(p_{j,s})$ of node $j$ reaching node $s$

$Q$; (* a priority queue sorted in descending order of the probability of nodes reaching $s$ *)
$L$; (* the set of already visited nodes, used to avoid tracing loops *)
Initialize ( $Q$ ); (* initialize nodes priority queue to the empty queue *)
FOR each node $v$ in $V$ DO
    $p_{v,s} \leftarrow \epsilon$; (* set default probability to $\epsilon$*)
    Insert ( $Q$ , $v$ , $p_{v,s}$ ) (* initialize the priority queue *)
END;
$p_{s,s} \leftarrow 1$; Increase( $Q$ , $s$ , $p_{s,s}$ ) (* update priority of $s$ with $p_{s,s}$ *)
$S_s \leftarrow Empty$ (* presume all elements are safe *)
$L \leftarrow Empty$
FOR $i \leftarrow 0$ TO $N-1$ DO
    $a^* \leftarrow GetMax(Q)$ (* get the element with the highest priority *)
    IF $p_{i,a^*} > \epsilon$ THEN
        $S_s \leftarrow S_s \cup a^*$; $L \leftarrow L \cup a^*$; $\prod(p_{a^*,s}) \leftarrow p_{a^*,i}$
        FOR every node $a$ in $V - S_s - L$ that is adjacent to $a^*$ DO
            IF $p_{a^*,s} * R_{a,a^*} > p_{a,s}$ THEN
                $p_{a,s} \leftarrow p_{a^*,s} * R_{a,a^*}$;
                Increase ( $Q$ , $a$ , $p_{a,s}$ )
            END
        END
    ELSE (* corresponds to $p_{a^*,i} < \epsilon$*)
        EXIT;
    END
END
```

Fig. 18.4 Backward tracing algorithm to determine the source of threat

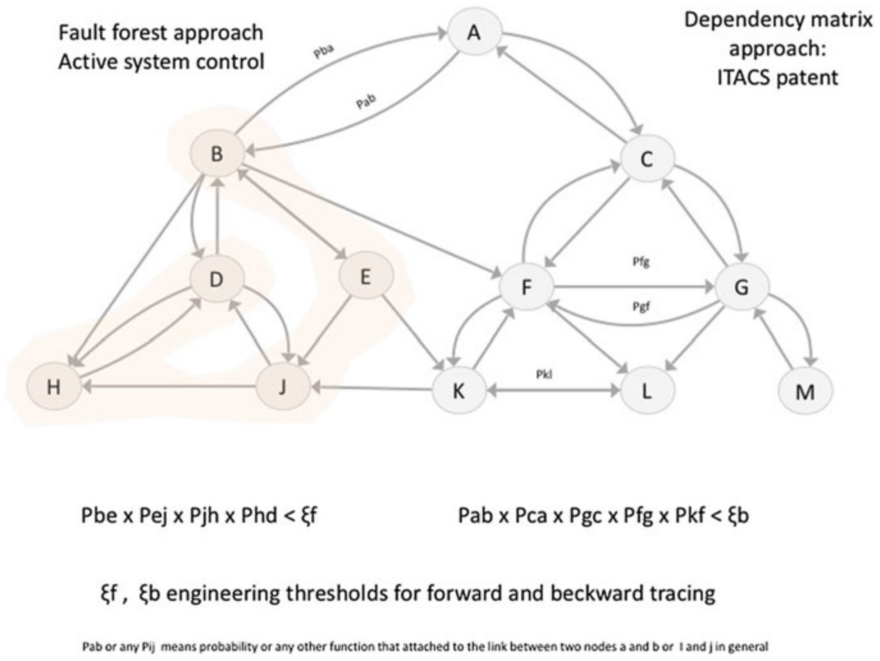

Fault forest approach
Active system control
Dependency matrix approach: ITACS patent
Pba
Pab
Pfg
Pgf
Pkl
A
B
C
D
E
F
G
H
J
K
L
M
Pbe x Pej x Pjh x Phd < ξf
Pab x Pca x Pgc x Pfg x Pkf < ξb
ξf , ξb engineering thresholds for forward and beckward tracing
Pab or any Pij means probability or any other function that attached to the link between two nodes a and b or I and j in general

Fig. 18.5 Visualization of forward tracing algorithm

When a DCS failure has detected, the only action known and applied so far is a system restart. This is a common practice in networked system monitoring. In contracts to it presented here Backward Tracing Algorithm enables the DCS recovery limited by the "area of damages".

A combination of both algorithms forms the area of damages enabling network management procedures substantially reduced the cost of recovery for DCS as a whole. Note that for FTA and BTA engineering thresholds ε_f and ε_b can be set differently and have different values.

Figures 18.5 and 18.6 visualize how both algorithms work, they were kindly given by IT-ACS Ltd. from the materials of the patent [10] and materials of IPO case against Russian Government attempt to endorse it [13].

Recovery procedures from a list of legitimate and relevant events should be applied automatically, or in highly sensitive cases under supervision of maintenance engineers. For reliability, security, and maintenance efficiency purposes it is also important to record and save the results of recovery procedures [4, 13].

The checking procedure of a threat on DCS might be activated by various reasons:

- A signal indicating that there is an inconsistency in data
- Detected deviation in system or element behavior in one or more vertexes.
- Maintenance team
- Maintenance automatic procedures

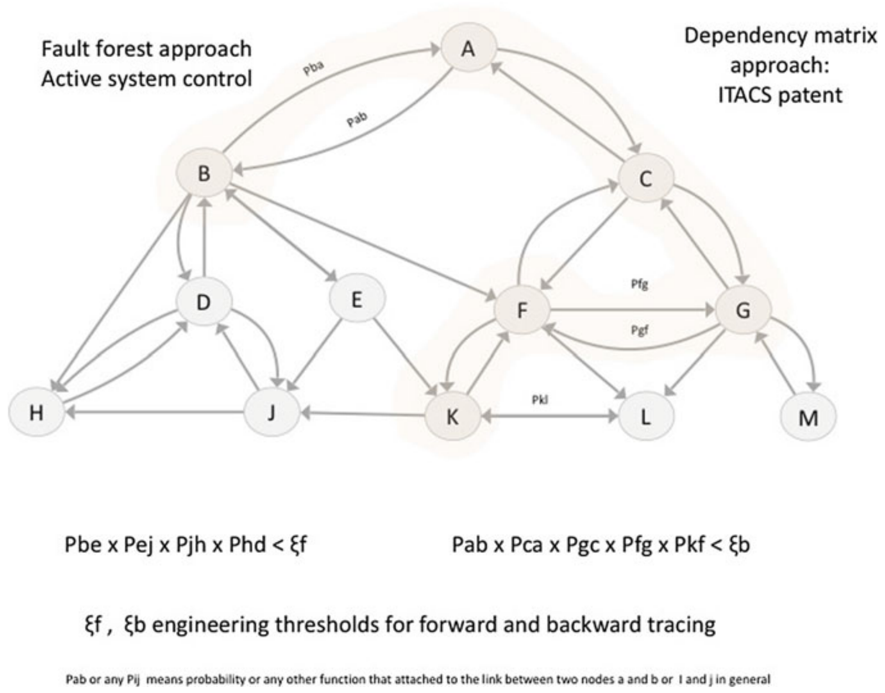

Fault forest approach
Active system control
Dependency matrix approach: ITACS patent
A
B
C
D
E
F
G
H
J
K
L
M
Pba
Pab
Pfg
Pgf
Pkl
Pbe x Pej x Pjh x Phd < ξf
Pab x Pca x Pgc x Pfg x Pkf < ξb
ξf , ξb engineering thresholds for forward and backward tracing
Pab or any Pij means probability or any other function that attached to the link between two nodes a and b or I and j in general

Fig. 18.6 Visualization of backward tracing algorithm

Therefore, when an element is included in a list of suspects, a sequence of well-oriented actions should be put in place to interpret the unusual behavior or event and prevent further propagation of the thread that can catastrophic damage to the whole system [4, 5]. This scheme of detection–reconfiguration-recovery is named Threat Monitor and operates on the dependency matrix of the DCS.

For the purpose of quality and integrity of networked systems—maintenance procedures should be organized in a way to check the condition at every vertex of the system, providing 100% coverage.

18.4 Recoverability Costs

Recoverability to work well in networked systems needs the deployment of many new processes, starting from management to online checking of the networked system conditions and application of algorithms of Forwarding and Backward Tracing. In the case of online checking, this is a real-time process running over a networked system's main elements and links, system software and application software [4, 13].

The goal of checking within networked system is to detect the degradation or changes in the behavior of the elements and, initialization, as a result, therefore, enabling the recovery of the suspected elements and subsequent conservation of the networked system with regards to its reliability, availability, and consequent extension of RT applicability as a whole [1, 3–5].

When full recovery cannot be achieved, preventive measures can reduce the level of danger, risk, and allows graceful degradation of services running on networked systems.

For networked systems the Process of Checking (PC) reliability and consequent availability degradation, and the execution of Forward and Backward Tracing are independent in principle. Therefore, they can be considered as sequential as well in parallel. The PC (Process of Checking) which can also be regarded as Process of Maintenance (PM), can be started any time when required, possible or convenient [4, 5].

The aim is to perform PC well in advance when current mission reliability of networked system is higher than threshold reliability defined and in such a manner that when critical applications are running the networked system does not reach the predefined threshold reliability. Note that the combination of processes of checking and the recovery have an impact on the reliability of a networked system as a whole. The gradient associated to the reliability change is a function of the quality of checking, success of recovery and the quality of maintenance [4, 5].

The Process of Maintenance should also include the execution of both (Forward and Backward) Tracing algorithms with harder conditions and reconfiguration of networked systems if necessary. According to [4, 5] it would be interesting to shift networked system in *real-time maintenance* with preventive measures against threats. This approach enables to increase mission reliability and availability of networked systems.

Recoverability could be measured using comparisons of standard networked system against a system with RT maintenance implementation. For the analysis and evaluation of the impact of recoverability implementation on networked system let us follow the notation on [4, 5] and assume:

(a) A constant failure rates
(b) Maintenance is not ideal and coverage is less than 100%
(c) Minimum acceptable reliability threshold is introduced as before.

There are other assumptions related to checking process that need to be considered. For instance, the mission reliability function with introduced recoverability and online checking for networked systems is grounded on the following assumptions:

(a) The coverage is not 100%. The coverage is $100\alpha\%$, $0 < \alpha < 1$, and this parameter is assumed to be constant during al preventive maintenance actions.
(b) Preventive maintenance should be instantaneous and might not have a negative impact on the networked system.
(c) The MR_0 which represents the threshold of acceptable mission reliability is provided.
(d) T_{PM} represents a function of several variables, such as α, λ and MR_0.

Therefore, the mission reliability for networked System can be calculated using the following expressions:

$$MR(t) = \alpha^n e^{-\lambda(t - \sum_{i=0}^{n} T_{PM}(i))}, \sum_{i=1}^{n} T_{PM(i)} < t \tag{18.4}$$

$$MR\left(\sum_{i=1}^{n} T_{PM}(i)\right) = MR_0 \tag{18.5}$$

The reliability curve derived from the above expression is shown in Fig. 18.7, if one considers that maintenance is performed when the networked system reaches the threshold of mission reliability, for instance when $MR(t) = MR_0$.

The solid curve represents the mission reliability curve, whereas the dashed line is the threshold defined by requirements of networked systems operations, and the dot-and-dash line indicates the ideal reliability state for a networked system. The latest system states are measured and analyzed when the checking period arrives. The confidence on the system grows and consequently the reliability curve if after each online- checking no faults are detected (the system is free of faults) [3–5].

Maintenance should always be carried out when the reliability of the networked system reaches the threshold, for that one can use existing network monitoring schemes and tools. The rate of mission reliability degradation is directly linked to the gain of recoverability implementation in networked systems. When there is lack of preventive maintenance for long periods of time the reliability of networked system reduces drastically to the lower threshold R_0 [3, 5, 14].

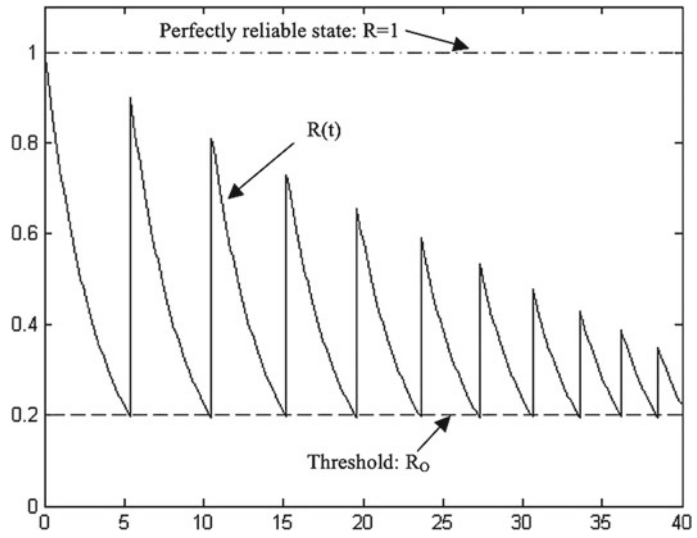


Fig. 18.7 Mission reliability for a CC system with incomplete coverage

18.4.1 Checking and Recovery: An Efficiency of Implementation

As mentioned previously, checking and recovery actions might be scheduled independently from the other processes inside the DCS; see Fig. 18.8.

In addition, Fig. 18.8 shows an example of mission reliability function for networked systems where periodic preventive maintenance is done but the coverage of checking is not complete.

Real-time online checking and recovery should be part of the monitoring process. As mentioned previously online checking is a process executed in real-time where the main elements in the system are checked in real-time. This includes hardware (vertices and edges) and software. The main goal of online checking is detection of changes in behavior or degradation, and if necessary recovery of the suspected element(s). This will certainly improve the system's mission reliability, performance, and availability.

Again, is worth to emphasize that is very important to carry out checking and preventive maintenance well in advance in order to provide the networked system with highest mission reliability in real-time. The mission reliability function for DCS and other networked systems with real-time maintenance needs to be calculated with realistic assumption about coverage of real-time maintenance is limited, so coverage is $\alpha 100$ as it was described in [3–5] and further elaborated in [14].

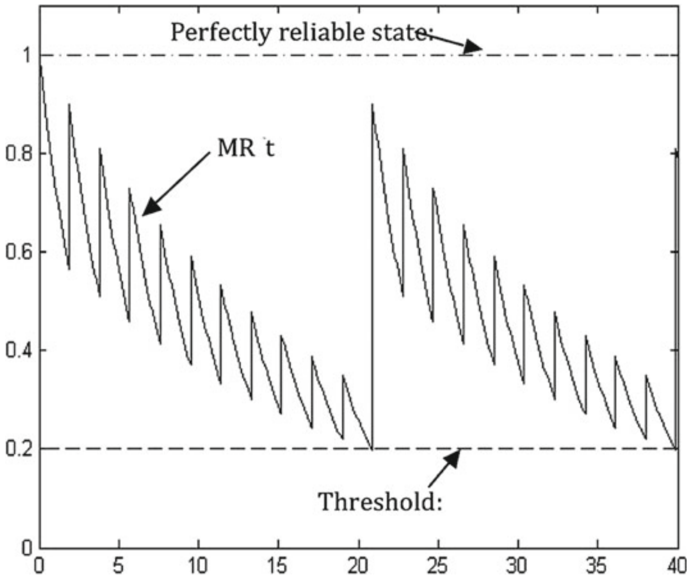


Fig. 18.8 Periodic maintenance initiated for DCS within complete coverage

18.5 Summary and Conclusions

Distributed systems can use existing resources much more efficiently.

We show that system natural or deliberately introduced redundancy can be used for reconfigurability.

Reconfigurability can serve for performance, reliability and energy-efficiency and if widely used be even traded for all three mentioned properties.

An extended generalized algorithm of threat handling using introduced resources is proposed and explained.

Distributed computer systems require own algorithms of threat handling that consider property of "distributiveness", two of them: forward and backward tracing ones are explained and evaluated.

Both algorithms are proven to be the most efficient to define damaged areas as soon as possible (ASAP) and as local as possible (ALAP).

A system availability applying described algorithms is estimated, showing great improvement of real-timeness.

References

1. Blaeser L, Monkman S, Schagaev I Vision on reconfigurable systems, chapter 10
2. Schagaev I, Monkman S (2013) Redundancy + Reconfigurability = Recoverability. Electronics 2:212–233. https://doi.org/10.3390/electronics2030212
3. Castano V, Schagaev I Resilient computer system design. https://doi.org/10.1007/978-3-319-15069-7
4. Schagaev I, Kaegi-Trachsel T Software design for resilient computer systems. https://doi.org/10.1007/978-3-319-29465-0
5. Schagaev I, Kirk B Active system control. https://doi.org/10.1007/978-3-319-29465-0
6. https://faculti.net/system-software-support-of-hardware-efficiency/
7. https://www.academia.edu/34010977/IAP_Symposium_2017
8. Leon-Garsia A et al Communication networks. McGraw Hill, ISBN-0-07-246352
9. https://www.bitpipe.com/tlist/Internet-Architecture-Board.html
10. Flynn MJ et al (1996) Parallel architectures. ACM Comput Surv 28(1)
11. Schagaev I (1990) Yet another approach to classification of redundancy. In: CIM IMEKO symposium 1990, Helsinki, pp 117–124
12. Schagaev I, Sogomonoyan E (1988) Hardware and software for a fault tolerant computing system. Automat Remote Control 49(2), Part 1, Pergamon Press
13. http://it-acs.co.uk/files/Patent%20breach%20failed.pdf
14. http://it-acs.co.uk/files/Grant_for_a_patent.PDF

Chapter 19
Distributed Systems: Resilience, Desperation

Igor Schagaev and Stephen Farrell

Abstract Major drawbacks of existing distributed systems, including networking are discussed. To cope with them we introduce a new design concept of performance-, reliability-, energy- smart system, applied for distributed system and networking. Our approach of distributed system handling was called *a desperation control*, which introduce and explain. It assumes introduction of new routing algorithms for routing and for package handling inside routers. We claim that state of a system as a whole as well as state of a package should be considered and treated in concert varying ways of package handlings. Shown that proposed approach at order of magnitude boosts performance and reliability of network making it suitable for real-time packages and widening applications for really important applications: existing technologies and billions of devices should serve and make people better. Using elements of graph theory, we analyze and compare proposed algorithms with known, explaining our advantages. Weighted costs of each segment of package path, combined with vector heuristic introduced and enriched by graph-logic extension for each routing table makes proposed concept the best so far from all known. Resilience of distributed computer systems as well as networking therefore becomes improved, extending application domain to the really important applications like safety critical, military, banking, health real-time monitoring, transport and similar.

19.1 Introduction

Computer system resilience we have introduced in [1]. But this book does not cover one important and even booming segment of computer systems—distributed computer systems and network computer systems. Clear, that having thousands of manufacturers and overcrowded society of applications to argue about unified hardware design is a kind of naïve. But following our mantra through all our works: ICT must be PRE-smart—read: Performance, Reliability and Energy efficient and smart. “Smartness” here means an ability to trade resources available for achieving requested functionality in spite of all odds: read resilience of the system is

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_19

considered as an essential requirement for both—stand alone and distributed computer system and, therefore and must be provided.

Distributed systems, networks, Internet declared as already resilient ones. Indeed—these systems can cope with traffic jamming, even recover, to some extent after subsystems collapse. Simple timeouts and refreshed tables of routing (every 200 ms) do the work.

Unfortunately, this solution cannot match requirements of new applications such as safety critical, real time, health monitoring and other applications with high demands of guarantee of service.

That is why we turned to an idea of *desperation management*—similar to human physiology actions. It is natural—if our applications require from ICT systems a certain flexibility this should be reflected in package handling and structure organization.

For this reason, we turn to analyze human desperation at first, seeking further how it might correspond with distributed system behavior.

19.2 On Desperation

In our life we instinctively define a desperation as:

A fear of impossibility to achieve wanted or desired (D1)

Desperation also is implicitly reflected in various descriptions of our desires. Regarding desires we are willing to know more to reduce fear and frustration:

- "knowing more worrying less"—every language has the same proverb or similar
- "better safe than sorry", etc.
- "I am happy to run extra mile for it"—means actions for reducing panic

The definition (D1) is general, but it is useful and even explicit for everyday life, social communication, literature and art. At the same time, for analytic purposes we should add that *desperation* should be measured and what is even better—quantified. Where to start? An analysis and further measure as we explained in [1–3] should be exercised using three following prime categories that applied for description of an object or phenomena:

- time (t)
- structure (s)
- information (i)

All three categories usually in research are represented by independent variables as described in elements of theory of redundancy elsewhere [4–6], since our papers in Automatic and Remote control about for fault tolerant computer systems design [1–3]. This might be considered as starting point, to see how to quantify desperation.

Then, from informal discussion of *desperation* as a word from the literature and art domain we will be able to define and apply it formally for analysis of complex technological objects like computers, distributed systems and Internet, to name a few.

In theory of redundancy mentioned and developed in [1–7] for the purpose of system improvement we also were considering an efficiency of system using three categories mentioned above: *time, structure* and *information*. The aim was as mentioned above is making systems smarter in terms of *performance*, *reliability* and *energy* (or cost), using available time, structure and information. This framework let us see and enables to making evolvable systems.

This was called PRE-smart [5, 7] system, and further thought through as evolvable systems [6, 7]. References presented indicate just starting point in this direction of research.

In principle, networking is nothing new in terms of computer science, because a foundation of distributed computing and distributed computer systems (DCS) were developed over 60 years ago [8–10]. Differences or advances in performance of links and nodes did not change logic and concepts of synchronization, a-synchronization and ordering of messaging in the network [9–12].

Queuing theory and works of Kleinrock were dealing with this since 20th mid-century [11]. At the same time, a desperation concept was not addressed as it should. The main reason for this, we guess, was an absence of real-time-ness requirements for packages and/or sequence of packages.

19.3 Desperation in Networking

In networking, when we deal with real time-ness of messaging we suggest to move from generic definitions above or using formulas with all three independent variables (*s, i, t*) down to addressing "time" as a prime parameter. The idea here is about seeking what information we can have or use to see measurable gain. Thus, for real time-ness of networking in terms of probabilities we might say:

Desperation is a probability of non-reaching a destination within scheduled time (D2)

Naturally, in DCS a desperation depends on the following parameters, Table 19.1.

Table 19.1 Parameters for desperation estimation

Name	Meaning
L	The length of a journey, given as distance between source and destination
T_{RT}	Time frame (or constraints) to be "within scheduled time"
T_{cost}	Current cost, or "local time" for the process,—in other words, how much time spent or which length is covered by process—at any particular moment

Here: ***L***: The "length of a journey" is a kind of estimated length of paths between source and destination

Note that it is not always known precisely. This is because, again, the structure of Internet is defined globally and hierarchically, while at the local level we are facing the fact that routing information is limited by the size of routing tables. Above all, principle design of the whole network as *asynchronous* system, with no decision core makes impossible to obtain the precise value of paths. Updating distances for each routing table executed by networking protocol that serves just this—refreshing periodically values of distances between nodes and distributing them to the relevant "customers"—routers.

Therefore, one of the first questions is about how we know the whole distance between source and destination. Considering network as a graph deciding about handling a package it is worth to have the whole distance as well as awareness about the size and structure of the whole network.

Further in special section we will consider some elements of graph theory that might be applied for implementation of desperation control.

$\boldsymbol{T_{RT}}$: Time frame is predefined time slot within which a package should or must reach a destination. We denote is as T_{RT}—time constraints ("real time-ness") for a chosen process;

Note that during a package journey we can treat T_{RT} not as a given limit or requirement but as a parameter that reduces its value along the package passing through network.

Then T_{RT} is *a function* of current time for package journey: $T_{RT}(t)$

$\boldsymbol{T_{cost}}$: Introduced "The current cost of the journey" is how much time we have spent so far. Clear, the more we waste time in traffic or queuing the less chances is left to reach the destination timely.

Naturally, when we perform well and pass through the main path having sufficient time reserve (in terms of T_{RT}) we should not worry much or be desperate. Thus:

$$Desperation = P\left(\sum_{i=1}^{n} d_i, T_{RT}, \sum_{j=1}^{n} d_j,\right) \tag{19.1}$$

Here the first sum represents the whole distance between source and destination;
T_{RT}—time constraints ("real time-ness") for a chosen process;
The second sum presents the current cost of journey for a process j.

19.3.1 How Desperation Grows

As long as journey of the package continues we are facing, amongst all others, two big questions to answer:

- *How desperation grows,* and
- *How to handle it*

When desperation property is introduced in DCS or any other system actually, other very interesting questions remain and still unanswered:

- *Are we desperate at the starting point?*
- *Is it enough to known only shortest distances and total time constraint introduced?*
- *Is it good to know inside the package the time constraint and how to deal with it?*
- *How desperate we are when a package time inequality—below—stands?*

Let us consider desperation graphically, Fig. 19.1 illustrates two cases of successful and late package. Constraints are introduced by Yc—time given for a package, while Xt is time that package spent in the network.

Clear, that for the case ***a*** case when a distance covered by a package to its destination took more time than permitted by constraint Yc the package is late;

Thus for the package ***a*** a distance covered is shorter than required within constraints and this package fail to be delivered in real time, failing real-time-ness property or requirement.

In turn, for the package ***b*** a distance covered and time spent is less than limit and this package complies the time constrains;

Normal delivery of a package is when between a current distance of a packages and time constraint given we can be either below or match a linear dependency such as

$$y = kx, \tag{19.2}$$

where kx at Xt satisfies:

$$\mathrm{kXt} < \mathrm{Yc} \tag{19.3}$$

Yc here is a defined time limit for a package, Xt a distance to a destination, k is a coefficient such as

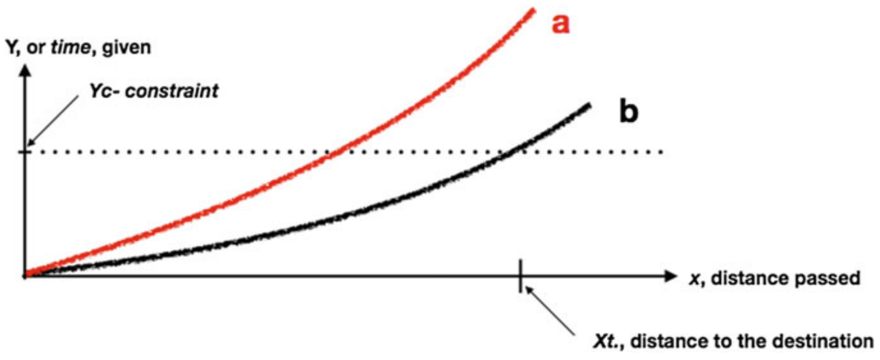


Fig. 19.1 Desperation of a package

$$\mathrm{k} = \mathrm{Xt}/\mathrm{Yc} \tag{19.4}$$

The biggest difference between real-time traffic packages and regular is that for the first ones the timing is strict: a package must be delivered within allocated period of time and before the predefined moment of time;

It is also clear that within this category of real-time packages the importance of them might differ as well.

Thus, for video or audio messaging the loss of some packages (if they are not clustered) might be almost invisible, while for control of an unmanned aircraft, a health-monitoring system or nuclear reactor control system, military or financial messages cannot be lost in principle. Desperation and real-time-ness should be analyzed along the DCS traffic as a whole.

There are two views to think about real time traffic:

- global and
- local

Global assumes that we know not only time for package within which it should be delivered but delays which each router and line along the package path contributes. Assuming "a global view" we are capable to estimate all delays and having this information can make decision "globally".

Thus, "globally" speaking, when our packages are highly unlikely are able to reach a destination before deadline we should be discarding them well before the real moment of deadline comes. It saves a lot of space and time for others and "smooth" traffic as a whole for the whole DCS or Internet. Or,—especially for real-time messaging we should make all our efforts to accelerate packages along the paths available.

Our approach assumes a creation of a "weighted" contribution for each segment of a package path and estimate it according the whole journey;

Let's say a segment j share of total delay is a division of time spent within this segment divided by total time allowed for messaging from the source to destination:

$$j\text{-th}\, share = \frac{t_{j\text{th}}}{\sum_1^n t_i} \tag{19.5}$$

Then we are able to analyze the situations such as: "are we really late even if we are late at this one, particular, but maybe insignificant segment?" It is interesting, there is no doubt, but requires some global knowledge, awareness about updated traffic information involved.

Also, probably we should consider all agents involved. Sender with its computer drivers involved, as well as all along routers with their programs, including in and output drivers and policies of reservation and making pull decisions.

19.4 Researching Desperation

Some good research questions here are worth to consider:

– *Probability of reaching the destination in terms of time remains, how it changes?*
– *How an internal state of each router and its organization impact on performance of network for real-time routing?*
– *Is it right to have a decision to hold or pull based on the level of time left?*
– *Should be better service (priority higher) when the threshold gets closer?*
– *How about steepness of a curve of success reaching a destination in time?*

Regretfully, this global view is over-optimistic, as delays due to traffic vary. Additionally, hardware malfunctions and path restrictions also contribute in uncertainty of delivery time. We consider this "global" view in a bit more details further.

In turn, the "Local view", or local approach is based on the interaction of elements inside the router represented such routing tables, queues, operating system, memory, processing power as well as hardware interruption schemes.

Every router—it is worth to remind—has to deal with packages of various types, i.e. mixed traffic of: normal, regular and real-time packages.

There are several potential questions that might have research interest:

– What we know at the level of router? (see examples above, extended)
– Do we know the distance covered and how much time left? (at least ruffled?)
– A distance covered?
– Time left?
– Time spent?

Answering these questions, we note: for real-time packages we have limitations that we are aware of: time given time spent at each segment.

Sure, we have to create a policy of pulling out packages that are approaching their deadline (of presence in the router we are talking about). The closer to the threshold the more urgent pull actions should be exercised.

This is what exactly has been called by as *level of desperation* or *desperation* policy.

Instinctively, an algorithm at the local (router) level to deal with real-time packages is obvious, Fig. 19.2.

In combination with a parameter of the package known as TTL (time to live), that describes maximum permitted either time (is milliseconds) or in numbers (number of hops permitted) time, for each package we can create a simple formula that combines three parameters:

– *Resource available*
– *Resource used*
– *Resource required*

```
Begin: get all real time packages inside the router queues (might be several queues)
- calculate a derivative – how fast each package is approaching its limit of stay in
  the router (it is possible to do very fast with 4-7Ghz processors in the routers)
- create the list of packages with order to pull out accordingly the "desperation"
- execute the most emergent ones
- return to the Begin
```

Fig. 19.2 Algorithm of desperation control

Note that these variables are not summed up like

$$Resource\,available = Resource\,required - Resource\,used \tag{19.6}$$

This is simply because "Resource used" is a derivative of TTL and "Resource required" is defined by type of package during formation of package, while "Resource available" is an internal property of a router. Note that TTL for real-time packages should be set much tighter than for normal package.

In other words, if we have more time to deal with a package without "a stress" the level of desperation is low:

$$Resource\,available \gg Resource\,required - Resource\,used \tag{19.7}$$

Then, the closer we are approaching the equality the higher level of desperation:

$$Resource\,available \geq Resource\,required - Resource\,used \tag{19.8}$$

And when

$$Resource\,available < Resource\,required - Resource\,used \tag{19.9}$$

We are, obviously, already late. We might introduce a "probability" of unsuccessful/successful delivery of package as

$$P_i = 1 - \frac{\sum_{i=1}^{n} x_i}{D} \tag{19.10}$$

where x_i is a cost of i-th segment; D—"a diameter" of network, or full distance of package transaction in it. This *probability*, in other words, indicates that the longer we travel the less chances to be on time.

Equation 19.11, in turn, defines a workload for i-th package delivery in time required from the network in terms of relative cost:

$$P_{is} = \frac{\sum_{i=1}^{n} x_i}{D} \tag{19.11}$$

And, as above, irregularity should satisfy:

$$\sum_{i=1}^{n} x_i < resource\ required \tag{19.12}$$

There is an engineering suggestion that might be useful in handling of desperation level up to the last moment of package handling—so-called "epsilon" ε_{Ra}. This "epsilon" can be used as estimate of speed approaching the equality of resource required and available, and when we see that almost nothing left: resources available $dR_a \to 0$ it is required to evaluate relative speed of its consumption, then

$$\frac{dR_a}{R_a} \to \varepsilon_{Ra}$$

means nothing else but relative "share" of the package requirements along the total cost of the journey. Physically speaking it means that if almost nothing left $dR_a \to 0$ but we are at the very end of the journey $R_a \to 0$ it is worth to have for this case and engineering threshold to consider for the last push priority and policy.

Additionally, as a subject of special further research it is worth to analyze a question like:—"what if we arrive too early…". This is important as the load of inside a node package Shandling might grow enormously. The importance of it grows—because we might create a queue inside the router and a problem of sorting packagers to put them in order ease their handling at the destination side. This, in turn, causes or might cause internal delays along the path of package.

19.5 Graph Theory Elements for Desperation Model

Well, division of the cost of the link on the weight of the total journey is good in terms of analysis of contribution of this link—share of it in the whole distance between chosen nodes or vertices.

It is also useful to consider this distance thinking about graph cardinality—i.e. how many nodes or vertices are in the graph as a whole. Internet as a graph model is pretty rich in number of connections and astronomical in number of terminals connected 2Bn+.

For the purpose of this research the structure of the Internet is sufficiently described in [9, 12, 13] indicating several layers of nodes. Each layer has different importance; further down Internet becomes a tree-like graph. New trends though change this established picture because each mobile terminal (device) now receives an option to become also router for others.

Saying this all gives us some argumentation that cardinal number and diameter of graph are good terms to discuss the role of number of nodes and cost (or weight) of path when we introduce different type of packages. Any estimation in particular, pre-estimation (before sending a package) provide information for evaluation of package travel success in presence of time constraints.

When we speak about directed and weighted graph *The diameter* of a graph G, is the maximum distance between two vertices or nodes. It usually denoted as Diam (G) [14].

Also, it is worth to consider how far a source node distant from any other vertices. For this, in graph theory a term *eccentricity* is used. *The eccentricity* of a vertex is the **maximum distance** from it to any other vertex.

Note here that packages are treated by routing algorithms and routers in terms of pursuing Hamiltonian property of the journey. Hamiltonian property of any path in graph also known when implemented as Hamiltonian path.

Tracing of the path in the graph for routing assumes that package is not repeating any of path and visits each vertex exactly once. Thus, for any node we can think about Hamiltonian cycle any of two nodes—sender receiver we A Hamiltonian cycle (or Hamiltonian circuit) is a Hamiltonian path that is a cycle. This is convenient to analyze when we seek a maximum length of routing for corporate networks which have known structure and costs of the links.

What we have said here about a distance on the graph, eccentricity, Hamiltonian paths or cycle and diameter it is worth to illustrate it on example. Figure 19.3 and Table 19.2 present a planar directed graph with weighted edges. Table 19.2 accompanies this graph presents all mentioned properties and distances.

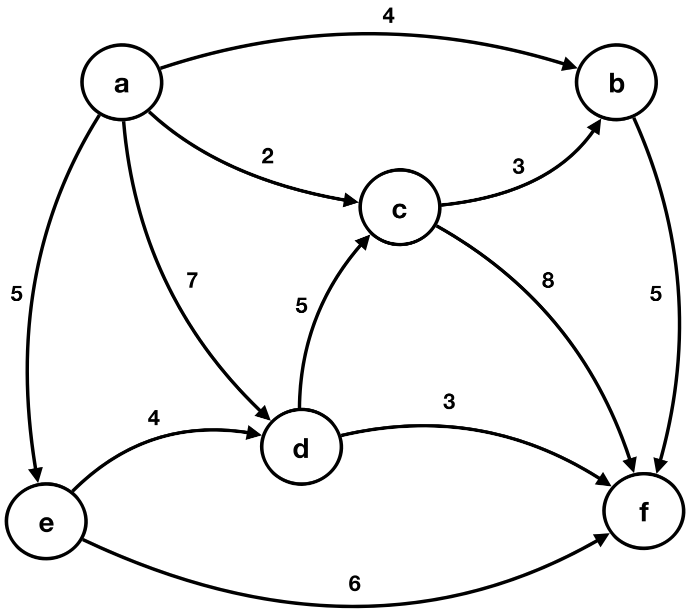


Fig. 19.3 Sample of the graph for cost-wise routing

Table 19.2 Graph description in terms of distances and shares

Name	Value	Comments
a-b	4	Distance source-destination
a-c	2	Distance source-destination
a-d	7	Distance source-destination
a-e	5	Distance source-destination
b-f	7	Distance source-destination
c-f	8	Distance source-destination
d-c	5	Distance source-destination
d-f	3	Distance source-destination
e-d	4	Distance source-destination
e-f	6	Distance source-destination
Hamiltonian cycle		
Diameter		
(a-b)/Diam		Share of edge within a network
(a-c)/Diam		Share of edge within a network
(a-d)/Diam		Share of edge within a network
(a-e)/Diam		Share of edge within a network
(b-f)/Diam		Share of edge within a network
(c-f)/Diam		Share of edge within a network
(d-c)/Diam		Share of edge within a network
(d-f)/Diam		Share of edge within a network
(e-d)/Diam		Share of edge within a network
(e-f)/Diam		Share of edge within a network

19.6 Desperation—Proposed Algorithms for Handling

In the algorithms of forward tracing and backward tracing (see for more details UK patent [15], papers [5–7] books [1–3] we have introduced on-going probabilistic reasoning to justify next step forward and next step backward. This enables us:

- to choose most likely further flow of events, predicting the future
- discover the most likely reason what went wrong (through backward tracing)

Both actions made possible preventing of unpleasant consequences and finding the way to change the structure and element(s) of the system.

In our books [1–3] and patent [15] we were considering an idea of introduction a realistic engineering constraints (with ε_f and ε_b) of possible further propagations of fault in the system.

Note here that an anomaly detected at one node (element) might cause serious consequences for other nodes adjacent to the first we start considering. Equations along the paths were based on multiplication of probabilities along the lists of adjacent nodes:

Variety of lists for possible propagation of a problem and possible reasons are based on a type of problem faced. For example, in case of leakage, water propagates along the system almost everywhere, in case of fire as well, while in case of shot-cut, due to energy discharge damage is limited by neighborhood nodes.

Thus graph model of a system was enriched by logic operators for each incoming and outcoming link. In this work, instead of fault propagation we are trying to apply the same principle for algorithms and methods of system description (graph, graph logic model (GLM) [16] to monitor "a level of desperation" for packages traveled across network.

Indeed, these days at router level various packages are handled differently, using the content of routing tables. Group of various algorithms introduced for that in [8–12] are applied sometimes even in combination to address the requirements of different types of packages. This all has been known as Quality of Service (QoS). Amount of reading about such algorithms exceeds (11.5 Million) and grows (see again previous chapters); surprisingly all of them are based on either Dijkstra or A* algorithms but scrutinized "to fit" QoS demanded by type of package.

Mentioned algorithms use routing tables. Routing tables, in turn, are using the protocol of network management protocol (NMP), that updates distance each 200 ms.

Modern routing is based on adding-on analysis of possible traces, paths and selection of the most relevant or suitable. Both Dijkstra's and A* algorithms form a basis for modern routing. Cisco [13] and other algorithms [12] does not have sufficient difference with Dijkstra and A* to be mentioned or considered.

In these circumstances introducing a level of urgency for service (or level of desperation as we call it) into routing of packages especially for real-time packages instead of known routing algorithms might be useful. This is simply because:

"*those who need the most should be treated first*"

But before to dive into details we should clarify what a difference of desperation in generic terms is with the one we are trying to be explicitly defined for the purposes of package handling. It is clear that the less time left the more effective service should be provided.

Sure, it is possible to think and introduce inside the router something like integer semaphores amongst all arrived packages with claims to be served and accessing it through critical section or we can call "service zone" instead of "waiting zone". We might, for example, provide for all queuing packages "a right to be served", following the rules of P, V semaphores [17] or even fault tolerant semaphores and [7]. When package (or process) is required to be served it requests to get through "critical section"—P, V semaphores ("NATO book" Jenui Ed., [17]. This solution is universal and it works. But number of voters is varying and complexity of voting scheme becomes too high at the router level and overall performance of it, therefore will be unavoidably poor.

Naturally, to implement "level of desperation"-driven service we need to evaluate timing for each package through the whole journey. At the same time, at the router level we do not know so far how many paths along the graph of network a package should take to reach its destination.

We might see in practice that some packages from so-called real-time ones sent arrived “almost” to their destination and thus can sit for a while in a queue before the last jump. Other packages in the middle of their journey might need “emergency treatment” and served first making their passing as smooth as possible.

We easily can generate different situations and suggest another dozen of algorithms [18–20] to address examples presented. But this is not practical—a performance [21] of the system as a whole requires thinning of service routines and relieve more time and space for user data (packages).

One of the solutions proposed here is to merge algorithm of forward tracing using weighted relative cost of the links with standard routing table existing in each router. An introducing the relative weight of path defined inside the routing table for chosen package.

Indeed, the size of Internet in numbers of nodes exceed 7 billion nodes, therefore, considering Internet as a set of nodes we can get a cardinal of this set—i.e. a number of elements in the set. Surprisingly, due to hierarchical organization between the most distant nodes there are no more than 20–25 routers, (from our examples using Ping we observed 11–14 routers). Additionally—assuming “round the Earth” distance of tracing one might for each router “weight” the share of this router in the path of a package d_i/D where d_i is a distance known from internal routing table and D is a “diameter” of an Internet.

19.7 Comparative Analysis of Proposed Algorithm and A*

Here we present an experiment and simulation of desperation control algorithm as described with known widely used in networking as Dijkstra and further modification as A* algorithm.

Let us repeat a Fig. 19.3, expanding it to our convenience as Fig. 19.4.

Dijkstra algorithm is based on need of expanding a horizon if visibility at each iteration of interaction with neighbors, gaining distances for steps to the further destinations. Knowledge about the network obtained at each step contacting neighbors. What we see, sitting at node ***a***? the distances 5, 7, 2, 4, to our direct neighbors: nodes ***e***, ***d***, ***c***, ***b***. What we chose knowing these numbers? Obviously, a link with cost 2, to the node ***c***.

Taking first step and considering that next searching of distance will be produced from the node c we create further list of distance from ***c***, taking into account (reading backward) a distance accumulated from starting point—node ***a***. If we aiming node ***f*** as our final destination we have to consider two distances—to node ***f*** directly—cost 8 and to node b, cost 3 with further unknown length written usually as:

$(\boldsymbol{b}, 3, \infty)$

Because a distance to b is shorter than distance of straight ***f*** step—(costs 8) it is possible to appoint as next marked node from where we will seek next step a node ***b***:

$\boldsymbol{b} \to \boldsymbol{f} : 5.$

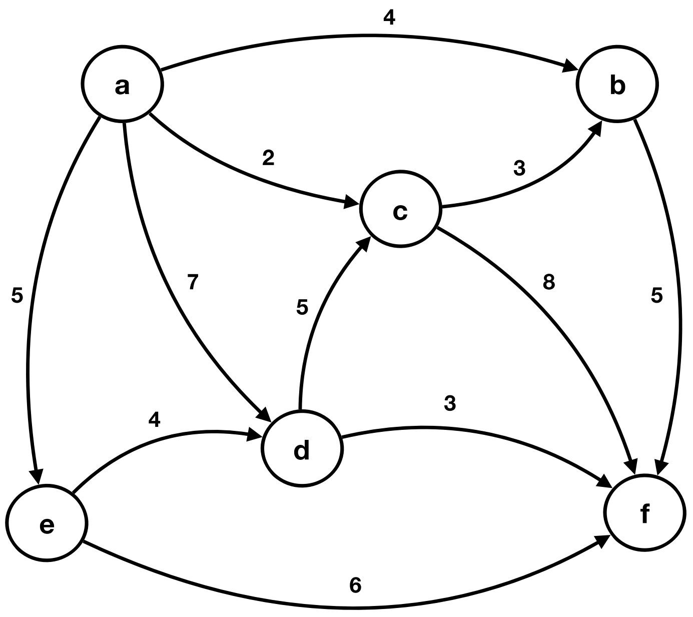


Fig. 19.4 An example of graph with cost of links

Thus we have two alternatives to reach from the node ***a*** destination ***f***

a $\rightarrow$ ***c*** $\rightarrow$ ***f***, total cost 2 + 8 = 10 and
a $\rightarrow$ ***c*** $\rightarrow$ ***b*** $\rightarrow$ ***f***, total cost 2 + 3+5 = 10

Dijkstra idea is about to move on and not stop when you have reached a destination. A *horizon*, expanded by your journey, at each and every step should be considered in both way—forward and backward: while node ***b*** is appointed as leading its backward link cost should be included in a joint table, giving us as a surprising example:

a $\rightarrow$ ***b*** $\rightarrow$ ***f***, with total cost 4 + 5 = 9;

This was not visible when we have chose the cheapest cost at the first iteration.

Thus, Dijkstra insists on visiting all nodes and creating a full link cost table when we have full and updated knowledge gained at each iteration. It is pretty good rule: expanding your own horizon to your neighbors is as well expanding theirs and when you have full knowledge of table you might see the first cheapest step advantage was short lived and lined up.

Much wider spread in networking a routing algorithm called A*, that we briefly describe further. Dijkstra algorithm is a core to understand the steps required in discovering routes, thus one can consider A* as modification of Dijkstra algorithm.

In turn, our Desperation Control Algorithm (DCA) is using a diameter of a graph or longest path as a parameter to estimate chances to reach a chosen destination. Let us assume that longest path between source and destination is known:

$\boldsymbol{a} \rightarrow \boldsymbol{e} \rightarrow \boldsymbol{f} := 11$

Then, relative weights to each link we have:

$\boldsymbol{a} \rightarrow \boldsymbol{c} \rightarrow \boldsymbol{f}$: (2/11) × (8/11) = 0.1322..
$\boldsymbol{a} \rightarrow \boldsymbol{d} \rightarrow \boldsymbol{f}$: (7/11) × (3/11) = 0.173…
$\boldsymbol{a} \rightarrow \boldsymbol{b} \rightarrow \boldsymbol{f}$: (4/11) × (5/11) = 0.1625

A-la "probabilistically speaking" and giving a threshold of success, say, 0.15 we might choose two bottom paths from this example.

Here, by the way it is worth to make some comment that only initially looks like a deviation.

We extract a bit of text about efficiency of design of embedded systems having technological constraints from [2]:

The on-board system options to implement both: fault tolerance and performance are limited by technologies. Power consumption is also a crucially limiting factor for any practical design of these systems.

Thus, we have generalized case of system design with constraints. The use of redundancy, therefore is very limited and no matter which scheme of design we choose, a maximum reliability can only be achieved if and only if the reliabilities of all elements of onboard systems are equal. Figure 19.5 illustrates about maximum reliability/performance design flow with constraints.

What does it mean for our algorithm with desperation control along paths? We are facing the same classic case—since ancient Greece problem of maximizing the rectangular size having a limited length of a line given: the ancient Greece legend says that King was giving privileges by giving his knights a rope: arguing get the land as much as the rope gives you. Clear that regarding rectangular the best option was a square. Having more options his generals could be of cause using a circle.

This scheme gives us some navigation for movement along the paths of the graph:

To preserve a probability of success for timely delivery along several paths we should seek the paths that "contribute" to the whole journey relatively similar weight.

Thus, if we have a distance between source and destination in number of nodes, say 6 then we should choose from available options at each node the path that as close as possible to L/6.

There is no doubt that it is not always possible and feasible but attempting it at each router will guarantee smoothness of traffic for critical packages in the long run.

Additionally, this concept might be supported by internal procedures and programs each router involved [18]. Thus, early arrived might be slightly delayed, while late one should be served ASAP; this at router contribution will smooth traffic in combination with proposed above optimisation. What are the options? Let us briefly highlight what actually router does.

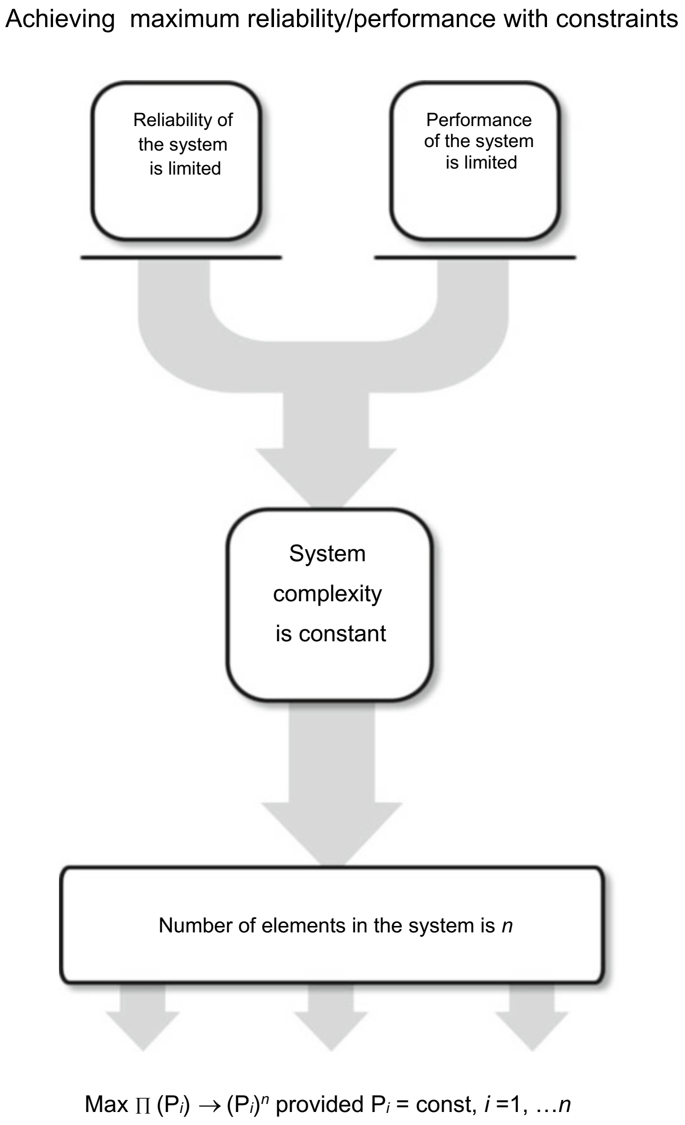

Achieving maximum reliability/performance with constraints
Reliability of the system is limited
Performance of the system is limited
System complexity is constant
Number of elements in the system is n
Max Π (P_i) → (P_i)^n provided P_i = const, i =1, …n

Fig. 19.5 Design of a system with constrains

*19.7.1 Briefly About A**

Before presenting the result of comparative simulation for A* and our proposed "desperation" algorithm we should briefly explain what A* means and how it works. When walking along the graph searching a shortest path to destination is good to know how much efforts were spent and how much is also required. All efforts are also useful to know. We have been describing similar parameters for our "desperation" approach. A* uses so-called *cost plus heuristics* function f that includes for each node the past cost function $g(x)$ and future cost function which is acceptable estimate also called "heuristic estimate".

$g(x)$ is dealing with past cost of the walk for the node x from a destination, while *h(x)* is dealing with an estimation—we stress here, an acceptable value—of future costs to the destination. A* therefore estimates each and every node (x) of the graph using a function:

$$f(x) = g(x) + h(x) \tag{19.13}$$

At each iteration "g" part of this function is updated and all potential "clients" are analyzed in terms of least cost. Further formal details of this algorithm can be found in [12].

19.8 Result of Comparative Simulation

Further simulation of network using the same principle with generated 20 nodes 100 links graph demonstrate that probability of weighted links gives very good advantage in comparison with A*. Initial setting of weights and heuristics is presented in Fig. 19.6 (Table 19.3).

The way A* works is demonstrated by red node marking at each iteration, until it reaches a destination node. For this particular layout we observe 22 iterations at cost of each defined by an amount of adjacent links to the current active node.

Let us consider now how Desperation Control Algorithm works on the same layout. Immediately it becomes visible that amount of iterations shrinks and, therefore an estimation of path finding that network is doing after every 200 ms synchronization of links length (using network service protocol) becomes several times shorter: QED (Table 19.4).

19.9 We Have More: A Vector Heuristic

Using a diameter of graph that we consider a matrix and adjusting weights relatively as di/L with multiplication along the paths and balancing as above attempting to find almost equal contribution of each path to the journey to destination. At the same time, we were using a heuristic that gave us estimation how long journey we still have ahead. But this is not the whole story.

Network can be considered as a matrix and in this sense, we are having two-dimensional space with heuristic at each path that give us an indication "how

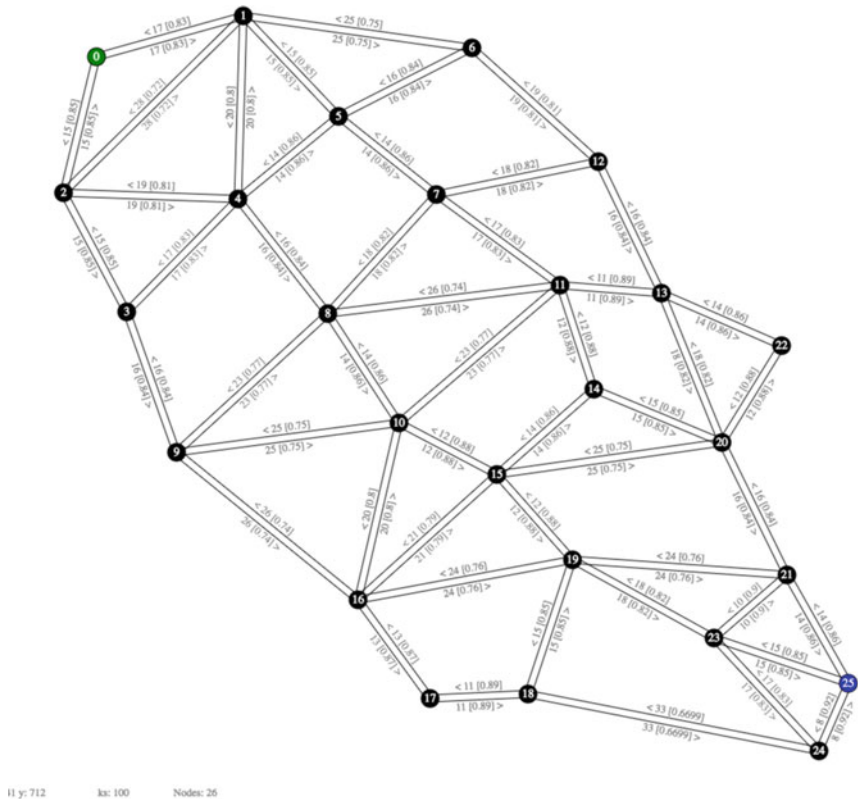


Fig. 19.6 Initial setting for A* and Desperation Control Algorithms

much is left". But having matrix we are facing borders that might be in close proximity and in this sense heuristic as a number does not help us much and always.

Thus, Stephen Farrell has suggested to read heuristics as a *vector* and at *each step* of path finding or routing search about destination we have to consider two values that might navigate us toward destination more efficient way, provided of cause—that "global heuristic" is also vector.

In other words, sitting in one corner of the network and having a direction to the destination indicated and adjusted at every step we can move along keeping the direction and watching—using local vector heuristic at each step. This enables us finding much more efficient routing by all means. Figure 19.7 and comments for each illustrates how it works.

But this all was and is about network level. How about inside job?

Table 19.3 Evolving process of A* path finding

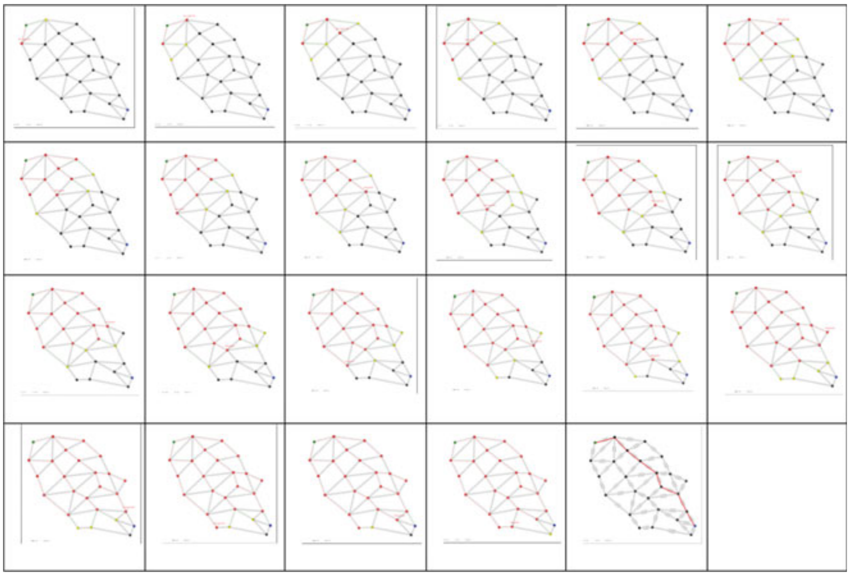
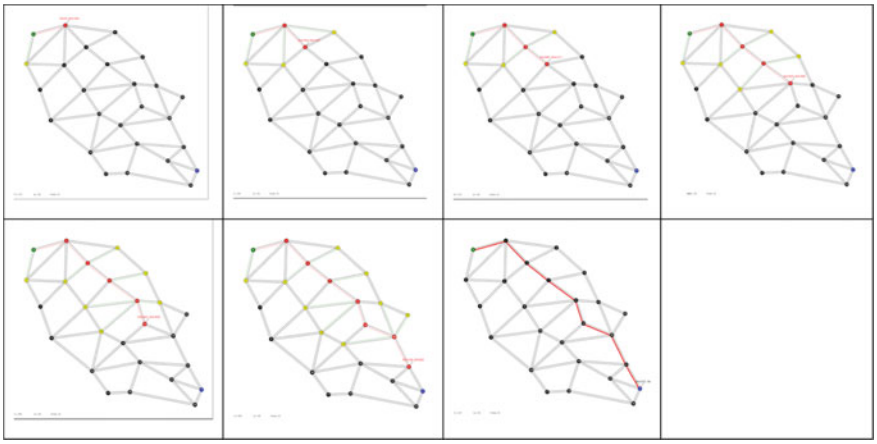
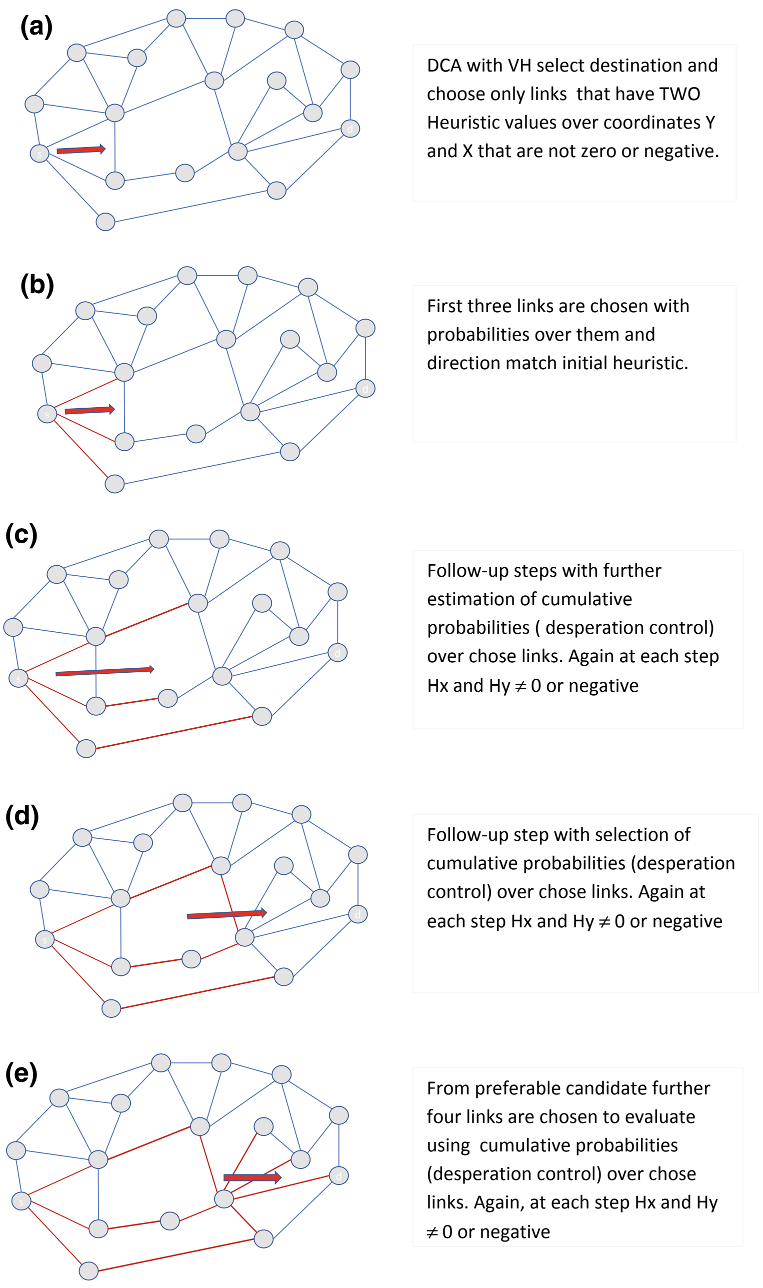


Table 19.4 Desperation control algorithm for path finding





DCA with VH select destination and choose only links that have TWO Heuristic values over coordinates Y and X that are not zero or negative.

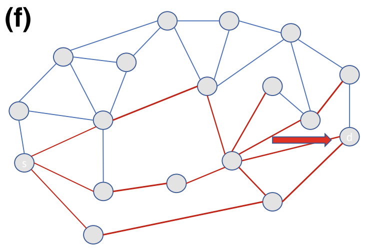
First three links are chosen with probabilities over them and direction match initial heuristic.

Follow-up steps with further estimation of cumulative probabilities (desperation control) over chose links. Again at each step Hx and Hy ≠ 0 or negative

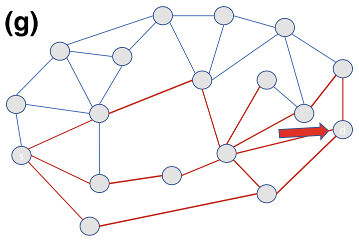
Follow-up step with selection of cumulative probabilities (desperation control) over chose links. Again at each step Hx and Hy ≠ 0 or negative

From preferable candidate further four links are chosen to evaluate using cumulative probabilities (desperation control) over chose links. Again, at each step Hx and Hy ≠ 0 or negative

Fig. 19.7 **a** Initial state s—source, d—destination. **b** First three selections along the direction. **c** Selected only links with min overall desperation and Hx, Hy $\neq 0$. **d** Selected only links with min overall desperation and Hx, Hy $\neq 0$. **e** Selected only links with min overall desperation and Hx, Hy $\neq 0$. **f** Destination has been achieved, and Hx, Hy $\neq 0$, check the rest inputs. **g** Destination has been achieved, all inputs has been checked



Destination has been reached. Still, other Hx, Hy cumulative probabilities (desperation control) over chosen links should be evaluated. Again, at each step Hx and Hy ≠ 0 or negative



Job done. DCA with VH works.

Fig. 19.7 (continued)

19.10 Router Level of Desperation Control

Let us summarize features of routers might use for desperation control. Routers consist of basic components like network interfaces, processing modules, buffering modules (memory), and an internal interconnection unit (switch fabric or bus). Inbound network interfaces receive packets and forward them to the processing modules where they will be processed and stored in buffering modules.

Further packages will be forwarded to the outbound interface, to the destination through the process of path determination.

The major tasks of a router are to determine path from routing table, maintenance, and reachability propagation (route processing).

Both internet protocol: IPv4 and IPv6 still require package processing, packet forwarding, and some router special services. Searching for a route in the routing table the longest prefix that matches the destination address of the packet is one of heaviest function of routing. Nowadays 7Bn IP addresses, in turn, even cluster-organized and hierarchical making routing tables quite large, and processing of them is time consuming.

In classless inter-domain routing (CIDR) [19, 20] a block of same class addresses is arranged as a single routing table entry. When a packet arrives, a routing table lookup is performed by the router using the packet's IP destination address. This process returns the best-matching routing table entry, which identifies to the router the IP address of the next hop for the specific packet and from which interface to forward the packet out.

These brief description of basic router duties highlights that a package inside might be reorganized, delayed, queued at the input and at the output with absolutely different reasons.

19.10.1 Router Level—Inside Job

Every package awaiting processing inside a router can be described by states 1, 2, 3… n, Fig. 19.8a. The number of "states" allowed depends on a type of a package. For example:

- Packages with emails might wait, two hundred milliseconds of delay for this type of messaging is not critical;
- Real-time messaging requires "smooth delivery", when some packages might be even dropped but the rest should be delivered in the strict time slot (video- and audio messaging);
- Safety-critical messaging and related packages (when the Internet is applied in real-time applications) vary in the level of restrictions and urgency.

Thus we can consider forced us to think about handling of "states" allowed for each type of package. Changing of states for a package might depend on time this package queued in the router. Clear that various type of packages can be given different number of states per package. Instinctively we simply must keep the delay for a package with high demand as short as possible and process package that is reaching its critical limit ASAP. Processing here means:

- receiving
- routing
- pulling out from incoming queue
- pushing in allocated output queue

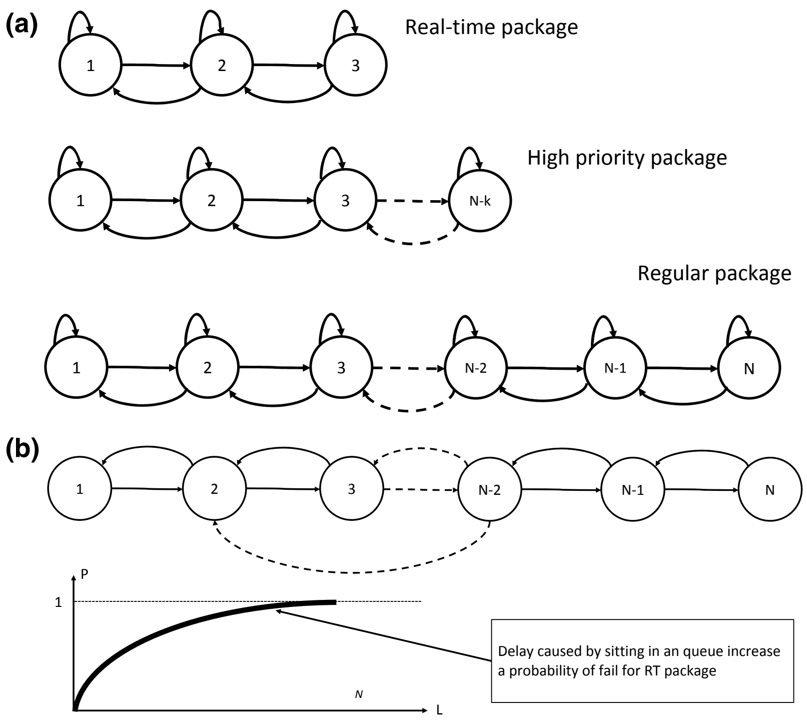
This kind of monitoring in terms "who sits here long enough" introduces a policy to handle queues accordingly "the level of desperation", for each and every package.

For a hard real time we simply must have the shortest queuing and fastest service from input to output, when priority is high, but not really real-time type of traffic, the queueing might be a little bit longer, and finally when a package is standard the waiting time might be much longer, Fig. 19.8a illustrates qualitatively behavior of router in terms of different type packages.

What kind of support the principle of desperation control requires from a router? It is fairly simple:

A priority within a class or even across different classes of packages should be *considered and treated as variables*.

This changing should depend on time of package presence within a queue and "history of the package". Thus, "High priority" package in any queue should be



Knowing the shape of a curve P(L) and current state of desperation for i-thpackages $P_i(t)$ it is possible to create an *algorithm of desperation handling (ADH)* elevating a package from, say, P_{N-2} to P_2 , etc.

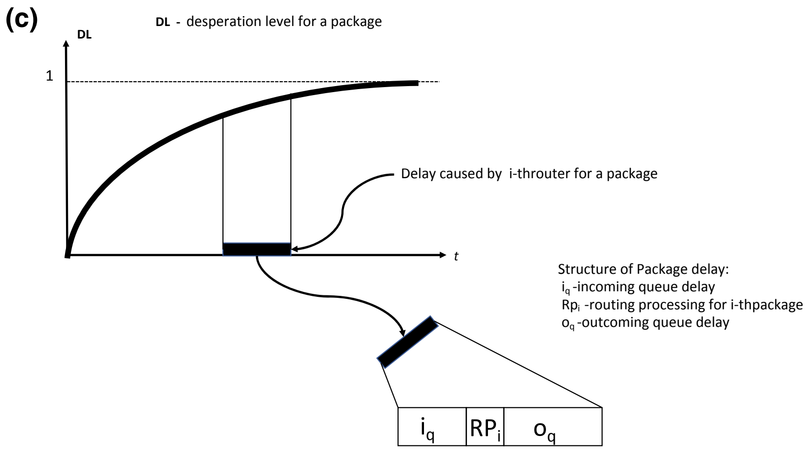

(c)
DL - desperation level for a package
DL
1
Delay caused by i-throuter for a package
t
Structure of Package delay:
i_q -incoming queue delay
Rp_i -routing processing for i-thpackage
o_q -outcoming queue delay
i_q
RP_i
o_q

Fig. 19.8 **a** Various lengths of queues for different types of packages. **b** Handling level of package desperation at router level. **c** Illustration of desperation level for a package processed even before real-time package if the priority and timing for this package become critical.

Router queues are organized so far as Fig. 19.8a qualitatively illustrates. The theory of this kind of queues is well developed by Kleinrock [11]. This theory estimates the position in a queue for the process with different states.

What is not considered in [11] is that what is proposed here: queued element (package in a queue) in real world should change its priority:—from very low to highest possible. Our desperation principle is just about it. Priority for RT packages is growing as long as its time left to destination is shrinking; Eq. 19.14 defined a so-to-say "a probability" of loss is growing (in other words, desperation goes beyond the chart).

In terms of implementation we might consider that packages should be described by two priorities—a static as given initially and dynamic—that is changing along the package journey in the network and a good router should handle and deal with both of them. We can write it symbolically:

$$\mathit{While}\, P \uparrow, L \rightarrow 0 \tag{19.14}$$

In plain English: when desperation of the package grows a position in the queue should be changed toward the output, leaving the queue (Fig. 19.8b).

As it was mentioned already, an "inside job" should be seriously revised to assist implementation of a principle of desperation control at the router level. Let us check what is possible here.

We need to elaborate this rule a bit further and see how to see it and implement it: Fig. 19.8c.

DL stands for desperation level of a package for the whole journey. Inside an i-th router delay for this package consists of input queueing delay I_q, processing delay (routing processing) RP_i and Output queueing delay O_q. All three might and should be revised in structure and overheads accordingly desperation level of a package.

Note that routing delay RP_i is squeezed as shown in previous sections of this work. Further delays I_q and O_q should be managed as Fig. 19.8b shows—moving a packages across the queuing accordingly desperation level.

Obviously that performance gain is scaled at order of magnitude, but for explicit results the whole structure of desperation control scheme should be analyzed using network simulation tool.

19.11 Aftermath—Instead of Conclusion

- We have shown how "*level of desperation*" works. We have shown also that desperation control in combination with *vector heuristic* perform by far better than known routing algorithms.

- We suggest to consider that priority of a message (that might include several packets) should be treated as variable. This variable can change both ways: decrease or grow along the journey of a package through the network, depending the network and package states.
- Our approach assumes that:

(A) Introduced limit of permitted processing time for each message is known
(B) Combining with overheads caused by state of internal router priority state message
knowledge about own progress

Thus, our result in one sentence is:

The proposed new schemes and algorithms increase efficiency of package handling and overall resilience, real-timeness, elasticity and reliability of network.

This is kind of system level solution. What else can be done here? Well, in [1–7] we have proposed a Graph Logic Model that enabled to see monitor incoming and outcoming conditions for every vertex of graph. It is potentially very useful for routing as well.

GLM unique features that fit for restructuring of routing table are:

- GLM-upgraded routing table can be adjusted, reflecting change of load and traffic
- Instead of cost of the link GLM introduces “probabilities” of the link (relative weight of the link in the network), this enables to estimate much faster the success of the journey
- Logic operator for each incoming and outcoming links should be modifiable.

Thus, when the system is overloaded that becomes visible by refusing to accept further incoming traffic—(a good example here is a so-called “denial of service”) GLM logic operator (for incoming link) can be modified from XOR down to NOP, and when traffic changes, can return back. For this purposes GLM introduces XOR (to accept all from this particular direction and nobody else) or NOP (nothing from this direction is welcomed) operators at the input and outputs of a node. At the node level the condition to process incoming and output flows might change and routing table outcome operators accordingly can accept all or several types (AND, OR) outcomes.

These all combined can handle different packages in various types of network load and conditions by the most efficient way.

This combination creates a kind of framework for network handling, considering both: types of packages and router conditions.

References

1. Castano V, Schagaev I. Resilient computer system design. https://doi.org/10.1007/978-3-319-15069-7
2. Schagaev I, Kaegi-Trachsel T. Software design for resilient computer systems. https://doi.org/10.1007/978-3-319-29465-0
3. Schagaev I, Kirk B. Active system control. https://doi.org/10.1007/978-3-319-29465-0
4. Schagaev I (1990) Yet another approach to classification of redundancy. In: CIM IMEKO symposium, Helsinki, pp 117–124
5. Schagaev I, Monkman S (2013) Redundancy + Reconfigurability = Recoverability. Electronics 2:212–233. https://doi.org/10.3390/electronics2030212
6. Schagaev I, Sogomonoyan E (1988) Hardware and software for a fault tolerant computing system. In: Automation and remote control, vol 49, no 2, part 1. Pergamon Press, 10 July 1988
7. Blaeser L, Monkman S, Schagaev I. Vision on reconfigurable systems, Chapter 10, [1]
8. Brockmeyer E, Halstrom HL, Jensen A (1948) The life and works of A. K. Erlang, 2nd edn. Danish Academy of Technical Science
9. Jackson JR (1957) Networks of waiting lines. Oper Res 5:518–521
10. Morse PM (1958) Queues, inventories and maintenance. Wiley, New York
11. Kleinrock L (1964) Communication nets: stochastic message flow and design. McGraw-Hill, p 220. ISBN 978-0486611051
12. Leon-Garsia A et al. Communication networks. McGraw Hill. ISBN-0-07-246352
13. https://www.bitpipe.com/tlist/Internet-Architecture-Board.html
14. Chartrand G, Lesniak L (2005) Graphs and digraphs, 4th edn. Chapman &Hall/CRC. ISBN 1-58488-390-1
15. http://it-acs.co.uk/files/Grant_for_a_patent.PDF
16. Schagaev I https://www.academia.edu/7685617/Control_Operators_vs_Graph_Logic_Model_-_WorldComp_2014_Presentation_and_Paper
17. Dijkstra EW (1968) Cooperating sequential processes. In: Hansen PB (ed) The origin of concurrent programming. Springer, New York, NY
18. Azan et al. System software support for router reliability. In: 30th IFAC workshop on real-time programming and 4th international workshop on real-time software (WRTP/RTS'09)
19. Fuller V, Li T, Yu J, Varadhan K (1993) Classless inter-domain routing, IETF RFC 1519
20. http://www.cisco.com/c/en/us/td/docs/routers/10000/10008/configuration/guides/qos/
21. Schagaev I et al. On performance of distributed computer systems. In: WorldComp16, 28 July 2016. https://www.academia.edu/25248500/On_Performance_of_Distributed_Computer_Systems

Chapter 20
ERRIC Machine Code Simulator

Aleksey Gospodchikov, Sergey Koziakov, Danila Romanov and Eugene Zouev

Abstract The architecture and implementation of the ERRIC instruction code software simulator is presented. The main property of the simulator is providing a tool for developing real ERRIC-based software even before a real ERRIC processor appears, or to develop software without a processor at hand. The overall structure of the software tool is described, and the key data structures and algorithms of the simulator are explained and discussed.

20.1 Overall Description

The principal architecture of the ERRIC code simulator is presented below, Fig. 20.1.

As Fig. 20.1 shows the developed software system consists of:

- an interpreter library capable of running machine code compatible with ERRIC command set
- a console and graphical clients that use the library to run executable files and show the results.

Together, they form the ERRIC Machine Code Simulator.

20.2 Basic Usage

The interpreter library is capable of interpreting a certain version of ERRIC machine code, the details of which were introduced in the Chap. 14, Sect. 14.2. The interpreter works primarily with self-contained structures, allowing for great flexibility, such as the option of simulating multi-core processors. The library also allows the user to read executable files in order to fill the structures to execute.

The console client executes the given executable from the start until the end on a single-processor machine and creates a memory dump after the execution.

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5_20

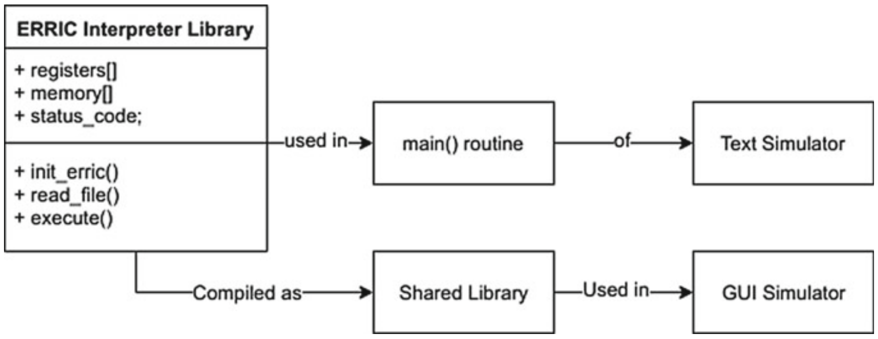


Fig. 20.1 Architecture of ERRIC simulator

The graphical client is more sophisticated—it allows the code to be executed not only from start to finish, but also step-by-step (e.g., for debugging purposes). It also shows the changes in memory and registers in real time during step-by-step execution. The special option provides the original assembler version for each instruction.

Both clients provide similar options: the user can change the amount of memory allocated for the virtual CPU and choose the file to execute.

An example of code that simulator can run is program that calculates the n-th Fibonacci number. The corresponding code snippet written in the system-level language (see Chap. 11) is shown in Listing 20.1, source code in assembly language is shown in Listing 20.2, and machine code produced from it can be seen in the next section.

Listing 20.1: Pseudocode for the Fibonacci algorithm

```
const DesiredNumber = 10
int n1 := 0
int n2 := 1
int t := 0
int curr := 1

loop
     t := n1 + n2
     n1 := n2
     n2 := t
     curr++
while
     curr < DesiredNumber
end
return n2
```

Listing 20.2: Assembly source code for the Fibonacci routine

```
<DesiredNumber>
     DATA 10    # Desired Fibonacci number index

     R1 := 0
     R2 := 1
     R4 := 1    # Increment
     R5 := 1    # Index of current Fibonacci number

<Loop>
     R3 := 0
     R3 += R1
     R3 += R2   # R3 now contains next Fibonacci number
     R1 := R2
     R2 := R3   # Shift numbers
     R5 += R4   # Increment counter
     R6 := DesiredNumber
     R6 := *R6
     R6 ?= R5   # Compare desired number and current Fibonacci number
     R7 := 2    # Binary mask
     R6 &= R7   # leave only the second bit
     R8 := Loop
     if R6 goto R8   # if R6 > R5 then second bit of R6 is 1
                     # then we need to jump
     *R29 := R2  # Push Fibonacci number to the stack
     R8 := 2
     R29 += R8   # And don't forget to increment the stack pointer
```

20.3 Simulator GUI

An example of writing a user interface for an interpreter library, besides one supplied with it, is a simple GUI that fully utilizes almost every available library function.

This graphical interface allows the user to load an arbitrary ERRIC binary file and execute the code from it on a single ERRIC processor, either completely, from start to finish, or step-by-step.

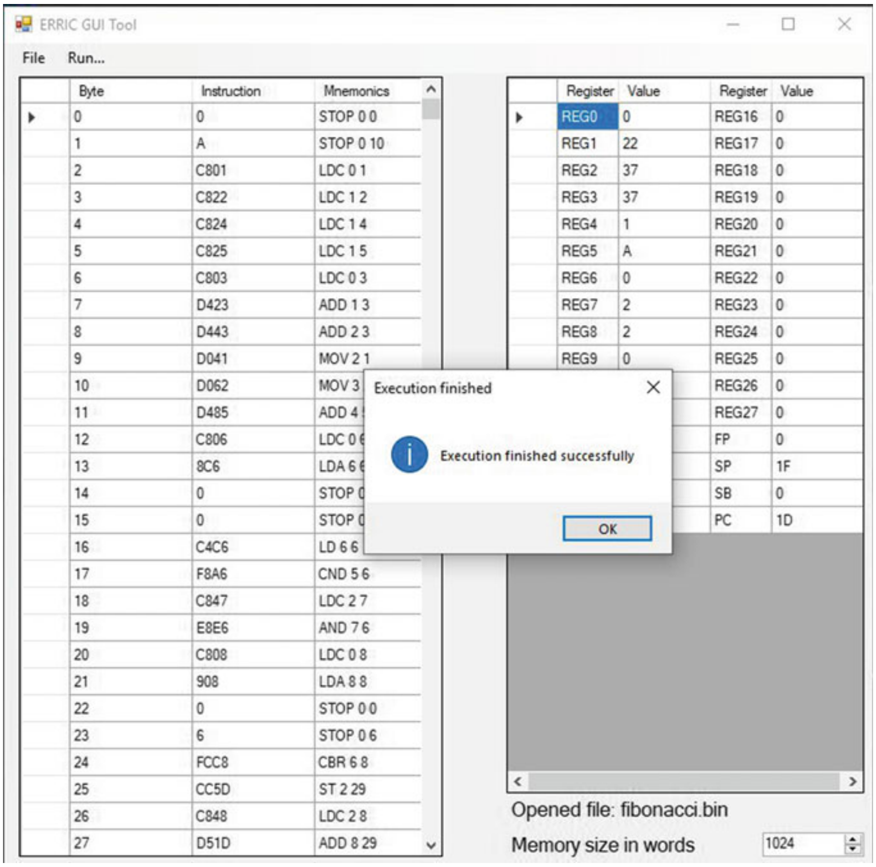
The interface also makes it simple to check individual memory words and the states of the registers during the course of the program's execution.

The top bar allows loading an ERRIC binary file via `File->Open`, and run it via the `Run...` submenu.

In the middle of the GUI is the memory map—it shows every word in the memory of the processor. It is automatically updated during the step-by-step execution, and it reflects the final state after completing execution.

To the right of the memory map is the information about registers—it shows all 32 registers and is similar to the memory map—Table 20.1.

Table 20.1 Memory map during execution



Byte	Instruction	Mnemonics
0	0	STOP 0 0
1	A	STOP 0 10
2	C801	LDC 0 1
3	C822	LDC 1 2
4	C824	LDC 1 4
5	C825	LDC 1 5
6	C803	LDC 0 3
7	D423	ADD 1 3
8	D443	ADD 2 3
9	D041	MOV 2 1
10	D062	MOV 3
11	D485	ADD 4
12	C806	LDC 0
13	8C6	LDA 6
14	0	STOP
15	0	STOP
16	C4C6	LD 6 6
17	F8A6	CND 5 6
18	C847	LDC 2 7
19	E8E6	AND 7 6
20	C808	LDC 0 8
21	908	LDA 8 8
22	0	STOP 0 0
23	6	STOP 0 6
24	FCC8	CBR 6 8
25	CC5D	ST 2 29
26	C848	LDC 2 8
27	D51D	ADD 8 29

Register	Value	Register	Value
REG0	0	REG16	0
REG1	22	REG17	0
REG2	37	REG18	0
REG3	37	REG19	0
REG4	1	REG20	0
REG5	A	REG21	0
REG6	0	REG22	0
REG7	2	REG23	0
REG8	2	REG24	0
REG9	0	REG25	0
		REG26	0
		REG27	0
		FP	0
		SP	1F
		SB	0
		PC	1D

Execution finished

Execution finished successfully

Opened file: fibonacci.bin

Memory size in words 1024

Below the registers information, there are two buttons to execute the code, the filename of the opened binary file and the memory size field, which allows the user to resize the available memory.

Doing so will restart the execution and reset the memory and the registers back to normal.

The execution follows the process expected by the emulator—it can run as long as there are no changes in the status codes. When an error occurs or the STOP instruction is encountered, the GUI shows the associated message (either an error message describing the problem or a notification that a STOP instruction was encountered).

If an error is encountered, the user can get all the needed information from the memory map and the registers, the PC register in particular.

20.4 Simulator Internal Structure

Main data structure of the interpreter is `erric_t`, which describes the single virtual CPU based on ERRIC architecture, with its registers, memory, a status code describing the current status of the CPU and other auxiliary components, Fig. 20.2.

This structure is passed to most, if not all, functions available in the interpreter library. Therefore, the functions do not need to store any state regarding the structure, and thus can be used to simulate multi-core processors by working with multiple ERRIC CPUs, either asynchronously or by running them in turn.

For more detailed description and explanation on usage of the library, refer to Appendix 2.

20.5 Interpreter: The Implementation

20.5.1 Design Decisions and Caveats

- The project started with incomplete requirements and lack of full information on ERRIC architecture (previously called ERA). Because of this, some design decisions and assumptions may not match the current architecture. In particular, the RETI instruction is not supported yet.
- The original machine specification utilizes nearly all of the space for instructions, meaning that it is difficult to add additional machine commands.

```
struct erric_t {
      // common registers
      lword_t *registers;

      // machine memory
      word_t *memory;

      // size of the memory (in words)
      uint32_t memory_size;

      // instruction register
      word_t IR;

      // Status code field, checked after every instruction
      sword_t status_code;
};
```

Fig. 20.2 Main data structure of the interpreter

- The interpreter library provides only one automatic way of properly filling the `erric_t` structure needed for execution—by reading from an executable file. Functions to simplify filling the structure or to fill it from memory were considered, but not implemented to achieve the maximum flexibility.

First caveat one may encounter is differences between listed ERRIC architecture and the one implemented within the simulator. However, the interpreter library is easily modifiable to accommodate for any revision of ERRIC. Please refer to the Appendix 2 for the internal description of the library.

Another important aspect to keep in mind is that software was developed and tested primarily on little-endian machines, since big-endian machines are not particularly popular now.

However, to make reading memory dumps and manual debugging easier, it was decided to simulate Big Endianness for virtual ERRIC machine. This introduces some difficulties to understand code snippets, which, however, helps make this code endianness-independent.

20.5.2 Extensibility

As mentioned before, the machine code interpreter is implemented as a library. This allows other developers to build various kinds of user interfaces that may provide additional features independently of the library or use the library to execute ERRIC machine instructions directly.

One of the executable format files was also designed with extensibility in mind - assembly compilers and interpreters can take advantage of the empty space possible by the format to insert additional data needed for them.

The library is open-source, meaning that anyone can contribute to its development and expand the functionality.

20.5.3 Machine Code File Formats

During the implementation process, capability for interpreter library to read and understand different file formats was added, along with two of those file formats—simpler one and flexible one.

The formats specify the following information in the file header:

- File format version
- How much data (i.e. not code) is there, or where it starts and where it ends
- How much code is there (right after data block), or where it starts and where it ends

This allows adding new file formats, with, maybe, some extra debug information, in the future.

Using file formats opens the possibilities for extending both architecture and interpreter in the future, while allowing for keeping backwards compatibility. Formats described in more details in Appendix 1.

As mentioned before, the simulator supports two different file formats, but it is capable of recognizing up to 256 formats in the current design and can be safely extended to 65,535 different types.

Appendix 1: Internal Structure of ERRIC Interpreter and API Description

A.1 The Main Structure

The `erric_t` structure is the primary component of the library, Fig. 20.3: it represents a single CPU running the ERRIC architecture and contains the memory chunk and the registers for the processor:

Using the structure, the memory of the ERRIC machine can be accessed through `erric_t->memory` word array, and registers via `erric_t->registers.`

```
struct erric_t {
      // common registers
      lword_t *registers;

      // machine memory
      word_t *memory;

      // size of the memory (in words)
      uint32_t memory_size;

      // instruction register
      word_t IR;

      // Status code field, checked after every instruction
      sword_t status_code;
};
```

Fig. 20.3 ERRIC_t structure

A.2 Functions, Operators and Instruction Execution

Main functions and simulated operators are not directly bound to a particular instance of `erric_t`; rather, they operate on the single copy of the `erric_t` structure that is passed to them via parameter. There are also some helper functions, like those that unify similar instructions in one code block to remove redundant code.

All execution functions first call `parse_instruction()` routine, that fetches and decodes instruction pointed to by the PC register. Then, the instruction represented by the `instruction_t` structure, Fig. 20.4, format is passed to the `execute()` function, which executes it using attached operator function.

Apparently, this structure closely resembles one of the ERRIC instruction, described in specifications.

`format_t` enumeration type represents instruction format, as described in the ERRIC specification. They are named after their function in mathematical operations. Note that due to 10 format being reserved, only 3 formats are listed (Fig. 20.5).

A.3 Source Files and Their Contents

`erric_types.h` describes data types used throughout the ERRIC interpreter.

`erric_t` structure, defined in `erric_struct.h`, represents the ERRIC machine (as explained above). Its main parts are memory, registers, and machine status code.

`*_operators.h` files describe operators that ERRIC CPU can perform, and their helper functions.

`erric_interpreter.c` contains mapping from operation instructions to the function that executes given instruction, and also functions to fetch instruction in `instruction_t` structure and parse the instruction.

`erric_util.h` contains helper functions, for example functions that can make memory access easier.

```
struct instruction_t
{
        enum format_t format;
        sword_t code;

        sword_t i;
        sword_t j;
};
```

Fig. 20.4 Instruction routine

```
enum format_t {
     F_8_BIT,
     F_16_BIT,
     F_32_BIT,
};
```

Fig. 20.5 Format types

A.4 Executable File Formats

Both formats that are supported for the moment, start with the unsigned 8-bit integer version number and are big-endian (bytes are read from left to right).

Version 0: small and simple, but inflexible. File consists of:

- Version (8-bit unsigned integer), followed by 8-bit padding
- Global/static data length in words (32-bit unsigned integer)
- Global/static data fixed-size section
- Code and code data fixed-size section

In this case, we simply read as many words as specified in Data Length section, and then read the rest as a Code. This method has few possibilities to break, but doesn't allow the user to place the sections in an arbitrary order while also being poorly adaptable and not allowing additional metadata to be inserted into the file.

File structure example (version 0, Table 20.2).

Please notice that data and code are nonsensical, and are here just for example.

Version 1: more comprehensive and much more flexible version, even allowing for compiler/debug metadata:

- Version number (8-bit unsigned integer), followed by 8-bit padding
- Global/static data start position relative to the beginning of the file in **bytes** (32-bit unsigned integer)
- Global/static data length in **words** (32-bit unsigned integer)
- Code start position relative to the beginning of the file in **bytes** (32-bit unsigned integer)
- Code length in **words** (32-bit unsigned integer)
- Global/static data fixed-size section, code and code data fixed-size section in any order

This version is very flexible. Sections can be put anywhere and in any order. Additional elements, such as additional headers and meta-information, can be put in

Table 20.2 File structure example

2 bytes Version + Padding	00 00
4 bytes data length	00 00 00 02
4 bytes static data	FF F2 FF F4
6 bytes code	AD DC 56 69 F7 B1

Table 20.3 File structure Version 1

2 bytes Version + Padding	`01 00`
4 bytes data start	`00 00 00 20`
4 bytes data length	`00 00 00 02`
4 bytes code start	`00 00 00 18`
4 bytes code length	`00 00 00 03`
6 bytes ignored	`13 37 10 01 DE AD`
6 bytes code	`AD DC 56 69 F7 B1`
2 bytes ignored	`00`
4 bytes static data	`FF F2 FF F4`

empty spaces within the file, allowing for greater flexibility. This version is also a bit more fragile and difficult to understand.

File structure example (version 1, Table 20.3).

Notice that data and code are nonsensical, and are here just for example.

Appendix 2: Practical Introduction and Developer's Guide

The following part will describe how to use the interpreter library for those who wish to use it in their projects.

The interpreter library consists of a structure representing a single CPU running the ERRIC architecture, as well as a number of functions that statelessly operate on an instance of the function.

This allows the library to be flexible and to support different execution methods, such as multiprocessor hardware setups. The structure of the library is implemented in a way that makes easily understanding and use by all kinds of users: for those who want to simply use the interpreter and for those who want to call individual functions for specific purposes.

In order to use all of the main features of the emulator, the `erric_interpreter.h` header file needs to be included. Files `*_operators.h` allow the user to call particular functions manually.

Initialization

In order to work with the library, an `erric_t` structure is required. The creation of the structure instance can be done in two ways:

1. By the call `init_erric().` The function creates the structure with the default memory size of 65536 words, or
2. By the call `init_erric_m(uint32_t _mem_size)`. The function allows the user to specify the size of the memory needed.

The memory size is defined only during the creation and is not changeable while using the library. This was done to make sure that the hardware can be properly

```
    {
        struct erric_t* default = init_erric();
        struct erric_t* sized = init_erric_m(1024);
    }
```

Fig. 20.6 The memory size structure

simulated, since the software might behave differently under different memory sizes, Fig. 20.6.

Freeing

After completing working with the particular instance of the `erric_t` structure, the memory taken by it should be freed by using the `free_erric(struct erric_t*erric)`: (Fig. 20.7).

Memory setup

After the structure for the processor has been created, its memory should be filled with needed data and code, which can be done either by loading a file with

```
read_file(char *filename, struct erric_t *erric)
```

or by setting it up manually, Fig. 20.8.

```
    {
        struct erric_t* erric = init_erric();
        // ...
        free_erric(erric);


    }
```

Fig. 20.7 The memory release

```
    {
        struct erric_t * erric = init_erric();
        read_file(filename, erric);
        // ...
        free_erric(erric);
    }
```

Fig. 20.8 The memory setup

```
{
    struct erric_t* erric = init_erric();
    read_file(filename, erric);
    step(erric);
    execute(erric);
    free_erric(erric);
}
```

Fig. 20.9 Execution struct

Function will automatically detect and load file version specified within the file.

The only way to set the memory up by hand is to directly modify the memory and register arrays—refer to the reference for the ERRIC hardware documentation to determine the needed registers and instruction codes.

Execution

The library offers 2 functions to execute the code—`step(struct erric_t* erric)` and `execute(struct erric_t* erric)`, Fig. 20.9. The `step()` function executes the instruction pointed to by the Program Counter register and directly returns the status code returned by the instruction. The `execute()` function executes instructions one-by-one, starting from PC until it encounters any erroneous status code. The final status code can be accessed by `erric_t->status_code` field.

Index

I. Schagaev et al., *Software Design for Resilient Computer Systems*,
https://doi.org/10.1007/978-3-030-21244-5