ASPECT-ORIENTED TECHNOLOGY FOR DEPENDABLE OPERATING SYSTEMS

Dissertation

zur Erlangung des Grades eines

DOKTORS DER INGENIEURWISSENSCHAFTEN

der Technischen Universität Dortmund
an der Fakultät für Informatik

von

CHRISTOPH BORCHERT

Dortmund

2017

Tag der mündlichen Prüfung:	4. Mai 2017
Dekan:	Prof. Dr.-Ing. Gernot A. Fink
Gutachter:	Prof. Dr.-Ing. Olaf Spinczyk
	Prof. Dr.-Ing. habil. Wolfgang Schröder-Preikschat

ABSTRACT

Modern computer devices exhibit transient hardware faults that disturb the electrical behavior but do not cause permanent physical damage to the devices. Transient faults are caused by a multitude of sources, such as fluctuation of the supply voltage, electromagnetic interference, and radiation from the natural environment. Therefore, dependable computer systems must incorporate methods of fault tolerance to cope with transient faults. Software-implemented fault tolerance represents a promising approach that does not need expensive hardware redundancy for reducing the probability of failure to an acceptable level.

This thesis focuses on software-implemented fault tolerance for operating systems because they are the most critical pieces of software in a computer system: All computer programs depend on the integrity of the operating system. However, the C/C++ source code of common operating systems tends to be already exceedingly complex, so that a manual extension by fault tolerance is no viable solution. Thus, this thesis proposes a generic solution based on Aspect-Oriented Programming (AOP).

To evaluate AOP as a means to improve the dependability of operating systems, this thesis presents the design and implementation of a library of aspect-oriented fault-tolerance mechanisms. These mechanisms constitute separate program modules that can be integrated automatically into common off-the-shelf operating systems using a compiler for the AOP language. Thus, the aspect-oriented approach facilitates improving the dependability of large-scale software systems without affecting the maintainability of the source code. The library allows choosing between several error-detection and error-correction schemes, and provides wait-free synchronization for handling asynchronous and multi-threaded operating-system code.

This thesis evaluates the aspect-oriented approach to fault tolerance on the basis of two off-the-shelf operating systems. Furthermore, the evaluation also considers one user-level program for protection, as the library of fault-tolerance mechanisms is highly generic and transparent and, thus, not limited to operating systems. Exhaustive fault-injection experiments show an excellent trade-off between runtime overhead and fault tolerance, which can be adjusted and optimized by fine-grained selective placement of the fault-tolerance mechanisms. Finally, this thesis provides evidence for the effectiveness of the approach in detecting and correcting radiation-induced hardware faults: High-energy particle radiation experiments confirm improvements in fault tolerance by almost 80 percent.

PUBLICATIONS

The ideas and findings presented in this dissertation have partly been published in the following peer-reviewed journals and proceedings of international conferences and workshops:

[26] Christoph Borchert and Olaf Spinczyk. Hardening an L4 microkernel against soft errors by aspect-oriented programming and whole-program analysis. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS '15)*. ACM Press, October 2015. doi: 10.1145/2818302.2818304

Acceptance rate: 44 %, also appeared in ACM Operating Systems Review [27]

[28] Christoph Borchert, Daniel Lohmann, and Olaf Spinczyk. CiAO/IP: A highly configurable aspect-oriented IP stack. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*. ACM Press, June 2012. doi: 10.1145/2307636.2307676

Acceptance rate: 18 %

[29] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Protecting the dynamic dispatch in C++ by dependability aspects. In *Proceedings of the 1st GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*. German Society of Informatics, September 2012. URL: `http://subs.emis.de/LNI/Proceedings/Proceedings208/521.pdf`

[30] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generative software-based memory error detection and correction for operating system data structures. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*. IEEE Press, June 2013. doi: 10.1109/DSN.2013.6575308

Acceptance rate: 20 %

[31] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Return-address protection in C/C++ code by dependability aspects. In *Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*. German Society of Informatics, September 2013. URL: `http://subs.emis.de/LNI/Proceedings/Proceedings220/2519.pdf`

[32] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generic soft-error detection and correction for concurrent data structures. *IEEE Transactions on Dependable and Secure Computing*, 14(1):22–36, January 2017. doi: 10.1109/TDSC.2015.2427832

Open access, journal impact factor: 1.59

[109] Martin Hoffmann, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Rüdiger Kapitza, Olaf Spinczyk, and Daniel Lohmann. Effectiveness of fault detection mechanisms in static and dynamic operating system designs. In *Proceedings of the 17th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '14)*. IEEE Press, June 2014. doi: 10.1109/ISORC.2014.26

Acceptance rate: 63 %

[159] Arthur Martens, Christoph Borchert, Tobias Oliver Geißler, Daniel Lohmann, Olaf Spinczyk, and Rüdiger Kapitza. Crosscheck: Hardening replicated multithreaded services. In *Proceedings of the 4th International Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology (DCDV '14)*. IEEE Press, June 2014. doi: 10.1109/DSN.2014.98

[160] Arthur Martens, Christoph Borchert, Manuel Nieke, Olaf Spinczyk, and Rüdiger Kapitza. CrossCheck: A holistic approach for tolerating crash-faults and arbitrary failures. In *Proceedings of the 12th European Dependable Computing Conference (EDCC '16)*. IEEE Press, September 2016. doi: 10.1109/EDCC.2016.29

Acceptance rate: 46 %

[211] Thiago Santini, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Martin Hoffmann, Olaf Spinczyk, Daniel Lohmann, Flávio Rech Wagner, and Paolo Rech. Effectiveness of Software-Based Hardening for Radiation-Induced Soft Errors in Real-Time Operating Systems. In *Proceedings of the 30th International Conference on Architecture of Computing Systems (ARCS '17)*, Springer, April 2017. doi: 10.1007/978-3-319-54999-6_1

Acceptance rate: 45 %

[213] Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. Rapid fault-space exploration by evolutionary pruning. In *Proceedings of the 33rd International Conference on Computer Safety, Reliability and Security (SAFECOMP '14)*. Springer, September 2014. doi: 10.1007/978-3-319-10506-2_2

Acceptance rate: 27 %

[214] Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*. IEEE Press, June 2015. doi: 10.1109/DSN.2015.44

Acceptance rate: 22 %

[218] Muhammad Shafique, Philip Axer, Christoph Borchert, Jian-Jia Chen, Kuan-Hsun Chen, Björn Döbel, Rolf Ernst, Hermann Härtig, Andreas Heinig, Rüdiger Kapitza, Florian Kriebel, Daniel Lohmann, Peter Marwedel, Semeen Rehman, Florian Schmoll, and Olaf Spinczyk. Multi-layer software reliability for unreliable hardware. *it - Information Technology*, 57(3):170–180, June 2015. doi: 10.1515/itit-2014-1081

CONTENTS

1 INTRODUCTION 1
- 1.1 Motivation and Relevance 1
- 1.2 Objectives and Research Questions 3
- 1.3 Scientific Contributions 4
- 1.4 Outline of this Thesis 4
- 1.5 The Author's Contribution 6
- 1.6 Typographical Conventions 8

2 RELIABILITY OF COMPUTER HARDWARE 9
- 2.1 Transient Faults in Semiconductor Devices 9
  - 2.1.1 Sources and Composition of Natural Radiation 10
  - 2.1.2 Effects of Radiation on Silicon Transistors 11
  - 2.1.3 Single-Event Upsets in Memory Circuits 12
  - 2.1.4 Single-Event Transients in CMOS Logic 15
  - 2.1.5 Overview of Transient-Fault Rates 16
- 2.2 The Concept of Fault Tolerance 17
  - 2.2.1 Terms and Definitions 17
  - 2.2.2 Error Detection and Recovery 18
  - 2.2.3 Redundancy 19
- 2.3 Hardware-Implemented Fault Tolerance 22
  - 2.3.1 Device-Level Hardening 22
  - 2.3.2 Circuit-Level Hardening 22
  - 2.3.3 System-Level Hardening 23
- 2.4 Software-Implemented Fault Tolerance 23
  - 2.4.1 Error Detection Mechanisms 24
  - 2.4.2 Error Correction and Recovery Mechanisms 29
  - 2.4.3 Comparison of the Approaches 33
  - 2.4.4 Fundamental Limitations 36
- 2.5 Chapter Summary 37

3 PROBLEM ANALYSIS 39
- 3.1 The Methodology of Fault Injection 40
  - 3.1.1 Modeling of Hardware Faults 41
  - 3.1.2 Probability of Failure 42
  - 3.1.3 Exhaustive Fault-Space Scanning 44
- 3.2 Baseline Dependability Assessment 46
  - 3.2.1 Experimental Setup 47
  - 3.2.2 Case Study: eCos 48
  - 3.2.3 Case Study: L4/Fiasco.OC 53
- 3.3 Interpretation of the Results 55
- 3.4 Suggested Approach 56
- 3.5 Chapter Summary 56

4 ASPECT-ORIENTED PROGRAMMING 59
- 4.1 General Considerations 60
  - 4.1.1 Crosscutting Concerns 60
  - 4.1.2 AOP – Quantification and Obliviousness 62
  - 4.1.3 Language Support 62
  - 4.1.4 Weaving 63
- 4.2 The AspectC++ Language and Compiler 64
  - 4.2.1 Language Concepts 64
  - 4.2.2 Case Study: The CiAO Operating System 68
- 4.3 Related Aspect Languages 71
  - 4.3.1 AspectJ and Descendants 71
  - 4.3.2 AOP Extensions to C, C++, and C# 72
  - 4.3.3 AspectAda 73
- 4.4 Criticism of AOP 73
- 4.5 Prior Work on Fault Tolerance using AspectC++ 75
- 4.6 Chapter Summary 75

5 ASPECTC++ 2.0 – LANGUAGE EXTENSIONS 77
- 5.1 Advice for Built-in Operators 78
  - 5.1.1 Pointcut Function for Built-in Operators 78
  - 5.1.2 Example: Range Check of Function Pointers 79
- 5.2 Advice for Access to Variables 81
  - 5.2.1 Name Pointcut Expressions for Variables 81
  - 5.2.2 Pointcut Functions for Access to Variables 81
  - 5.2.3 Example: Bounds Check of Arrays 84
- 5.3 Generic Introductions 87
  - 5.3.1 Join-Point API of Introductions 87
  - 5.3.2 Example: Run-Time Type Checking 89
  - 5.3.3 Join-Point Template Library 91
- 5.4 Whole-Program Analysis 92
  - 5.4.1 Project Repository 93
  - 5.4.2 Example: Control-Flow Reachability Analysis 95
- 5.5 Chapter Summary 96

6 LIBRARY OF DEPENDABILITY ASPECTS 99
- 6.1 Design of the Library 100
- 6.2 Symptom Detection 102
  - 6.2.1 Checking of Pointers, Arrays, and Class Types 102
  - 6.2.2 Integer-Overflow Checking 103
- 6.3 Return-Address Protection 105
  - 6.3.1 Implementation 106
  - 6.3.2 Whole-Program Optimization 109
- 6.4 Virtual-Function Pointer Protection 111
  - 6.4.1 C++ Object Layout of GCC and LLVM/Clang 111
  - 6.4.2 Implementation 111
- 6.5 Generic Object Protection 115
  - 6.5.1 Generative Redundancy 118

6.5.2 Exploiting Object-Oriented Program Structure 122
6.5.3 Concurrent Error Detection 127
6.6 Chapter Summary 135

7 evaluation 137
7.1 Case Study: Hardening eCos 138
7.1.1 Symptom Detection 139
7.1.2 Return-Address Protection 142
7.1.3 Generic Object Protection 145
7.1.4 Combining the Dependability Aspects 149
7.2 Case Study: Hardening L4/Fiasco.OC 153
7.2.1 Effectiveness: Error Detection and Correction 154
7.2.2 Efficiency: Runtime 155
7.2.3 Memory Footprint: Static Binary Size 157
7.3 Case Study: Hardening Memcached 157
7.3.1 Experimental Setup 158
7.3.2 Effectiveness: Error Detection and Correction 161
7.3.3 Efficiency: Runtime 163
7.3.4 Memory Footprint 164
7.4 Chapter Summary 164

8 discussion 167
8.1 Reality Check: Neutron-Beam Testing 167
8.1.1 Experimental Methodology 168
8.1.2 Experimental Results 170
8.1.3 Interpretation of the Results 172
8.2 Software Maintainability 173
8.2.1 Scattering and Tangling in the Implementation 173
8.2.2 Separation of Concerns in the Compiler 175
8.3 Limitations 176
8.4 Future Work 176
8.5 Chapter Summary 177

9 summary and conclusions 179
9.1 Contribution 1: AspectC++ 2.0 – Language Extensions 180
9.2 Contribution 2: Library of Dependability Aspects 180
9.3 Contribution 3: Evidence for Effectiveness 181
9.4 Final Remarks 182

a appendix: baseline dependability assessment 183
a.1 eCos 183
a.2 L4/Fiasco.OC 188

b appendix: hardening ecos 189

List of Figures 203

List of Tables 205

bibliography 207

ACRONYMS

ABFT	Algorithm-Based Fault Tolerance
AOP	Aspect-Oriented Programming
API	Application Programming Interface
CFC	Control-Flow Checking
CMOS	Complementary Metal-Oxide-Semiconductor
CPU	Central Processing Unit
CRC	Cyclic Redundancy Check
DRAM	Dynamic Random-Access Memory
ECC	Error-Correcting Code
FIT	Failures In Time
GCC	GNU Compiler Collection
GNU	GNU's Not Unix!
GOP	Generic Object Protection
IP	Internet Protocol
IPC	Inter-Process Communication
JPTL	Join-Point Template Library
JVM	Java Virtual Machine
LANL	Los Alamos National Laboratory
LANSCE	Los Alamos Neutron Science Center
MMU	Memory Management Unit
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
RAII	Resource Acquisition Is Initialization
RAM	Random-Access Memory
RAP	Return-Address Protection
RISC	Reduced Instruction Set Computing
ROM	Read-Only Memory

SDC	Silent Data Corruption
SET	Single-Event Transient
SEU	Single-Event Upset
SIL	Safety Integrity Level
SRAM	Static Random-Access Memory
STU	Single Translation Unit
TMR	Triple-Modular Redundancy
TSO	Total Store Ordering
UDP	User Datagram Protocol
UML	Unified Modeling Language
VGA	Video Graphics Array
VLSI	Very-Large-Scale Integration
VPP	Virtual-Function Pointer Protection
XML	Extensible Markup Language

1 INTRODUCTION

Most people know that cosmic rays exist, yet, few people are aware that cosmic rays cause failures of computer systems at ground level. Recent incidents show that safety-critical computer systems often implement an insufficient degree of fault tolerance, even though human lives are at stake. As a remedy, the aspect-oriented programming language AspectC++ represents a suitable general-purpose technology for automatically improving the dependability of computer software – especially operating systems – and, thus, provides economical mitigation of radiation-induced hardware faults.

Thesis statement

1.1 MOTIVATION AND RELEVANCE

> “Reliability turns into *the* major design constraint for on-chip systems as scaling moves on. After almost a decade of research, the problems are far from being solved.”
>
> – Jörg Henkel and associates [104, p. 9]

The effects of cosmic rays manifest at ground level as ambient neutron radiation, of which a single particle suffices to disturb the electrical behavior of modern computer devices. Thus, computer devices occasionally exhibit transient faults that typically do not cause any permanent physical damage to the device. However, transient faults jeopardize correct program execution. For example, digital memory circuits are particularly susceptible to radiation-induced faults, which irrecoverably corrupt the stored information. Therefore, transient faults often turn into bit flips of computer memory. This problem worsens with the continuous downscaling of Complementary Metal-Oxide-Semiconductor (CMOS) technology [69, p. 488], as ever increasing amounts of memory are integrated on a single computer chip.

Section 2.1 discusses the radiation effects on semiconductor devices in detail.

Ignoring such faults in safety-critical systems causes catastrophic consequences. For example, recent legal proceedings [246] reveal that the electronic throttle control system of the Toyota Camry vehicle (model 2005) contains memory circuits without built-in error detection and correction. The software of that electronic control unit, however, implements duplicated program variables to detect certain errors, but the operating system is left out:

> “Inside this operating system . . . , we found that . . . critical data structures aren’t protected in any way. . . . a bit-flip there, will have the effect of killing one of the tasks.”
>
> – Michael Barr [246]

Therefore, if the software task (thread of control) responsible for regulating the engine speed is turned off abruptly, the vehicle can get out of control. In fact, this particular deficiency is considered as the root cause for several fatal car accidents due to unintended acceleration [246].

To prevent such incidents, adequate methods of *fault tolerance* are essential, which can be implemented at the hardware or software level or both. As an example of the former, radiation-hardened CMOS technology [153, 89] and memory cells [40, 103] are typically used in aerospace systems; high-end computer systems contain built-in circuits for the detection and correction of memory errors [128, 117, 170], in addition to redundant processors that operate in lockstep [226].

> " Many of these solutions have, however, been prohibitively expensive and difficult to justify in the mainstream commodity computing market. "
>
> – Shubu Mukherjee [172, p. 2]

For that reason, much research focuses on fault tolerance at the software level to compensate for unreliable, yet inexpensive, hardware. A particularly successful approach is redundant multi-threading [251, 221, 70]; that is, application programs are executed repeatedly and synchronized on operating-system calls, on which the individual program outputs are checked for consistency. Such an approach, however, does not cover the operating system itself.

Another solution is represented by special-purpose compilers that automatically apply instruction-level duplication [25, 200, 184, 176] and error-detecting codes [182, 138]. These compilers tend to become so complex that none of them works correctly for multi-threaded programs. For example, compiler transformations that duplicate program variables provoke race conditions because the variables cannot be written atomically anymore. Thus, consistency checks can fail if a concurrent thread overwrites the original and duplicate variables one after the other. As operating-system code is typically concurrent and asynchronous due to interrupt processing, these compiler-based solutions are also not applicable to operating systems.

It turns out that there is a rich body of literature on fault tolerance of user-level programs (see Section 2.4), whereas operating systems receive less attention – just as in the case of Toyota. However, the operating system is the most important piece of software, because all user-level programs depend on it. A crash of the operating system thwarts any user-level fault tolerance. Thus, the operating system represents a critical single point of failure.

1.2 OBJECTIVES AND RESEARCH QUESTIONS

The aforementioned state-of-the-art approaches to *automatic* fault tolerance at the software level do not cover operating systems. This problem can be approached by manually implementing a special-purpose operating system from scratch with built-in transient error detection [110]. However, the resulting source code becomes highly tangled with error-detection functionality, which is scattered over the source-code files of that operating system. Such an approach severely impairs the maintainability of the source code.

The objective of this thesis is to investigate a semiautomatic solution that can be applied *ex post* to existing operating systems without modification of any source code. This thesis evaluates whether Aspect-Oriented Programming (AOP) [131] is suitable for achieving this objective. In brief, AOP is a general-purpose programming technology that provides extra language support for the transparent extension of existing programs by user-defined functionality, which can be specified as a separate program module. A compiler for the AOP language extends the existing program by the specified functionality. Thus, it might become feasible to integrate fault-tolerance mechanisms into operating systems in an automated and transparent way.

Chapter 4 elaborates on AOP.

Several studies [90, 8, 5, 7, 126] indicate that AOP is a promising technology for implementing fault tolerance at the application level. To the best of my knowledge, only Afonso and associates [4] consider AOP for checking a specific precondition of a semaphore data structure of an operating system. Therefore, the question whether AOP is suitable for improving the fault tolerance of operating systems at all is not yet completely answered. Likewise, the extent to which AOP can improve fault tolerance is still poorly understood. In summary, this thesis addresses the following research questions:

QUESTION 1: Are contemporary AOP languages suitable for improving the fault tolerance of existing operating systems? If the languages are not applicable or not expressive enough, can they be extended to achieve this goal? Which AOP language features are essential in this regard?

QUESTION 2: Which steps are required to support asynchronous and multi-threaded operating-system code? Is there a generic and reusable solution that can be applied to various operating systems without affecting maintainability?

QUESTION 3: How effective and how efficient is an AOP-based solution? Is there evidence that the solution provides protection from radiation-induced hardware faults? What are the limits of the approach?

1.3 SCIENTIFIC CONTRIBUTIONS

This thesis provides answers to the research questions stated in the previous section. To this end, I focus on the aspect-oriented programming language AspectC++ [231] as motivated in Chapter 4. I evaluate the suitability of that language for improving the fault tolerance of two off-the-shelf operating systems: the embedded configurable operating system (eCos) [162] and the L4/Fiasco.OC microkernel [139]. Both operating systems are used in a broad range of production systems with dependability constraints (see Section 3.2). In summary, this thesis advances the state-of-the-art by making the following scientific contributions:

CONTRIBUTION 1: By analyzing the previously mentioned operating systems, I identify missing language features of AspectC++ that are essential for achieving fault tolerance. Subsequently, I propose and specify four distinct language extensions that led to the AspectC++ 2.0 language and compiler.

CONTRIBUTION 2: I describe the design and implementation of a library of generic fault-tolerance mechanisms using the AspectC++ 2.0 technology. This library can be applied automatically to protect the most critical data structures of both operating systems. I present a wait-free synchronization algorithm and correctness proof that enables support for asynchronous and multi-threaded operating-system code.

CONTRIBUTION 3: Finally, I evaluate the AspectC++ 2.0 technology and library of fault-tolerance mechanisms on the basis of both operating systems and one additional user-level program. Exhaustive fault-injection experiments show an excellent tradeoff between runtime overhead and fault tolerance. I validate the findings by high-energy particle radiation experiments conducted at Los Alamos National Laboratory, USA, which attest effective improvements in fault tolerance by almost 80 percent.

1.4 OUTLINE OF THIS THESIS

This thesis is structured in nine chapters.

CHAPTER 2: *Reliability of Computer Hardware* (pages 9–38)
The second chapter provides an introduction to radiation effects on semiconductor devices and summarizes the frequency of transient faults that affect contemporary memory technologies. Subsequently, the chapter briefly introduces the principles and terminology of *fault tolerance* and, thereafter, establishes a taxonomy of software-implemented fault tolerance that represents the *related work* of this thesis. The chapter concludes with

an evaluation of the related work and points out several shortcomings in the *state-of-the-art*.

CHAPTER 3: *Problem Analysis* (pages 39–57)
In general, the methodology of *fault injection* is fundamental to the assessment of fault tolerance. Thus, I apply this methodology to examine the memory segments of the two operating systems eCos and L4/Fiasco.OC. In short, the third chapter analyzes the failure mode, effects, and criticality of each individual memory location of both operating systems. On that basis, the chapter formulates a *suggested approach* to selective protection of the most critical data.

CHAPTER 4: *Aspect-Oriented Programming* (pages 59–76)
The fourth chapter introduces AOP and reviews several AOP languages. Furthermore, the chapter presents a case study on AspectC++ and evaluates the suitability of that language for the domain of dependable operating systems. After recapitulating frequent points of criticism concerning AOP, the chapter examines *prior work* on fault tolerance using AspectC++ and concludes with the need for several language extensions.

CHAPTER 5: *AspectC++ 2.0 – Language Extensions* (pages 77–97)
In the fifth chapter, I propose and specify four distinct language extensions to AspectC++. These extensions remedy the deficiencies of prior work on fault tolerance using AspectC++ and, thus, represent the essential building blocks for the development of highly generic fault-tolerance mechanisms. The chapter illustrates each language extension by functional examples, such as checking of pointer ranges, checking of array bounds, and runtime type checking.

CHAPTER 6: *Library of Dependability Aspects* (pages 99–135)
The sixth chapter presents the design and implementation of four highly generic mechanisms for the detection and correction of memory errors in operating systems. These mechanisms use the AspectC++ 2.0 technology to achieve complete transparency: The target operating systems need not be aware of – and not prepared for – these mechanisms. To this end, the mechanisms support asynchronous and multi-threaded code by wait-free synchronization, which is also addressed in that chapter.

CHAPTER 7: *Evaluation* (pages 137–165)
I evaluate the approach of this thesis in the seventh chapter, which comprises case studies on eCos, L4/Fiasco.OC, and one user-level application. Each case study involves extensive fault-injection experiments to evaluate the fault tolerance of the three software systems. In addition, the chapter measures the effi-

ciency of the aspect-oriented fault-tolerance mechanisms regarding runtime overhead and memory footprint.

CHAPTER 8: *Discussion* (pages 167–178)
The eighth chapter discusses the validity of the results obtained from fault injection. For that purpose, the chapter confirms the results by neutron-beam testing at Los Alamos National Laboratory, USA. Subsequently, the chapter examines the software maintainability of the approach taken in this thesis. After recapitulating the fundamental limitations of the aspect-oriented approach, the chapter outlines directions for future work.

CHAPTER 9: *Summary and Conclusions* (pages 179–182)
The final chapter summarizes and reviews my work and concludes this thesis.

Two appendices succeed the nine main chapters and present supplemental fault-injection results.

APPENDIX A: *Baseline Dependability Assessment* (pages 183–188)
The first appendix presents further fault-injection results of the baseline dependability assessment of eCos and L4/Fiasco.OC as described in Chapter 3.

APPENDIX B: *Hardening eCos* (pages 189–202)
The second appendix shows the complete fault-injection results of all benchmarks used in the case study on eCos in Chapter 7.

1.5 THE AUTHOR'S CONTRIBUTION

According to §10(2) of the examination regulations of the department of computer science, TU Dortmund, 2011, a doctoral dissertation must include a separate list of the author's contribution to the scientific results that were obtained in cooperation with other persons. As such, the following list provides a chapter-wise overview of my contributions to results that were obtained in cooperation.

CHAPTER 3: The third chapter presents the reliability metric for fault injection used throughout this thesis. I developed the foundations of that metric in cooperation with Horst Schirmeier and we published the results [214]. In Section 3.1.2, I present an improved formalization of that reliability metric. The fault-injection results of eCos that are shown in Section 3.2.2 are parts of three publications [30, 31, 32], of which I am the principal author. My contribution to these fault-injection results is the setup of eCos and the interpretation of the results.

CHAPTER 4: The case study on the CiAO operating system in Section 4.2.2 is based on a publication of Martin Hoffmann and colleagues [109], of which I am a coauthor. My contribution is the setup of eCos (for comparison) and the design and implementation of the network stack of the CiAO operating system as described in another publication [28], of which I am the principal author.

CHAPTER 5: I developed the concepts of the AspectC++ language extension by *advice for built-in operators* (see Section 5.1). The integration of that language feature into the AspectC++ compiler is a result of the bachelor thesis of Simon Schröder, whom I supervised. Likewise, I developed the language extension by *advice for access to variables* (see Section 5.2), and the integration into the AspectC++ compiler is a result of the diploma thesis of Markus Schmeing, whom I supervised as well. The third language extension by *generic introductions* (see Section 5.3) is presented in a publication [26] of which I am the principal author. I developed the concepts of that language feature in cooperation with my adviser Olaf Spinczyk, who was involved in many details of the implementation in the AspectC++ compiler.

CHAPTER 6: The library of *dependability aspects* – that is, generic fault-tolerance mechanisms based on AspectC++ – is partly described in five publications [29, 30, 31, 26, 32], of which I am the principal author. As such, my contribution is the whole idea, design, and implementation of all these dependability aspects. Moreover, the wait-free synchronization algorithm described in Section 6.5.3 has been quoted verbatim with permission from an IEEE journal publication [32]; the quoted passages have been written exclusively by myself.

CHAPTER 7: First, the evaluation of the eCos operating system in Section 7.1 is partly described in three publications [30, 31, 32], of which I am the principal author. However, I completely re-evaluated the eCos operating system in this thesis using improved versions of all the dependability aspects and the AspectC++ compiler. Second, the evaluation of the L4/Fiasco.OC microkernel is based on a publication [26] of which I am the principal author. I carried out all the fault-injection experiments. Third, the hardening of the user-level program Memcached is based on the work of Arthur Martens and our publication [160]. My contribution are the dependability aspects and the concept of the ptrace-based fault-injection tool, which was implemented as a prototype during the bachelor thesis of Christopher Kukkel, whom I supervised.

CHAPTER 8: The neutron-beam testing at Los Alamos National Laboratory was carried out by Paolo Rech and colleagues [211]. My contribution to that work is the setup and hardening of eCos.

1.6 TYPOGRAPHICAL CONVENTIONS

Margin notes highlight important terms or provide supplemental information, such as cross-references.

Throughout this thesis, I use the *italic* font shape to emphasize a particularly important term or fragment of a sentence. Source code is typeset in a `monospace` font; in-text references to source-code identifiers are also typeset that way. The labels of descriptions and headers of tables are depicted by SMALL CAPS. The same typesetting applies to benchmark programs with cryptic names, such as BIN_SEM2.

2 RELIABILITY OF COMPUTER HARDWARE

> “As the dimensions and operating voltages of computer electronics are reduced to satisfy the consumer's insatiable demand for higher density, functionality, and lower power, their sensitivity to radiation increases dramatically.”
>
> – Robert Baumann [24, p. 258]

In short, the goal of this thesis is to evaluate AOP as a means to improve the dependability of computer operating systems. In particular, this thesis focuses on mitigation of hardware faults that are induced by the natural radiation environment. First, this chapter provides an introduction to radiation effects on semiconductor devices in Section 2.1, including various memory technologies and the core logic of a CPU. This background information is essential for the development of effective fault-tolerance mechanisms, as discussed in the following chapters. For example, Chapter 8 describes high-energy particle radiation experiments conducted at Los Alamos National Laboratory, USA, to evaluate the fault-tolerance mechanisms developed in this thesis. For that reason, Section 2.1 deals with the physics of semiconductors exposed to radiation.

Second, this chapter briefly introduces the principles and terminology of *fault tolerance* in Section 2.2. The terminology defines the vocabulary that is important for understanding this thesis.

Third, Section 2.3 reviews the benefits and costs of hardware solutions for fault tolerance. Subsequently, Section 2.4 elaborates on software solutions that provide low-cost alternatives. That section discusses the *related work* of this thesis and concludes with a comparative study of the *state-of-the-art* in software-implemented fault tolerance.

Finally, Section 2.5 summarizes the findings of this chapter.

2.1 TRANSIENT FAULTS IN SEMICONDUCTOR DEVICES

> “Intermittent and transient faults are expected to represent the main source of errors experienced by VLSI circuits.”
>
> – Cristian Constantinescu [57, p. 18]

The reliability of digital semiconductor devices is confronted with transient faults that disturb the electrical behavior of the devices. Transient faults are caused by a multitude of sources, such as transistor variability during fabrication [33], fluctuation of the supply voltage [57, 33], temperature variation affecting transistor switching

Sources of transient faults

speed [33], electromagnetic interference [57, 230], and ionizing radiation [259, 258, 23]. In addition, several aging effects cause a gradual degradation of semiconductor devices, including hot-carrier injection [158], negative bias temperature instability [158], electromigration [57], and time-dependent dielectric breakdown [158], all of which typically start out as intermittent faults and turn into permanent failure over time [57, 223].

Most of these problems relate to the semiconductor fabrication process, whereas transient faults induced by ionizing radiation depend on the natural radiation environment. The following sections focus on radiation-induced faults, "because, uncorrected, they produce a failure rate exceeding that of all other reliability mechanisms combined." [24, p. 260]

2.1.1 *Sources and Composition of Natural Radiation*

The natural radiation environment varies with altitude and, thus, differs between the Earth's atmosphere and ground level. Above the Earth's upper atmosphere, *primary cosmic rays* arrive from outer space. Primary cosmic rays mostly consist of positively charged particles: 93 percent protons, 6 percent alpha particles, and a remainder of heavier elements [125, p. 130]. They arrive in large quantities of about 36,000 particles per cm^2 per hour [172, p. 22].

Once the galactic particles enter the atmosphere, they collide with air nuclei and produce cascades of secondary particles by nuclear spallation reactions [125, p. 129]. The air nuclei thereby emit charged particles and neutrons with high kinetic energy, which in turn can strike other air nuclei once again. As these cascades travel down to the Earth's surface, the charged particles constantly lose energy to the electrons of the air molecules and then get lost from the cascades. Thus, more than 95 percent of the particles that finally reach the ground level are neutrons, which penetrate the air without interaction of electrons [258]. The resulting particle flux at sea level, for example in New York City, amounts to about 13 high-energy neutrons per cm^2 per hour [118, p. 56].

High-energy neutrons

Besides neutrons from the atmosphere, radioactive impurities in semiconductor devices emit charged alpha particles during alpha decay. For example, even highly purified chip materials satisfying the *ultra-low alpha* grade emit about 0.001 alpha particles per cm^2 per hour [134, p. 207], which corresponds to less than one radioactive atom per 10 billion [23, p. 307]. A further reduction of radioactive emission is considered as extremely difficult [125, 172] and prohibitively expensive [23]. Alpha particles from the natural radiation environment are shielded by the chip packaging and cannot reach the silicon transistors – only those few alpha particles originating from residual radioactive activity in the device itself are of concern.

Alpha particles

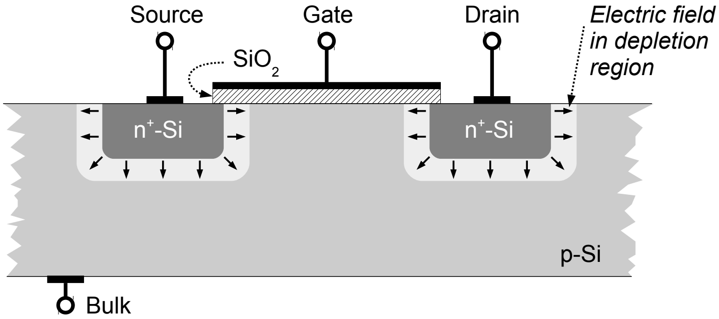


Figure 2.1: Schematic diagram showing the cross section through a MOSFET. The *electric fields* across the p-n junctions arise from different chemical potential of the n-type silicon (*n^+-Si*) and the p-type substrate (*p-Si*). The resulting *depletion region* is the most radiation-sensitive area of the transistor.

2.1.2 *Effects of Radiation on Silicon Transistors*

Neutrons with high kinetic energy penetrate a semiconductor device as easily as the Earth's atmosphere and cannot be shielded effectively, except by several meters of concrete [23, p. 309]. When a high-energy neutron passes through a semiconductor device, the neutron can collide with the silicon nuclei causing a nuclear spallation reaction. The silicon nucleus emits thereby additional neutrons, protons, and alpha particles [23, p. 308]. It is only these *secondary*, charged protons and alpha particles that disturb the electrical behavior of a silicon transistor in addition to the *primary* alpha particles emitted from the chip material.

Figure 2.1 depicts the cross section through a Metal-Oxide-Semiconductor Field-Effect Transistor (MOSFET) with p-type silicon substrate (bulk). A thin layer of silicon dioxide insulates the gate terminal. The source and drain terminals are each connected to implanted regions of highly doped n-type silicon (*n^+-Si*). At the junction between the p-type and n-type silicon, the different chemical potential forces several electrons from the n-type silicon into the p-type substrate to reach thermodynamic equilibrium. This once-only charge transfer forms a static electrical field spanning a *depletion region* (see Figure 2.1) across the p-n junctions in which no mobile charge carriers remain.

Depletion region

When a charged particle – alpha or proton, produced indirectly from a neutron or directly from radioactive decay – penetrates the silicon lattice, the particle frees electrons from the silicon atoms along its moving direction [125, p. 130f]. Such electrons leave ionized atoms behind, containing *holes* where electrons are missing. Figure 2.2 (❶) illustrates a cylindrical ion track of electron-hole pairs caused by a charged particle.

The electron-hole pairs, altogether, are neutral, unless they occur in the depletion regions across the p-n junctions. There, the electric field rapidly collects free electrons and makes them drift into the n-type

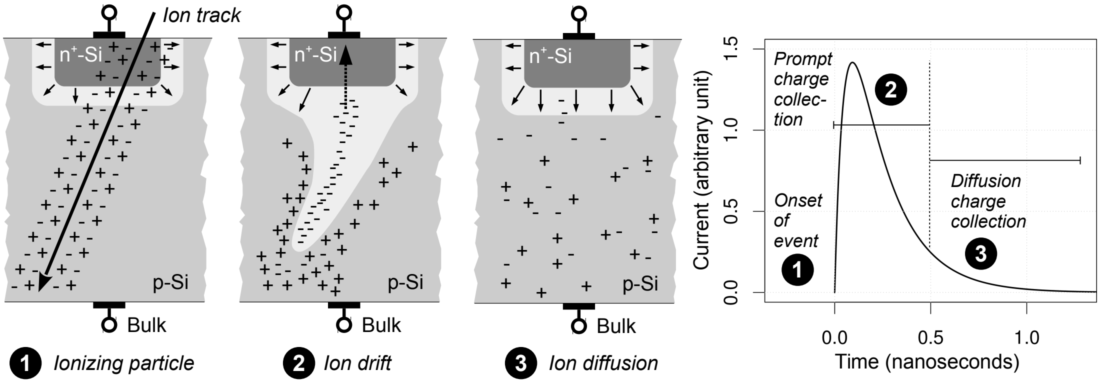


Figure 2.2: Particle strike at the depletion region of a MOSFET. The ionizing particle creates a cylindrical track of electron-hole pairs (❶), which are separated by the electric field at the p-n junction. Electrons drain off via the upper terminal, whereas holes flow to the bulk terminal (❷). After most charge carriers have been collected, some electrons slowly diffuse into the n^+ region (❸). The upper terminal registers a current pulse, as shown in the rightmost diagram. Adapted from Baumann [24, p. 259].

silicon, eventually flowing off via the connected terminals (source or drain). Figure 2.2 (❷) shows the electron drift, which leaves holes behind and, thus, temporarily extends the electric field deeper into the substrate [24, p. 259], collecting more and more electrons. The remaining holes flow to the bulk terminal and cause no further harm.

After sufficient charge carriers have drifted away, the electric field shrinks to its initial extent, as indicated in Figure 2.2 (❸). In the meantime, residual electrons slowly diffuse into the depletion region and continue with the charge collection until no free electrons remain.

Charge collection

The diagram on the right-hand side of Figure 2.2 shows the resulting electric current generated during the three phases (❶ to ❸). At the source or drain terminal of the struck transistor, the depicted current pulse appears for a few nanoseconds. If the current pulse transfers enough charge, the circuit elements connected to the affected transistor terminal can be disturbed. For example, if the collected charge exceeds the amount of charge stored in a memory circuit, the stored information is lost.

2.1.3 *Single-Event Upsets in Memory Circuits*

Single-Event Upset (SEU) [96] denotes the information loss of a memory cell caused by particle-induced electric current. This electric current exists only for a few nanoseconds, yet it can cause a persistent disturbance of one or more bits in the memory circuit. Such events are referred to as *soft errors*, because the error can be resolved by overwriting the affected memory cells with new data, making the cells usable thereafter again. However, the information stored prior to the particle strike is lost.

This section quantifies the susceptibility to SEUs of contemporary CMOS memory technologies.

2.1.3.1 *SRAM*

> “ The majority of observed SRAM faults in the field are transient faults from particle strikes. ”
>
> – Vilas Sridharan and associates [235, p. 298]

The cache memory and the register file of most CPUs are implemented in Static Random-Access Memory (SRAM) technology [117, p. 257ff]. A standard 1-bit SRAM cell consists of two cross-coupled CMOS inverters and two access transistors, needed when the cell is read or written to. The inverter pair implements a regenerative feedback loop that preserves one bit of information by voltage differential between the outputs of both inverters. Each inverter comprises two CMOS transistors.

When the SRAM cell is in storage mode, that is, it is not read or written to, the four transistors of the inverters are highly sensitive to particle strikes. A radiation-induced voltage glitch at one transistor propagates into the inverting feedback loop, and if the opposite inverter responds before the particle-induced current runs off, the SRAM cell flips its stored voltage differential [71, p. 588f]. The slow ion-diffusion phase shown in Figure 2.2 (❸) is extremely critical for SRAM cells, giving the opposite inverter plenty of time to latch the disturbed voltage. Hence, an observable SEU manifests as *bit flip*, with both $0 \mapsto 1$ and $1 \mapsto 0$ transitions being equally probable [16, p. 2262].

SRAM fault rate

The fault rate of SRAM amounts to the order of 10^{-4} to 10^{-2} FIT[1]/bit at sea level (New York City) and has remained almost constant while technology scaled from the feature size of 250 nm to 65 nm [225]. However, further downscaling from 65 nm to 40 nm increases the fault rate for individual SRAM cells by about 30 percent [69, p. 488].

Real-time measurements confirm that SEUs occur randomly distributed over the area of large SRAM arrays [16, p. 2262], but a single particle can strike multiple physically adjacent cells. Autran and associates [16, p. 2262] report that, for 65-nm cells, 25 percent of the SEUs affect 2 to 7 physically adjacent bits, and the remaining 75 percent cause single bit flips. Dixit and Wood [69, p. 491] show that 40-nm cells exhibit multiple bit upsets with a relative frequency of almost 40 percent. Thus, as the cell geometry shrinks and the density of transistors per area increases, multiple bit upsets tend to become more frequent.

[1] The FIT (Failures In Time) rate measures the number of failures per 10^9 hours of operation.

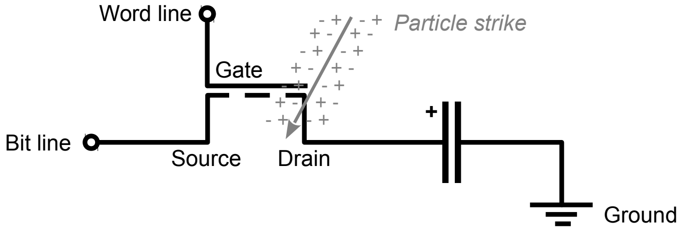


Figure 2.3: Equivalent circuit diagram for a 1-bit DRAM cell. The capacitor stores one bit of information as electric charge. Particle strikes to the access transistor can directly disturb the stored charge of the connected capacitor and result in SEU.

2.1.3.2 *DRAM*

Dynamic Random-Access Memory (DRAM) [117, p. 353ff] is the dominant technology for main memory in terms of storage density. One bit of information is stored as electric charge in an integrated capacitor, insulated by one access transistor (see Figure 2.3). In contrast to the SRAM cell, there is no active regeneration of the stored charge that leaks through the capacitor over time. Thus, the passively stored charge of the DRAM cell must be refreshed every few milliseconds by a read-out and rewrite procedure.

Figure 2.3 shows that the storage capacitor is directly connected to the drain terminal of the access transistor. Thus, particle-induced charge collection at that drain region (see Section 2.1.2) directly disturbs the stored electric charge of the capacitor. To cause an SEU, it suffices to degrade the stored charge to a level outside the noise margin – there is no need to flip the entire electric state.

A particle strike can easily relax the stored charge [164, p. 4], and if a particle passes through both the source and drain region of the access transistor, the capacitor can also be charged accidentally [71, p. 587]. Thus, *bit flips* occur in both directions.

DRAM fault rate

Several large-scale studies quantify the fault rate of contemporary DDR-2 and DDR-3 DRAM devices [150, 232, 234, 235]. These four studies agree on a fault rate in the order of 10^{-8} FIT/bit. Compared to SRAM, the per-bit fault rate is notably smaller, mostly because an individual DRAM cell occupies less chip area, thereby reducing the chance of being hit by an incident particle. The high density of DRAM cells, on the other hand, increases the frequency of multiple bit faults, ranging from 21 percent [235, p. 301] up to 50 percent [232]. This is in line with laboratory tests, reporting an average number of 1.0 to 1.5 cell upsets per neutron strike [34]. Considering the spatial and temporal distribution, Sridharan and associates conclude:

> “DRAM faults in our system are consistent with a uniform random distribution of faults in a device, implying that DRAM faults are equally likely to occur in any region of a DRAM device.”
>
> – Vilas Sridharan and associates [234, p. 6]

2.1.3.3 *Flash*

Computer systems without hard-disk drive usually use nonvolatile flash memory [42] to store persistent data such as program code. A flash memory cell consists of a single transistor like the one depicted in Figure 2.1, with the distinction that a floating gate – a piece of metal not connected electrically – is embedded into the silicon-dioxide layer beneath the gate terminal. The floating gate, surrounded by oxide, stores information as electric charge that can be placed into the floating gate by *hot electron injection* or *Fowler-Nordheim tunneling* of electrons through the isolating oxide [42, p. 153ff]. Once stored in the floating gate, the electric charge remains insulated from the depletion regions of the transistor (see Figure 2.1). Thus, charge collection at the transistor's depletion regions does not cause SEUs in flash memory.

In the programmed cell state (binary zero), the electrons stored in the floating gate form an electric field across the surrounding oxide. Hence, electron-hole pairs generated by a particle strike can be separated by that electric field, which redirects a few hundred electrons out of the floating gate into the oxide [39]. This charge loss can relax the programmed reference charge of about 1,000 electrons [39, p. 553] to a level outside the noise margin, causing an SEU with $0 \mapsto 1$ transition. An erased memory cell (binary one), on the other hand, is virtually immune to SEU due to the absence of an electric field around the floating gate [44].

Flash memory fault rate

Several studies report a fault rate without Error-Correcting Code (ECC) in the order of 10^{-8} FIT/bit [86, 44, 122]. Unlike DRAM and SRAM cells, flash memory cells exhibit a degradation of the gate oxide with repeated program–erase cycling [42, p. 399ff]. Therefore, an array of flash memory cells typically contains redundant cells that implement a strong ECC, such as the correction of 8 defect cells out of 539 bytes [91, p. 401]. Thus, for flash memory, "the atmospheric-neutron induced failure rate is still (and is expected to remain so for the foreseeable future) within the correcting capabilities of current ECC algorithms." [44, p. 4]

2.1.4 *Single-Event Transients in CMOS Logic*

> "Dealing with logic soft errors is less critical since there are fewer logic gates than memory cells and many of the logic errors are masked."
>
> – Charles Slayman [225, p. 4]

The core logic of a CPU consists of sequential elements, such as flip-flops and latches, connected to combinational logic circuits. Flip-flops and latches are like SRAM cells (see Section 2.1.3.1) and exhibit a similar SEU rate per individual node [69, p. 486].

Combinational logic gates, on the other hand, forward any particle-induced electric charge to their outputs, causing a Single-Event Transient (SET). An SET is a short-lived noise pulse that propagates to subsequent gates. Only if the SET is captured by a subsequent flip-flop or latch, the SET turns into an SEU (bit flip). For this to happen, there must be an active path from the struck gate to a storage element without any *logical masking* in between. For example, if the SET arrives at the input of a NOR gate, but the other input is already high (1), the SET is irrelevant for the output of the NOR gate and gets masked. Furthermore, the SET must arrive within the setup and hold time of a flip-flop, which captures incoming signals only on one clock edge. Otherwise, *temporal masking* of the SET takes place. Finally, *electrical masking* [151] attenuates the strength of the noise pulse with the number gates it traverses, so that an SET may disappear before reaching a storage element.

Masking of SETs

Because of these masking effects, only a small fraction of SETs eventually corrupts the state of a flip-flop or latch [151, 208, 92]. Even if an SEU occurs at a flip-flip or latch, the corrupted state is in turn subject to logical masking in the following clock cycle. For instance, Cho and colleagues [53] find that 68.86 percent of flip-flop SEUs are logically masked. Altogether, Slayman [225, p. 4] estimates the frequency of logic errors to be ten times lower than for SRAM.

2.1.5 *Overview of Transient-Fault Rates*

Table 2.2 summarizes the individual fault rates of CMOS circuits exposed to atmospheric radiation at sea level. Besides the fault rate per bit or logic gate, the quantity of those circuit elements defines the total impact on chip-level reliability. Typically, the chip area of a microprocessor is dominated by memory cells [117, p. 14]. Thus, the various memory technologies are the most important factor limiting the overall system reliability.

CIRCUIT	FAULT RATE	EFFECTS	MULTI-BIT UPSETS
SRAM	10^{-4} to 10^{-2} FIT/bit	Bit flip	25 % to 40 %
DRAM	10^{-8} FIT/bit	Bit flip	21 % to 50 %
Flash	10^{-8} FIT/bit	$0 \mapsto 1$ bit flip	Unknown
Logic	10× less than SRAM	Bit flip or CPU failure	Frequent [53]

Table 2.2: Overview of fault rates caused by atmospheric radiation at sea level

For instance, the impact of 1 MiB SRAM cache and 10 GiB DRAM is about 850 FIT each.[2] The ever-increasing chip complexity and memory size, as predicted by Moore's Law, worsens this reliability problem. To put these numbers into perspective, a study reports that the aggregated DRAM of a computing cluster already causes "one failure approximately every six hours" [232, p. 5].

2.2 THE CONCEPT OF FAULT TOLERANCE

> "Our increasing dependence on computing systems brings in the requirement for fault tolerance."
>
> – Algirdas Avižienis and associates [18, p. 29]

Fault tolerance is the only mean for building a dependable system from unreliable components. The reliability of the whole system can *improve* thereby compared to the reliability of the individual components. In general, fault tolerance introduces redundancy to compensate for faults, thus, involving more resources than strictly necessary for the fault-free case.

In a computer system, *hardware fault tolerance* covers hardware faults, whereas *software fault tolerance* addresses software defects [74, p. 24]. This distinction is orthogonal to the differentiation between *hardware-implemented fault tolerance* (Section 2.3) and *software-implemented fault tolerance* (Section 2.4) [74, p. 24]. Based on that classification, this thesis focuses on *software-implemented hardware fault tolerance (SIHFT)*. The following sections introduce the basic terminology to elaborate on the concept of fault tolerance.

2.2.1 *Terms and Definitions*

Avižienis and associates [18] define the fundamental terms *fault*, *error*, and *failure* in the following meaning:

FAULT: A *fault* is the "adjudged or hypothesized cause of an error" [18, p. 13]. If a fault gets *activated*, it influences the state of a system. Otherwise, a fault can occur but stays *dormant*.

ERROR: An *error* is the deviation of a system's internal state because of an activated fault. If an error is irrelevant for subsequent system states, it gets *masked*.

FAILURE: The propagation of an internal error to the system interface leads to a service *failure* – a deviation from correct behavior. A failure can cause faults in other systems that depend on the failed service.

[2] The sample calculation assumes an SRAM fault rate of 10^{-4} FIT/bit.

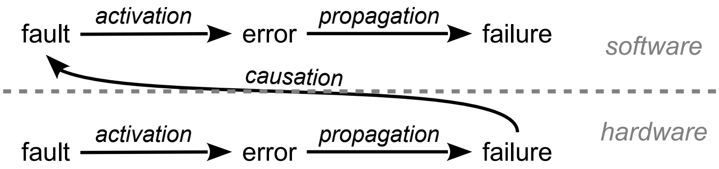


Figure 2.4: The pathology of failure in hardware and software layers

Figure 2.4 illustrates the relationship between these terms in a layered architecture. The terms fault, error, and failure have different meanings in the hardware layer and software layer, respectively. For example, hardware-implemented fault tolerance expects radiation-induced noise (faults), corrects errors in the data path of the CPU and, thus, avoids bit flips in memory (failures). On the other hand, software-implemented fault tolerance assumes bit flips in memory (faults), corrects errors in program variables, and prevents thereby application crashes (failures). Hence, a bit flip in memory is a failure of the hardware *and*, at the same time, causes a fault for the software layer, so that the interpretation of these terms depends on the context in which they are used.

The term *soft error* originates from the hardware perspective and precisely describes the activation of a transient physical fault. A soft error that further propagates to the hardware-software interface manifests as bit flips (SEUs). Thus, a soft error also refers to unexpected bit flips observed by the software layer.

2.2.2 *Error Detection and Recovery*

Fault tolerance denotes the *detection of errors* along with *recovery from errors* to avoid service failures [18, p. 24]. Error detection can take place either *concurrently* during normal operation, or the system is suspended for *preemptive* detection. Recovery is the corrective action after an error has been detected and subsumes the following measures [18, p. 25]:

ROLLBACK: Restoring the system to a previously saved state (checkpoint) that is assumed error free.

ROLLFORWARD: Bringing the system to a valid, error-free state by exploiting application-specific knowledge.

COMPENSATION: Using redundancy to mask the error (preventing error propagation) and to correct the erroneous system state.

Hence, *redundancy* is an integral part of each recovery measure and, thus, a fundamental requirement for fault tolerance.

2.2.3 *Redundancy*

Protective redundancy refers to those resources explicitly allocated for fault tolerance, whereas *unintentional* redundancy may also contribute to fault tolerance [18, p. 20]. Considering protective redundancy, Echtle [74, p. 51ff] describes four characteristic features:

STRUCTURAL REDUNDANCY: Additional components, for example replicas of existing components, are structurally redundant.

FUNCTIONAL REDUNDANCY: Additional functionalities, such as algorithms implementing fault tolerance, provide functional redundancy.

INFORMATION REDUNDANCY: Data encoding with more bits than strictly necessary, or data that can be reconstructed from other sources, implements information redundancy.

TIME REDUNDANCY: A system with time redundancy provides slack time that is not used for the actual functionality, but which can be used for the execution of fault-tolerance mechanisms.

In general, a fault-tolerance mechanism requires each kind of redundancy to a certain degree. For example, the *Hamming code* [98] is an error-correcting code that adds several extra bits to a datum to allow for the correction of a single bit error. Thus, the additional bits increase the demand on physical storage (*structural redundancy*) and require algorithms to encode and decode the data (*functional redundancy*). A corrupt bit can be reconstructed (*information redundancy*) and the data encoding takes some amount of time (*time redundancy*).

Section 2.3 and Section 6.5.1.2 on page 120 pick up on the Hamming code, which is therefore introduced in the following example.

Example: Hamming Code

The Hamming code [98] is a linear block code that enables the detection and correction of a single bit error by information redundancy. Therefore, k check bits are appended to n data bits so that $n \leq 2^k - k - 1$ holds. Thus, for each number n, there is a *Hamming(n+k, n)* code that encodes n data bits into a code word of the length n+k. A code is *separable* if the data bits are preserved and not intermixed with the check bits [136, p. 57].

The individual check bits are computed by arithmetic over the finite field of two elements, that is, integer arithmetic modulo 2. In other words, addition corresponds to the logical `XOR` operation, and multiplication is equivalent to the logical `AND` operation. Then, a Hamming code is a linear mapping that can be represented by a transformation matrix. A bit vector $\vec{x}$ is mapped to a code word $\vec{c}$ by multiplication with the transformation matrix G:

Section 6.5.1.2 on page 120 takes up the matrix notation.

$$G \cdot \vec{x} \;=\; \vec{c}$$

For example, the shown matrix G encodes four bits of data $x_1, \ldots,$ x_4 into a separable Hamming(7,4) code as follows:

$$\underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} \end{pmatrix}}_{G} \cdot \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \underbrace{\begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \\ \mathbf{1}x_1 + \mathbf{1}x_2 + \mathbf{0}x_3 + \mathbf{1}x_4 \\ \mathbf{1}x_1 + \mathbf{0}x_2 + \mathbf{1}x_3 + \mathbf{1}x_4 \\ \mathbf{0}x_1 + \mathbf{1}x_2 + \mathbf{1}x_3 + \mathbf{1}x_4 \end{pmatrix}}_{\vec{c}}$$

The upper part of G consists of the identity matrix, which maps the four data bits to the first four elements of $\vec{c}$. The lower three rows of G contribute to the three check bits and are printed in bold face.

For instance, suppose $\vec{x}$ consists of four alternating bits. Then, the encoded bit vector $\vec{c}$ is:

$$G \cdot \underbrace{\begin{pmatrix} 0 & 1 & 0 & 1 \end{pmatrix}^{\mathrm{T}}}_{\vec{x}} = \underbrace{\begin{pmatrix} 0 & 1 & 0 & 1 & \mathbf{0} & \mathbf{1} & \mathbf{0} \end{pmatrix}^{\mathrm{T}}}_{\vec{c}}$$

where $\vec{x}^{\mathrm{T}}$ denotes the transpose of $\vec{x}$. The three check bits are again printed in bold face.

To decode $\vec{c}$ and check for bit errors, we need the *parity-check matrix* H obtained by solving $H \cdot G = 0$. The columns of H are the binary representations of the integers $1, 2, ..., 2^k - 1$ for a Hamming(n+k, n) code with $n = 2^k - k - 1$ [152, p. 313], but not necessarily in ascending order. Every permutation of the columns of H defines a valid Hamming code. For a separable code, the first n columns of H are identical to the last k rows of G (bold face), and the remaining columns of H form the identity matrix.

Given the parity-check matrix, we compute the *syndrome* $H \cdot \vec{c}$, which yields information on potential bit errors. Taking the example from above, we obtain:

$$\underbrace{\begin{pmatrix} \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & 1 & 0 & 0 \\ \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & 0 & 1 & 0 \\ \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 & 0 & 1 \end{pmatrix}}_{H} \cdot \underbrace{\begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \end{pmatrix}}_{\vec{c}} = \begin{pmatrix} \mathbf{0} \\ \mathbf{0} \\ \mathbf{0} \end{pmatrix}$$

If the syndrome is the zero vector, as shown above, then $\vec{c}$ is free of any single bit error. Otherwise, the syndrome is identical to one column of H, say the ith column. In that case, c_i – the ith bit of $\vec{c}$ – is the only bit that can be affected by a single error. For example, flipping the first bit in $\vec{c}$ yields:

$$\underbrace{\begin{pmatrix} \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} & 1 & 0 & 0 \\ \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} & 0 & 1 & 0 \\ \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} & 0 & 0 & 1 \end{pmatrix}}_{H} \cdot \underbrace{\begin{pmatrix} \textcolor{red}{\mathbf{1}} \\ 1 \\ 0 \\ 1 \\ \mathbf{0} \\ \mathbf{1} \\ \mathbf{0} \end{pmatrix}}_{\vec{c}} = \begin{pmatrix} \mathbf{1} \\ \mathbf{1} \\ \mathbf{0} \end{pmatrix}$$

The syndrome equals the *first* column of H and, thus, only the first bit of the code word $\vec{c}$ can be invalid under the assumption of a single bit error.

By construction of the Hamming code, any single bit error can be detected and corrected. Moreover, the *Hamming distance* of the Hamming code is three [73, p. 106]; that is, any pair of different code words differs by at least three bit positions. This property guarantees that any error pattern involving less than three bits (the distance) can be detected.

However, on decoding a code word, both single and double bit errors produce a nonzero syndrome and cannot be differentiated. Thus, to decide whether a single bit error occurred – and correction is possible – or whether multiple bit errors are present that cannot be corrected, the Hamming code must be extended by another check bit that covers all data bits. This extra check bit implements an overall *parity* by appending an all-ones row to the transformation matrix G, and increases thereby the Hamming distance of the code to four [195, p. 119]. Thus, a single bit error leads to a nonzero syndrome and odd parity, whereas a double bit error produces a nonzero syndrome but even parity. Such an *extended Hamming code* provides single-error correction and double-error detection (SEC-DED), which is commonly used in memory circuits, as discussed in the following section.

2.3 HARDWARE-IMPLEMENTED FAULT TOLERANCE

The reliability issue of semiconductor devices calls for the development of hardware-implemented fault tolerance. To this end, a certain degree of hardware redundancy is used that, depending on the amount of redundancy, inevitably increases the costs of hardware. Redundancy can be implemented at multiple hardware layers, so that Dodd and colleagues [72, p. 1756f] distinguish between hardening at the device level, circuit level, and system level.

2.3.1 *Device-Level Hardening*

Radiation hardening of individual transistors aims at *fault avoidance* and requires a custom semiconductor fabrication process. For instance, Honeywell fabricates radiation-hardened CMOS devices at a feature size of 150 nm [89] with a silicon-on-insulator process [153], which inserts an insulating oxide into the silicon substrate beneath the transistors. However, these devices lag several technology generations behind commercial state-of-the-art microprocessors, implying less computing power.

Another approach is to add resistors or capacitors to the feedback loop of SRAM cells, flip-flops and latches [72, p. 1757]. The time required to switch a logical state increases thereby, so that those devices can recover from short-lived particle-induced voltage spikes. Unfortunately, the reduced switching speed directly degrades the performance of the device, and additional resistors or capacitors increase the energy consumption [172, p. 70].

2.3.2 *Circuit-Level Hardening*

Fault tolerance at the circuit level changes the *design* of the individual SRAM cells, flip-flops, and latches [72, p. 1757]. Such storage cells can be designed with additional transistors that act as redundant memory cells. For example, the *Dual Interlocked Storage Cell* [40] uses twelve transistors instead of six. Such a hardened cell is at least ten times more robust against SEUs; however, the primary and redundant cells are close to each other, so that a single particle can still strike both simultaneously [103]. Besides the area overhead, such cells consume about 30 to 40 percent more energy [103, p. 1541].

2.3.3 *System-Level Hardening*

At the system architecture level, error detection and correction can be applied to various hardware modules, including main memory, caches, and other parts of a microprocessor. SRAM caches are nowadays protected with parity and SEC-DED [117, p. 302f]. For example, the AMD Opteron [128, p. 72] processor implements an extended Hamming(72,64) code for its caches; that is, eight check bits cover 64 bits of data at the cost of 12.5 percent storage overhead. The very same code is typically used for ECC DRAM in the server market [170, p. 2].

To cope with word-wise multi-bit errors, IBM promotes *chipkill memory* [117, p. 875ff], which interleaves a 128-bit datum and 16 check bits on independent DRAM chips, at the cost of reduced performance and up to 30 percent higher energy consumption due to forced narrow-I/O configuration [257]. In this case, a burst error of 4 bits can be corrected, but two or more uncorrelated bit errors remain uncorrectable.

Chipkill memory

More powerful codes with stronger correction capabilities are typically not used for SRAM caches [117, p. 880] and DRAM [224, p. 401] due to performance and economic constraints.

Another system-level approach is to replicate a hardware module to perform the same computation multiple times. *Triple-Modular Redundancy (TMR)* [136, p. 20ff] uses three instances of a hardware module in combination with majority voting, so that a single hardware error gets masked. This technique can be applied to entire computer systems to obtain a high degree of reliability, however, increasing costs by at least a factor of three. The IBM S/390 G5 processor [226, p. 20] uses a duplicated processor pipeline, which operates in lockstep and compares the computations. On mismatch, the processor reverts to a checkpoint of the entire microarchitectural state (rollback). While providing adequate fault tolerance, such solutions come at high costs:

Triple-Modular Redundancy

> “The overheads associated with these conventional solutions are prohibitively expensive for budget-conscious designers with less demanding reliability requirements.”
>
> – Shuguang Feng and associates [79, p. 398]

2.4 SOFTWARE-IMPLEMENTED FAULT TOLERANCE

The preceding sections pointed out natural radiation as a major cause for transient faults in computer systems, but hardware-implemented countermeasures tend to be too expensive for commonplace usage. This problem has motivated many researchers to investigate the feasibility of software-implemented hardware fault tolerance. The term *software-implemented* denotes that the software introduces redundancy to handle errors, whereas *hardware fault tolerance* indicates that the approach covers hardware defects[3]. Because such approaches also cover software defects to some extent, I omit the word *hardware* in this section, reducing the unwieldy term to *software-implemented fault tolerance*. Nevertheless, the focus of this section remains on hardware defects.

We shall see in the following chapters that this thesis proceeds with software-implemented fault tolerance. Therefore, this section mainly reviews related work. I do not claim to be comprehensive here – Wensley and associates [255] pioneered the notion of software-implemented fault tolerance in the 1970s; since then, a rich body of literature has evolved. For example, Goloubeva and colleagues [94] devote a complete textbook to this topic.

Taxonomy of software-implemented fault tolerance

As a starting point for structuring the literature, I developed a taxonomy of software-implemented fault tolerance. Figure 2.5 illustrates the taxonomy, which subdivides the area of research into *error detection* and *error correction*, each of which being further differentiated into five categories. Each category comprises numerous of related approaches that are discussed in the following sections.

The goal of these sections is to introduce the state-of-the-art approaches to software-implemented fault tolerance. Subsequently, Section 2.4.3 evaluates these approaches and rigorously compares them to identify potential shortcomings. Finally, Section 2.4.4 discusses the fundamental limitations of software-implemented fault tolerance in general.

2.4.1 *Error Detection Mechanisms*

The ability to detect an error but not correct it solves one-half of the problem raised by fault tolerance. First, just error detection does not prevent any service failure but avoids incorrect program output. Thus, the goal of error detection is to reduce *silent data corruption*. Second, error detection is usually required *prior* to error correction and recovery – the other half of the problem, covered in Section 2.4.2. As shown in Figure 2.5, I differentiate between five distinct approaches to software-implemented error detection, which are successively described in the following sections.

2.4.1.1 *Computation Duplication*

Error detection, implemented by computation duplication, relies on the assumption that a specific transient fault does not reoccur on repeated execution. Thus, the sequential repetition of computation exploits *time redundancy*. If both the primary and duplicated execution compute an identical result, the computation is likely to be correct. Duplication can be implemented at different levels of abstraction, in-

[3] See Section 2.2 for further distinction between these terms.

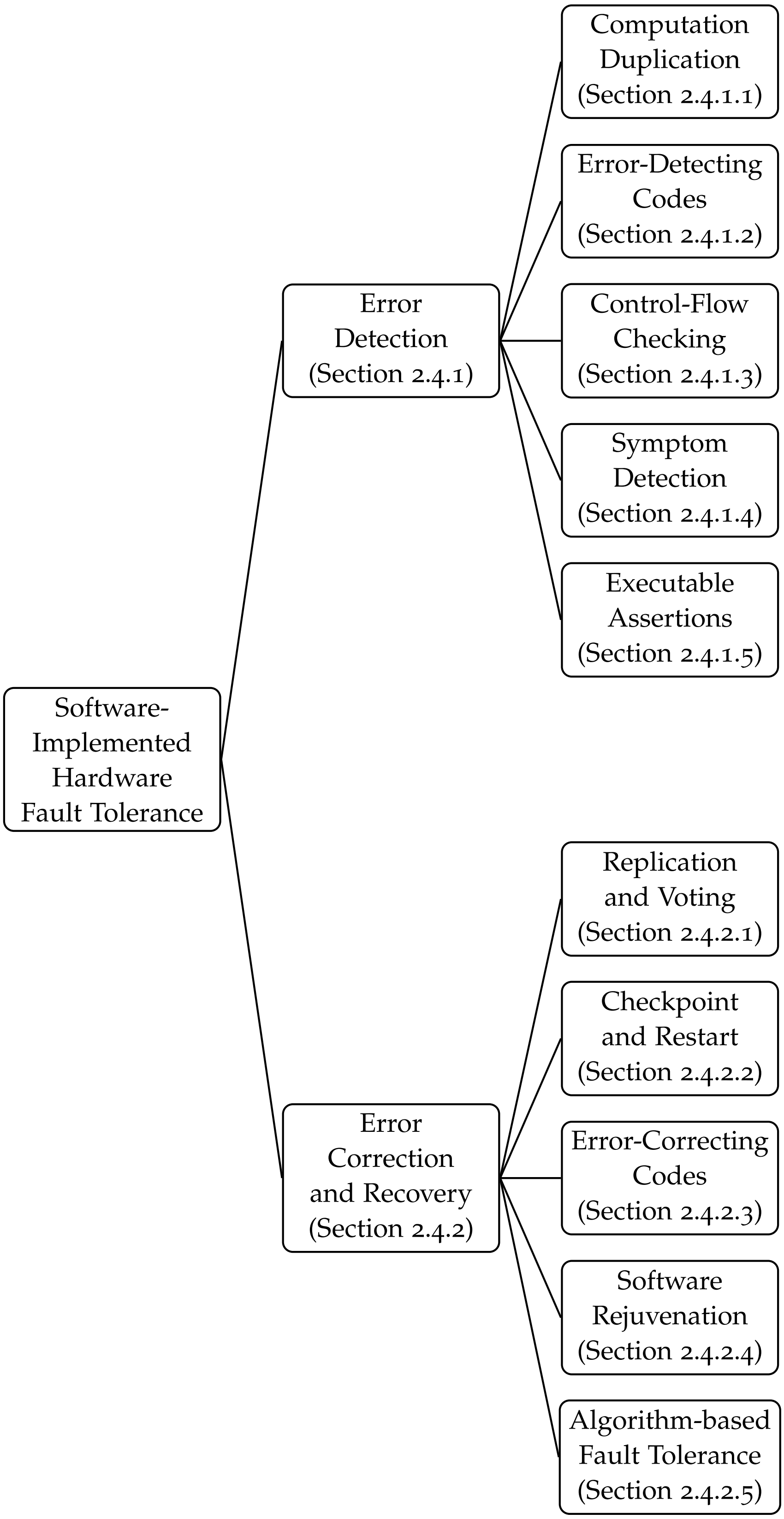

Software-Implemented Hardware Fault Tolerance
Error Detection (Section 2.4.1)
Computation Duplication (Section 2.4.1.1)
Error-Detecting Codes (Section 2.4.1.2)
Control-Flow Checking (Section 2.4.1.3)
Symptom Detection (Section 2.4.1.4)
Executable Assertions (Section 2.4.1.5)
Error Correction and Recovery (Section 2.4.2)
Replication and Voting (Section 2.4.2.1)
Checkpoint and Restart (Section 2.4.2.2)
Error-Correcting Codes (Section 2.4.2.3)
Software Rejuvenation (Section 2.4.2.4)
Algorithm-based Fault Tolerance (Section 2.4.2.5)

Figure 2.5: Taxonomy of software-implemented hardware fault tolerance. The taxonomy subdivides the state-of-the-art into *error detection* and *error correction*, each of which being further differentiated into five categories. Each category is discussed in a dedicated section.

cluding the operating-system level, program level, procedure level, and instruction level.

OPERATING-SYSTEM DUPLICATION: A whole operating system, including all applications, can be duplicated by hypervisor-based virtual machines [35]. Both a primary and a secondary operating system can be executed on the same hardware, and the hypervisor coordinates their I/O operations. When both virtual machines operate in lockstep [119], errors in the guest operating systems are detected by comparing the I/O operations of each virtual machine. This approach does not detect errors that affect the hypervisor itself.

PROGRAM DUPLICATION: Repeated execution of application software [135, p. 468] can be implemented at the process or thread level [251]. Software-based redundant multi-threading [251], for example, uses a compiler to generate duplicate threads that compare their computation outputs on system calls and on access to shared memory. The operating system is responsible for scheduling the redundant processes and threads, so that errors that affect the operating system itself cannot be detected.

PROCEDURE-CALL DUPLICATION: At the level of the source code of a program, procedure (function) calls can be executed twice to detect errors by comparing the computation results [181]. If a procedure modifies a global variable or uses pointers, the procedure code, the global variables, and the pointed-to variables must be cloned.

HIGH-LEVEL INSTRUCTION DUPLICATION: Rebaudengo and associates [199] propose duplication of the individual statements and variables of a program. Each write operation is performed on both the original variable and on the duplicate, and both variables are checked for consistency on each read operation. Conditional statements (`if`, `while`, ...) cannot be duplicated, but the condition can be evaluated once again after the conditional statement. These duplication rules can be automated by a source-to-source compiler [25, 200, 176]; however, compiler optimization must be disabled to preserve the duplication [199, p. 212], resulting in a slowdown of the execution time by a factor of five [199, p. 218].

ASSEMBLY-LEVEL INSTRUCTION DUPLICATION: In principle, the duplication of machine-code instructions is like high-level instruction duplication. Oh and colleagues [184] describe a compiler that duplicates every program variable in the main memory. The original assembly instructions are duplicated and use different CPU registers to operate on the duplicate variables. A check for consistency is inserted before storing a value to memory and before a branch instruction. Reis and associates [202] propose a similar approach that only duplicates variables once they are loaded into a CPU register and assumes the main memory to be protected by other mechanisms. Oh and associates [182] also show that, if the duplicate variables are additionally encoded with an AN code (see Section 2.4.1.2), permanent hardware faults are detected as well.

2.4.1.2 *Error-Detecting Codes*

Error-detecting codes introduce *information redundancy* to allow for the detection of erroneous data (error correction is covered in Section 2.4.2.3). In general, n bits of data are encoded into a larger code word with $c > n$ bits. Thus, not all 2^c binary combinations are valid code words. Usually, any single bit error turns a valid code word into an invalid one, and, depending on the coding scheme, multiple bit errors can be detected as well.

The purpose of this section is to discuss the *application* of error-detecting codes, whereas the coding theory is covered in further literature [37, 194, 136, 163, 212]. Error-detecting codes are classified into *checksums* and *arithmetic codes*.

CHECKSUMS: Maxino and Koopman [163] give an overview on algorithms that fall into the broad category of *checksums*, such as parity codes, addition checksums, and Cyclic Redundancy Check (CRC) codes. Using a checksum for read-only data, such as the program instructions, is considered as best practice in the field of embedded systems [22, p. 74ff]. Nicolescu and associates [177] provide evidence for the effectiveness of checking a CRC code at regular intervals to detect memory errors affecting the program instructions. In addition, programmers can manually maintain a checksum for mutable data structures [36]. In the context of the Java programming language, the Java Virtual Machine (JVM) can be extended to augment heap objects by a checksum [46, 48]. Likewise, the JVM can maintain parity codes for object references and object headers [238].

ARITHMETIC CODES: An arithmetic code has the characteristic of being preserved under arithmetic operations. For example, the AN code [37] encodes the datum N by multiplication with an odd constant A. Thus, an integer X is encoded by $A \cdot X$. The resulting code word is valid if and only if it is a multiple of A. The sum of two encoded integers results again in a valid code word: $A \cdot X + A \cdot Y = A \cdot (X + Y)$. Hence, there is no need for decoding prior to arithmetic, so that arithmetic errors are detectable (for more information, see Schiffel [212]). AN codes can be applied either manually [248, 110], by special compilers [80, 138], or by binary translation [253]. The drawbacks of AN codes are that larger data types must be used to store the encoded values (32-bit values usually become 64-bit values after encoding), and that bit operations are not preserved.

2.4.1.3 *Control-Flow Checking*

In contrast to duplication and encoding that cover data errors, Control-Flow Checking (CFC) monitors the sequence of executed CPU instructions to detect illegal branches. Errors that manifest as data corruption go undetected. The integrity of the program's control flow can be checked at the granularity of *functions* (procedures) or *basic blocks*.

FUNCTION TOKENS: A function token is a memory location that stores a unique ID representing the function that is currently executing [186, p. 231]. Before a function is called, the token is assigned the ID of the invoked function. Within the function, the token is regularly compared with the function's ID. A mismatch signals an invalid branch to another function. When a function returns, the token is restored to the value of the caller. Alexandersson and colleagues [7, p. 306f] extend this scheme by a second ID that is stored on function return. The caller checks thereby whether the expected function has returned. Both approaches need information on the call target at compile time, thus, ruling out any indirect function call.

BASIC-BLOCK SIGNATURES: Like function tokens, a *basic block* can be identified by a unique ID (the *signature*). A basic block is a sequence of CPU instructions that are always executed in the specified order; that is, there is exactly one entry and one exit point. Then, a memory location can store the signature of the basic block that is currently executing, which can be checked before branching. The advantage over function tokens is that illegal branches within the same function can be detected. Several approaches have been proposed [185, 171, 199, 9, 183, 129], which differ in the way the signatures are computed and validated. Goloubeva and associates [94, p. 63ff] devote an entire book chapter on that topic. However, a recent study suggests that the exclusive use of CFC might be ineffective because the critical data remain unprotected [220].

2.4.1.4 *Symptom Detection*

Some hardware errors can be detected by observing abnormal software *behavior* at runtime. For example, unexpected traps or excessive time spent in the OS kernel are *symptoms* that indicate an error [148]. Wang and Patel [252] argue that arithmetic overflows, branch mispredictions, and even cache misses can be considered as symptoms.

Because such symptoms also occur intentionally, symptom detection is usually paired with speculative recovery that restores the system state to an older checkpoint (see Section 2.4.2.2). A symptom that reoccurs after recovery is considered as *false alarm*.

Racunas and colleagues [197] pick up the idea of Hangal and Lam [100] to use program profiling to determine likely invariants, such as data ranges of program variables. After profiling, range checks are inserted into the program. A symptom is signaled when an invariant is violated by a variable exceeding its expected data range. Several invariant detectors have been proposed in the literature [209, 190, 102].

Finally, security mechanisms that detect program bugs are related to symptom detectors. For instance, a soft error that corrupts a pointer variable can cause a buffer overflow, which can be detected by stack-protection mechanisms [60, 52, 137]. Modern production compilers such as the GNU Compiler Collection (GCC) and LLVM/Clang include optional *sanitizers*, which perform run-time type checking [245] and memory-access checking [216] to detect bugs.

The drawback of symptom detection is that only erroneous *behavior* is detected while the corruption of critical data can go undetected. In addition, symptoms include false alarms, which complicate error recovery.

2.4.1.5 *Executable Assertions*

Application-specific semantics of a program, such as loop invariants, can be checked for consistency by executable assertions. For example, Skarin and Karlsson [223] show the effectiveness of range checks on a software integrator algorithm for automotive brake-by-wire applications. In a similar manner, Hannius and Karlsson [101] point out the benefits of range checks on parameters of a jet-engine controller to detect soft errors. Plausibility checks on program output, such as verification whether the output of a sorting algorithm is indeed sorted, can also detect several hardware errors [203]. However, Rela and associates [203, p. 402] argue that assertions need to be complemented with CFC (see Section 2.4.1.3) to prevent jumps over the assertions.

Finally, manually implemented assertions involve high engineering costs, because appropriate invariants must *exist*, which are often difficult – or even impossible – to specify.

2.4.2 *Error Correction and Recovery Mechanisms*

Once an error has been detected, fault tolerance requires a corrective measure to avoid a service failure. The following sections briefly introduce such measures, classified into five categories as depicted in Figure 2.5 on page 25. Most approaches are direct extensions of their error-detecting counterparts and enable error compensation, whereas *checkpoint and restart* (see Section 2.4.2.2) as well as *software rejuvena-*

tion (see Section 2.4.2.4) are complementary to error detection and only implement error recovery.

2.4.2.1 *Replication and Majority Voting*

An extension to computation duplication (see Section 2.4.1.1) is N-fold replication, such as *triplication*. Under the assumption that at most one error occurs, at least two out of three independent program executions compute correct results. Thus, majority voting between the three results chooses a correct result and masks a single error. In general, *M-of-N* systems consists of N (replicated) modules of which M must operate correctly, so that $N - M$ errors can be tolerated. N-modular redundancy (NMR) implements M-of-N systems with N odd and $M = \lceil N/2 \rceil$ to minimize the redundancy for tolerating a certain number of errors [136, p. 20ff]. Thus, Triple Modular Redundancy (TMR) is a 2-of-3 system that masks one error.

Replication and majority voting can be applied in the same way as computation duplication, that is, to all the same levels of abstraction described in Section 2.4.1.1. This section discusses replication at the program level, procedure-call level, and instruction level in more detail.

PROGRAM REPLICATION: Redundant execution and majority voting of software tasks have been used for decades [254, 6]. To overcome a manual implementation, Shye and colleagues [221] propose binary rewriting of single-threaded Linux processes to redirect all system calls to a voter process. Döbel and associates [70] improve on that by providing operating-system services that support the replication of multi-threaded applications. Another variant of program replication is N-version programming [17], which aims at tolerating software bugs by voting on the output of N *different* program implementations for the same problem.

PROCEDURE-CALL REPLICATION: Alexandersson and colleagues [7] propose to execute program procedures three times for comparing the results. The program variables also need to be triplicated, so that each procedure run operates on its own set of variables. The approach uses the AspectC++ compiler (see Chapter 4) to automate the procedure-call replication, however, only *global* variables are replicated thereby [7, p. 305].

INSTRUCTION REPLICATION: The duplication of program instructions (see Section 2.4.1.1) can easily be extended to instruction triplication and majority voting. For example, the SWIFT-R compiler technique [45] loads each value into three distinct CPU registers, and replicates each assembly instruction to repeat its operation on the register copies. Majority voting is carried out before load and store instructions, procedure calls, and branch decisions. In the same study, Chang and associates [45] point out that arithmetic coding (see Section 2.4.2.3) can be used additionally to reduce the number of redundant instructions: If a computation of the original instructions differs from one AN-encoded copy, the AN code can be checked to infer which one is faulty.

2.4.2.2 *Checkpoint and Restart*

Checkpointing is the proactive measure taken to save the entire state of a program at regular intervals. When an error is detected by one of the mechanisms described in Section 2.4.1, the program state can be recovered to a checkpoint prior to the occurrence of the error (*rollback*), and the program can then be restarted to continue on the restored data. Koren and Krishna [136, p. 193ff] provide a book chapter on checkpointing in general, so that the following discussion is limited to transparent checkpointing at the system level, application level, and object level.

SYSTEM-LEVEL CHECKPOINTING: Transparent checkpointing can be implemented at the operating-system level. For example, *TICK* [93] is a Linux kernel module that periodically creates incremental checkpoints of application programs. The approach copies those virtual memory pages that have been modified since the last checkpoint. In a similar manner, *Remus* [61] implements checkpointing in a hypervisor for virtual machines. In either case, the checkpointing implementation itself is not protected, that is, the operating system or hypervisor, respectively.

APPLICATION-LEVEL CHECKPOINTING: Checkpointing in scientific computing is commonly implemented by program libraries [113, 175]. However, checkpointing libraries are not fully transparent to the user, as the user must insert library calls into critical areas of the program to initiate checkpointing. Thus, compiler-assisted checkpoint insertion has been proposed [147, 79].

OBJECT-LEVEL CHECKPOINTING: Compiler-based checkpointing of object-oriented C++ programs has been studied by Kasbekar and colleagues [127]. Their approach uses the OpenC++ compiler [50] to generate checkpointing code for individual data objects. Lawall and Muller [143] propose incremental checkpointing for Java programs optimized by automatic program specialization. Finally, the JVM can be extended to create checkpoints of data objects [47].

2.4.2.3 *Error-Correcting Codes*

One of the most widely used error-correcting codes (ECC) is the Hamming code, which has been already introduced in Section 2.2.3. Detailed information on coding theory for error correction is provided by Peterson and Weldon [195].

In principle, the application of an ECC is like the application of checksums (see Section 2.4.1.2). Shirvani and associates [219] propose to perform periodic error detection and correction of read-only data segments, such as the program instructions. The error-correcting code for read-only data can be created in advance. For volatile data that change during program execution, the programmer must use a special Application Programming Interface (API) to invoke error detection and correction. Likewise, Nicolescu and colleagues [177] manually apply a Hamming code to individual program variables. Other examples are the triplication of program variables [145] and heap memory [189]. In C++, wrapper classes comprising ECCs can be implemented to substitute the integral data types `int`, `float`, and so on by means of operator overloading [165]. As mentioned earlier, the Java virtual machine can be augmented to extend data objects by checksums, and, for error correction, by additional object duplicates [49]. At the operating-system level, software-implemented Hamming codes can be applied to virtual memory pages being withdrawn temporarily from an application process [81].

2.4.2.4 *Software Rejuvenation*

Software rejuvenation [136, p. 152ff] aims at proactively removing accumulated errors and freeing unused resources, such as locks and memory blocked by a crashed program. Thus, a software component is stopped and restarted to create a clean program state (*rollforward*). Parts of a system can be rejuvenated one after another, leading to *microreboots* [41], as opposed to rebooting the whole system.

Rejuvenation can be initiated either periodically or on demand. As an example of the former, device drivers can be designed for periodic device re-initialization [161]. *Nooks* [241], on the contrary, enables driver restart *after* driver failures. Song and associates [229] describe the design of an operating system focused on microreboots, so that various low-level system components are designed for restarts. However, the kernel of that operating system is not considered for microreboots [229, p. 27].

2.4.2.5 *Algorithm-Based Fault Tolerance*

Huang and Abraham [112] coined the term Algorithm-Based Fault Tolerance (ABFT), which exploits the *semantics* of algorithms for error detection and correction. In their early work on linear matrix operations, they encode matrices by appending one additional checksum row and one additional checksum column. Each element of the checksum row is computed as the sum over the original matrix elements that belong to the same column, and, the other way around, the elements of the checksum column represent the sum of the original matrix elements of the same row. By construction, the checksum rows and columns are preserved by several linear matrix operations, such as addition and multiplication. Because each matrix element is covered by both a row and column checksum, a single erroneous element can be detected and corrected.

On this basis, ABFT has been devised for various numerical problems, including QR factorization [206], fast Fourier transform [20], and solvers for partial differential equations [207]. Fault-tolerant sorting and binary-search algorithms have also been developed [83].

Robust data structures

Finally, robust data structures are closely related to ABFT. Taylor and colleagues [244] exploit the inherent redundancy of doubly linked lists and propose a recovery procedure to correct a single erroneous pointer. Likewise, binary trees [244], stack data structures [15], and priority queues [121] can be extended by redundant pointers to enable error correction.

In summary, ABFT describes a set of highly specialized techniques for specific algorithms. ABFT techniques cause a huge development effort – from the algorithm design to its implementation. Demsky and Rinard [66] propose to automate the implementation by only specifying consistency constraints that can be enforced at runtime, however, still requiring a thorough understanding of the algorithms and data structures.

2.4.3 *Comparison of the Approaches*

The previous sections introduced numerous approaches to software-implemented fault tolerance, subdivided into error detection (see Section 2.4.1) and error correction (see Section 2.4.2). Besides, these approaches differ in terms of applicability, for example:

> “If only the application state is corrupted, it can likely be recovered through application-level checkpointing (for which there is a rich body of literature). However, OS state corruptions can potentially be difficult – software-driven OS checkpointing has been proposed only for a virtual machine approach so far.”
>
> – Man-Lap Li and associates [148, p. 272]

Thus, the approaches can be distinguished by *applicability to operating-system kernels* – a field that seems to be underrepresented in the literature. The same applies to approaches based on virtual machines that protect the guest operating systems, but leave the hypervisor unprotected.

Considering operating systems, which manage multiple threads of control and asynchronous interrupt routines, the question arises whether the presented fault-tolerance mechanisms can be applied to multi-threaded programs in general. For instance, naive replication of global program variables causes false alarms during majority voting if the replicated variables are concurrently modified by another thread. Thus, several approaches – as proposed in the literature – do not work correctly for multi-threaded programs. Hence, *support for multi-threading* is another criterion that differentiates the approaches.

Echtle [74, p. 298] argues that the design of a fault-tolerance mechanism should focus on *transparency*. A transparent mechanism is independent of the application software, so that the mechanism can be reused for multiple applications, reducing the overall development costs [74, p. 13]. For example, a *compiler* that automatically inserts redundancy for error detection achieves transparency, whereas manually implemented assertions are not transparent. Therefore, the presented approaches can be further classified by transparency, possibly implemented by a compiler.

Finally, Section 2.1 pointed out that memory circuits are highly susceptible to soft errors, yet, the approaches differ in *coverage of memory errors*.

Because of these major differences, Table 2.3 evaluates all the approaches to software-implemented fault tolerance presented in Section 2.4.1 and Section 2.4.2 regarding the following four features:

1. Coverage of memory errors

2. Transparency, for example, compiler (support)

3. Support for multi-threading

4. Applicability to OS kernels (and hypervisors, respectively)

Related approaches are grouped into a single row of Table 2.3, and the distinction between error detection and error correction is given up. Acronyms, if available, refer to the terms used in the respective publications.

Considering Table 2.3 allows drawing the following conclusions:

- *Application-level fault tolerance is a problem mostly solved.* For example, redundant multi-threading (SRMT and Romain) as well as checkpointing are transparent, cover memory errors, and support multi-threaded applications.

- *OS-kernel fault tolerance receives less attention in the literature.* Only $^{11}/_{24}$ of the approaches are applicable to OS kernels in theory, yet, few works address this problem in practice [135, 64, 110].

  - Of these approaches, mere six support multi-threading, of which only two cover memory errors (*periodic checking of read-only memory* and *assertions*). None of them is transparent.

Approach	Coverage of memory errors	Transparency, e.g., compiler	Support for multithreading	Applicability to OS kernels
Virtual lockstep [119]	Yes	Yes	Yes	No
SRMT [251], Romain [70]	Yes	Yes	Yes	No
SPCD [181], TTR-FR [7]	Yes	Yes	No	Yes
High-level instruction replication [25, 200, 145]	Yes	Yes	No	Yes
EDDI [184], ED^{4}I [182]	Yes	Yes	No	Yes
SWIFT [202], SWIFT-R [45]	No	Yes	Yes	Yes
Periodic checking of read-only memory [219, 177]	Yes	No	Yes	Yes
JVM-based hardening [143, 46, 48, 47, 49, 238]	Yes	Yes	Unknown	No
Encoded processing [253, 80, 138]	Yes	Yes	Unknown	Unknown
Control-flow checking [171, 185, 199, 9, 186, 183, 7, 129]	No	Yes	Yes	Yes
Symptom detection [252, 197, 209, 148, 190, 102]	No	Yes	Yes	Yes
Buffer-overflow detection [60, 52, 137]	No	Yes	Yes	Yes
Sanitizer [216, 245]	No	Yes	Yes	No
Assertions [203, 223, 101]	Yes	No	Yes	Yes
Thread replication [254, 6, 248]	Yes	No	Yes	No
PLR [221]	Yes	Yes	No	No
Checkpointing [147, 113, 127, 93, 61, 175, 79]	Yes	Yes	Yes	No
Samurai [189]	Yes	Yes	No	No
Encoding by C++ operator overloading [165, 110]	Yes	No	No	Yes
Page-level ECC [81]	Yes	Yes	Yes	No
Software rejuvenation [41, 241, 229, 161]	Yes	No	Unknown	No
ABFT [112, 20, 206, 207, 83]	Yes	No	Unknown	No
Robust data structures [244, 15, 66, 121]	Yes	No	Unknown	Yes

Table 2.3: Comparison of approaches to software-implemented hardware fault tolerance as proposed in the literature. The entry *Unknown* denotes that the respective publications do not clearly state whether a particular feature is supported or not.

The latter conclusion clearly points out a severe shortcoming of the state-of-the-art in software-implement fault tolerance. To the best of my knowledge, *there is no transparent approach to detection and correction of memory errors in multi-threaded OS kernels and hypervisors, yet*.

This thesis aims at filling the knowledge gap in this area of research. But prior to that, the next section discusses the fundamental limitations of software-implemented fault tolerance in general.

2.4.4 *Fundamental Limitations*

Three fundamental limitations apply to software-implemented hardware fault tolerance – in consideration of the underlying operating system.

2.4.4.1 *Perfect Fault Tolerance is Impossible*

In his textbook, Echtle argues that fault tolerance can improve dependability but can never guarantee the absence of failures at all [74, p. 1]. This is especially true for software-implemented fault tolerance. Consider a sequence of CPU instructions that stores the results of a computation into a memory-mapped I/O device.

> “ No matter what sophisticated software checking is performed just before a conventional store instruction, it will be undone if a fault strikes between the check and execution of the store instruction. This is the conundrum of the *Time-Of-Check-Time-Of-Use* (TOCTOU) fault. ”
>
> – Frances Perry and associates [192, p. 45]

Thus, perfect software-implemented fault tolerance is impossible, implying that only *probabilistic* reliability guarantees can be provided. The intended purpose of fault tolerance is to reduce the probability of failure to an acceptable level.

2.4.4.2 *Dependence on the Operating System*

Fault tolerance at the application level, such as checkpointing and redundant multi-threading, is necessarily incomplete.

> “ Errors generated on unprotected software modules will make useless the high levels of protection provided by other (thought) more critical modules. ... This point is particularly relevant in the design of the Operating System or Run-time Kernels where the fault-tolerant application is being run, since unreliable system services may thwart the dependability objectives. ”
>
> – Mário Rela and associates [203, p. 403]

Hence, fault tolerance at the application level *depends* on the correct functionality of the underlying operating system. This calls for a fault-tolerant operating system. Engel and Döbel [76] coined the term *Reliable Computing Base* to describe this dependency: The kernel of an operating system needs to be reliable in any case.

2.4.4.3 *Error Detection in OS Kernels is Insufficient*

> “Over 65% of detected faults corrupt OS state before detection.”
>
> – Man-Lap Li and associates [148, p. 273]

In general, error detection without correction is poorly suited for OS kernels. On error detection, the whole system must be restarted, terminating thereby all running applications and leaving persistent data, such as file systems, in a defective state. As an alternative, restoring the kernel to a previous checkpoint is also disadvantageous. Consider the state of a network connection: just rolling back the server state causes inconsistency with the client, for example, by mismatching sequence numbers.

Microkernel operating systems designed for fault tolerance, such as MINIX 3 [105, 242] and CuriOS [65], mitigate this limitation by executing most system services as isolated processes that can be restarted individually. However, the microkernel itself is still subject to the limitation that error detection is insufficient. Fault tolerance of the OS kernel requires error correction or error masking to guarantee system availability and avoid corruption of persistent data.

2.5 CHAPTER SUMMARY

The goal of this chapter was to discuss the problem of hardware unreliability and its implications for computer systems. Atmospheric neutrons and alpha particles have been identified as root causes for transient hardware faults, which pose a severe threat to the dependence on computer systems. Section 2.1.3 pointed out that CMOS memory technologies, especially SRAM and DRAM circuits, are highly susceptible to radiation-induced faults. Such faults do not cause permanent physical damage but primarily manifest as bit flips – *soft errors* – that can be recovered by overwriting the defective bits.

Unfortunately, conventional hardware solutions tend to be prohibitively expensive for commonplace usage, because redundant memory cells and redundant transistors increase the production costs and power consumption.

As a remedy, software-implemented fault tolerance can provide a low-cost alternative. Section 2.4 reviewed *related work* in the domain of software-implemented fault tolerance, culminating in a detailed evaluation of the *state-of-the-art* and its fundamental limitations. The conclusion is that there is a *knowledge gap* on transparent approaches to detection and correction of memory errors in multi-threaded OS kernels. This shortcoming has also been acknowledged lately by scientists from Microsoft Research:

> "More work is warranted towards an operating system designed with faulty hardware as a first-order concern."
>
> – Edmund Nightingale and associates [179, p. 353]

3 PROBLEM ANALYSIS

> "The only DRAM bit errors that cause system crashes are those that occur within the roughly 1.5% of memory that is occupied by kernel code pages."
>
> – Edmund Nightingale and associates [179, p. 344]

In the previous chapter, Section 2.1.3 pointed out that CMOS memory technologies, especially SRAM and DRAM circuits, are highly susceptible to radiation-induced transient faults. The kernel of an operating system is the most important piece of software regarding system availability, because a crash of the operating system – as a consequence of a transient memory fault – terminates all user-level applications. Thus, the memory regions used by the kernel of an operating system are especially critical. A software-implemented error-correction mechanism for kernel memory would offer an enormous potential for improving the system's dependability.

The goal of this chapter is to analyze the problem of software-implemented fault tolerance for operating-system kernels. To this end, Section 3.2 examines the memory segments of two different operating systems. The failure mode, effects, and criticality of each individual memory location is evaluated, revealing the most frequent points of failure.

The following problem analysis is based on the methodology of *fault injection*, which is briefly introduced in Section 3.1. After that, Section 3.2 applies the methodology in two case studies to evaluate the reliability of two off-the-shelf operating systems. The results and implications are discussed in Section 3.3. On that basis, Section 3.4 formulates a *suggested approach* to address the problem of operating-system fault tolerance. Finally, Section 3.5 summarizes the problems analyzed in this chapter and concludes with a solution proposal.

RELATED PUBLICATIONS

The findings presented in this chapter have partly been published in:

[30] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generative software-based memory error detection and correction for operating system data structures. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*. IEEE Press, June 2013. doi: 10.1109/DSN.2013.6575308

Acceptance rate: 20 %

[213] Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. Rapid fault-space exploration by evolutionary pruning. In *Proceedings of the 33rd International Conference on Computer Safety, Reliability and Security (SAFECOMP '14)*. Springer, September 2014. doi: 10.1007/978-3-319-10506-2_2

Acceptance rate: 27 %

[214] Horst Schirmeier, Christoph Borchert, and Olaf Spinczyk. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*. IEEE Press, June 2015. doi: 10.1109/DSN.2015.44

Acceptance rate: 22 %

3.1 THE METHODOLOGY OF FAULT INJECTION

> “Fault injection is the de facto standard technique for system dependability benchmarking.”
>
> – Erik van der Kouwe and associates [249, p. 126]

The characterization of software behavior in the presence of hardware faults is impractical to carry out by deploying the system in the field. Even if a hardware fault occurs, field operation hardly yields statistically authoritative evidence for an asserted software behavior. For example, hardware faults induced by the natural radiation environment are neither controllable nor reproducible, providing little information on the reliability of a specific software component.

Fault injection [12] is a method for experimental testing by the deliberate introduction of faults, effectively speeding up their occurrence. The term *fault injection* describes a broad range of techniques, such as the injection of hardware faults [13] and software faults [249]. Furthermore, hardware faults can be injected either physically by irradiation from particle accelerators, by electromagnetic interference, by signals connected to the pins of an integrated circuit, or by software that emulates a hardware fault by changing the contents of memory or CPU registers [13]. This chapter focuses on the latter technique, which offers perfect controllability, repeatability, and reproducibility [13, p. 1129]. However, we shall come back to irradiation from a particle accelerator in Chapter 8.

Arlat and colleagues [12, p. 168] identify two complementary goals of fault injection. First, fault injection provides a means for *quantitative evaluation* of a target system's reliability. Second, fault injection gives *design aids* for the development of fault-tolerance mechanisms. The following chapters of this thesis cover quantitative evaluation, whereas this chapter focuses on design aids obtained by fault injection into two operating systems.

3.1.1 *Modeling of Hardware Faults*

The emulation of hardware faults by software is restricted to the type of faults that can be injected. In general, software has access to the memory subsystem and a few CPU registers, whereas a large portion of the internal microarchitecture is hidden. Thus, a *fault model* is needed that approximates real hardware faults by the manipulation of memory contents and CPU registers.

Atmospheric neutrons from the natural radiation environment penetrate every region of an integrated circuit at the same rate. Since the chip area of a microprocessor is dominated by the memory cells of caches [117, p. 14], the memory subsystem can be considered as the major contributor to transient faults (see Section 2.1.5). Cho and associates [53, p. 5] remark that even 12.5 percent of the nonmasked CPU-internal faults also propagate into main memory and circumvent any hardware-implemented redundancy of the memory subsystem. Therefore, this section proceeds with a simplified fault model that concentrates on memory faults and disregards some CPU-internal faults.

In Section 2.1.5, we identified that transient SRAM and DRAM faults manifest as bit flips. About 50 to 80 percent of such faults only affect a single bit (see Table 2.2 on page 16). Hence, the simplified fault model that is established in this section can be further confined to single bit flips.

In addition, the fault model must describe the spatial and temporal distribution of hardware faults. A recent large-scale study by Sridharan and colleagues concludes that "DRAM faults ... are consistent with a uniform random distribution." [234, p. 6] Similar observations hold for SRAM faults [16, p. 2262]. Moreover, Li and associates argue that "it is reasonable to assume that the time to the next high energy particle strike is independent of the previous strike" [149, p. 268]. Thus, the following commonly accepted[1] fault model serves as a working basis for the remainder of this chapter.

FAULT MODEL: In this thesis, transient hardware faults are modeled as *independent single bit flips in memory that follow a uniform random distribution.*

It is paramount to recapitulate that this fault model is a simplification and, thus, accepts certain inaccuracies. However, Cho and associates point out that even an inaccurate fault model "can be very useful as long as it is effective in driving the correct design decisions for building robust systems." [53, p. 6]

The working hypothesis for this chapter is that higher vulnerability to independent, uniformly distributed single bit flips in memory

[1] The great number of studies [124, 88, 78, 199, 25, 177, 201, 167, 48, 49, 256, 238] that adopt this particular fault model indicates its common acceptance.

implies higher vulnerability to radiation-induced hardware faults. We shall discuss the validity of this working hypothesis in Section 8.1.

3.1.2 *Probability of Failure*

The simplified fault model of *independent* single bit flips allows calculating the probability of exactly k fault occurrences in n possible fault locations by applying the binomial distribution:

$$B_{n,p}(k) = \binom{n}{k} \cdot p^k \cdot (1-p)^{n-k} \tag{3.1}$$

The parameter p denotes the probability of a single bit flip, which is given by the fault rate g of the memory technology (see Table 2.2 on page 16) based on the time of operation Δt. The probability of a DRAM fault is about

$$p = g \cdot \Delta t = 10^{-8}\,\frac{\text{FIT}}{\text{bit}} \cdot \Delta t = \frac{10^{-8} \cdot \Delta t}{10^9\,\text{h} \cdot \text{bit}} = 10^{-17}\frac{\Delta t}{\text{h} \cdot \text{bit}}$$

For example, consider a program (operating system) that runs for one hour ($\Delta t = 1\,\text{h}$) and uses $\Delta m = 1\,\text{MiB} = 2^{23}$ bits of memory. Thus, there are $n = \Delta m$ possible fault locations. Table 3.2 takes this example and shows the individual probabilities for the occurrence of $k = 0, 1, 2,$ or more DRAM faults during one program run.

k	$B_{n,p}(k\,\text{Faults})$	k	$B_{n,p}(k\,\text{Faults})$
0	0.9999999999161	3	$9.8 \cdot 10^{-32}$
1	$8.4 \cdot 10^{-11}$	4	$2.1 \cdot 10^{-42}$
2	$3.5 \cdot 10^{-21}$	...	...

Table 3.2: Binomial probabilities for $k = 0, 1, 2,$ or more faults occurring in one program run, assuming a DRAM fault probability of $p = 10^{-17}$ per bit and $n = 1\,\text{MiB}$ possible memory locations.

It is remarkable that the probability of two or more faults is ten orders of magnitude lower than for one fault. Because the occurrence of exactly one fault is so much more likely, the total probability of a failure $P(\text{Failure})$ – as a consequence of faults – is dominated by the probability of *one* fault $P(1\,\text{Fault})$. For now, a failure can be a program crash or data corruption, yet, Section 3.2 defines the term *failure* of the operating system more precisely. By applying the law of total probability, we can decompose $P(\text{Failure})$ and derive an upper bound as follows[2]:

[2] The argumentation is based on the idea of Horst Schirmeier [214, p. 326]. In addition, this section derives an *upper bound* to estimate the error of the subsequent approximation. Furthermore, the following formalization avoids the error introduced by Poisson approximation of binomial probabilities [214, p. 327].

$$\begin{aligned}
P(\text{Failure}) &= \sum_{k=1}^{n} P(\text{Failure} \mid k\,\text{Faults}) \cdot P(k\,\text{Faults}) \\
&= P(\text{Failure} \mid 1\,\text{Fault}) \cdot P(1\,\text{Fault}) \\
&\quad + \sum_{k=2}^{n} P(\text{Failure} \mid k\,\text{Faults}) \cdot P(k\,\text{Faults}) \\
&\leq P(\text{Failure} \mid 1\,\text{Fault}) \cdot P(1\,\text{Fault}) + \sum_{k=2}^{n} 1 \cdot P(k\,\text{Faults}) \\
&= P(\text{Failure} \mid 1\,\text{Fault}) \cdot P(1\,\text{Fault}) \\
&\quad + (1 - P(0\,\text{Faults}) - P(1\,\text{Fault})) \\
&= P(\text{Failure} \mid 1\,\text{Fault}) \cdot P(1\,\text{Fault}) + \epsilon(n, p)
\end{aligned}$$

The notation $\epsilon(n, p)$ represent the term $1 - P(0\,\text{Faults}) - P(1\,\text{Fault})$, which is an upper bound for the probability of a failure caused by multiple faults. This term is negligible for sufficiently low fault probabilities p. Taking the aforementioned example shown in Table 3.2 gives $\epsilon(2^{23}, 10^{-17}) = 4 \cdot 10^{-17}$, which is six orders of magnitude lower than $P(1\,\text{Fault})$. Thus, the absolute error of the following approximation is bounded by the negligible small $\epsilon(n, p)$:

$$P(\text{Failure}) \approx P(\text{Failure}|1\,\text{Fault}) \cdot P(1\,\text{Fault}) \tag{3.2}$$

The probability of exactly one fault occurrence $P(1\,\text{Fault})$ can be calculated by the binomial distribution $B_{n,p}(1)$. Inserting Equation 3.1 in Equation 3.2 gives

$$\begin{aligned}
P(\text{Failure}) &\approx P(\text{Failure}|1\,\text{Fault}) \cdot n \cdot p \cdot (1 - p)^{n-1} \\
&\leq P(\text{Failure}|1\,\text{Fault}) \cdot n \cdot p
\end{aligned} \tag{3.3}$$

Hence, the only unknown term in Equation 3.3 is $P(\text{Failure}|1\,\text{Fault})$, which needs to be measured by fault-injection experiments. For such experiments, it suffices to run the program independently for N times and to inject exactly one bit flip randomly into the memory area Δm during each program run. F denotes the number of fault-injection experiments that exhibit a program failure. For a sufficiently large number of samples N, the fault-injection experiments yield an accurate statistical estimator $\hat{P}(\text{Failure}|1\,\text{Fault}) = {}^{F}/_{N}$. Thus, we can rewrite Equation 3.3 as

$$P(\text{Failure}) \approx \frac{F}{N} \cdot n \cdot p = \frac{F}{N} \cdot \Delta m \cdot \Delta t \cdot g \tag{3.4}$$

The term Δt refers to the program runtime and g denotes the fault rate of the memory technology (see page 42).

Equation 3.4 provides the fundamental *reliability metric* for the evaluation based on fault injection throughout this thesis. The total probability of failure is directly proportional to the number of failed program runs F obtained by independent fault-injection experiments.

3.1.3 *Exhaustive Fault-Space Scanning*

The fault model of independent, uniformly distributed single bit flips in memory spans a two-dimensional *fault space* of possible fault locations. On the one hand, a fault can affect every memory cell. On the other hand, a fault can occur at any time. The continuous time can be discretized for synchronous circuits with a resolution of CPU clock cycles. Whenever a fault occurs within the interval of a clock cycle, the fault is not read before the next clock edge.

Figure 3.1a illustrates the discrete fault space of a hypothetical program that runs for $\Delta t = 12$ clock cycles on a machine with $\Delta m = 9$ bits of memory. Thus, there are $12 \cdot 9 = 108$ possible fault locations, which are denoted by black circles at each coordinate in Figure 3.1a. An exhaustive assessment of the individual criticality of each memory location requires conducting one independent fault-injection experiment for each coordinate. In each experiment, one bit of memory is flipped at the respective cycle–memory coordinate while the program is running. Thus, an exhaustive fault-space scan requires $N = \Delta m \cdot \Delta t$ fault-injection experiments. Inserting this value for N into Equation 3.4 gives

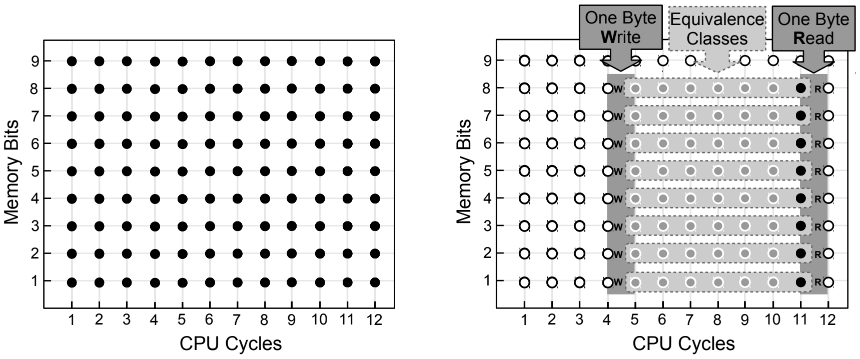
$$P(\text{Failure}) \approx \frac{F}{N} \cdot \Delta m \cdot \Delta t \cdot g = F \cdot g \qquad (3.5)$$

Hence, for an exhaustive fault-space scan, the probability of failure depends only on the number F of failed fault-injection experiments, scaled by the fault rate g of the memory technology.

However, conducting experiments for each coordinate of the whole fault space is practically infeasible for real-world programs. Therefore, Smith and associates [227] as well as Güthoff and colleagues [97] propose an optimization based on memory access traces that identifies equivalent fault locations in the fault space. For example, Figure 3.1b shows the memory access pattern obtained from a trace. The dynamic CPU instruction starting in the fourth clock cycle writes eight bits to main memory; the datum is read back into a CPU register in the eleventh clock cycle.

Trace-based optimization (def–use analysis)

First, this information can be exploited to identify *dead* fault locations that are not read. For example, the ninth bit of memory in Figure 3.1b is never used, and all other memory cells are overwritten in



(a) Each coordinate (cycle, bit), denoted as a black circle, represents a possible fault location. Without further information on the program, an exhaustive fault-space scan requires conducting a fault-injection experiment for each of the 108 coordinates.

(b) With information on the memory access pattern of the program, several coordinates can be grouped into equivalence classes. Fault injections at perforated coordinates (white area enclosed by a circle) can be omitted, as a fault there is overwritten or is never read. A black circle represents a class of equivalent faults (light-gray coordinates).

Figure 3.1: Illustration of a program's fault space spanned by *CPU Cycles* × *Memory Bits*. Every discrete (cycle, bit) coordinate denotes a possible location for the occurrence of a single bit flip in memory.

the fourth clock cycle. The injection of faults into *dead* fault locations cannot result in a program failure, so that experiments for those fault locations can be omitted.

Second, Figure 3.1b shows that each memory fault has the same effect no matter whether it occurs in the fifth, sixth, . . . , or tenth clock cycle, because the fault is not read before the eleventh clock cycle. Thus, these fault locations can be grouped into equivalence classes, which only a single fault-injection experiment needs to be conducted for. Figure 3.1b shows eight equivalence classes – one for each used memory location around the clock cycles five to ten. For example, the rightmost fault location, depicted as a black circle, can be chosen as a representative for an equivalence class.

By exploiting memory access traces, the number of fault-injection experiments needed for an exhaustive fault-space scan is reduced from 108 to mere 8 in the example illustrated in Figure 3.1. Even though only one experiment is conducted for each equivalence class, the results of *all fault locations count* in the calculation of the probability of failure according to Equation 3.5.

A limitation of the trace-based optimization is that the program runs must be carried out deterministically; that is, the memory access patterns of repeated program runs must match the previously recorded trace. This can be achieved in practice by using a hardware simulator that allows replaying external events, such as interrupts, at the exact same point in time during each program run.

3.2 BASELINE DEPENDABILITY ASSESSMENT

This section comprises two case studies that assess the dependability of two operating systems by means of fault injection based on the fault model of single bit flips. The technique of exhaustive fault-space scanning, described in the previous section, allows analyzing and comparing the vulnerability of each memory location. The goal of this section is to identify the most vulnerable spots of both operating-system kernels. This information provides design aids for the development of tailored fault-tolerance mechanism. In short, this section reveals those operating-system internals that would benefit the most from software-implemented fault tolerance.

The following case studies cover two complementary operating systems that address different domains.

ECOS: The *embedded configurable operating system (eCos)* [162] is an off-the-shelf operating system for real-time applications. eCos offers configurability at compile time of more than 750 system components, such as file systems and networking, resulting in roughly one million lines of C and C++ code. Targeting the domain of deeply embedded systems, eCos supports a multitude of hardware architectures, including microprocessors without Memory Management Unit (MMU). eCos has been deployed in numerous automotive and aerospace systems[3]. Dependability is of particular concern for such systems.

L4/FIASCO.OC: The L4/Fiasco.OC [139] operating system is a state-of-the-art microkernel that facilitates hardware-based isolation of system components. As such, L4/Fiasco.OC requires a hardware architecture with MMU, and currently supports x86, x86-64, and ARM platforms. The principle of the microkernel is that only a small piece of code – about 57,000 lines of C++ code for L4/Fiasco.OC – runs in the privileged mode of the CPU. It is only the microkernel that needs to be trusted, whereas other system components, such as device drivers and file systems, run in the unprivileged mode. L4/Fiasco.OC has been deployed in governmental projects with stringent security requirements [193].

Both operating systems have been used in a broad range of production systems with dependability constraints – from deeply embedded real-time applications to highly secure computing. To the best of my

[3] An overview of industrial products that use eCos can be found at: `http://www.ecoscentric.com`

knowledge, however, there is no previous study that rigorously evaluates the reliability of those operating systems regarding transient hardware faults. This section proceeds with such an evaluation.

3.2.1 *Experimental Setup*

In this chapter, I use the fault-injection framework FAIL* [215] for the dependability assessment of the eCos and L4/Fiasco.OC operating systems. FAIL* supports the x86-hardware emulator Bochs [144], which is capable of executing both eCos and L4/Fiasco.OC. Therefore, this chapter focuses on the x86 hardware architecture, whereas Section 7.3 and Chapter 8 generalizes the dependability assessment by using native x86-64 and ARM platforms.

FAIL fault-injection framework*

Bochs emulates an x86 CPU on a behavioral level with a simple timing model of one CPU instruction per cycle. The only exception is the `HLT` instruction, which suspends the CPU for multiple cycles until an interrupt occurs. Unless stated otherwise, the clock frequency of the emulated CPU is set to 2.666 GHz, representing a modern x86 CPU. Bochs does not simulate a cache hierarchy. Therefore, timing effects and fault masking caused by caches cannot be analyzed with this hardware emulator.

Emulated x86 CPU

FAIL* implements the fault model of independent single bit flips in memory. Furthermore, the trace-based fault-space optimization described in Section 3.1.3 is implemented as well, reducing the number of fault-injection experiments for an exhaustive fault-space scan.

In the following two case studies, Read-Only Memory (ROM) segments are excluded from fault injection for two reasons: First, deeply embedded systems targeted by eCos use flash memory as ROM storage. Flash memory already implements strong redundancy to compensate for cell degradation (see Section 2.1.3.3). Second, the usage of checksums [22, 177] and error-correcting codes [219] for ROM segments is already considered as best practice [22, p. 74ff]. For example, L4/Fiasco.OC implements a checksum for read-only kernel data, which are checked during bootstrapping. It is straightforward to perform that check periodically, as proposed by Shirvani and colleagues [219]. In short, integrity of ROM is a problem already solved.

To evaluate the operating systems under load, I use a set of benchmark and test programs that are bundled with each operating system. These programs only serve for activating and testing the operating system's functionalities. As such, the benchmarks finish after a predefined test procedure and report on success or failure on the serial port. The outcome of a single fault-injection experiment is classified into one of the four categories:

BENIGN: The injected fault does not change the result of the benchmark program, which finishes in due time and reports on success.

SDC: A benchmark run that finishes after fault injection but does not report on success is considered as Silent Data Corruption (SDC). In addition, if the benchmark run finishes in less than 90 percent of the expected runtime, or overwrites data outside the data segments, the fault-injection experiment counts as SDC.

TIMEOUT: A fault-injection experiment is aborted if ten times more CPU instructions are executed compared to a fault-free run. Moreover, a benchmark run that exceeds the fault-free runtime by $^1/_{18.2}$ seconds (one complete timer interval) is aborted. Such a situation occurs if the kernel is stuck in the idle loop.

CPU EXCEPTION: This category comprises occurrences of processor exceptions in kernel mode after injecting a fault. Silberschatz and associates [222, p. 561f] refer to processor exceptions as *nonmaskable* events that signal various error conditions; technically speaking, the first 32 entries of the x86 interrupt-vector table [222, p. 562] represent exceptions, such as the general protection fault (number 13) and double fault (number 8). In other words, this means a crash of the operating system.

In the following two case studies, I subsume the three categories SDC, timeout, and CPU exception under the common category *failure*. As discussed in Section 2.4.4.3, even though an error is *detected* by an exception handler or by a potential watchdog timer, the state of the operating system is likely corrupted as well. Thus, an SDC, timeout, and CPU exception counts toward the total number F of failed fault-injection experiments in Equation 3.5 on page 44. That equation also implies that benign faults are irrelevant for an exhaustive fault-space scan.

3.2.2 *Case Study: eCos*

The first case study evaluates the reliability of eCos 3.0 with its default configuration. The architecture of eCos centers on a single address space. Both the kernel and all applications share a common address space and run in the privileged processor mode. eCos is designed as a static library, which is linked with the applications to produce an executable image. A large share of kernel data is allocated statically and can be identified by a symbol name in the executable image.

eCos is bundled with a kernel test suite, which contains various benchmark programs that exercise the kernel's functionalities. This section takes two benchmark programs as examples to evaluate the eCos kernel. Appendix A.1 confirms the findings by evaluating 13 additional benchmarks from the kernel test suite.

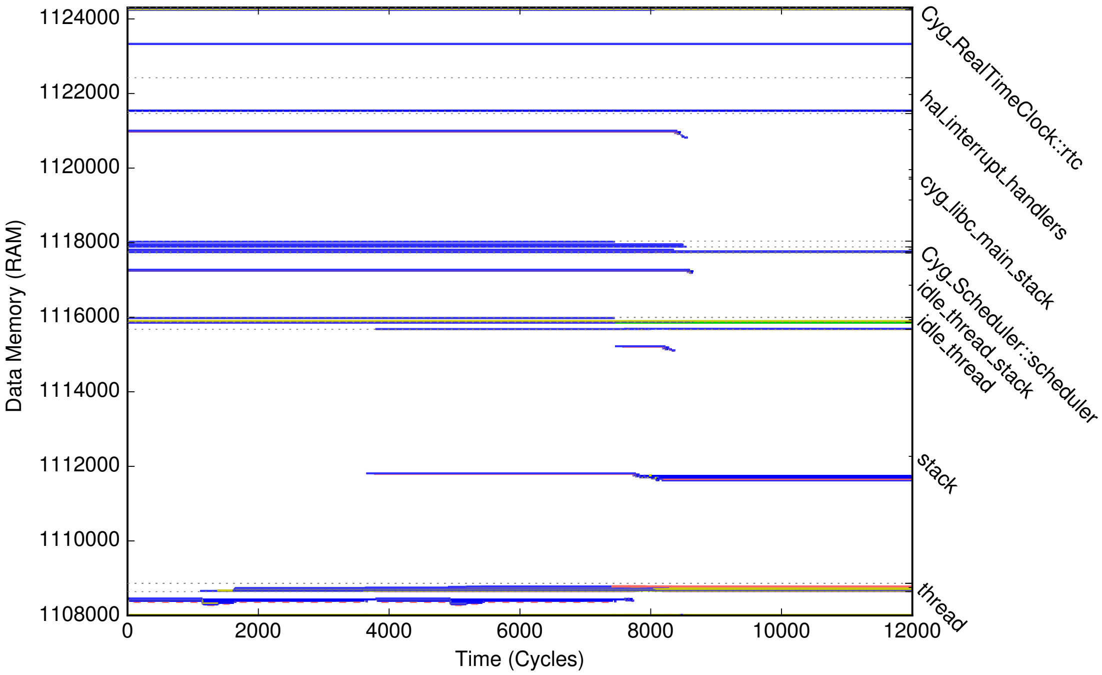


Figure 3.2: Excerpt of the fault space spanned by eCos running the thread1 benchmark. Each point denotes the outcome of an independent benchmark run after injecting a single bit flip at a specific time and memory coordinate. Injections in *white* areas are benign. *Blue* marks illegal memory accesses outside the data segments. CPU exceptions are colored *red* and timeouts *yellow*, respectively. *Green* data points show SDC.

3.2.2.1 *The eCos Benchmark* thread1

The first benchmark program thread1 serves the purpose of testing the interleaved execution of two threads. Therefore, thread1 executes a predefined test procedure of the thread-related system calls `suspend`, `resume`, `sleep`, and `wake`. At the same time, the benchmark repeatedly checks whether both threads are in their expected states, such as *running* or *sleeping*.

Figure 3.2 shows an excerpt from the exhaustive fault-space scan of eCos running the thread1 benchmark. Each coordinate denotes one independent fault-injection experiment, which injects a single bit flip at the respective point in time and memory location. If the operating system reads the corrupted memory location at a later point in time, the fault can turn into a failure, and then the coordinate of the fault injection is shown as colored point.

On the one hand, the clear majority of fault injections are benign, as indicated by the large uncolored areas in Figure 3.2. On the other hand, the colored horizontal lines reveal that failures often originate from the very same memory locations. The right y-axis assigns the program's symbol names to several consecutive memory locations.

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
thread	Cyg_Thread[]	264	$9.20 \cdot 10^9$	39.5 %
Cyg_RealTimeClock::rtc	Cyg_RealTimeClock	52	$4.29 \cdot 10^9$	18.4 %
stack	cyg_uint64[]	5,088	$3.31 \cdot 10^9$	14.2 %
Cyg_Scheduler::scheduler	Cyg_Scheduler	132	$8.53 \cdot 10^8$	3.7 %
comm_channels	int (*[][])()	96	$8.53 \cdot 10^8$	3.7 %
pt1	Cyg_Thread*	4	$8.53 \cdot 10^8$	3.7 %
Cyg_Interrupt::dsr_list_tail	Cyg_Interrupt*	4	$8.53 \cdot 10^8$	3.7 %
hal_interrupt_objects	cyg_uint32*[]	896	$8.53 \cdot 10^8$	3.7 %
hal_interrupt_handlers	cyg_uint32 (*[])()	896	$8.53 \cdot 10^8$	3.7 %
Cyg_Scheduler::sched_lock	cyg_ucount32	4	$8.53 \cdot 10^8$	3.7 %

Table 3.3: Quantitative fault-injection results of eCos running the THREAD1 benchmark. The ten most failure-prone symbols (continuous memory regions) amount to 97.8 percent of the total failures. In summary, the Cyg_Thread and Cyg_RealTimeClock data types are the most critical.

These symbols, separated by dotted lines, point out that *most failures cluster around a few kernel data structures*.

Table 3.3 lists the ten most failure-prone symbols from the exhaustive fault-space scan. Beside the symbol name and its data type, the table lists the accumulated number of failures per symbol next to the percentage of all failures.

97.8 percent of the total failures are caused by only these ten program symbols. In particular, 39.5 percent and 18.4 percent originate from instances of the Cyg_Thread and Cyg_RealTimeClock data types, respectively. Thus, these two data structures are the most critical. In addition, the stack accounts for 14.2 percent of the total failures, followed by several member variables, pointers and arrays thereof.

The baseline evaluation of the benchmark THREAD1 suggests that the reliability of the eCos kernel primarily depends on a small subset of kernel data structures. Especially these kernel data structures should be provided with software-implemented fault tolerance to improve the reliability significantly.

3.2.2.2 *The eCos Benchmark* MUTEX1

The second benchmark program MUTEX1 from the kernel test suite exercises the kernel's synchronization mechanisms to coordinate the execution of three threads. MUTEX1 invokes a test sequence of the system calls lock, trylock, unlock, signal, wait, and broadcast using two mutex variables and three condition variables. The benchmark monitors the activities of the three threads and checks for consistency with a predefined schedule.

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
stack	cyg_uint64[]	7,632	1,687,203	31.8 %
thread_obj	Cyg_Thread[]	416	1,487,037	28.0 %
cvar2	Cyg_Condition_Variable	8	287,551	5.4 %
cvar1	Cyg_Condition_Variable	8	226,016	4.6 %
m0	Cyg_Mutex	20	207,538	3.9 %
comm_channels	int (*[][])()	96	195,392	3.7 %
m1	Cyg_Mutex	20	184,180	3.5 %
cvar0	Cyg_Condition_Variable	8	177,824	3.3 %
Cyg_Interrupt::dsr_list	Cyg_Interrupt*	4	155,904	2.9 %
Cyg_Scheduler::scheduler	Cyg_Scheduler	132	148,663	2.8 %

Table 3.4: Quantitative fault-injection results of eCos running the MUTEX1 benchmark. The ten most failure-prone symbols (continuous memory regions) amount to 89.6 percent of the total failures. In summary, the stack and the Cyg_Thread data types are the most critical.

Table 3.4 summarizes the results from an exhaustive fault-space scan of eCos running the MUTEX1 benchmark. Like the findings of the previous benchmark, the reliability of the eCos kernel primarily depends on a few kernel data structures. However, because MUTEX1 exhibits a different application profile, the ten most failure-prone symbols are also different. In this case, the stack and instances of the Cyg_Thread data structure are the most critical. In addition, failures originating from instances of Cyg_Condition_Variable and Cyg_Mutex total at about 20 percent.

The stack symbol accounts for almost one third of the total failures, and comprises three independent stacks – one for each thread. Figure 3.3 on the next page depicts the fault-injection results of the exhaustive fault-space scan. The larger diagram identifies the memory locations that belong to the stack symbol on the right y-axis. The zoomed-in section reveals that a large share of failures is caused by faults in *return addresses* and *frame pointers*. These return addresses and frame pointers are generated by the GNU C++ compiler to implement subroutine calls, and represent the most homogeneous reasons for failures caused by the stack memory.

Return addresses and frame pointers

In summary, the evaluation of the benchmark MUTEX1 confirms the findings of the previous benchmark, and Appendix A.1 provides further evidence by presenting the results of 13 additional benchmarks. The reliability of the eCos kernel primarily depends on a small *application-specific* subset of kernel data structures: The instances of a few C++ classes and the stack, which stores return addresses and frame pointers, are the most critical.

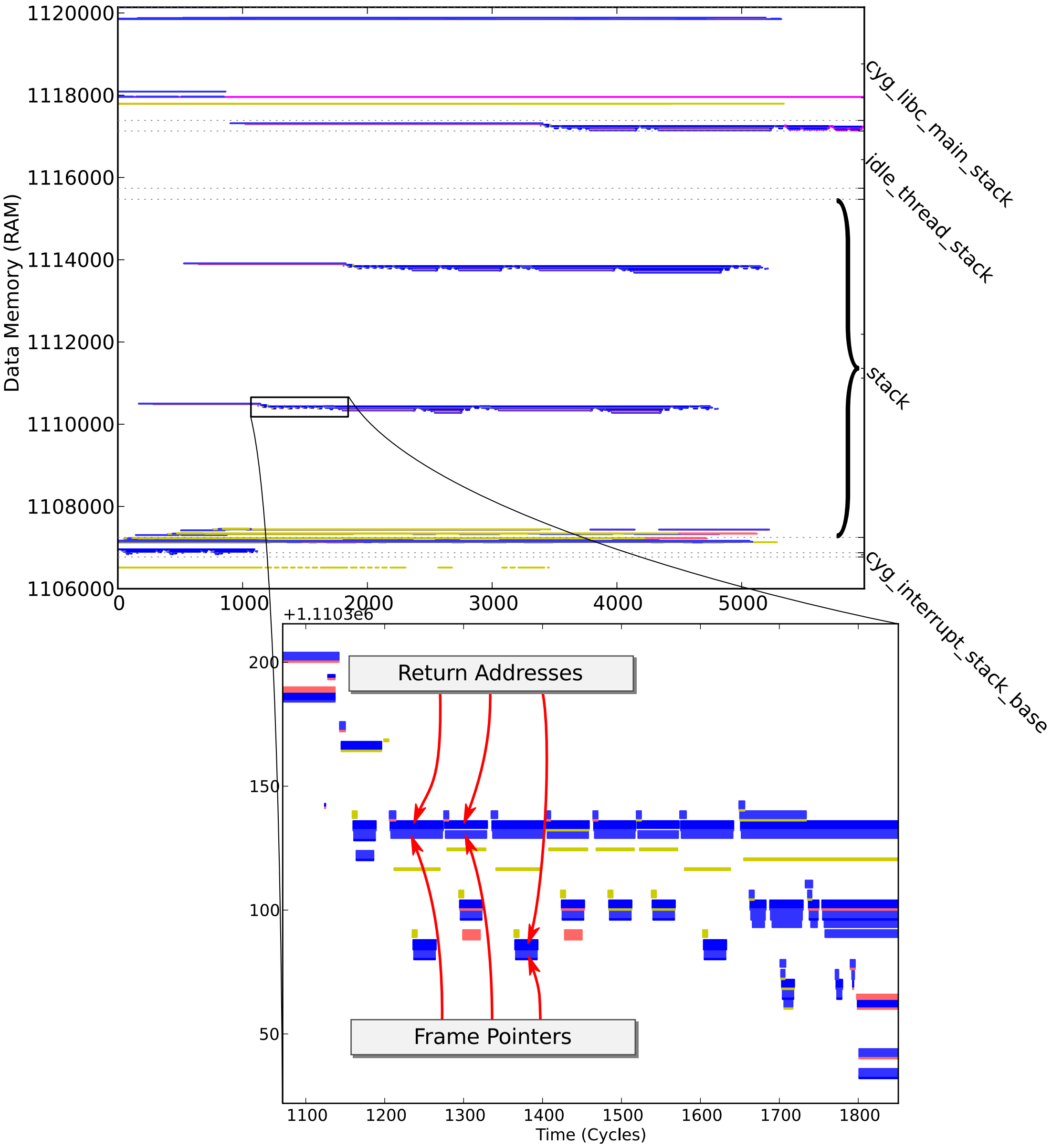


Figure 3.3: Fault space spanned by eCos running the MUTEX1 benchmark. Each point denotes the outcome of an independent benchmark run after fault injection at a specific time and memory coordinate. Injections in *white* areas are benign. *Blue* marks illegal memory accesses outside the data segments. CPU exceptions are colored *red* and timeouts *yellow*, respectively. *Magenta* data points show SDC. The zoomed-in section identifies return addresses and frame pointers as homogeneous reasons for failures.

3.2.3 *Case Study: L4/Fiasco.OC*

The second case study repeats the baseline evaluation with the operating system L4/Fiasco.OC to validate the previous findings. The microkernel architecture of L4/Fiasco.OC provides multiple address spaces with hardware-based isolation; the microkernel occupies a reserved address space that is always mapped to the virtual memory addresses `0xC0000000` to `0xFFFFFFFF`. Faults in other address spaces cannot corrupt the kernel because of the hardware-based isolation. Thus, I confine the fault injection to the kernel address space.

In this case study, I use the revision 64 of the Fiasco.OC microkernel, which is bundled with the *L4 Runtime Environment (L4Re)* that implements basic system services such as program loading. The L4Re repository contains several example programs, which I use for evaluating the microkernel under load. This section takes the program UIRQ as an example. Appendix A.2 confirms the findings of this section by presenting the evaluation results of two more programs.

The program UIRQ tests the interrupt functionality of the kernel. At first, one thread allocates a kernel object that emulates a hardware interrupt. After that, the same thread triggers a virtual interrupt several times, while another thread waits and receives the interrupt signals. Both threads catch runtime errors by C++ exceptions.

Table 3.5 summarizes the results from an exhaustive fault-space scan of the kernel address space[4]. The ten most failure-prone symbols account for 97.2 percent of the total failures. On the one hand, 53.4 percent originate from one instance of the data type `Ready_queue`, which implements priority-based scheduling. On the other hand, 37.6 percent stem from a large memory region called `Physmem` that represents a dynamically managed *heap* of kernel objects. For example, the kernel dynamically allocates an instance of the data type `Thread` for each thread of control that is started at runtime. Likewise, the symbol `Capabilities` refers to another heap that stores instances of data structures that implement access control. Future work needs to analyze these heaps in more detail, yet, most failures are caused by a few statically allocated data structures.

A further, manual analysis of the statically allocated data structures reveals that their four lowest bytes are particularly vulnerable. For instance, the four lowest bytes of the symbol `_fcon` cause 49.2 percent of the symbol's failures. Likewise, the four lowest bytes of `the_timeslice_timeout` and `Kconsole::_c` amount to 30.4 percent and 19.8 percent of their symbol's failures, respectively. The debugging information shows that the GNU C++ compiler uses these bytes as *virtual-function pointers (vptrs)* that implement the dynamic dispatch of virtual C++ functions [240, p. 67f]. Figure 3.4 illustrates

[4] The CPU clock frequency of the x86 emulator Bochs, used by FAIL*, is set to 1.0 GHz. In contrast to eCos, L4/Fiasco.OC refuses bootstrapping on an emulated CPU with 2.666 GHz.

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
`Sched_context::rq`	`Ready_queue`	1,036	$2.47 \cdot 10^{13}$	53.4 %
`Physmem`	*Heap*	62,914,560	$1.74 \cdot 10^{13}$	37.6 %
`Timeout_q::timeout_queue`	`Timeout_q`	48	$6.07 \cdot 10^{11}$	1.3 %
`Capabilities`	*Heap*	41,943,040	$5.94 \cdot 10^{11}$	1.3 %
`Kconsole::_c`	`Kconsole`	56	$4.84 \cdot 10^{11}$	1.0 %
`vga`	`Vga_console`	72	$3.32 \cdot 10^{11}$	0.7 %
`the_timeslice_timeout`	`Timeslice_timeout`	28	$3.15 \cdot 10^{11}$	0.7 %
`_fcon`	`Filter_console`	80	$1.95 \cdot 10^{11}$	0.4 %
`Cpu::cpus`	`Cpu`	224	$1.92 \cdot 10^{11}$	0.4 %
`Rcu::_rcu_data`	`Rcu_data`	44	$1.92 \cdot 10^{11}$	0.4 %

Table 3.5: Quantitative fault-injection results of the L4/Fiasco.OC microkernel running the UIRQ program. The ten most failure-prone symbols (continuous memory regions) amount to 97.2 percent of the total failures. The `Ready_queue` data type is the most critical. In addition, the large memory regions `Physmem` and `Capabilities` represent dynamically allocated memory (*heap*). These heaps also store critical kernel data structures.

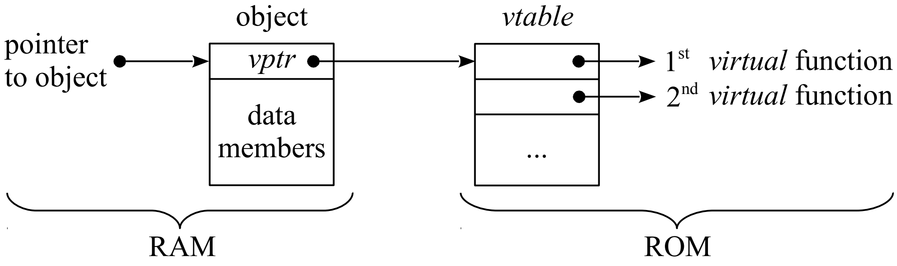


Figure 3.4: Memory layout of C++ objects, virtual-function pointers (*vptrs*) and virtual-function tables (*vtables*) used by common C++ compilers, such as the GNU C++ compiler. The integrity of the *vptr* is critical for correct control flow.

Virtual-function pointers

the C++ object layout used by the GNU C++ compiler for instances of data structures with virtual functions. On a virtual-function call, the *vptr* is dereferenced to locate the virtual-function table (*vtable*), which contains pointers to functions. Thus, a fault in the *vptr* causes a control-flow error.

In summary, the evaluation of the L4/Fiasco.OC microkernel confirms the findings of the *eCos* case study. The reliability of the kernel primarily depends on a few kernel data structures, such as the scheduler's ready queue. Compiler-generated virtual-function pointers represent another homogeneous reason for failures.

3.3 interpretation of the results

The previous case studies identified the individual kernel data structures that are critical to an extremely high degree. These critical data structures alone account for almost all failures. Therefore, applying software-implemented fault-tolerance mechanisms to these data structures would avoid almost all failures and, thus, would significantly improve the reliability of the operating systems.

In summary, the following types of kernel data structures exhibit an exceeding criticality:

- Instances of C++ classes (allocated statically or dynamically)
- Static data members of C++ classes
- Stacks (including return addresses and frame pointers)
- Pointers (including virtual-function pointers)

A software-implemented fault-tolerance mechanism for operating systems must address these data types. The exceeding criticality of C++ classes is a consequence of the object-oriented implementation of both the eCos kernel and L4/Fiasco.OC. However, the class feature of the C++ language itself is *not* the reason for failures – it is the runtime data that are stored in the individual data members of class instances.

The uneven distribution of kernel failures, which mostly stem from a few data structures, can be explained by reconsidering the *probability of failure* derived in Section 3.1.2. According to Equation 3.4 on page 43, the probability of failure is directly proportional to the runtime Δt. This proportionality implies that data structures that are used over a longer period of time have a higher potential of causing failures than data structures with short lifetime. In other words, the criticality of a data structure depends on its lifetime. This correlation explains why the statically allocated kernel data structures, such as the scheduler, are so very critical – their lifetime is virtually unbounded.

On the other hand, the lifetime of many kernel data structures is bound to the application programs. For example, the lifetime of an instance of the class `Cyg_Thread` (eCos) begins when a thread of control is started and ends on thread termination; the class `Cyg_Mutex` does not cause any failure if there is no application program that uses it (see Section 3.2.2.1). Thus, the criticality of many kernel data structures depends on the application profile.

This calls for a selective placement of fault-tolerance mechanisms based on application knowledge. Unnecessary overhead is avoided if uncritical data is left unprotected. The avoidance of runtime overhead is crucial for the effectiveness of fault tolerance, as the runtime is directly proportional to the probability of failure (see Equation 3.4 on page 43). A fault-tolerance mechanism, applied to a subset of data structures, must not excessively increase the lifetime of the other data structures.

3.4 SUGGESTED APPROACH

The selective, application-specific placement of fault-tolerance mechanisms into the kernel rules out any manual implementation in C++: Whenever an application program changes, the kernel of the operating system would need manual adaptation. Available compiler-based solutions are also too inflexible in that they do not provide any means for a selective placement of fault-tolerance mechanisms into kernel data structures.

The goal of this thesis is to develop and evaluate a hybrid approach between these two extremes. On the one hand, the introduction of fault-tolerance mechanisms into the kernel should be automated by a software tool like a compiler. On the other hand, the fault-tolerance mechanisms themselves should be implemented in a general-purpose programming language to allow for user-defined extensions. Such a programming language must provide means to implement *generic* fault-tolerance mechanisms, and the programmer must be able to specify *where* these generic mechanisms shall be applied.

Aspect-Oriented Programming (AOP) is a promising candidate for accomplishing these goals. AOP provides language support for the transparent extension of a program by user-defined functionality, and a compiler for the AOP language automates the extension process.

This thesis evaluates the suitability of AOP for the implementation of generic fault-tolerance mechanisms that shall be applied selectively to critical parts of operating systems. Moreover, this thesis investigates whether AOP allows implementing fault-tolerance mechanisms in such a generic way so that they can be applied to various kernel data structures of both eCos and L4/Fiasco.OC.

3.5 CHAPTER SUMMARY

This chapter introduced the methodology of fault injection to evaluate the reliability of operating systems with respect to transient memory errors, and identified thereby two central problems:

PROBLEM 1: Several kernel data structures, such as instances of C++ classes, stacks, and pointers, exhibit an exceeding vulnerability to transient memory errors. Integrity of these kernel data structures is crucial for the reliability of the operating systems.

PROBLEM 2: The individual criticality of the kernel data structures depends on the application profile. The lifetime of many kernel data structures is bound to the application programs that use

> them. As applications change, the individual criticality of the kernel data structures varies dramatically.

Furthermore, Section 3.1.2 affirmed that runtime overhead has a negative impact on reliability. Therefore, I propose a selective, application-specific placement of fault-tolerance mechanisms into the kernel to avoid unnecessary overhead. Such an approach, however, is impractical with traditional techniques and tools.

AOP might be a solution for these problems. Hence, the following chapters evaluate the suitability of AOP for implementing a library of generic fault-tolerance mechanisms. This thesis evaluates thereby whether AOP is an effective and efficient technology towards resolving the following shortcoming:

> “ There is a need for application-specific, fault-tolerant techniques that offer a trade off between the reliability improvement and amount of overhead. ”
>
> – Tanay Karnik and associates [125, p. 140]

4 ASPECT-ORIENTED PROGRAMMING

In the 1970s, Dijkstra [68, p. 211] describes the *separation of concerns* as important principle of thinking in general and programming in particular. A concern is an area of interest, such as fault tolerance. The separation of concerns permits a programmer to deal with the many concerns of a complex software system one by one. Such a separation of concerns emerges from the programmer's mind; however, it depends on the programming language whether the individual concerns can be implemented also as independent software modules.

Separation of concerns

Aspect-Oriented Programming (AOP) [131] provides extra language support for the separation of concerns at the implementation level. The goal of AOP is to enable a modular program structure even where traditional programming paradigms, such as object-oriented programming, deny modularity.

Software-implemented fault tolerance is a concern that is hard to implement as independent module by object-oriented programming, as shown in Section 4.1 on the following page. Subsequently, this chapter introduces the fundamental principles of AOP. Section 4.2 shows these principles in practice by taking the AspectC++ programming language as an example and evaluates its impact on the fault tolerance of operating systems. Such an evaluation indicates whether the AspectC++ programming language is suitable for the domain of fault tolerance.

Section 4.3 discusses further aspect-oriented languages. Section 4.4 addresses frequent points of criticism concerning AOP in general. Finally, Section 4.5 reviews prior work on fault tolerance using AspectC++ and Section 4.6 summarizes this chapter.

RELATED PUBLICATIONS

The findings presented in this chapter have partly been published in:

[28] Christoph Borchert, Daniel Lohmann, and Olaf Spinczyk. CiAO/IP: A highly configurable aspect-oriented IP stack. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*. ACM Press, June 2012. doi: 10.1145/2307636.2307676

Acceptance rate: 18 %

Acceptance rate: 63 %

[109] Martin Hoffmann, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Rüdiger Kapitza, Olaf Spinczyk, and Daniel Lohmann. Effectiveness of fault detection mechanisms in static and dynamic operating system designs. In *Proceedings of the 17th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '14)*. IEEE Press, June 2014. doi: 10.1109/ISORC.2014.26

4.1 GENERAL CONSIDERATIONS

The major goal of AOP is to manage a separation of concerns at the implementation level. AOP addresses thereby one of the shortcomings of object-oriented programming: the lack of means to implement certain concerns as independent modules so that they do not cut across each other. The AOP literature refers to this deficiency as the problem of *crosscutting concerns* [131].

4.1.1 *Crosscutting Concerns*

The canonical example of a crosscutting concern is program *tracing*. Symptoms of such a concern are the *scattered* and *tangled* source code of the implementation.

SCATTERING: The implementation of a concern is *scattered* if multiple modules contain pieces of code associated with the same concern. For example, the repetition of similar code fragments in many modules causes scattering.

TANGLING: The implementation of a concern is *tangled* if it embraces multiple concerns simultaneously. The intermixing of code fragments from multiple concerns in one program module results in tangling.

Both symptoms refer to the same problem that originates from a limitation of the programming language. Although a concern can be well modularized in the programmer's mind – for example, *tracing* – the programmer has no means to implement that concern without interfering with many program modules. Thus, as scattering and tangling depend on the programming language, a concern that causes scattering and tangling is said to be a *crosscutting concern because of the programming language*.

Figure 4.1 illustrates the problem of crosscutting concerns by taking the example of a simplified linked list, which is implemented in object-oriented C++. In addition to the basic functionality of the list, the concern of *error detection by duplication* (see Section 2.4.1.1 on page 24) is implemented, highlighted on colored background. The im-

```
 1 #include <exception>
 2 class Item {
 3   Item* next;
 4   Item* redundant_next;
 5 public:
 6   Item() : next(0),
 7     redundant_next(0) {}
 8   void set_next(Item* next) {
 9     this->next = next;
10     redundant_next = next;
11   }
12   Item* get_next() const {
13     if (next != redundant_next)
14       throw std::exception();
15     return next;
16   }
17 };
18 class List {
19   Item* head;
20   Item* redundant_head;
21 public:
22   List() : head(0),
23     redundant_head(0) {}
24   void push_front(Item* item) {
25     item->set_next(front());
26     head = item;
27     redundant_head = item;
28   }
29   Item* front() const {
30     if (head != redundant_head)
31       throw std::exception();
32     return head;
33   }
34 };
```

Figure 4.1: Scattering and tangling in the C++ implementation of a simplified linked list. The basic functionality of the pointer-based list is printed on white background. In addition, the concern of *error detection by duplication* is implemented, which replicates each pointer variable. Lines of source code that belong to the error-detection concern are highlighted on colored background. The scattering and tangling impairs the readability of the source code.

plementation of the concern of error detection notably cuts across the basic functionality of the linked list even in this simplified example.

In particular, the member variables of the classes `Item` and `List` require duplicate counterparts (lines 4 and 20); these duplicates need initialization (lines 7 and 23) and regular updates (lines 10 and 27); consistency checks are needed before read access (lines 13 and 30), coupled with proper error handling (lines 14 and 31).

On the one hand, the code scattering over the classes `Item` and `List` makes the concern of error detection hard to read and impossible to study in isolation. For instance, you might want to verify that no consistency checks are missing. On the other hand, the tangling substantially complicates the source code of the basic functionality.

Kiczales and associates [132] further differentiate between two types of crosscutting in object-oriented programming languages according to the influence on the program:

static crosscutting: The concern overlapping becomes manifest in the static program structure, that is, in the data types of the program. For example, a concern implementation that extends existing C++ classes by additional data and function members exhibits static crosscutting. The same applies to a scattered declaration of compile-time assertions. In summary, the runtime behavior of the program is not affected.

DYNAMIC CROSSCUTTING: The interaction of multiple concerns at runtime represents dynamic crosscutting. This type of crosscutting becomes manifest in the execution of a tangled sequence of program statements that belong to different concern implementations. In short, the program execution flow is subject to concern overlapping.

Static crosscutting is typically associated with dynamic crosscutting. In Figure 4.1, the variable duplication (lines 4 and 20) *statically* cuts across both classes, and so does the `#include` directive in the first line. On this basis, the remaining colored lines represent *dynamic* crosscutting that influences the runtime behavior by executing concern-specific program statements.

4.1.2 *AOP – Quantification and Obliviousness*

The motivation for AOP is to manage crosscutting concerns in a modular way. Therefore, AOP addresses the symptoms of code scattering and tangling. Filman and Friedman define AOP as the combination of "quantification and obliviousness" [82, p. 21].

QUANTIFICATION: The ability of quantification denotes that a multitude (a specific *quantity*) of source-code locations are augmented transparently with additional functionality. The functionality itself, however, is encapsulated in a separate module. Therefore, quantification avoids code scattering.

OBLIVIOUSNESS: The source code of a program needs not be aware of – and not prepared for – the augmentation. Thus, obliviousness resolves code tangling in the implementation.

An AOP language must provide means to specify the quantification property in terms of language constructs. Consequently, these language constructs allow for a separation of the concern functionality (the *what*) from the points of instantiation (the *where*). This separation is the fundamental characteristic of AOP.

4.1.3 *Language Support*

AOP languages revolve around the notion of *join points* and *advice* [132]. A join point represents an identifiable location in the source code of a program (the *where*), such as the execution of a member function. Join points statically and dynamically cut across the implementation of a program. *Advice*, on the other hand, specifies the desired actions (the *what*) that shall take place at selected join points.

Essentially, advice provokes the implicit invocation of additional functionality.

The individual join points, which advice shall be applied to, are captured by a *pointcut description language*. Primarily, the pointcut description language provides means for quantification. As such, a *pointcut* declaratively identifies a set of join points. The set can be potentially open if the pointcut description language supports wildcard symbols.

A pointcut description language together with an advice construct are the building blocks that form an AOP language. An *aspect* – in the terminology of AOP – is then a module that encapsulates advice, pointcut expressions, and other implementation artifacts of a crosscutting concern.

4.1.4 *Weaving*

Weaving [131] describes the process of composing the final system from the individual modules. The control flow at selected join points must be transferred to the aspect code and back. This process requires tool support in form of an aspect *weaver*. Essentially, the weaver makes sure that the advice is carried out. Thus, advice can be considered as transformation rules that are processed by the weaver, which can be realized as compiler or interpreter. In either case, the weaver does not modify the original source code. Figure 4.2 depicts the weaving process schematically.

In general, weaving can be distinguished between *static weaving*, which is carried out at compile time, and *dynamic weaving* after compilation. Static weaving, on the one hand, can be implemented efficiently as source-to-source translation; the individual source-code files of a system are woven together into temporary source code that can be compiled with commodity compilers into an executable. Therefore, weaving can be applied without any overhead on the final executable [231].

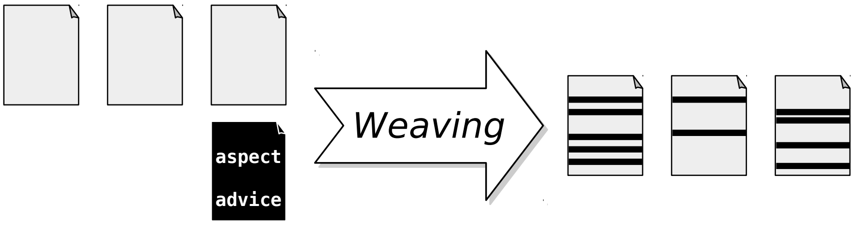


Figure 4.2: Schematic of aspect weaving. On the left-hand side, three files of source code coexist with one aspect module, which is shown in black and encapsulates a crosscutting concern. The weaving process instruments the final program at the relevant join points. On the right-hand side, the black bars indicate those locations in the source code that are augmented with aspect code.

Dynamic weaving, on the other hand, takes place at program load time [196] or at runtime [77, 243]. Therefore, dynamic weaving necessarily incurs runtime costs and, thus, is unfavorable for the domain of fault tolerance, because any runtime overhead has a negative impact on fault tolerance (see Equation 3.4 on page 43).

4.2 THE ASPECTC++ LANGUAGE AND COMPILER

AspectC++ [231] is a general-purpose language extension to C++. As such, AspectC++ is a superset of C++, implying that every valid C++ program is also a valid AspectC++ program. The language extension comprises a minimal set of new language constructs to enable AOP, which integrate seamlessly into the complex C++ grammar. AspectC++ aims at complying with the C++ philosophy and its peculiarities.

The AspectC++ language is bundled with an open-source[1] compiler that implements a source-to-source translation from AspectC++ to standard C++. Thus, the weaving process is carried out statically and integrates well into existing C++ tool chains. The compiler also supports dynamic weaving; however, the join points are still prepared statically for dynamic instrumentation. This thesis focuses on static weaving, which is more efficient in general.

4.2.1 *Language Concepts*

This section briefly introduces the basic language concepts of AspectC++ 1.0, whereas the next chapter covers the advances of the version 2.0. Throughout this section, I refer to Figure 4.3 for illustrating the individual language concepts. That figure resembles the previous example of the simplified linked list with crosscutting error detection (compare to Figure 4.1 on page 61). However, Figure 4.3 shows an implementation using AspectC++: The concern of error detection is encapsulated in a single module on the right-hand side. The `aspect` keyword (line 3) defines a `class`-like entity that permits the declaration of *pointcut expressions* and *advice*.

4.2.1.1 *Pointcut Description Language*

As mentioned in the previous section, a pointcut description language is one of the building blocks of an AOP language. Such a description language allows declaring pointcut expressions that identify sets of join points. The pointcut description language of AspectC++ distinguishes between *name pointcut expressions* and *code pointcut expressions*.

[1] The AspectC++ compiler is available at `http://aspectc.org/` under the terms of the GNU General Public License.

```
class Item {
  Item* next;
public:
  Item() : next(0) {}
  void set_next(Item* next) {
    this->next = next;
  }
  Item* get_next() const {
    return next;
  }
};

class List {
  Item* head;
public:
  List() : head(0) {}
  void push_front(Item* item) {
    item->set_next(front());
    head = item;
  }
  Item* front() const {
    return head;
  }
};
```

(a) `List.h`

```
#include "List.h"
#include <exception>
aspect ErrorDetection {
  advice "Item" || "List" : slice class {
    Item* redundant_ptr;
  };
  advice construction("Item" || "List")
  : before() {
    tjp->target()->redundant_ptr = 0;
  }
  advice execution("Item* Item::get_next()"
    || "Item* List::front()")
  : after() {
    Item* item = *tjp->result();
    if(item != tjp->target()->redundant_ptr)
      throw std::exception();
  }
  advice execution("% Item::set_next(Item*)"
    || "% List::push_front(Item*)")
  : after() {
    Item* item = *tjp->arg<0>();
    tjp->target()->redundant_ptr = item;
  }
};
```

(b) `ErrorDetection.ah`

Figure 4.3: Implementation of a simplified linked list with error detection by duplication using AspectC++. In contrast to Figure 4.1 on page 61, which shows a scattered and tangled implementation using pure C++, the implementation at hand is modularized. (a) The implementation of the basic functionality is clear and concise; the classes `Item` and `List` are oblivious of any error detection. (b) The concern of error detection is encapsulated in a separate `aspect` that transparently augments the classes `Item` and `List`. Details of this example are discussed on the pages 64 to 68.

Name pointcut expressions

Name pointcuts are composed of *quoted match expressions*, which refer to identifiers of a C++ program, such as names of classes and functions. For example, the quoted expression `"List"` represents a name pointcut that matches only the class `List`, whereas the expression `"% List::push_front(Item*)"` matches a member function with a respective signature, that is, with arbitrary result type and only one argument of the type `Item*`. The percent symbol (`%`) serves as wildcard for any name. An ellipsis (`...`) matches any sequence of names, such as a list of function arguments. In summary, name pointcuts describe the static program structure.

Code pointcut expressions

Code pointcuts, on the contrary, refer to dynamic events in the program execution flow, such as the execution of a function. Code pointcut expressions result from applying certain predefined *pointcut functions* to name pointcuts. For instance, the code pointcut expression `execution("% List::push_front(Item*)")` refers to the dynamic execution of the respective function. Likewise, `construction("List")` matches the execution of the `List`'s class constructor. Furthermore, the predefined pointcut function `destruction()` refers to the execution of a class destructor, and `call()` identifies the place of function invocation.

Pointcut algebra

Pointcut expressions of the same type can be combined by the set-theoretic operations of union (`||`), intersection (`&&`), and complementation (`!`). Figure 4.3 exemplifies the union operation on name pointcuts (lines 4, 7, 11f, and 18f). For instance, the pointcut expression `construction("Item" || "List")` in line 7 matches the constructor execution of both classes.

All pointcuts shown in Figure 4.3 are anonymous, that is, they are defined at the place of their usage. However, pointcut expressions can also be *named* and, thus, are reusable by other pointcut expressions. Furthermore, such pointcuts can be declared as `virtual` and may be redefined by derived aspects. Section 6.1 on page 100 gives examples for these advanced pointcut features.

4.2.1.2 *Introductions*

Class slice

The purpose of pointcut expressions is to describe the join points for advice. Name pointcuts, for instance, are the basis for extensions of the static program structure. This language feature is called *introductions*. Syntactically, the `advice` keyword provokes a `slice` of class members to be introduced into the classes that match the given name pointcut. Figure 4.3 illustrates such an introduction. The piece of advice in line 4 inserts the data member `redundant_ptr` (line 5) into the classes `Item` and `List`. Similarly, member functions and base classes can be introduced. Thus, the language feature of introductions provides means to avoid static crosscutting.

4.2.1.3 *Code Advice*

Advice based on code pointcuts, which refer to dynamic events in the program execution flow, is differentiated between `before`, `after`, and `around`. These three keywords specify the temporal relationship between the advice and the dynamic event. For instance, advice specified with the `before` keyword is carried out ahead of the event. The `construction` advice shown in Figure 4.3 (line 7f) initializes a variable `before` the execution of the specified class constructor. On the other hand, the remaining pieces of code advice in Figure 4.3 take place `after` the execution of the respective functions. The *body* of code advice contains ordinary C++ program statements. In both cases, the respective events are executed as usual. However, the `around` keyword can be used to replace the event completely with the advice body. In summary, code advice provides means to invoke the advice body implicitly on certain events to avoid dynamic crosscutting.

before, after, and around

4.2.1.4 *Order Advice*

Given that multiple pieces of advice affect the same join point, the keyword `order` can be used to define a precedence of aspects for that join point. AspectC++ allows defining a partial order of pointcut expressions. For example, the following order definition specifies that an aspect `A` takes precedence over an aspect `B` concerning the given code pointcut:

```
advice execution("% List::front()") : order("A", "B");
```

4.2.1.5 *Join-Point API*

AspectC++ provides a rich join-point API for code advice. The programmer can retrieve thereby context information from the specific join point. In the body of code advice, the keyword `JoinPoint` provides uniform access to compile-time information, such as the involved data types. Furthermore, the keyword `tjp` points to an instance of `JoinPoint` and provides additional runtime information.

For example, `tjp->target()` yields a statically typed pointer to the object involved with the particular code join point. In the case of `construction`, `destruction` and `execution` advice, `tjp->target()` yields the C++ `this` pointer of the affected join point. Figure 4.3 exemplifies the usage of `tjp->target()` in lines 9, 15, and 22.

The join-point API is an extremely powerful mechanism in combination with quantification. In Figure 4.3, all pieces of advice affect both the classes `Item` *and* `List`. Yet, the join-point API provides different information at each join point: Depending on whether the class `Item` or `List` is affected, `tjp->target()` yields a typed pointer to *either* an `Item` or `List` object. Thus, the advice behaves polymorphically with respect to the point of instantiation. The data types are resolved at compile time and enable *static polymorphism* [250, p. 238].

compile-time information	description
`Target`	type of the callee object
`That`	type of the caller object (identical to `Target` except for `call` advice)
`Arg<i>::ReferredType`	type of the *i*th argument
`Result`	type of the function's return value
`JPID`	unique identifier per join point

runtime functions	description
`Target *target()`	pointer to callee object
`That *that()`	pointer to caller object (identical to `target()` except for `call` advice)
`Arg<i>::ReferredType *arg<i>()`	pointer to *i*th argument value
`Result *result()`	pointer to return value
`void proceed()`	invoke replaced join point (only `around` advice)

Table 4.2: Excerpt from the AspectC++ join-point API. In the body of code advice, the keyword `JoinPoint` provides access to compile-time context information (upper part of the table). At the same place, the keyword `tjp` can be used to query runtime values via the listed functions (lower part of the table). Both `JoinPoint` and `tjp` utilize static type information based on the point of advice instantiation. A complete documentation of the join-point API is available online at: `http://aspectc.org/`

AspectC++ supports thereby *generic advice* [155] based on the compile-time join-point API.

In addition to `target()`, the example in Figure 4.3 uses `result()` and `arg<0>()` to obtain the function's return value and first argument, respectively. Table 4.2 summarizes the elements of the join-point API that are relevant for this thesis.

4.2.2 *Case Study: The CiAO Operating System*

This case study evaluates the suitability of AspectC++ for the domain of dependable operating systems. AspectC++ has been used already in the development of the CiAO operating system [156, 154, 157], which is a library operating system that implements the automotive OSEK specification [146]. The primary design goal of CiAO is *strict tailoring* at compile time. CiAO is, like eCos (see Section 3.2), statically configurable for customization of the operating system to the sole requirements of an application. The distinguishing characteristic of CiAO is, however, the use of *aspect-oriented design principles* [156, p. 221], such as loose coupling of all system components. Individual configuration options are implemented modularly without code scat-

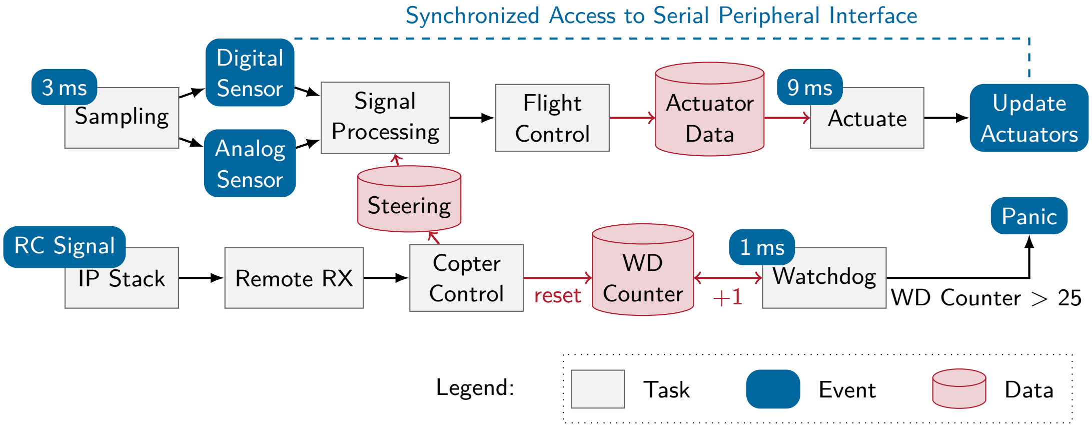


Figure 4.4: Simplified illustration of the I4Copter software architecture. The flight-control application consists of three periodic tasks (1 ms, 3 ms, and 9 ms) and several event-driven tasks. Adapted from Hoffmann and colleagues [109, p. 231].

tering and tangling. Thus, CiAO provides an even higher degree of configurability than eCos.

Two studies point out the superior efficiency of the CiAO kernel [156, p. 225] and its networking subsystem [28, p. 443ff], suggesting that the general-purpose AspectC++ language and compiler are well suited for the domain of resource-constrained operating systems. The reason is that the static weaving, as carried out by the AspectC++ compiler, practically induces no runtime overhead on the resulting executable [231, p. 648ff]: The advice body becomes *inlined* at the affected join points.

Besides performance considerations, the fault tolerance of AspectC++ programs determines the applicability to the domain of dependable operating systems. Therefore, this case study compares the *inherent* fault tolerance of eCos and CiAO by means of fault injection with the experimental setup of the baseline dependability assessment described in Section 3.2.1 on page 47. For comparison, both eCos and CiAO execute the same workload: The flight-control application of the I4Copter [247] quadrotor helicopter represents a real-world safety-critical application that serves as evaluation scenario.

I4Copter flight-control application

Figure 4.4 illustrates the software architecture of the I4Copter. The control application consists of three periodic tasks that are activated at intervals of 1 millisecond, 3 milliseconds, and 9 milliseconds, respectively. These periodic tasks activate further processing tasks via kernel synchronization mechanisms. In addition, one interrupt-driven task receives incoming remote-control (RC) signals via UDP/IP packets and forwards the steering data to a chain of processing tasks. The `Watchdog` task observes the remote-control communication for conti-

FAILURE MODE	ECOS DEFAULT	ECOS BITMAP	CIAO
SDC	$1.50 \cdot 10^{11}$	$1.24 \cdot 10^{11}$	$2.07 \cdot 10^{10}$
Timeout	$5.39 \cdot 10^{10}$	$5.76 \cdot 10^{10}$	$1.62 \cdot 10^{10}$
CPU exception	$1.71 \cdot 10^{11}$	$1.40 \cdot 10^{11}$	$2.42 \cdot 10^{10}$
Total failures	$3.75 \cdot 10^{11}$	$3.22 \cdot 10^{11}$	$6.12 \cdot 10^{10}$

Table 4.3: Quantitative fault-injection results of eCos and CiAO running the I4Copter flight-control application. Each number shows the accumulated count of a particular failure mode that occurs in the exhaustive fault-space scan (see Section 3.2.1 for details on the experimental setup and failure modes). The only difference of the experimental setup is that the simulated CPU is set to 50 MHz. In summary, the CiAO operating system is the most robust because both eCos variants exhibit notably more failures than CiAO.

nuity. In summary, a set of 14 tasks interacts in a strict execution order by using various kernel synchronization mechanisms.

For the fault-injection experiments using the FAIL* framework (see Section 3.2.1 on page 47), a set of predefined inputs and sensor data is provided. The mission of both eCos and CiAO is to reproduce the exact sequence of task execution in the presence of memory errors. Therefore, the task schedule is recorded and compared at the end of each application run, which stops after three hyper periods. A divergence from the expected task schedule disturbs the flight-control logic and counts as Silent Data Corruption (SDC).

Table 4.3 summarizes the results from an exhaustive fault-space scan of the kernel memories. The eCos kernel supports two different scheduler implementations: The *default* multi-level queue implementation uses a pointer-based list data structure to manage an arbitrary number of threads at runtime, whereas the more efficient *bitmap* scheduler supports only a fixed number of threads specified at compile time. CiAO solely implements a bitmap scheduler. The total number of failures shown in the bottom row indicates a clear trend: Tailoring of the operating system at compile leads to more robust systems. The bitmap variant of eCos exhibits 14 percent fewer failures than the default variant. Furthermore, CiAO fails even 81 percent less often than the bitmap variant of eCos. Thus, static tailoring leads to *fault avoidance*. Fewer pointer indirections at runtime imply fewer potential points of failure, decreasing the operating system's vulnerability to memory errors.

The results suggest that AspectC++, the implementation language of CiAO, is suitable for the domain of dependable operating systems. Compared to the pure C++ implementation of the eCos kernel, the CiAO implementation achieves a highly competitive robustness. However, $6.12 \cdot 10^{10}$ points of failure remain in the strictly tailored

CiAO setup. Because the kernel state is condensed to a bare minimum, almost every bit becomes relevant and leads to a failure if corrupted. In other words, fault avoidance by static tailoring alone is not enough: A dependable operating system requires fault-tolerance mechanisms for protection of the critical kernel state. In conclusion, this case study motivates that AspectC++ is a promising candidate for developing such fault-tolerance mechanisms.

4.3 RELATED ASPECT LANGUAGES

In the previous section, AspectC++ has been presented as an example for an AOP language in practice. Yet, there are other programming languages that also address AOP. First and foremost, the introduction of AspectJ [132] marks the rise of AOP. AspectJ uses the Java programming language as *base language* and adds language constructs for pointcut and advice. Section 4.3.1 provides details on AspectJ.

AspectJ inherits the strengths and weaknesses of Java. Likewise, AOP extensions to C, C++, and C# share the properties of the respective base languages. The goal of this section is to give an overview of AOP languages beyond AspectC++ and to discuss their applicability to the domain of dependable operating systems.

4.3.1 *AspectJ and Descendants*

AspectJ established the notion of pointcut, advice, and aspects (see Section 4.1.3). AspectJ defined thereby the understanding of AOP as duality of aspects and classes. Consequently, AspectJ represents a general-purpose *language extension* to the Java programming language and aims at "programmer compatibility" [132, p. 329].

The idea of AOP as a language extension has been adopted by many other languages. AspectC++, for instance, transfers the AspectJ approach to C++. A notable difference between AspectC++ 1.0 and AspectJ is that the latter lacks quantification for introductions [140, p. 95f]. On the other hand, AspectJ allows capturing access to member variables by the additional pointcut functions `get()` and `set()` [132, p. 332]. CaesarJ [11], Josh [51], Sally [99], and LogicAJ [133] build on the AspectJ approach and represent further Java-based AOP languages.

Overall, AspectJ and its Java-based descendants are inapplicable to the domain of dependable operating systems. The reason is that the Java Virtual Machine (JVM) typically interprets Java programs or compiles them just in time. Thus, the JVM runs *prior* to the actual Java program, which is impossible if the program was an operating system. Ahead-of-time compilation of Java is less common and still requires an interpreter or a just-in-time compiler because of dynamic class loading [173]. To the best of my knowledge, there is no off-the-shelf operating system written in Java.

4.3.2 *AOP Extensions to C, C++, and C#*

Several programming languages based on C, C++, and C# adopt the AspectJ model of AOP. This section provides a brief overview with focus on static weaving.

4.3.2.1 *C-Based Languages*

Most operating systems are written in the C programming language because of its runtime efficiency. At first glance, a C-based AOP approach would represent an ideal candidate for the development of fault-tolerance mechanisms that cover operating systems. However, C provides only a limited set of programming abstractions. Its type system neither supports static polymorphism nor dynamic polymorphism. This shortcoming remarkably restricts the potential for generic advice compared to AspectC++ (see Section 4.2.1.5 on page 67). Thus, C is a poor base language for AOP. Nevertheless, several AOP extensions have been proposed.

AspectC [55, 54], on the one hand, is an almost one-to-one mapping of the AspectJ concepts to C. The lack of polymorphism in C is reflected by no quantification in AspectC at all. Furthermore, AspectC neither supports introductions nor pointcut functions for variable access. The C4 approach [85] bases on AspectC but further drops the obliviousness characteristic.

Mirjam/WeaveC [174] and Aspicere [2], on the other hand, overcome the lack of polymorphism by logic meta programming. Their pointcut description languages incorporate the *Prolog* language and enable a binding of logic meta variables to join-point context information. Both languages provide quantification and generic advice, but notably increase the complexity of the pointcut description language. Yet, there are no means to capture variable access as in AspectJ. In summary, these languages are *less programmer compatible* than AspectJ.

Program transformation tools for C also provide obliviousness and quantification to some extent and, thus, are loosely related to AOP. The Bossa framework [1], for instance, allows specifying source-code rewrite rules based on a variant of temporal logic. The Semantic Patch Language (SmPL) [187] provides a more convenient way for writing program transformations independent of the affected source-code locations. Coccinelle [38] implements SmPL and internally applies temporal logic for matching control-flow sequences. In contrast to AOP, Bossa and SmPL focus on robust patches rather than general-purpose programming.

4.3.2.2 *C++-Based Languages*

Besides AspectC++, AspectX/XWeaver [205] addresses AOP for C++ programs. The pointcut description language of AspectX refers to nodes in an XML representation of the C++ program and resembles XPath queries. Thus, the language AspectX is conceptually independent of the base language C++; only the XWeaver implementation needs adaptation to other base languages. The drawback of this approach is that the programmer needs considerable knowledge of the XML representation of the program.

4.3.2.3 *C#-Based Languages*

The Yiihaw weaver [120] and PostSharp [95] are examples for AOP with the C# programming language. C# programs are typically compiled just-in-time because of dynamic class loading, which complicates ahead-of-time compilation. Thus, C# remains an unfavorable language for operating systems. For example, the C#-based Singularity [114] kernel still contains about 6 percent of C++ code.

4.3.3 *AspectAda*

The Ada programming language is particularly successful in the domain of dependable systems, especially in avionics [21]. There is also a real-time operating system implemented in Ada [204]. Ada supports polymorphism, and AspectAda [191] extends this base language by AspectJ-like constructs. Unfortunately, AspectAda provides no means for accessing join-point context information as required by generic advice. In addition, AspectAda neither supports introductions nor pointcuts for variable access. In summary, AspectAda is a promising approach but is not yet technically mature.

4.4 criticism of aop

Despite the many AOP languages – or rather, because of them – the idea of AOP has become a controversial subject. AOP has been "considered harmful" [58], and Steimann raises the following three objections in his essay on "The Paradoxical Success of Aspect-Oriented Programming" [237]:

1. *"AOP adds another dimension of not knowing what just happened"* [237, p. 490]. This point concerns the obliviousness property that thwarts the mental tracing of program execution while looking at the source code of a single module (modular reasoning [130]). "Since an aspect can plug into just about any point of execution of a program, one can never tell the previous (or following) statement of any statement" [237, p. 490]. In other

words, "the net effect on program understandability is not indisputable." [237, p. 493]

2. *"AOP ... breaks modularity"* [237, p. 493]. Steimann argues that granting an aspect access to join-point context information contradicts Parnas' original notion of information hiding [188]. The problem is an implicit dependency from the aspect to data hidden in a module, which leads to a strong coupling between the aspect and the module, impairing independent development [237, p. 488]. This issue also becomes manifest as *fragile pointcut problem* [239] if a module's implementation changes so that an aspect silently becomes defective, for instance, because a pointcut does not match anymore.

3. *"The number of useful aspects is not only finite, but also fairly small"* [237, p. 482]. Steimann brings forwards that AOP only addresses technical, nonfunctional concerns, such as tracing, logging, synchronization, caching, and so on [236, p. 175f]. According to that, "aspects are not domain level abstractions and thus lack a significant source of diversity." [237, p. 482]

I agree that, in general, AOP *can* cause the first two problems. However, this thesis uses AOP in particular to implement *transparent* fault-tolerance mechanisms. Transparent fault-tolerance mechanisms do not interfere with the mainline program semantics and, thus, do not cause the first problem concerning modular reasoning. A transparent fault-tolerance mechanism respects program invariants and does not change any mainline program state in the fault-free case; error correction is yet the only way of potential data interference. The motivating example in Figure 4.3 on page 65 shows that modular reasoning of the mainline program is not affected at all by the aspect-oriented error-detection mechanism. Dantas and Walker coined the term *"harmless advice" [63]* for describing such pieces of advice that obey to a weak non-interference property. Harmless advice may change the timing and termination behavior, but does not otherwise affect the results of a program. Likewise, I consider the detection and correction of memory errors as harmless in the sense that the mainline program semantics remains unaffected. Thus, this thesis pursues a disciplined approach to AOP in the style of harmless advice for facilitating modular reasoning and program understandability.

Harmless advice

The second point of critique (coupling between aspects and mainline code) is true for the motivating example in Figure 4.3 on page 65: The shown introduction in line 5 replicates a pointer variable of the hard-coded type `Item*`, and the pieces of execution advice depend on the semantics of the advised member functions. Consequently, changes to the mainline code likely break the exemplified aspect, which is therefore *not* truly modular. The following chapter presents language extensions to AspectC++ 1.0 to avoid such a coupling between aspects and mainline code. Thus, the second point of critique can be ruled out as well.

Finally, Steimann argues that AOP only addresses technical, nonfunctional concerns that "lack a significant source of diversity" [237, p. 482]. Indeed, software-implemented fault tolerance is a very technical and nonfunctional concern. However, Section 2.4 (pages 23 to 37) provides evidence that the domain of software-implemented fault tolerance is a huge area of research on its own with enormous diversity. Implementing all these fault-tolerance mechanisms clearly calls for a general-purpose programming paradigm such as AOP. The next section presents examples for actually "useful aspects" [237, p. 482].

4.5 PRIOR WORK ON FAULT TOLERANCE USING ASPECTC++

Several studies investigate the feasibility of software-implemented fault tolerance, as reviewed in Section 2.4, using the AspectC++ language. Gal and colleagues [90] propose the replication of message passing in distributed systems. Alexandersson and Karlsson [7] implement the replication of procedure calls, whereas Afonso and associates [5] address the replication on the thread level. Alexandersson and Öhman [8] also discuss the implementation of control-flow checking, executable assertions, and checkpointing. Karol and colleagues [126] propose arithmetic coding via manual introduction of wrapper classes augmented by aspects. Yet, these approaches have been applied only to the application level, and some are not applicable to the kernel of an operating system at all, such as program replication on the thread level [5]. Furthermore, they do not support either error correction or multi-threading (see Section 2.4.3).

To the best of my knowledge, only Afonso and associates [4] address aspect-oriented fault tolerance of an operating-system kernel. They propose error detection for a semaphore data structure like the motivating example in Figure 4.3 on page 65. Hence, their approach suffers from the very same coupling problem pointed out in the previous section and, thus, is not applicable to other data structures. Moreover, their approach verifies only a semaphore-specific precondition and does not cover error correction.

4.6 CHAPTER SUMMARY

This chapter showed in Section 4.1 that software-implemented fault tolerance is a concern whose implementation often cuts across the business logic of multiple program modules. The root cause of this problem is the inability of object-oriented programming to modularize such a concern in a way that it can be separated from other program modules. A remedy to this defect is AOP.

Section 4.2 took the AspectC++ language as an example to illustrate the fundamental principles of AOP. In addition, that section evaluated the inherent fault tolerance of the CiAO operating system, which is written in AspectC++ and outperforms the pure C++ implementation of eCos. Thus, AspectC++ is a preferable aspect language for developing dependable operating systems.

Finally, Section 4.5 reviewed prior work on software-implemented fault tolerance using AspectC++. Yet, these approaches are either not applicable to the kernel of an operating system or do not support the correction of memory errors. Moreover, most of these approaches suffer from a strong coupling between the aspects and mainline code – a frequent point of critique on AOP in general (see Section 4.4). The next chapter describes the essential language extensions to AspectC++ to overcome these issues.

5 ASPECTC++ 2.0 – LANGUAGE EXTENSIONS

The previous chapter introduced AspectC++ 1.0 and reviewed state-of-the-art approaches to fault tolerance using that language. Yet, these approaches are not generic enough to be broadly applicable to the kernels of various operating systems. On the one hand, the approaches do not cover the exceedingly critical data types identified in Section 3.3, such as pointers and data members of C++ classes. On the other hand, most of the approaches are coupled to the mainline program code and, thus, are not truly modular (see Section 4.5). This chapter addresses these shortcomings and presents AspectC++ language extensions to overcome these issues.

The following table summarizes the requirements identified in Section 3.3 and Section 4.5 that are not supported by AspectC++ 1.0. For each requirement, I propose a respective language extension, which is discussed in a separate section as follows:

REQUIREMENT	LANGUAGE EXTENSION
Fault-tolerant pointers	Advice for built-in operators (Section 5.1)
Fault-tolerant data members	Advice for data access (Section 5.2)
Redundancy without coupling	Generic introductions (Section 5.3)
Minimal runtime overhead	Whole-program analysis (Section 5.4)

These four language extensions have been developed in the course of this thesis. After all, the extensions led to the official AspectC++ 2.0 language and compiler release in 2016, which is publicly available online at: `http://aspectc.org/`

RELATED PUBLICATIONS

The findings presented in this chapter have partly been published in:

[26] Christoph Borchert and Olaf Spinczyk. Hardening an L4 microkernel against soft errors by aspect-oriented programming and whole-program analysis. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS '15)*. ACM Press, October 2015. doi: 10.1145/2818302.2818304

Acceptance rate: 44 %, also appeared in ACM Operating Systems Review [27]

5.1 ADVICE FOR BUILT-IN OPERATORS

A close reexamination of the eCos case study in Section 3.2.2 shows that five of the ten most failure-prone program symbols are of pointer data type (see Table 3.3 on page 50). For example, eCos invokes the kernel's interrupt routines indirectly via the global array of function pointers `hal_interrupt_handlers`, which causes control-flow failures if corrupted by memory errors. Pointers are basic data types defined by the C++ standard [116, § 3.9.2], but AspectC++ 1.0 does not provide any means to specify advice for such basic data types. Thus, a fault-tolerance mechanism for pointers requires an appropriate language extension for basic data types.

Built-in operators

According to the C++ standard [116, § 13.6], *built-in operator functions* are used for pointer dereferencing and other computations with basic data types. For instance, the expression `1+2` is evaluated by a built-in operator of the form `"int operator +(int, int)"`, which represents the addition of basic integer (`int`) operands. The fundamental idea of an AspectC++ language extension for basic data types is to capture the invocation of such built-in operator functions.

Overloaded operators

In addition, operator functions can be overloaded for user-defined data types; the C++ standard specifies that "uses of overloaded operators are transformed into function calls" [116, § 5]. Therefore, AspectC++ 1.0 already captures the invocation of overloaded operators by the `call` pointcut function. For example, the pointcut expression `call("% ...::operator %(...)")` captures any overloaded operator.

As a language extension for basic data types, I propose a complementary pointcut function for built-in operators, such as the pointer dereference operator.

5.1.1 *Pointcut Function for Built-in Operators*

Table 5.3 defines the syntax and semantics of the pointcut function `builtin`, which extends the pointcut description language of AspectC++. This pointcut function captures the invocation of built-in operators much as the pointcut function `call` captures user-defined functions and overloaded operators (see Section 4.2.1). Likewise, the pointcut function `builtin` transforms a name pointcut expression into a code pointcut expression as required for code advice. The only argument of the pointcut function `builtin` is a name pointcut expression that describes the signature of one or more built-in operator functions as specified by the C++ standard [116, § 13.6]. For example, `builtin("int operator +(int, int)")` captures all invocations of the binary operator that adds two signed integers at runtime.

Pointcut function `builtin`

By this means, code advice can be specified either `before`, `after`, or `around` the invocation of built-in operators. Because the body of code advice can contain arbitrary program statements, including noncon-

POINTCUT	DESCRIPTION
builtin(pointcut)	The pointcut function captures all code join points where a built-in operator function is called that matches the pointcut expression. Calls to built-in operator functions within constant expressions that evaluate at compile time are not captured.

Table 5.3: Specification of the pointcut function builtin, which extends AspectC++'s pointcut description language.

stant expressions that evaluate at runtime, advice for built-in operators within constant expressions that evaluate at compile time is not supported[1].

The uniform join-point API of code advice (see Section 4.2.1.5) is also available for advice based on the builtin pointcut function. For example, the join-point API provides typed pointers[2] to the built-in operator's arguments and return value. In summary, the builtin pointcut function represents a minimal – yet powerful – language extension to AspectC++ for dealing with basic data types.

5.1.2 *Example: Range Check of Function Pointers*

Although the builtin pointcut function is a general-purpose language feature, this thesis focuses on fault tolerance. For example, Figure 5.1 shows the complete implementation of a highly generic mechanism that applies a *range check* on all function pointers. The valid address range of function pointers is confined by the GNU program linker, which usually defines the two symbols[3] __executable_start and _etext that enclose the code segment of program instructions (see line 2 in Figure 5.1).

Any indirect function call via function pointer involves a dereference operation of the pointer prior to the call. The unary built-in operator of the form "T& operator *(T*)" performs the dereference operation for any data type T. The quoted match expression in line 6 describes *any* dereference operator that takes a function pointer as argument using the wildcard symbols percent and ellipsis. Thus, the

[1] The exclusion of constant expressions from advice does not impair the development of software-implemented fault-tolerance mechanisms, because the result of a constant expression is precomputed at compile time. Thus, the respective built-in operator function is not subject to any faults of the target hardware.

[2] The C++ standard prohibits pointers to certain data types, such as bit fields [116, § 5.3.1]. Advice for built-in operators whose arguments or return value cannot be represented in the join-point API is omitted. The AspectC++ 2.0 language reference manual provides a complete list of limitations and is available online at: http://aspectc.org/

[3] The linker script of eCos defines the symbol _stext instead of __executable_start.

```
1  #include <functional> // for exception: std::bad_function_call
2  extern "C" void __executable_start(), _etext(); // linker symbols
3
4  aspect FunctionPointerCheck {
5    // capture the dereference operator of any function pointer
6    advice builtin("% operator *( % (*)(...) )") : before() {
7      // use the join-point API to get the function pointer
8      void (*func_ptr)() = (void (*)()) *tjp->arg<0>();
9
10     if (func_ptr < &__executable_start || func_ptr > &_etext)
11       throw std::bad_function_call(); // range check failed
12   }
13 };
```

Figure 5.1: Implementation of a range check of function pointers using AspectC++ 2.0. The `builtin` pointcut function in line 6 captures the invocation of the built-in dereference operator *. In the same line, the quoted match expression selects those operators whose argument is a function pointer. The type conversion in line 8 is necessary for the subsequent comparison with the linker-defined symbols that are of different type than the actual function pointer, obtained by `*tjp->arg<0>()`.

advice definition in line 6 refers to all[4] built-in dereference operations of function pointers in a program, and adds the desired range check `before` the dereference operation is carried out.

Line 8 shows the initialization of the local variable `func_ptr` by using the join-point API for retrieving the actual function pointer to be dereferenced. Then, the pointer value is checked for validity (line 10), and an exception is thrown for error handling if necessary (line 11).

In summary, the example in Figure 5.1 illustrates the high degree of expressiveness of the `builtin` pointcut function. Only thirteen lines of code are necessary to check *every* function pointer in a generic, modular, and portable way. The shown implementation can be customized easily by providing other range limits, custom error handling, or a restriction to certain types of function pointers. In addition, a similar check of data pointers could be implemented as well. Finally, advice for the numerous other built-in operators, such as the addition operator `+`, enable the development of further fault-tolerance mechanisms as shown in the next chapter.

[4] In C/C++, a function call using a function pointer can be provoked without *explicit* invocation of the operator *. The pointcut function `builtin` also captures the *implicit* dereferencing of function pointers as invocation of the operator * since AspectC++ 2.1.

5.2 ADVICE FOR ACCESS TO VARIABLES

The second language extension to AspectC++ enables advice for access to program variables. In particular, member variables of C++ classes are exceedingly critical as pointed out in Section 3.2: The instances of classes and their member variables clearly dominate the ten most failure-prone program symbols of eCos (see Table 3.3 on page 50) and L4/Fiasco.OC (see Table 3.5 on page 54). Thus, a fault-tolerance mechanism must address these member variables.

AspectJ supports advice for member variables by the pointcut functions `get` and `set` [140, p. 74], whereas AspectC++ 1.0 lacks such a functionality. This section describes an AspectJ-like extension to AspectC++ for capturing access to member variables. Moreover, this section deals with the peculiarities of the C++ language, such as global variables and aliasing of variables.

5.2.1 *Name Pointcut Expressions for Variables*

Quoted match expressions for variables

The pointcut description language of AspectC++ 2.0 allows declaring name pointcut expressions for referring to variables. Such a name pointcut expression must contain the *data type*, an optional *scope* pattern, and the *name* of the variable. For example, the expression `"int Semaphore::count"` describes a variable of the basic data type `int`. Furthermore, that variable is a member of the class or namespace `Semaphore`, which is in turn located in the global namespace, and the variable is called `count`. The wildcard symbols percent and ellipsis are also applicable. For instance, `"% ...::%"` matches variables of any type (`%`), declared as member of any sequence of nested scopes (`...`) including the global namespace, and any name (`%`). In other words, the latter pointcut expression matches any variable in any namespace or class scope. Global variables can be described by omitting the scope pattern.

5.2.2 *Pointcut Functions for Access to Variables*

A name pointcut expression, which describes one or more variables, represents the input for the pointcut functions `get` and `set`, which extend the pointcut description language of AspectC++ in an AspectJ-like fashion. Table 5.4 defines the syntax and semantics of both pointcut functions, which capture read access and write access, respectively. The following example illustrates three join points captured by the `get` and `set` pointcut functions:

```
class Semaphore { public: int count; } sema;
sema.count = 0; // set join point for "count"
sema.count++; // get join point prior to another set join point
```

POINTCUT	DESCRIPTION
get(pointcut)	The get pointcut function captures all code join points where a global variable or data member is read that matches the pointcut expression. Such join points occur at implicit lvalue-to-rvalue conversions according to the C++ standard [116, § 4.1], within all built-in compound-assignment operators, and within the built-in increment and decrement operators.
set(pointcut)	The set pointcut function captures all code join points where a global variable or data member is modified that matches the pointcut expression. Such join points occur within all built-in assignment operators, and within the built-in increment and decrement operators. The initialization of a global variable or data member provides no join point.
ref(pointcut)	The ref pointcut function captures all code join points where a reference (reference type or pointer) to a global variable or data member is created that matches the pointcut expression. Such join points occur within the built-in address-of operator &, within implicit array-to-pointer conversions according to the C++ standard [116, § 4.2], and before the initialization of a variable of reference type, including return values. Moreover, binding a reference argument of a function, including default values, exposes a join point.

Table 5.4: Specification of the pointcut functions get, set, and ref, which extend the pointcut description language of AspectC++ by means for capturing access to program variables.

Note that variables of class type are not supported directly by the get and set pointcut functions. Hence, there are no join points for the variable sema itself, however, its individual data members provide the actual join points. For example, the pointcut expression get("% Semaphore::%") indirectly captures all join points where a variable of the class type Semaphore is read.

The get and set functions transform a name pointcut expression into a code pointcut expression. Thus, advice is applicable before, after, and around the captured event. The uniform join-point API of code advice (see Section 4.2.1.5) is also available for advice based on the get and set pointcut functions. As such, JoinPoint::Target refers to the type of the encapsulating class of a member variable, and tjp->target() yields a pointer to the respective class instance. The

compile-time information	description
`Entity`	type of the accessed variable (or function)
`MemberPtr`	type of a pointer to member for the accessed variable (or function)
`DIMS`	number of array dimensions if available, otherwise zero
`Dim<i>::Idx`	type of the index of the *i*th array dimension
`Dim<i>::Size`	size of the the *i*th array dimension

runtime functions	description
`Entity *entity()`	pointer to the accessed variable (or function)
`MemberPtr memberptr()`	pointer to member for the accessed member variable (or function)
`Dim<i>::Idx idx<i>()`	value of the *i*th index of an array access

Table 5.5: Extension to the join-point API of AspectC++, implemented by AspectC++ 2.0. In the body of code advice, the keyword `JoinPoint` provides access to compile-time context information (upper part of the table). At the same place, the keyword `tjp` can be used to query runtime values via the listed functions (lower part of the table). Both `JoinPoint` and `tjp` utilize static type information based on the point of advice instantiation.

join-point API of AspectC++ 2.0 provides additional functionality for retrieving a typed pointer[5] to the accessed variable, as summarized in Table 5.5. In addition, the extended join-point API provides detailed information on array variables, such as the number of array dimensions, their size, and runtime indices.

Limitations of `get` and `set`

The `get` and `set` pointcut functions do not capture any join points within constant expressions that evaluate at compile time, and, as in AspectJ [140, p. 47], local variables expose no join points at all. Moreover, the `get` and `set` pointcut functions capture only direct variable access by name; indirect access to a variable using a pointer or reference is *not* captured. The reason is that, in general, it is undecidable at compile time to which variable a pointer refers to [141]. Thus, the AspectC++ compiler cannot implement the necessary code transformation without resorting to a runtime system, which would negatively affect the performance of AspectC++ programs.

[5] The C++ standard prohibits pointers to bit fields [116, § 5.3.1] and reference variables [116, § 8.3.2]. Advice for access to variables that cannot be represented in the join-point API is omitted.

Pointcut function `ref`

For dealing with variable access using pointers and references, AspectC++ 2.0 complements the `get` and `set` pointcut functions by another pointcut function: The `ref` pointcut function captures the *aliasing* of a variable, that is, the event that creates the address of a variable or initializes a reference to the variable. Table 5.4 on page 82 defines the syntax and semantics of that pointcut function. The following example illustrates two join points that are captured by the `ref` pointcut function:

```
class Semaphore { public: int count; } sema;
int *pointer = &sema.count; // ref join point for "count"
int &reference = sema.count; // ref join point for "count"
```

Prohibit aliasing of variables

The `ref` pointcut function can be used to prohibit aliasing of certain variables. Advice based on the `ref` pointcut function can specify an unsatisfiable compile-time assertion by using a `static_assert` declaration [116, § 7]. The advice only gets woven if a respective join point exists, that is, if the considered variable is effectively aliased. Thus, aliasing of the variable causes a compilation failure; however, successful compilation guarantees that the `get` and `set` pointcut functions capture every access to the variable. The example in the following section illustrates the `get`, `set`, and `ref` pointcut functions including such a compile-time assertion that prohibits aliasing.

5.2.3 *Example: Bounds Check of Arrays*

The pointcut functions for access to variables are versatile extensions to AspectC++. For example, Figure 5.2 shows the complete[6] implementation of a generic *bounds check* of array variables. The shown aspect declaration starts in line 22 with advice that prohibits the aliasing of any array variable to make sure that every access to an array can be checked. The used pointcut expression `"% ...::%"` matches all variables; however, the `static_assert` declaration in line 23 uses the join-point API to ascertain that an aliased variable must be dimensionless. Thus, only aliasing of arrays violates the static assertion at compile time.

Likewise, the piece of advice in line 26 affects every access to a variable by using the union operation (||) of `get` and `set` pointcut expressions. The advice body (lines 27 to 30) invokes the template function `out_of_bounds` and passes a pointer to the join-point API (`tjp`) as function argument. Furthermore, the value of `JoinPoint::DIMS` chooses between two class templates that implement the function `out_of_bounds` differently: A value of zero chooses the default implementation that returns `false` (line 18). For instance, if the affected variable was no array, the compiler chooses that default implemen-

[6] Because of a bug in the AspectC++ 2.0 compiler, the workaround statement `tjp->idx<0>();` must be added to the body of the second piece of advice in Figure 5.2, for example, after line 28.

```
 1 #include <stdexcept> // for exception: std::out_of_range
 2
 3 template<unsigned int DIMS> // primary template
 4 struct ArrayBounds {
 5   enum { DIM = DIMS-1 }; // current dimension
 6   template<typename JP>
 7   static bool out_of_bounds(JP *tjp) {
 8     if (tjp->template idx<DIM>() >= JP::template Dim<DIM>::Size)
 9       return true; // runtime index exceeds compile-time bound
10     else if (tjp->template idx<DIM>() < 0)
11       return true; // negative index
12     else return ArrayBounds<DIM>::out_of_bounds(tjp); // recurse
13   }
14 };
15 template<> // template specialization ...
16 struct ArrayBounds<0> { // ... for no more dimensions (zero)
17   template<typename JP>
18   static bool out_of_bounds(JP *tjp) { return false; }
19 };
20
21 aspect ArrayBoundsCheck {
22   advice ref("% ...::%") : before() { // prohibit aliasing
23     static_assert(JoinPoint::DIMS == 0, "aliasing of array");
24   }
25
26   advice get("% ...::%") || set("% ...::%") : before() {
27     if (ArrayBounds<JoinPoint::DIMS>::out_of_bounds(tjp))
28       throw std::out_of_range(JoinPoint::signature());
29   }
30 };
```

Figure 5.2: Implementation of a bounds check of array variables using AspectC++ 2.0. The `get` and `set` pointcut functions in line 26 capture access to any variable and instantiate the *template metaprogram* `ArrayBounds<>`, which generates a multi-dimensional bounds check. The template specialization in line 15ff makes the advice applicable to all variables, including variables that are no arrays. To prevent access using a pointer or reference, the advice based on the `ref` pointcut function in line 22f prohibits aliasing of array variables.

tation because `JoinPoint::DIMS` equals zero. Otherwise, the primary implementation, presented in lines 7 to 13, is chosen.

The primary implementation of the function `out_of_bounds` performs the bounds checking for one array dimension `DIM` at a time. It retrieves the runtime index of the array access using `idx<DIM>()` from the extended join-point API and compares that index with the compile-time bound retrieved by `Dims<DIM>::Size`, as shown in line 8. Subsequently, line 10 checks for a negative array index. If the bounds check is satisfied, the function `out_of_bounds` recursively invokes itself to check the next array dimension (line 12). The class' template parameter `DIMS` (line 3) is decremented in each recursive step (see line 5), and eventually approaches zero, which terminates the recursion by choosing the default implementation in line 18.

Template metaprogramming

In other words, the class `ArrayBounds<>` represents a *template metaprogram* [62, p. 397ff] that generates one bounds check for each array dimension. The point of instantiation (line 27) invokes a *recursive instantiation* (line 12) of the class template `ArrayBounds<>` until the template specialization `ArrayBounds<0>` gets instantiated. C++ template instantiation happens at compile time; the number of array dimensions that limits the recursion depth is also a compile-time constant value (`JoinPoint::DIMS`). Thus, the template metaprogram *runs* completely at compile time. However, the result is the *generated* function `out_of_bounds` that implements a multi-dimensional bounds check at runtime. Lohmann and colleagues [155, p. 62] use the term *generative advice* to describe pieces of advice that instantiate template metaprograms based on information provided by the join-point API, such as the advice defined in lines 26 to 29.

Generative advice

Optimizing compilers, such as GCC and LLVM/Clang, perform function inlining and constant propagation [19] for optimization of C++ templates and, thus, eliminate the bounds check if the used array index was a compile-time constant value. Moreover, the bounds check compiles to only two CPU instructions per array dimension on the x86 architecture: a `cmp` instruction that *compares* the runtime index with an immediate value and a subsequent `ja` instruction (*jump if above*) that conditionally jumps to the exception code. Thus, there is no indirection at runtime caused by AspectC++ 2.0.

In summary, the example in Figure 5.2 illustrates the power of advice for variable access. The example implements bounds checking for array variables with only 30 lines of code. A limitation of the shown implementation is that dynamically allocated arrays, which are unnamed and whose bounds are not known at compile time, are not checked. In addition, arrays of class-type objects are not checked either; however, the following section presents a complementary mechanism to check objects of class type as well. Finally, the next chapter builds on the exemplified `get`, `set`, and `ref` pointcut functions to develop generic, modular, and efficient error-correction mechanisms.

5.3 generic introductions

The third language extension resolves the coupling problem of introduction advice identified in the previous chapter. Introductions in AspectC++ 1.0 – and AspectJ – can be used to introduce additional members into existing classes; however, the introduced members are not supplied with any context information depending on the join point. For example, Figure 4.3 on page 65 shows an aspect that introduces a redundant pointer of the type `Item*` into the classes `Item` and `List`. Therefore, the aspect is tightly coupled to both classes, because adding such a specific pointer to other classes is nonsense. Introductions in AspectC++ 1.0 are not as generic as code advice.

Hanenberg and Unland [99] propose *parametric introductions* as improvement over the AspectJ pointcut description language. Their parametric aspect language incorporates the *Prolog* language and enables a binding of logic meta variables to join-point context information, such as the class type being extended by an introduction. Kniesel and Rho [133] generalize such an approach to code advice and use the term *generic introductions* in the style of *generic advice* [155]. Unfortunately, both approaches [99, 133] add another dimension of complexity by expanding the pointcut description language to logic metaprogramming.

However, there is no need to incorporate another metaprogramming language for generic introductions in AspectC++, because the base language C++ already supports metaprogramming by the template mechanism (see Section 5.2.3). For that reason, the join-point API of AspectC++ 1.0 builds on C++ templates for implementing generic code advice. Consequently, the following section presents a generalization of the existing join-point API to cover introductions as well.

5.3.1 *Join-Point API of Introductions*

In AspectC++ 2.0, the built-in keyword `JoinPoint` provides uniform access to the join-point API in the body of both introductions and code advice. In the body of an introduction, the keyword `JoinPoint` provides information on the class type that receives the introduction. This information is purely static in nature, because a class type is solely a compile-time concept. Thus, the *compile-time* join-point API suffices for introductions.

Table 5.6 shows an excerpt from the join-point API of introductions, which provides information on the join point *prior* to the introduction. For example, `JoinPoint::That` refers to the type of the incomplete class that receives the introduction. Besides, the API enumerates the number of base classes (`JoinPoint::BASECLASSES`), data members (`JoinPoint::MEMBERS`), functions, constructors, and destruc-

COMPILE-TIME INFORMATION	DESCRIPTION
`That`	type of the (incomplete) class that receives the introduction
`HASHCODE`	integer hash value of the class type that receives the introduction
`BASECLASSES`	number of base classes
`BaseClass<`*i*`>::Type`	type of the *i*th base class
`BaseClass<`*i*`>::spec`	specifiers of the *i*th base class
`MEMBERS`	number of data members
`Member<`*i*`>::Type`	type of the *i*th data member
`Member<`*i*`>::spec`	specifiers of the *i*th data member
`FUNCTIONS`	number of member functions
`Function<`*i*`>::spec`	specifiers of the *i*th member function
`CONSTRUCTORS`	number of user-defined constructors
`DESTRUCTORS`	number of user-defined destructors
`Destructor<0>::spec`	specifiers of the single user-defined destructor

RUNTIME FUNCTIONS	DESCRIPTION
`Member<`*i*`>::pointer(That*)`	typed pointer to the *i*th data member

Table 5.6: Excerpt from the join-point API of introductions, implemented by AspectC++ 2.0. In the body of introduction advice, the keyword `JoinPoint` provides access to compile-time context information (upper part of the table) and static functions for querying runtime values (lower part of the table). The specifier `::spec` is either one of `AC::SPEC_NONE`, `AC::SPEC_STATIC`, `AC::SPEC_MUTABLE`, or `AC::SPEC_VIRTUAL`. A complete list of all information provided by the join-point API of introductions, although not relevant for this thesis, is available online at: `http://aspectc.org/`

tors of the receiving class. For each particular base class, there is a nested type `JoinPoint::BaseClass<`*i*`>` that further describes the *i*th base class, where *`i`* is a compile-time constant index. For example, `JoinPoint::BaseClass<0>::Type` refers to the type of the first base class, if available. Such nested types also describe each data member (`JoinPoint::Member<`*i*`>`), member function, constructor, and destructor. For data members, the nested types also provide the static function `Member<`*i*`>::pointer(That *obj=0)` that converts a pointer to an object into a typed pointer to its *i*th data member. If the *i*th data member is declared as `static`, the object pointer is not required.

In other words, the join-point API of introductions enables *compile-time introspection*: An introduction can introspect for existing members and inheritance hierarchies. Based on the provided information, an introduction can instantiate generic class templates that adapt themselves to the particular join point. The instantiation of template metaprograms can even generate join-point–specific code and assemble tailored data types by *generative programming* [62] techniques. Thus, introductions in AspectC++ 2.0 are both generic and generative[7].

Introspection

As a remark, the combination of compile-time introspection and template metaprogramming essentially implements *computational reflection* at compile time [14]. This is like Java's runtime reflection [87]; however, compile-time reflection causes no overhead at runtime.

Reflection

5.3.2 *Example: Run-Time Type Checking*

Generic introductions are a powerful language extension to AspectC++. For example, Figure 5.3 shows the complete implementation of a highly generic mechanism that applies *run-time type checking* to all instances of class type. The mechanism checks at run time whether a class instance corresponds to its associated data type.

First, the aspect defines in line 13 the named pointcut `where()`, which matches any data type except the class template `TypeCode<>`. Second, lines 15 to 17 show the generic introduction that extends all classes except `TypeCode<>` by the single data member `type_code`. That member instantiates the class template `TypeCode<>` and binds its integer template parameter to `JoinPoint::HASHCODE`, which is a compile-time hash value of the class type that receives the introduction, provided by the join-point API (see Table 5.6). Thus, the introduction is generic, because the instantiation of the class template depends on context information provided by the join-point API.

The functionality of the introduced data member is shown in lines 3 to 10. Basically, the introduced member stores the compile-time hash value in a member variable at run time. The function `invalid()` in line 9 indicates whether the stored value is not identical to the expected compile-time value. If a mismatch occurs, the class instance is invalid.

Finally, the pieces of advice shown in lines 19 to 26 make sure that the type checking is carried out at run time. The function `invalid()` is invoked whenever a non-static data member is read (line 19) or modified (line 20), whenever a class instance is deleted (line 21), and whenever a virtual function is called (line 22). Calls to non-virtual functions are not checked because they do not depend on the dynamic data type. The expression in line 24 retrieves a pointer to the involved class instance by `tjp->target()` and uses the introduced member `type_code` for type checking.

In summary, the centerpiece of this example is the generic introduction in line 16. The shown aspect extends thereby all class instances

[7] An introduction is *generative* if and only if it invokes a template metaprogram that depends on the join-point API.

```
 1 #include <typeinfo> // for exception: std::bad_typeid
 2
 3 template<int HASHCODE>
 4 class TypeCode {
 5   int code; // storage for object’s HASHCODE
 6 public:
 7   inline TypeCode() : code(HASHCODE) {} // initialize
 8   inline ~TypeCode() { code = 0; } // nullify on deletion
 9   inline bool invalid() const { return code != HASHCODE; }
10 };
11
12 aspect TypeCheck {
13   pointcut where() = "...::%" && !"TypeCode<%>"; // reusable, named pointcut
14
15   advice where() : slice class { // generic introduction
16     TypeCode<JoinPoint::HASHCODE> type_code;
17   };
18
19   advice (get(where()) && !get("static % ...::%") && target(!"void")) ||
20          (set(where()) && !set("static % ...::%") && target(!"void")) ||
21          (destruction(where())) ||
22          (call(where()) && call("virtual % ...::%(...)"))
23   : before() {
24     if (tjp->target()->type_code.invalid()) // generic advice
25       throw std::bad_typeid(); // exception for error handling
26   }
27 };
```

Figure 5.3: Implementation of run-time type checking using AspectC++ 2.0. The centerpiece of this example is the *generic introduction* in line 16. As described by the pointcut expression in line 13, each class type except `TypeCode<>` receives a join-point–specific variable that stores a hash value in each class instance. The join-point API of introductions provides such a hash value by `JoinPoint::HASHCODE`. The remaining pieces of code advice in lines 19 to 26 make sure that the hash value is checked on access to a class instance. In contrast to the C++ dynamic run-time type identification (RTTI), the shown aspect is *not* limited to polymorphic class types [116, § 5.2.8].

by a join-point–specific variable that enables the detection of corrupt pointers to the class instance. The reason for pointer corruption can be not only a hardware error, but also programming errors such as dangling pointers (use-after-free), dereference of uninitialized or null pointers, double frees, and incompatible type casts. In fact, by applying the aspect to the AspectC++ compiler itself, I found and resolved three[8] incompatible type casts.

Detection of programming errors

5.3.3 *Join-Point Template Library*

The previous example illustrates a generic introduction that only uses a single entry of the join-point API. A *generative* introduction, however, instantiates a template metaprogram that iterates over multiple entries of the join-point API at compile time.

Manually writing template metaprograms using the join-point API is a tedious and repetitive task. Thus, I implemented a library of reusable compile-time iterators for each API entry: the *Join-Point Template Library (JPTL)*. For example, the following iterator introspects for member functions and recursively invokes a user-defined action for each member function:

```
JPTL::FunctionIterator<typename TypeInfo, template <typename,
    typename> class Action>
```

The first template parameter must be bound to the join-point API by using the `JoinPoint` keyword, whereas the second template parameter `Action<>` accepts a user-defined class template. That class template is then instantiated for each member function and is supplied with the respective type information `JoinPoint::Function<i>` as first template parameter. The second template parameter of Action<> refers to the previous iteration and can be used to compute compile-time constant values.

For example, Figure 5.4 shows a class template that counts the number of virtual functions. Thus, in the body of an introduction, the number of virtual functions of the receiving class can be determined as follows:

```
JPTL::FunctionIterator<JoinPoint, VirtualFunctionCount>::EXEC::
    COUNT
```

Besides the nested `EXEC` data type, which computes compile-time constant values, an `exec()` function with arbitrary arguments can be defined that is invoked iteratively for each member function. Tailored program code can be generated thereby for each particular member function.

Section 6.4.2 shows the usage of the function `exec()`.

In addition, the JPTL provides uniform compile-time iterators for the join point's base classes, data members, constructors, and destruc-

[8] I resolved three incompatible type casts in the AspectC++ subversion revisions 192, 201, and 202, respectively.

```
template<typename FunctionInfo, typename LAST>
struct VirtualFunctionCount {
  struct EXEC {
    enum { COUNT = LAST::COUNT + // add 1 if virtual
          ((FunctionInfo::spec & AC::SPEC_VIRTUAL) ? 1 : 0) };
  };
};
template<typename FunctionInfo> // specialization for ...
struct VirtualFunctionCount<FunctionInfo, void> {
  struct EXEC { enum { COUNT = 0 }; }; // ... initial value
};
```

Figure 5.4: Class template that counts the number of virtual functions. This template can be used to instantiate an iterator of the JPTL for computing the number at compile time.

tors. Furthermore, there are four compound iterators, such as the `BaseFunctionIterator<>` that recursively iterates over the join point's functions *and* all functions of respective base classes.

In summary, generic and generative introductions in AspectC++ 2.0 are extremely powerful language features that enable a loose coupling of the introductions to the mainline program code. The JPTL dramatically relaxes the obstacles of template metaprogramming as imposed by the compile-time join-point API.

5.4 WHOLE-PROGRAM ANALYSIS

The fourth extension to AspectC++ concerns the technical compilation process. Currently, the AspectC++ compiler processes one translation unit at a time and evaluates a pointcut expression without information on other translation units. Thus, without abandoning the modular C++ compilation process, information on the *whole program* is just not available during compilation.

This lack of information disables more complex pointcut functions that incorporate precise static program analyses. For example, a static control-flow reachability analysis could identify those functions that may lead directly or indirectly to a context switch. Such functions do not return until the respective thread of control is resumed, so that the lifetime of their local variables and return addresses is prolonged. As data lifetime is directly proportional to soft-error susceptibility (see Section 3.3), the return addresses of such long-lasting functions are exceedingly critical.

We shall see in the following chapters that static analyses of the whole program can improve the efficiency of aspect-oriented fault-tolerance mechanisms. Therefore, I propose a framework for user-defined static analyses using the AspectC++ compiler without extending the pointcut description language. The framework creates

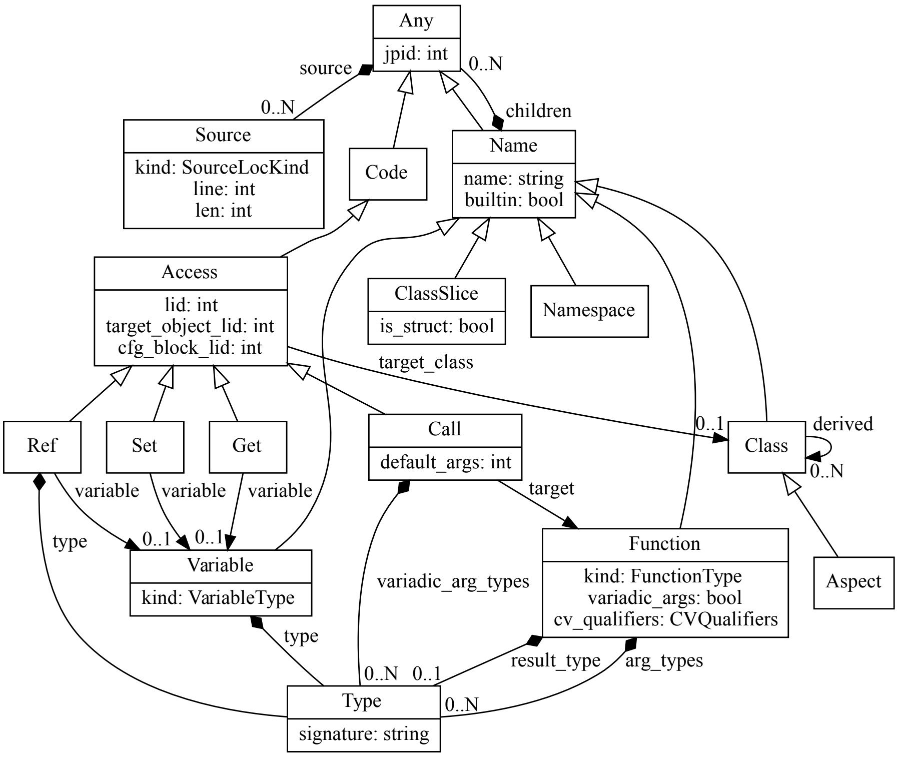


Figure 5.5: Excerpt from the AspectC++ *metamodel of a translation unit*. The class diagram illustrates the relationship between some elements of the model, which represent entities of the source code of one translation unit. The union of all translation-unit models of a program yields the *project repository*, which contains information on the whole program.

an XML-based representation of the whole program, so that general-purpose static analyses can be implemented rapidly in the XML query language *XQuery* [166].

5.4.1 *Project Repository*

The AspectC++ 2.0 compiler internally uses the Clang [142] parser to build an abstract syntax tree from the source code of one translation unit at a time. Afterwards, the AspectC++ compiler analyzes the syntax tree and builds a *model of the translation unit*, which is a tree-based data structure that aggregates semantic information. Figure 5.5 shows an excerpt from the class diagram of the *metamodel* of a translation unit. A concrete model of a translation unit contains numerous instances of the metamodel's classes. For example, an instance of the class `Call` represents one specific function call that is present in the source code being compiled. The `target` association of such a `Call` instance refers to the callee function – an instance of the class `Function`.

Model of a translation unit

Yet, the model of one translation unit only contains partial information, but the AspectC++ compiler can serialize the model into a shared XML file: the global *project repository*. After all translation units have been processed once, the models of the individual translation units merge into the global project repository, which eventually contains information on the whole program.

XQuery

Whole-program analysis using the global project repository can be implemented, for example, in the XML query language *XQuery*, which is a functional programming language that facilitates data extraction from XML documents. The result of an XQuery program can be an AspectC++ pointcut expression, which in turn can be included in the final aspect weaving process.

Figure 5.6 illustrates the compilation process with whole-program analysis using the AspectC++ project repository and XQuery. First, the AspectC++ compiler processes all individual source code files and populates the project repository with information obtained by a static pre-analysis (❶). After that, a user-defined XQuery program extracts the whole-program information from the project repository and stores the results in form of pointcut definitions in an aspect header file (❷). An aspect that makes use of the whole-program analysis includes these pointcut definitions. Finally, the AspectC++ compiler weaves such an aspect into the respective files (❸).

In other words, the depicted compilation process virtually enables the use of XQuery programs as pointcut functions. Rohlik and colleagues [205] as well as Eichberg and associates [75] propose similar approaches, however, the static pre-analysis (❶) improves over their works.

Control-flow analysis

In particular, I implemented control-flow and pointer-alias analyses in the AspectC++ 2.0 compiler, which stores the results from that analyses in the project repository. The control-flow analysis builds a traditional *control-flow graph* in which the nodes represent *basic blocks* [10]; a basic block is a linear sequence of program statements that are always executed in order. Thus, the control-flow graph defines a partition of program statements into basic blocks. The AspectC++ project repository reflects the partition into basic blocks by the member `cfg_block_lid` of the class `Access` (see Figure 5.5). That member represents an intra-procedural identifier for each basic block. For example, the class `Call` inherits from the class `Access`, so that each function call is mapped to a basic block.

Flow-sensitive pointer-alias analysis

Based on the control-flow analysis, I implemented a *flow-sensitive pointer-alias analysis* [108], which attempts to determine whether two pointers – or C++ references – refer to the same variable. The alias analysis consequently assigns an intra-procedural identifier to each variable. That identifier is stored in the AspectC++ project repository as the member `target_object_lid` of the class `Access` (see Figure 5.5). The class instance of a member-function call is identified thereby: When two function calls have an identical `target_object_lid`, then the same class instance is used with certainty. Different identifiers

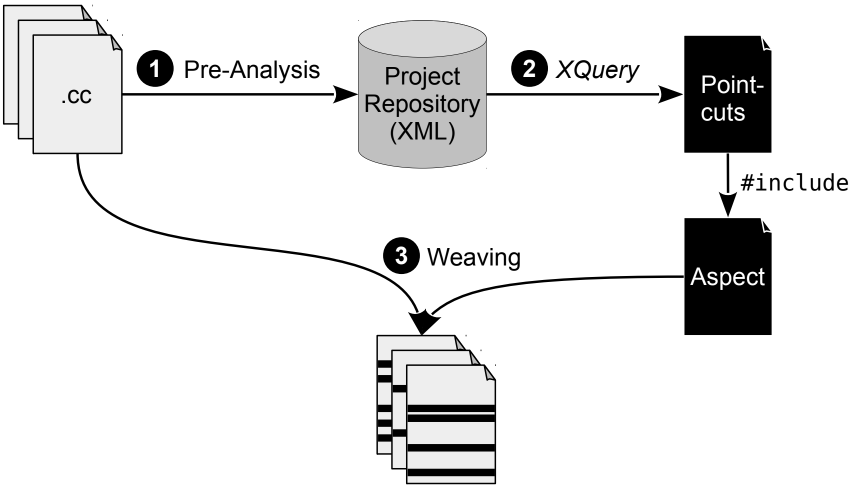


Figure 5.6: Process of the whole-program analysis using the AspectC++ project repository and XQuery. After filling the project repository with information obtained by a static pre-analysis (❶), a user-defined XQuery program extracts information on the whole program and stores the results in form of pointcut definitions in an aspect header file (❷). The final compilation (*weaving*) includes that pointcut definitions (❸) and, thus, makes use of the whole-program analysis.

indicate either different instances or that the pointer-alias analysis cannot prove their identity.

In summary, the control-flow analysis and pointer-alias analysis implement parts of the pre-analysis step (❶) shown in Figure 5.6. On that basis, a programmer can implement user-defined static analysis using the project repository, as exemplified in the following section.

5.4.2 *Example: Control-Flow Reachability Analysis*

This section picks up the demand for identification of those functions that may lead directly or indirectly to a context switch. For instance, the eCos kernel implements the context switch in the assembler routine `hal_thread_switch_context`. Thus, a static analysis must traverse the call graph of the whole operating system to identify those functions that lead transitively to an invocation of the routine `hal_thread_switch_context`.

The complete call graph is available once the AspectC++ project model has been set up. Hence, an XQuery program can implement a control-flow reachability analysis to determine the desired set of functions. Figure 5.7 shows an excerpt of a recursive XQuery implementation that computes the transitive closure of the *call* relation between functions. Each function in the AspectC++ project model is identified by a unique integer `id`. Thus, the shown implementation expects a sequence of integers (`xs:integer*`) as argument and returns such a sequence. The implementation iterates over all function calls listed

```
 1 declare function local:reachable($function_ids as xs:integer*) as
       xs:integer* {
 2   let $caller := for $call in $repo//Function/children/Call
 3                  where $call/@target = $function_ids
 4                  return $call/../../@id
 5   let $reachable_ids := distinct-values(($function_ids, $caller))
 6   return
 7     if ( count($function_ids) = count($reachable_ids) ) then
 8       $function_ids
 9     else
10       local:reachable($reachable_ids)
11 };
```

Figure 5.7: Excerpt from the implementation of a control-flow reachability analysis in the XQuery language. The shown code processes the XML-based project repository `$repo` in which each function can be identified by a unique integer `id`. By recursion (line 10), the implementation identifies all functions that transitively call one of the functions provided by the argument `$function_ids`.

in the project repository `$repo` (line 2) and selects the caller function `id` (line 4) if that function invokes one of the functions provided in the argument sequence (line 3). The implementation recurses (line 10) until a fixed point is reached (line 7).

Figure 5.7 omits several convenience functions that are needed to transform a function name, such as `hal_thread_switch_context`, into the corresponding integer `id` and vice versa. Similarly, the functionality that formats the result as AspectC++ pointcut expressions is not shown. Altogether, the complete reachability analysis consists of about 100 lines of XQuery code that additionally handles function calls whose call target is unknown, such as invocations of library functions and function calls using a function pointer. In summary, the AspectC++ project repository provides a rich framework for user-defined whole-program analyses that can be implemented rapidly in the XQuery language.

5.5 CHAPTER SUMMARY

The goal of this chapter was to introduce the new general-purpose language features of AspectC++ 2.0 that have been developed in the course of this thesis. These language features remedy the deficiencies of prior work on fault tolerance using AspectC++ by capturing join points of pointer variables and data members. Moreover, the compile-time join-point API of introductions decouples the aspects from the mainline program semantics.

The new language features represent the essential building blocks for the development of highly generic fault-tolerance mechanisms as illustrated by the functional examples: Checking of pointer ranges, checking of array bounds, and run-time type checking are implemented by only a few lines of code. Thus, AspectC++ 2.0 is a promising language for implementing *error-detection* mechanisms; however, this section provides limited insight into the domain of error correction and recovery. Therefore, the next chapter addresses the development of error-correction mechanisms using AspectC++ 2.0.

Finally, whether the presented example mechanisms really improve the fault tolerance of an operating system remains an open question. Are the example mechanisms effective and efficient? To answer these questions, Chapter 7 quantitatively evaluates these examples.

6 LIBRARY OF DEPENDABILITY ASPECTS

> "Dependability is ... a global concept that subsumes the usual attributes of reliability, availability, safety, integrity, maintainability, etc."
>
> – Algirdas Avižienis and associates [18, p. 11]

This chapter presents a library of generic software mechanisms that aim at facilitating *dependable* operating systems. As such, the presented mechanisms focus on error detection *and* correction to enhance the reliability, availability, and data integrity of operating systems. At the same time, the library approach guarantees maintainability, because the source code of an operating system remains as it is: The AspectC++ compiler applies the error-detection and error-correction mechanisms transparently.

Primarily, this chapter introduces four aspect-oriented mechanisms that use the AspectC++ 2.0 technology as presented in the previous chapter – referred to as just AspectC++ in this chapter. The individual mechanisms are highly generic and reusable, so that they can be applied to both eCos and the L4/Fiasco.OC operating system.

First, Section 6.1 describes the design of the library and its programming interface. After that, Section 6.2 continues with the symptom-detection mechanisms that served as examples in the previous chapter. Subsequently, Section 6.3 and Section 6.4 cover the protection from control-flow errors by detection and correction of memory errors that affect return addresses and virtual-function pointers, respectively. Finally, Section 6.5 presents a generative mechanism that detects and corrects memory errors in data members of C++ classes.

In summary, this chapter covers four *dependability aspects*, that is, aspect-oriented mechanisms that enhance the degree of dependability. The individual mechanisms advance the state-of-the-art in the domain of software-implement fault tolerance (see Section 2.4). In particular, the presented mechanisms address the exceedingly critical kernel data structures identified in Section 3.3. Finally, the next chapter quantitatively evaluates the library of dependability aspects.

RELATED PUBLICATIONS

The findings presented in this chapter have partly been published in:

[29] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Protecting the dynamic dispatch in C++ by dependability aspects. In *Proceedings of the 1st GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*. German Society of Informatics, September 2012. URL: `http://subs.emis.de/LNI/Proceedings/Proceedings208/521.pdf`

[31] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Return-address protection in C/C++ code by dependability aspects. In *Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*. German Society of Informatics, September 2013. URL: `http://subs.emis.de/LNI/Proceedings/Proceedings220/2519.pdf`

Open access, journal impact factor: 1.59

[32] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generic soft-error detection and correction for concurrent data structures. *IEEE Transactions on Dependable and Secure Computing*, 14(1):22–36, January 2017. doi: 10.1109/TDSC.2015.2427832

6.1 DESIGN OF THE LIBRARY

> “There is no good library without a strict emphasis on regular, coherent design.”
>
> – Bertrand Meyer [168, p. 69]

The main design goal of the library of dependability aspects is usability of the individual aspects without the need to deal with implementation details. On the one hand, the library's source code should be closed to modification. On the other hand, there should be a well-defined programming interface that allows a programmer to use the library and to extend it as necessary. The term *open-closed principle* [168, p. 57] refers to such an approach, in which openness means being “available for extension” [168, p. 57]. In particular, the library's interface should allow for extension at compile time to avoid any runtime overhead.

Pure virtual pointcuts

These design goals can be accomplished by *abstract aspects* in AspectC++, which are abstract data types like abstract classes. User-defined aspects can inherit from abstract aspects just as classes inherit from abstract base classes. An abstract aspect contains one or more *pure virtual* pointcut declarations, which must be defined by a derived aspect. Only if all pure virtual pointcut declarations of an abstract aspect are defined, the aspect can be instantiated and eventually gets woven by the AspectC++ compiler.

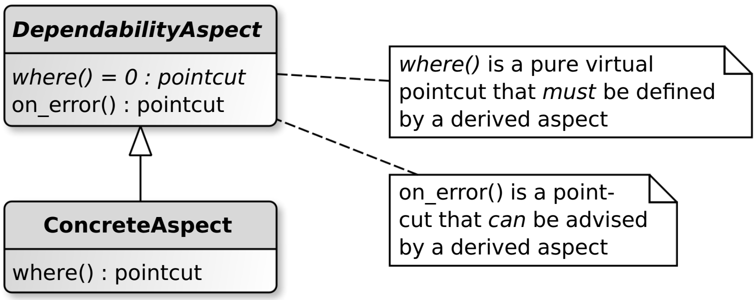

DependabilityAspect
where() = 0 : pointcut
on_error() : pointcut
ConcreteAspect
where() : pointcut
where() is a pure virtual pointcut that must be defined by a derived aspect
on_error() is a pointcut that can be advised by a derived aspect

Figure 6.1: Interface of the dependability aspects. The depicted aspects are modeled as classes by a UML class diagram in which member functions may yield `pointcut` expressions. The aspect *`DependabilityAspect`* is abstract because it declares the pure virtual pointcut *`where()`*, denoted by italics. `ConcreteAspect` inherits from the abstract aspect and defines the pure virtual pointcut to specify the placement of the dependability aspect. Abstract aspects are not considered for weaving by the AspectC++ compiler unless all pure virtual pointcuts are defined by a derived aspect.

The programming interface of the individual dependability aspects is based on pure virtual pointcut declarations. Each dependability aspect declares the pure virtual pointcut *`where()`*, which specifies the placement policy of the aspect. To instantiate such a dependability aspect, the programmer must define that pointcut by a derived aspect. For instance, to apply a certain aspect to all program elements within the global namespace, the user can define *`where()`* as follows: `pointcut where() = within("::");`

Figure 6.1 shows a UML class diagram in which aspects are modeled as classes with member functions that yield `pointcut` expressions. That figure illustrates the inheritance relation between the abstract aspect *`DependabilityAspect`*, which implements some fault-tolerance mechanism, and a derived aspect that defines the pure virtual pointcut *`where()`*. Abstract aspects and pure virtual members are denoted by italics.

In addition, the base aspect defines the pointcut `on_error()`, which captures all occurrences of errors that are detected but not corrected. The derived aspect can specify thereby advice for error handling, such as invoking a user-defined recovery routine, or throwing a C++ exception. For instance, the latter can be implemented as follows:

```
#include <exception>
aspect ConcreteAspect : public DependabilityAspect {
  pointcut where() = within("::"); // apply everywhere
  advice on_error() : before() {
    throw std::exception(); // user-defined error handling
  }
};
```

The AspectC++ compiler evaluates all pointcut expressions at compile time; even pure virtual pointcuts are bound to the concrete definitions at compile time. Thus, the pointcut-based configuration interface of the dependability aspects causes no runtime overhead at all. Each dependability aspect is highly configurable with respect to placement and error handling. Finally, as each dependability aspect declares at least the two mandatory pointcuts `where()` and `on_error()`, the library provides a regular and coherent programming interface.

6.2 SYMPTOM DETECTION

> “If the software operates on corrupt values, not only could the data result be incorrect, but the error could result in ‘side-effects’. For example, exceptions (memory access faults or arithmetic overflow) . . . can be caused by corrupt data values feeding into a pointer, arithmetic value, or branch instruction. These events are examples of invalid program behavior — events that should not occur in normal program execution.”
>
> – Nicholas Wang and Sanjay Patel [252, p. 32]

Symptom detection refers to a broad class of error-detection mechanisms that identify invalid software behavior at runtime (see Section 2.4.1.4 on page 28). Such mechanisms can be implemented modularly in the AspectC++ programming language as shown by the three examples in the previous chapter. Thus, the library of dependability aspects, described in this chapter, includes these example mechanisms under the umbrella of symptom detection. In addition, Section 6.2.2 on the facing page introduces another aspect-oriented mechanism that detects integer overflows in arithmetic operations.

6.2.1 *Checking of Pointers, Arrays, and Class Types*

The previous chapter already presented three examples for symptom detectors using AspectC++. In brief, these symptom detectors provide the following functionalities:

1. Range checking of function pointers (Section 5.1.2 on page 79)
2. Checking of array bounds (Section 5.2.3 on page 84)
3. Run-time type checking (Section 5.3.2 on page 89)

Each of these symptom detectors addresses abnormal behavior that can indicate a hardware error. In the library of dependability aspects, the implementation of these three symptom detectors is extended by the pointcut-based configuration interface described in Section 6.1.

6.2.2 *Integer-Overflow Checking*

Integer overflows of arithmetic operations can be considered as symptoms for hardware errors [252, p. 32]. Especially signed integer data types are of particular concern because the C++ standard permits undefined behavior on overflow [116, § 5]. As a remedy, overflow checking can be implemented in the AspectC++ programming language using the pointcut function `builtin` (see Section 5.1 on page 78), which captures all arithmetic operations on built-in data types.

Hardware support for overflow detection

Most processors provide hardware support for detecting arithmetic overflow by a condition code flag in a special CPU register: After an arithmetic operation, the condition code flag signals whether an overflow occurred. Such CPU-specific registers are generally inaccessible from a high-level programming language such as C++; however, the compilers GCC and LLVM/Clang offer several compiler intrinsics that make use of the CPU-specific condition code registers. For instance, the intrinsic function `__builtin_add_overflow` computes the sum of two integer operands and returns `true` if the operation causes an overflow.

Figure 6.2 shows the shortened implementation of a symptom detector for integer overflows using AspectC++. First, the shown aspect implements the pointcut-based configuration interface as described in Section 6.1 by declaring the pure virtual pointcut *`where()`* in line 3 and by defining the pointcut `on_error()` in line 4. Second, the pointcut `integer()` in line 10 filters out any join point that yields a floating-point value or pointer expression, because the aforementioned intrinsic function `__builtin_add_overflow` does not handle pointer arithmetic. Third, the aspect specifies its order of precedence in lines 12 to 13: Aspects derived from `IntegerOverflowCheck` should be applied to any join point involving built-in operators *after* all other aspects have been applied. This order of precedence makes sure that no other aspect interferes by silently modifying the operands after the overflow check has passed. Finally, the piece of generic advice in lines 16 to 19 exemplifies the overflow checking for the built-in operator `+` that adds two integer values. As mentioned earlier, the intrinsic function `__builtin_add_overflow` tests whether the addition of the first two operands, retrieved via the join-point API functions `tjp->arg<0>()` and `tjp->arg<1>()`, causes an integer overflow. Moreover, this intrinsic function stores the result of the addition in the variable pointed to by `tjp->result()`, which is returned by the advice code.

The complete implementation includes similar pieces of advice for subtraction and multiplication, for the built-in increment and decrement operators, and for the compound assignment operators `+=`, `-=`, and `*=`. In total, the implementation amounts to less than 130 lines of AspectC++ code. To put this into perspective, a comparable exten-

```
1  aspect IntegerOverflowCheck {
2  protected:
3    pointcut virtual where() = 0; // must be defined by derived aspects
4    pointcut on_error() = execution("void IntegerOverflowCheck::error()");
5
6  private:
7    inline void error() {} // explicit join point, exposed via on_error()
8    pointcut floats() = result("float" || "float&" || "double" || "double&");
9    pointcut pointer() = result("%*" || "%*&");
10   pointcut integer() = where() && !floats() && !pointer();
11
12   advice builtin("% operator %(...)") && integer() // lowest precedence:
13   : order(!derived("IntegerOverflowCheck"), derived("IntegerOverflowCheck"));
14
15   // addition of two integer operands:
16   advice builtin("% operator+(%, %)") && integer() : around() {
17     if (__builtin_add_overflow(*tjp->arg<0>(), *tjp->arg<1>(), tjp->result()))
18       error();
19   }
20   // similar pieces of advice for other built-in operators follow ...
21 };
```

Figure 6.2: Shortened implementation of integer-overflow checking. The shown aspect is *abstract* because it declares the pure virtual pointcut *where()*, which allows for the configuration at compile time. User-defined handling of integer overflows can be specified by advice for the pointcut on_error(). The advice definition in lines 16 to 19 replaces the invocation of the built-in operator + by a call to the compiler-intrinsic function __builtin_add_overflow, which signals whether an addition causes overflow. Similar pieces of advice for subtraction, multiplication, increment, decrement, and compound-assignment operators are not shown.

sion of the Clang compiler requires more than 1,600 lines of code [67, p. 763].

In addition, the concise aspect-oriented implementation is even more flexible and configurable than a compiler extension. The *where()* pointcut allows the programmer to confine the placement of the overflow check to certain functions and, for example, to exclude certain data types, such as unsigned integers. As pointed out by Dietz and associates [67], several programs intentionally exploit the wraparound[1] semantics of unsigned integers. The authors conclude that “tools for *detecting* integer numerical errors need to distinguish intentional from unintentional uses of wraparound operations—a challenging task—in order to minimize false alarms.” [67, p. 769]

Finally, the detection of integer overflows needs to be efficient. In other words, the detection should execute only a few additional CPU instructions. For example, consider a C++ function that just multiplies

[1] The C++ standard specifies that unsigned integers “shall obey the laws of arithmetic modulo 2^n where n is the number of bits in the value representation of that particular size of integer.” [116, § 3.9.1]

```
  int multiply(int a, int b) { return a*b; } // C++ equivalent

1 4005c0:       mov    %edi,%eax          # move first operand
2 4005c2:       imul   %esi,%eax          # multiply
3 4005c5:       jo     4005c8             # jump if overflow
4 4005c7:       retq                      # function return
5 4005c8:       mov    $0x1,%edi          # error handling:
6 4005cd:       push   %rax               # ...
7 4005ce:       callq  4004a0 <exit@plt> # invoke std::exit(1)
```

Figure 6.3: GNU x86-64 assembler instructions for overflow checking of one multiplication operation using the intrinsic function `__builtin_mul_overflow` of the GCC 5.4.0 compiler at optimization level -O2. The four instructions on colored background implement the aspect-oriented overflow checking, whereas the three instructions on white background are emitted by the compiler also without any overflow checking. Only the conditional jump instruction `jo` (*jump if overflow*) in line 3 gets executed in the error-free path, whereas the instructions in lines 5 to 7 implement the invocation of `std::exit(1)` on error.

two integer arguments as shown in the upper part of Figure 6.3. The lower part of that figure lists the compiled x86-64 CPU instructions for the respective C++ function after applying the `IntegerOverflowCheck` aspect, which simply calls `std::exit(1)` on error. Instructions on colored background implement the optional overflow checking, whereas the three instructions on white background are mandatory for the actual multiplication and, thus, are also present without any overflow checking. In total, the aspect adds only four CPU instructions per operation. Moreover, just the single conditional jump instruction `jo` (*jump if overflow*) in line 3 gets executed in the error-free case. The remaining three instructions in lines 5 to 7 implement the rarely used error handling.

Efficiency of the aspect-oriented solution

Optimizing compilers, such as GCC and LLVM/Clang, perform constant propagation [19] and eliminate thereby the overflow check at runtime if the operands were constant values at compile time. In summary, the presented overflow-checking mechanism could not be more efficient. As shown in Figure 6.3, there is no *per se* overhead of AspectC++, because the advice code gets heavily optimized at compile time. The same conclusion also holds for the other three aspect-oriented symptom detectors summarized in Section 6.2.1 on page 102.

6.3 Return-address protection

The case study on eCos in Section 3.2.2 revealed that the stack memory segments are highly susceptible to soft errors and that *return addresses* and *frame pointers* are the most homogeneous reasons for failures caused by stack memory. This section presents a dependability aspect that detects and corrects memory errors in these stack elements.

Stack frames

In brief, stack memory is typically organized as *stack frames* that store the local variables of a C/C++ function. C/C++ compilers automatically allocate a new stack frame for each function invocation. In addition, the x86 and x86-64 CPU architectures use the stack frame for passing the current instruction pointer as return address to the invoked function. Other CPU architectures, especially RISC CPUs, pass the return address by a dedicated CPU register; however, that register is spilled to stack memory on nested function calls as well.

The GCC and Clang compiler flag `-fomit-frame-pointer` *omits saving the* `%ebp` *register in some cases.*

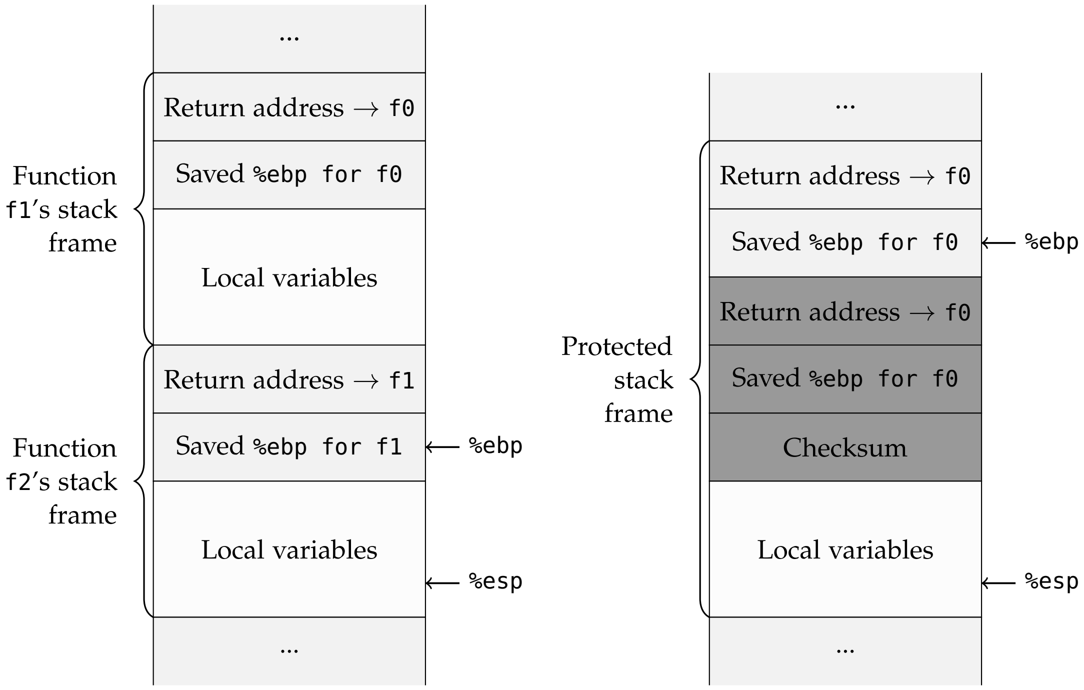
Figure 6.4a illustrates the layout of stack memory for the x86 architecture and shows the stack frame of a function `f1` that has invoked another function `f2`. Each stack frame starts with the return address of the caller function. For example, the return address of `f2`'s stack frame points to a program instruction of `f1`. The CPU register `%ebp` – called *base pointer* or *frame pointer* – points to the current stack frame and, thus, is defined at the beginning of a function execution. To be able to restore the previous `%ebp` register value on function return, the previous value is saved in the stack frame as well. On function return, the saved `%ebp` value is loaded into the `%ebp` register and the control flow jumps to the return address.

Hence, bit errors in the return address directly cause illegal control flow. Nicolescu and associates [178, p. 105] confirm that faults in return addresses cause program failures that even circumvent conventional control-flow checking mechanisms (see Section 2.4.1.3). As a remedy, this section presents an aspect-oriented mechanism that applies an error-correcting code to the return addresses and saved frame pointers.

6.3.1 *Implementation*

The return address and frame pointer together occupy only eight bytes of storage on the x86 architecture. Thus, data duplication in combination with a checksum is certainly the most efficient error-correcting code. A two's complement checksum detects all single bit errors and burst errors up to the length of the checksum [163]. These errors can be corrected by data duplication. Figure 6.4b illustrates a protected stack frame that stores a redundant return address, a redundantly saved `%ebp`, and a checksum of both values. This information redundancy is created at the beginning of a function execution. Right before the respective function returns, the actual return address and saved frame pointer are checked. Majority voting is carried out if the checksum indicates an error:

1. If the checksum matches the redundant return address and redundantly saved frame pointer, then the actual return address



(a) Typical layout of stack frames as used by the GCC and LLVM/Clang compilers.

(b) A protected stack frame stores a redundant return address and a redundantly saved %ebp in addition to a checksum of both elements.

Figure 6.4: Layout of stack frames for the x86 architecture. The CPU register %ebp points to the second element of the current stack frame, whereas the register %esp points to the end of the current stack frame. The x86-64 architecture implements a similar layout of stack frames using the 64-bit registers %rbp and %rsp.

or frame pointer is corrupt and can be repaired using the redundant ones.

2. If the redundant return address and redundantly saved frame pointer are identical to the actual return address and saved frame pointer, then the checksum is faulty and can be ignored.

3. Otherwise, the stack frame cannot be repaired and an error is signaled.

A drawback of majority voting is that memory areas filled with a recurring bit pattern, such as all zeros, are wrongly interpreted as valid stack frames. A countermeasure is to add a pseudo-random value to the checksum, and to exploit the knowledge that the actual return address and frame pointer must not be identical on a von Neumann architecture.

The sketched approach can be implemented as a dependability aspect using the AspectC++ programming language. Figure 6.5 shows the shortened implementation of the Return-Address Protection (RAP)

```
 1 aspect ReturnAddressProtection {
 2 protected:
 3   pointcut virtual where() = 0; // must be defined by derived aspects
 4   pointcut virtual correction() = where(); // optional error correction
 5   pointcut on_error() = execution("void ReturnAddressProtection::error()");
 6
 7 private:
 8   static inline void error() {} // explicit join point, exposed via on_error()
 9
10   // error detection and correction:
11   advice execution(where() && correction()) && !on_error() : around() {
12     RedundantRetAddr<JoinPoint::JPID> rra; // RAII idiom
13     tjp->proceed(); // continue the intercepted function’s execution (inline)
14   }
15
16   // error detection, only:
17   advice execution(where() && !correction()) && !on_error() : around() {
18     ChecksumRetAddr<JoinPoint::JPID> chksum; // RAII idiom
19     tjp->proceed(); // continue the intercepted function’s execution (inline)
20   }
21
22   advice execution(where()) : order(derived("ReturnAddressProtection"),
23     /* highest precedence */       !derived("ReturnAddressProtection"));
24 };
```

Figure 6.5: Shortened implementation of Return-Address Protection using AspectC++. The shown aspect provides the pointcut-based configuration interface in lines 3 to 5. The pointcut *where()* must be defined to capture those functions whose stack frames shall be protected, whereas the pointcut correction() allows choosing between error detection and error correction. In total, the complete implementation amounts to about 170 lines of AspectC++ code.

aspect, which provides the pointcut-based configuration interface described in Section 6.1 by declaring the pure virtual pointcut *where()* in line 3 and by defining the pointcut on_error() in line 5. In addition, the aspect defines the virtual pointcut correction() in line 4, which can be overridden to disable error correction for specific join points.

The pieces of advice in lines 11 to 20 use the around keyword to intercept the execution of functions captured by the pointcut expression *where()*. In particular, the upper piece of advice allocates an instance of the class template RedundantRetAddr<> as local variable before continuing to execute the intercepted function via tjp->proceed(). During allocation of the local variable, the constructor of the class template RedundantRetAddr<> (not shown) copies the return address and saved frame pointer from the current stack frame to volatile member variables and computes the checksum. For access to the current stack frame from the C/C++ language, the GCC and LLVM/Clang compiler provide the intrinsic function __builtin_frame_address(0).

Using __builtin_frame_address(0) within a function disables the optimization flag -fomit-frame-pointer for that function.

Once `tjp->proceed()` finishes, the destructor of the class template `RedundantRetAddr<>` is invoked implicitly to deallocate the local variable; the destructor also implements the checksum verification and majority voting.

The usage of a local variable and implicit destructor invocation is an instance of the C++ idiom *Resource Acquisition Is Initialization (RAII)* [240, p. 354ff], which provides exception safety because the destructor is invoked even if an exception is thrown.

The advice in lines 17 to 20 allocates a checksum for those functions that are not captured by the pointcut `correction()` and, thus, receive only error detection. Both pieces of advice pass the constant value `JoinPoint::JPID` to the respective class templates. That value represents a unique identifier per join point and serves as pseudo-random value that is added to the checksum to encode it in a function-specific way. Furthermore, adding one constant value to the checksum does not cause any runtime overhead on the x86 and x86-64 architectures, because the `lea` (*load effective address*) CPU instruction can add two register values and one immediate value in a single clock cycle.

Finally, the advice in line 22f makes sure that the RAP aspect gets the highest order of precedence, so that no other aspect can delay the protection. In summary, the shown aspect implements an error-detecting and error-correcting code that covers the return address and saved frame pointer of a stack frame as illustrated in Figure 6.4b. The implementation focuses on the x86 and x86-64 architectures; however, the aspect can be extended easily to other CPU architectures. For example, the ARM architecture uses the link register `r14` to store the return address. The RAP aspect only needs extension by a small piece of inline assembly code that accesses the register `r14` instead of using `__builtin_frame_address(0)`.

6.3.2 *Whole-Program Optimization*

The RAP aspect is a highly configurable mechanism with respect to its placement, which is specified by the pure virtual pointcut *`where()`* as shown in line 3 of Figure 6.5 on the facing page. Yet it makes no sense to select all functions for protection, because optimizing compilers typically perform *function inlining* [19]. Functions that become inline do not allocate a separate stack frame and, thus, do not benefit from RAP. The decision whether a function becomes inline is made by the C++ compiler after aspect weaving. Hence, the configuration pointcut *`where()`* must be confined to match only non-inline functions. This can be achieved by analyzing the symbol table of the compiled executable binary.

Function inlining

However, it is still unfavorable to apply the RAP aspect to all non-inline functions. For example, Figure 6.6 shows the call-stack histogram of the operating system eCos running the benchmark pro-

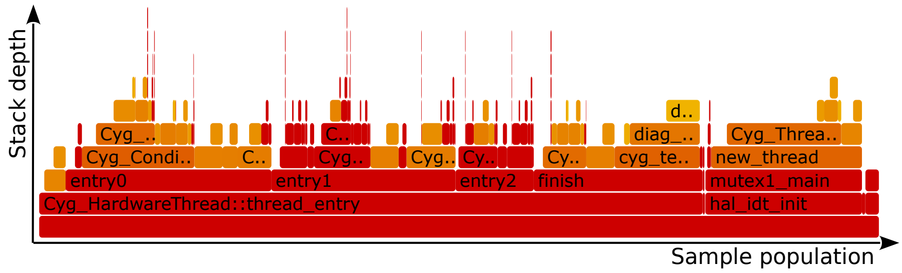


Figure 6.6: Call-stack histogram of the operating system eCos running the benchmark program MUTEX1. Each bar represents a function's stack frame. The x-axis illustrates the period of time that the function's stack frame spends on the call stack. Bright colors indicate a high number of invocations per function, whereas dark-red colors mark functions that are only called once. This *flame graph* reveals that a few functions spend a long time on the call stack, whereas a large share of functions return quickly or are called often or both.

gram MUTEX1 (see Section 3.2.2.2 on page 50). Some functions are called only a few times but spend a long time on the call stack (dark colors), whereas other functions are called often or return quickly or both (bright colors). As data lifetime is directly proportional to soft-error susceptibility (see Section 3.3), the stack frames of long-lasting functions are exceedingly critical. On the other hand, the chance that a soft error corrupts the return address of a short-running function is negligible. Thus, RAP is most efficient when applied to long-lasting functions only.

In general, the identification of long-lasting functions at compile time is an undecidable problem. Therefore, I propose the following heuristic as an approximation: It turns out that a context switch dramatically prolongs the lifetime of a function, because the function does not return until the respective thread of control is resumed. Hence, all functions that may request a context switch – directly or indirectly – are promising candidates for long-lasting functions. In other words, I consider a function as long-lasting if there is a potential control-flow transition from the function to a context switch. For instance, the eCos kernel implements the context switch in the assembler routine `hal_thread_switch_context`. Thus, the static control-flow reachability analysis presented in Section 5.4.2 on page 95 yields an approximation for the set of long-lasting functions.

In summary, inline functions and short-running functions can be excluded from Return-Address Protection to avoid excessive runtime overhead. The process of static whole-program analysis, as described in Section 5.4, implements an optimization heuristic that identifies the potentially long-lasting functions that should be protected. Finally, Chapter 7 evaluates the effectiveness and efficiency of the RAP dependability aspect and optimization heuristic.

6.4 VIRTUAL-FUNCTION POINTER PROTECTION

The L4/Fiasco.OC case study in Section 3.2.3 identified *virtual-function pointers (vptrs)* as a common reason for failures of the microkernel. Such *vptrs* are the usual technique for compilers to implement the dynamic dispatch of virtual C++ functions [240, p. 67f]. Thus, control-flow integrity depends on the integrity of *vptrs*, which should be provided with redundancy for error detection and correction.

The concrete implementation details of *vptrs* are compiler specific; however, L4/Fiasco.OC uses the popular GNU C++ compiler (GCC). Initially, this section outlines the *vptr* implementation scheme of the GCC. Subsequently, this section presents a dependability aspect that protects the *vptrs* from memory errors.

6.4.1 *C++ Object Layout of GCC and LLVM/Clang*

In short, each instance of a C++ class with virtual functions contains a *vptr* that refers to a table of function pointers (the *vtable*). The GCC and LLVM/Clang store the *vptr* in the lowest memory addresses of such class instances. Consider the three classes `A`, `B`, and `C` that declare some virtual functions, and the class `C` inherits from both `A` and `B`. Figure 6.7 illustrates these classes and shows that both `A` and `B` get an implicit *vptr*, because they have no base classes with virtual functions. However, the class `C` does not introduce another *vptr* but reuses `A`'s and `B`'s *vptrs* instead. The value of a *vptr* remains constant after object construction until object destruction, and it is used at runtime for locating the corresponding function pointer that is needed for invoking a virtual function. Thus, any memory error that corrupts a *vptr* can result in a program crash or even causes the invocation of a wrong function.

6.4.2 *Implementation*

The Virtual-Function Pointer Protection (VPP) is a dependability aspect that replicates the *vptrs* of C++ objects to allow for error detection and error correction. A pointcut-based configuration interface allows choosing between duplication and triplication of *vptrs*: Duplication enables error detection, whereas triplication applies majority voting for correcting any error that corrupts a single *vptr*.

Encoded replication

The replicated *vptrs* are stored in an encoded binary representation, because otherwise, memory areas filled with a recurring bit pattern, such as all zeros, would be wrongly interpreted as valid replicas. Moreover, the encoding is implemented in a type-specific way to detect *vptrs* of incompatible data types.

Figure 6.8 shows a shortened implementation of the VPP dependability aspect using AspectC++. The source code of the aspect starts

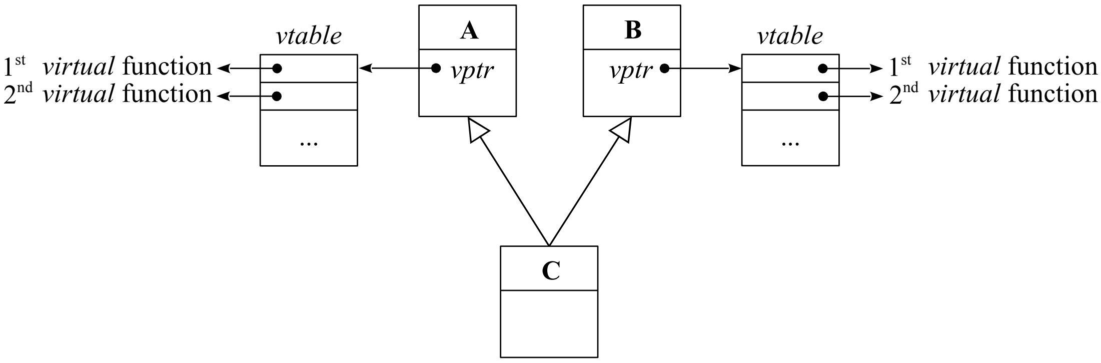


Figure 6.7: Virtual-function pointers (*vptrs*) and virtual-function tables (*vtables*) as implemented by the GCC and LLVM/Clang compiler. A class that declares a virtual function gets an implicit *vptr* if the class has no base class with virtual functions. In the depicted example, the classes `A` and `B` satisfy this condition and get a *vptr*, whereas the class `C` reuses the *vptrs* from its base classes.

in line 13 and declares the pointcut-based configuration interface in lines 15 to 16, which specifies the target classes whose *vptrs* are to be protected. By default, error correction is enabled (see line 16) and, for the sake of clarity, all further lines of code that implement only error detection are omitted in this listing. User-defined handling of uncorrectable errors can be specified by advice using the pointcut `on_error()` in line 17.

Section 5.3.3 describes the JPTL.

Discovery of vptrs

The centerpiece of the shown aspect is the generic introduction in lines 19 to 40. First, the generic introduction discovers at compile time whether the receiving class contains a *vptr*. For this purpose, the `JPTL::FunctionIterator<>` and `JPTL::DestructorIterator<>` types from the Join-Point Template Library (JPTL) are instantiated using the class template `VirtualFunctionCount<>`, which is presented in Figure 5.4 on page 92. The number of virtual functions declared in the receiving class is counted thereby (lines 21 to 23). Likewise, the number of virtual functions declared in the receiving class *and all* its base classes is calculated in lines 25 to 27. Recollecting the C++ object layout described in Section 6.4.1, the constant value `HAS_VPTR` (line 30) identifies whether the receiving class contains a *vptr*, that is, if the class declares a virtual function but has no base class with virtual functions.

Based on that information, the generic introduction declares in line 34 the member variable `red_vptrs`, which instantiates the class template `RedundantVptrs<>`. That class template (implementation not shown) provides storage for the redundant *vptrs* and encapsulates the error detection and correction functionality. One redundant *vptrs* is encoded by a bitwise `XOR` operation with the class-specific hash value `JoinPoint::HASHCODE`, whereas the other is encoded by a bitwise `XOR` operation with the bitwise inversion of `JoinPoint::HASHCODE`. How-

```
#include "JPTL.h" // use the Join-Point Template Library (JPTL)

template<typename TypeInfo, typename>
struct InitVptr { // 'Action' class template for use with JPTL::BaseIterator<>
  static inline void exec(typename TypeInfo::That* object) {
    object->red_vptrs.init(object); } };

template<typename TypeInfo, typename>
struct CheckVptr { // 'Action' class template for use with JPTL::BaseIterator<>
  static inline void exec(typename TypeInfo::That* object) {
    object->red_vptrs.check(object); } };

aspect VirtualFunctionPointerProtection {
protected:
  pointcut virtual where() = 0; // classes whose vptrs are to be protected
  pointcut virtual correction() = where(); // optional error correction
  pointcut on_error() = execution("void ...::vptr_error()") && within(where());

  advice derived(where() && correction()): slice class { // generic introduction
    enum {
    VIRT_FUNCTIONS = // number of virtual functions of the target class
     JPTL::FunctionIterator<JoinPoint, VirtualFunctionCount>::EXEC::COUNT +
     JPTL::DestructorIterator<JoinPoint, VirtualFunctionCount>::EXEC::COUNT,

    BASE_VIRT_FUNCTIONS = // virtual functions of the target class and its bases
     JPTL::BaseFunctionIterator<JoinPoint, VirtualFunctionCount>::EXEC::COUNT +
     JPTL::BaseDestructorIterator<JoinPoint, VirtualFunctionCount>::EXEC::COUNT,

    // discover at compile-time whether the target class introduces a vptr:
    HAS_VPTR = (VIRT_FUNCTIONS != 0) && (BASE_VIRT_FUNCTIONS == VIRT_FUNCTIONS)
    };

  public: // there is a template specialization for HAS_VPTR==false (do nothing)
    RedundantVptrs<JoinPoint::That, JoinPoint::HASHCODE, HAS_VPTR> red_vptrs;

    void vptr_error() {} // explicit join point, invoked by RedundantVptrs<>
    // use the JPTL to iterate over base classes and to access the redundancy
    void init_vptr() { JPTL::BaseIterator<JoinPoint, InitVptr>::exec(this); }
    void check_vptr() { JPTL::BaseIterator<JoinPoint, CheckVptr>::exec(this); }
  };

  advice construction(derived(where())) ||
         destruction(derived(where())) : before() {
    tjp->target()->init_vptr(); // iterate over base classes (see above)
  }

  advice call(derived(where())) && call("virtual % ...::%(...)") : before() {
    tjp->target()->check_vptr(); // iterate over base classes (see above)
  }
};
```

Figure 6.8: Shortened implementation of Virtual-Function Pointer Protection using AspectC++. This listing uses the class template `VirtualFunctionCount<>` that is presented in Figure 5.4 on page 92. The implementation details of the class template `RedundantVptrs<typename T, int HASHCODE, bool HAS_VPTR>` are not shown. In total, the complete implementation amounts to 275 lines of code.

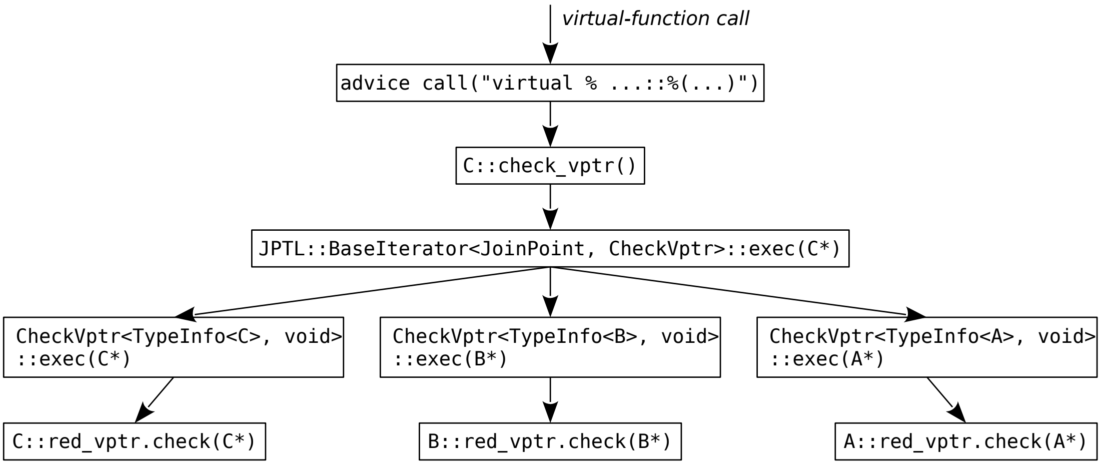

virtual-function call
advice call("virtual % ...::%(...)")
C::check_vptr()
JPTL::BaseIterator<JoinPoint, CheckVptr>::exec(C*)
CheckVptr<TypeInfo<C>, void>
::exec(C*)
CheckVptr<TypeInfo<B>, void>
::exec(B*)
CheckVptr<TypeInfo<A>, void>
::exec(A*)
C::red_vptr.check(C*)
B::red_vptr.check(B*)
A::red_vptr.check(A*)

Figure 6.9: Call graph initiated by a virtual-function call. The advice captures any virtual-function call and invokes the member function `check_vptr()` of the target class, for example, `C::check_vptr()`. Subsequently, the `JPTL::BaseIterator<>` instantiates the class template `CheckVptr<>` for each data type of the class hierarchy – `A`, `B`, and `C` – and then invokes the function `exec()`.

ever, there is also a template specialization for `HAS_VPTR==false`, so that no code is generated for classes without any *vptr*.

Finally, the generic introduction defines the two member functions `init_vptr()` and `check_vptr()` in lines 38 to 39. Both functions use the `JPTL::BaseIterator<>` to traverse the class hierarchy and invoke the `check()` and `init()` routines of the member `red_vptr`. For example, the member function `check_vptr()` in line 39 uses the class template `CheckVptr<>` for instantiating the `JPTL::BaseIterator<>`. Lines 8 to 11 show the implementation of the class template `CheckVptr<>`, which basically defines the static function `exec()` that gets a pointer to an object and invokes the `check()` routine of the object's data member `red_vptr`.

Traversal of the class hierarchy

Consider the aforementioned example in which the class `C` inherits from the base classes `A` and `B` (see Figure 6.7 on page 112): The resulting call graph initiated by an invocation of `C::check_vptr()` is illustrated in Figure 6.9. In brief, the `JPTL::BaseIterator<>` instantiates the class template `CheckVptr<>` for each data type of the class hierarchy and then invokes the function `exec()`. Each *vptr* in the class hierarchy gets checked thereby (see bottom row in Figure 6.9).

Last, but not least, the pieces of advice in lines 42 to 49 make sure that the introduced member functions are invoked as necessary: Whenever a class instance is constructed, the redundant *vptrs* are initialized (lines 42 and 44). The same applies to object destruction (line 43), because the GCC and LLVM/Clang compiler reset the *vptr* to match the type of the destructor being executed. Finally, the pointcut expression in line 47 captures any virtual-function call to classes specified by the pointcut *`where()`* and classes derived thereof. Such a call triggers the implicit invocation of the function `check_vptr()` as illustrated by the top of Figure 6.9.

Although multiple threads of control may access the objects with replicated *vptrs* concurrently, the VPP aspect causes no race conditions. This is because the value of a *vptr* remains constant after object construction until object destruction, and the same applies to the replicas. The only operation that modifies a *vptr* is error correction, however, this only happens if a quorum of two identical *vptrs* is found during majority voting. In this specific case, correcting the deviating *vptr* by overwriting is an idempotent operation. In other words, even if multiple threads of control initiate the correction of one erroneous *vptr* at the same time, the result is not affected. Thus, the VPP aspect is thread safe in every respect.

Thread safety

In summary, the VPP aspect replicates the compiler-generated *vptrs* in C++ objects and inserts code for error detection and error correction at the call sites of virtual functions. The implementation provides a pointcut-based configuration interface that enables a selective placement, offers the choice between error detection and error correction, and allows for user-defined error handling. Chapter 7 evaluates the effectiveness and efficiency of the VPP dependability aspect.

6.5 GENERIC OBJECT PROTECTION

The most frequent reasons for failures of both L4/Fiasco.OC and the eCos kernel are memory faults that corrupt critical kernel data structures, such as the process scheduler (see Section 3.2). Both kernels implement these data structures as instances of C++ classes, whose individual data members exhibit an exceeding criticality.

As a countermeasure, redundancy can be introduced into the class instances (objects) to allow for error detection and error correction. For example, a CRC code that is calculated over the values of all data members can be stored as additional data member in each object. Before any access to such an object, the CRC code can be verified to detect errors. After write access, the object's CRC code is updated. In principle, such an approach characterizes the Generic Object Protection (GOP) dependability aspect.

The essential AspectC++ 2.0 language features for GOP are generic introductions (see Section 5.3) and advice for access to variables (see Section 5.2). Using these language features, Figure 6.10 shows a highly simplified implementation that highlights the fundamental ideas of the GOP aspect. A user needs only to define the pure virtual pointcut *`where()`* in line 3 to apply the GOP aspect to a particular set of class data types. The generic introduction in lines 6 to 11 extends these class data types by the two data members `redundancy` (line 7) and `static_redundancy` (line 9), which both instantiate the class template `CRC<>`. In short, that class template provides storage and algorithms

Generic introduction of redundancy

for computing the CRC code for dynamic and static data members, as specified by the second template parameter. Section 6.5.1 describes the implementation of that class template in detail. Lines 44 to 45 define the previously declared static data member outside of the class scope as required by the C++ standard [116, § 9.4.2].

Generic advice for access to data members

The remaining pieces of generic advice make sure that the introduced redundancy gets checked and updated at runtime. On the one hand, lines 14 to 18 capture any access to non-static data members and use `tjp->target()` of the join-point API (see Section 5.2.2) to retrieve a pointer to the involved object. Subsequently, the respective pointer is passed as argument to the `check()` routine of the introduced member `redundancy`. Likewise, lines 21 to 24 invoke the `update()` routine after write access or constructor execution. On the other hand, the pieces of advice in lines 27 to 36 capture any access to static data members and invoke the `check()` and `update()` routines of the member variable `static_redundancy`, whose scope is provided by `JoinPoint::Target` of the join-point API. Because the pointcut functions `get` and `set` do not capture member access via pointer or reference, the advice in lines 38 to 40 specifies an unsatisfiable compile-time assertion using the pointcut function `ref` to prohibit aliasing of data members (see Section 5.2.2). Altogether, the respective redundancy gets checked before every access and gets updated afterwards.

However, it turns out that checking an object's redundancy on *every* access incurs a high runtime overhead. Therefore, I propose to omit any checking within member functions of the same class: An object can be checked once at the call site of a member function instead of repetitive checking within the member function. Section 6.5.2 describes such a call-based optimization and further presents a whole-program optimization that identifies temporally close checks that can be eliminated for performance reasons. Finally, Section 6.5.3 extends the GOP dependability aspect by a wait-free synchronization algorithm that guarantees thread safety and interrupt safety.

Aspect-oriented design of the GOP aspect

The GOP extensions are implemented as separate abstract aspects as illustrated in Figure 6.11: *`Introducer`* refers to an aspect that implements the generic introduction of redundancy, such as CRC codes or Hamming codes (see Section 6.5.1). *`GetSetAdviceInvoker`* resembles the highly simplified aspect shown in Figure 6.10 but omits any checking within member functions of the same class. The respective checks around calls of member functions are inserted by *`CallAdviceInvoker`*, which declares three additional pure virtual pointcuts that enable whole-program optimization. Likewise, the pure virtual pointcut *`synchronized()`* declared by the *`Synchronizer`* aspect specifies the application of the wait-free synchronization algorithm. After all, the abstract aspect *`GOP`* inherits from four base aspects and provides default expressions for the inherited pure virtual pointcuts except *`where()`*, which must be specified by the user to apply GOP.

```
 1 aspect GenericObjectProtection {
 2 protected:
 3   pointcut virtual where() = 0; // class types to be protected
 4   pointcut on_error() = execution("void ...::gop_error()");
 5 
 6   advice where() : slice class Intro { // generic introduction
 7     CRC<JoinPoint> redundancy; // for non-static members
 8     typedef CRC<JoinPoint, true> static_redundancy_t;
 9     static static_redundancy_t static_redundancy;
10     static void gop_error() {} // explicit join point (see above)
11   };
12 
13   // verify the redundancy before non-static member access:
14   advice (get(where()) && !get("static % ...::%")) ||
15          (set(where()) && !set("static % ...::%")) : before() {
16     if (! tjp->target()->redundancy.check( tjp->target() ) )
17       tjp->target()->gop_error();
18   }
19 
20   // update the redundancy after write access or initialization:
21   advice (set(where()) && !set("static % ...::%")) ||
22           construction(where()) : after() {
23     tjp->target()->redundancy.update( tjp->target() );
24   }
25 
26   // verify the static redundancy before static-member access:
27   advice (get(where()) && get("static % ...::%")) ||
28          (set(where()) && set("static % ...::%")) : before() {
29     if (! JoinPoint::Target::static_redundancy.check() )
30       JoinPoint::Target::gop_error();
31   }
32 
33   // update the redundancy after write access to static members:
34   advice (set(where()) && set("static % ...::%")) : after() {
35     JoinPoint::Target::static_redundancy.update();
36   }
37 
38   advice ref(where()) : before() { // prohibit aliasing
39     static_assert(JoinPoint::JPTYPE == 0, "aliasing of member");
40   }
41 };
42 
43 // definition of the introduced static data member:
44 slice GenericObjectProtection::Intro::static_redundancy_t
45       GenericObjectProtection::Intro::static_redundancy;
```

Figure 6.10: Highly simplified implementation of Generic Object Protection using AspectC++. The class data types specified by the pointcut expression *where()* receive a generic introduction (lines 6 to 11) that extends these classes by a non-static and by a static instance of CRC<>. These instances cover the dynamic and static data members of the receiving class. The remaining pieces of advice make sure that the CRC code gets checked before every object access and gets updated afterwards. In total, the complete implementation amounts to about 3,000 lines of code.

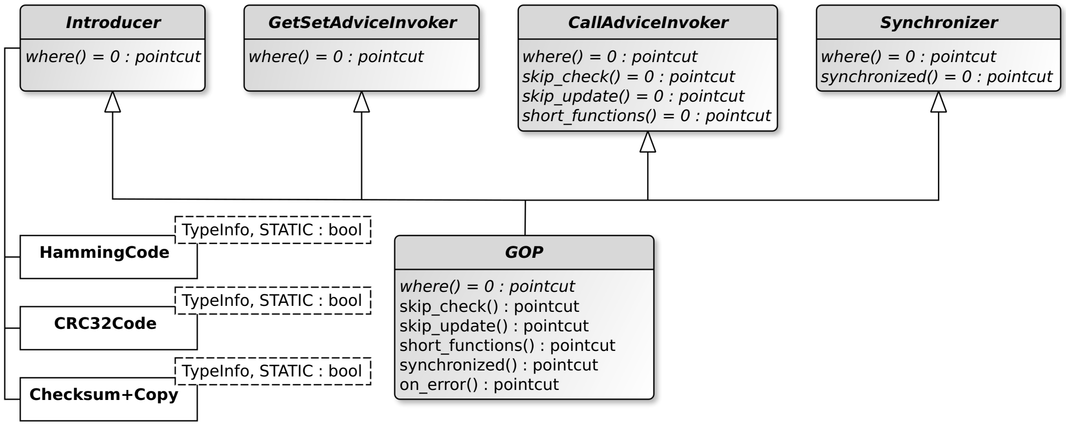


Figure 6.11: Design of Generic Object Protection. The abstract aspect `GOP` represents the user interface of the dependability aspect, which inherits from four base aspects that implement distinct features. Each base aspect is configurable at compile time via pure virtual pointcuts, such as `where()` and `synchronized()`. Moreover, the `Introducer` allows choosing from several types of redundancy.

In summary, GOP provides the following key features, which are discussed individually in separate sections:

- Generative redundancy (Section 6.5.1)
- Call-based optimization (Section 6.5.2)
- Wait-free synchronization for thread safety (Section 6.5.3)

6.5.1 *Generative Redundancy*

GOP inserts information redundancy as additional data members into the target classes (see lines 6 to 11 in Figure 6.10). Thus, redundancy becomes an integral part of each class instance and the compiler automatically allocates the needed memory whenever such an object is constructed. As illustrated by Figure 6.11, any kind of information redundancy can be introduced. This thesis exemplarily considers four different implementations of redundancy that can be used by the GOP aspect. Table 6.2 summarizes the four configuration options, which differ in error detection and correction capabilities.

Error detection and error correction capabilities

The CRC options detect any possible 1-bit, 2-bit, and 3-bit errors in data objects smaller than 256 MiB by the implemented CRC-32/4 code [43]. Moreover, any burst error up to a length of 32 bits is detected [163]. In addition, the CRC+Copy option transparently corrects such bursts and any 1-bit error by maintaining a shadow copy of each data member. The Sum+Copy option implements a computationally cheaper two's complement addition checksum and achieves the same error-correction capability, but arbitrary multi-bit errors can go unde-

OPTION	DESCRIPTION	LOC
CRC	CRC-32 implementation, leveraging Intel's SSE4.2 instructions (only error detection)	163
CRC+Copy	CRC (see above), plus one copy of each data member for additional error correction	210
Sum+Copy	32-bit two's complement addition checksum, plus one copy of each data member for error correction	198
Hamming	Extended Hamming code, processing up to 64 bits in parallel (see Section 6.5.1.2)	426

Table 6.2: Configuration options for redundancy to be introduced by the GOP aspect. The four options exemplarily show that GOP facilitates a concise implementation by only a few lines of code (LOC).

tected [163]. In addition to the correction of single bit errors and burst errors, the Hamming code reliably detects any 2-bit error.

Template-based code generation using the JPTL

I implemented each of these options as individual class templates that can be instantiated by a generic introduction. For example, the class template `CRC<>`, as shown in line 7 of Figure 6.10, is parameterized by AspectC++'s keyword `JoinPoint`, which provides access to the compile-time join-point API (see Section 5.3). The class template `CRC<>` obtains thereby type information on the target class and, thus, can use the join-point template library (JPTL). In particular, the `JPTL::MemberIterator<>` facilitates template-based code generation by recursive instantiation for each data member of the target class. In other words, that compile-time iterator adapts itself depending on the type information. By this means, a *generative* implementation of a CRC-32 code that covers all data members of *any* class requires only 163 lines of code.

In each recursive step, the `JPTL::MemberIterator<>` provides type information on the individual data member, such as its data type and a pointer to the member. Information on the data type allows calculating the size of the data member by applying the compile-time `sizeof` operator[2]. Altogether, the memory area occupied by each data member can be identified and, thus, can be covered by the CRC-32 code.

[2] The only exceptions are arrays of variable length, whose size cannot be determined at compile time, so that they cannot be protected by GOP. The C++ language does not support such arrays at all, because they break class inheritance [116, § 3.9]. However, many compilers accept such member declarations as an extension for compatibility with legacy C code.

6.5.1.1 *Object Composition*

C++ objects can be composed of several subobjects that complicate the GOP approach. For example, consider the class $\mathcal{C}$ that contains the member $\mathcal{C}_{sub}$ of class type and GOP redundancy: $\mathcal{C} = \{\mathcal{C}_{sub}, \ldots, \mathcal{R}\}$. The subobject – an instance of $\mathcal{C}_{sub}$ – could be protected twice, both by $\mathcal{R}$ and its own redundancy $\mathcal{R}_{sub}$. Yet, it suffices to protect objects only once. In addition, the subobject can be accessed independently of the parent object. Therefore, subobjects should be protected only by their own redundancy $\mathcal{R}_{sub}$.

Subobjects

GOP excludes members of class type from protection because they must be protected individually. The `JPTL::MemberIterator<>` provides the data type of each member, so that the standard C++ type trait `std::is_class<>` can be used to identify whether a member is of class type [116, § 20.9]. Likewise, arrays of class-type objects can be identified by first extracting the underlying object type via `std::remove_all_extents<>` before applying the former type trait. Whenever a member of class type or array thereof is found, GOP skips that member and proceeds with the subsequent one.

C++ type traits

6.5.1.2 *Adaptive Hamming Code*

Table 6.2 on the previous page indicates that the Hamming code is the most complex configuration option regarding lines of source code. The advantage of the Hamming code is that the amount of redundant memory grows only logarithmically with the protected object's size, whereas the other options for error correction exhibit linear growth.

Section 2.2.3 on page 19 presented a separable Hamming code that appends several check bits to a bit vector. However, such an approach requires many shifts and logical operations in software to isolate the individual bits for computation [219, p. 276]. Instead of computing such a Hamming code for one long bit vector, it is more efficient to slice the bit vector into chunks of several bits and to compute a separate Hamming code for each slice. The advantage is that the Hamming codes for all slices can be computed in parallel by using a word-wise `XOR` operation. Consider 128 bits of memory that are sliced into four words w, x, y, z of each 32 bits. Thus, 32 individual Hamming codes are calculated over the finite field of two elements[3] by multiplication with a transformation matrix G as follows:

Bit slicing

[3] In the finite field of two elements, addition corresponds to the logical `XOR` operation, whereas multiplication is equivalent to the logical `AND` operation.

$$G \cdot \begin{pmatrix} w \\ x \\ y \\ z \end{pmatrix} = \underbrace{\begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{1} \\ \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{0} & \mathbf{1} & \mathbf{1} & \mathbf{1} \end{pmatrix}}_{G} \cdot \begin{pmatrix} w_0 & w_1 & \dots & w_{31} \\ x_0 & x_1 & \dots & x_{31} \\ y_0 & y_1 & \dots & y_{31} \\ z_0 & z_1 & \dots & z_{31} \end{pmatrix}$$

$$= \begin{pmatrix} w_0 & w_1 & \dots & w_{31} \\ x_0 & x_1 & \dots & x_{31} \\ y_0 & y_1 & \dots & y_{31} \\ z_0 & z_1 & \dots & z_{31} \\ w_0 + x_0 + z_0 & w_1 + x_1 + z_1 & \dots & w_{31} + x_{31} + z_{31} \\ w_0 + y_0 + z_0 & w_1 + y_1 + z_1 & \dots & w_{31} + y_{31} + z_{31} \\ x_0 + y_0 + z_0 & x_1 + y_1 + z_1 & \dots & x_{31} + y_{31} + z_{31} \end{pmatrix}$$

Thus, each of the 32 columns gets a *vertical* Hamming code that can be computed in parallel if the processor architecture supports a 32-bit XOR operation[4]. For example, the last row of the result can be calculated by the two word-wise XOR operations $x + y + z$. In this example, only six word-wise XOR operations are required to compute all 96 check bits.

Furthermore, the Hamming code in each column can correct one bit error independently, so that up to 32 bit errors can be corrected if each of them occurs in a different column. Thus, all burst errors up to 32 bits can be corrected.

The remaining challenge for implementing such a vertical Hamming code is to obtain a suitable transformation matrix G, because the dimensions of the matrix depend on the number of data words to be transformed. GOP determines the size of each data member already at compile time via the sizeof operator, so that the transformation matrix for a particular class data type can be constructed by means of template metaprogramming at compile time.

Tailoring the transformation matrix

As discussed in Section 2.2.3 on page 19, the upper part of a transformation matrix consists of the identity matrix; only the lower rows are relevant for computing the check bits. In the previous example, the relevant rows of G are printed in bold face. These rows form a submatrix whose columns represent permutations of at least two ones. Thus, we have a certain degree of freedom when constructing the matrix: The matrix can be optimized to contain a minimal number

[4] A 64-bit processor can compute 64 bit slices in parallel.

of ones, because only ones cause an effective XOR operation. A zero element means that the template metaprogram does not need to generate any code for the element. For instance, the following matrix G_5 allows transforming five data words and contains a minimal number of ones:

$$G_5 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ \mathbf{1} & \mathbf{1} & \mathbf{1} & \mathbf{0} & \mathbf{0} \\ \mathbf{1} & \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{1} \\ \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{1} & \mathbf{0} & \mathbf{1} \end{pmatrix}$$

By constructing such a transformation matrix at compile time, a template metaprogram can generate an *adaptive* Hamming code tailored for each particular class data type. In summary, the bit-slicing technique for parallelization and the adaptation at compile time enable a highly efficient implementation of the Hamming code, which requires fewer redundancy than the other GOP options for error correction.

6.5.2 *Exploiting Object-Oriented Program Structure*

So far, GOP has been introduced as a mechanism that inserts runtime checks at *every* data access. Yet, the object-oriented paradigm encourages C++ developers to declare data members as `private` to afford fine-grained access control and to hide implementation details [169, p. 94A–98A]. Hence, access to many data members is typically restricted to member functions of the same class.

Checking of objects already at call sites of functions

Thus, a whole object can be checked already at the call site of a member function, so that access to `private` data members can be carried out without any individual checks. Repetitive checks are avoided thereby if the member function accesses multiple data members one after another. Likewise, the object's redundancy only needs to be updated once the member function returns. However, if the member function leaves the object in the meantime by calling another function, the object's redundancy needs to be updated as well, and the object should be checked after the other function has returned. For member functions declared as `const`, the redundancy does not need any updating unless there are `mutable` data members.

Figure 6.12 illustrates the insertion of `check()` and `update()` operations at the call sites of member functions by a UML sequence dia-

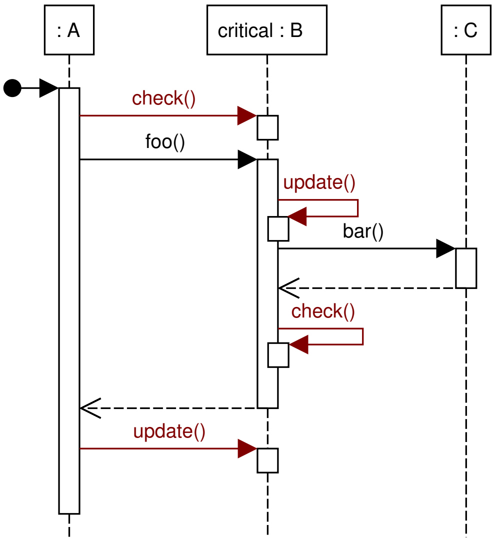


Figure 6.12: Call-based optimization of GOP. The UML sequence diagram depicts three objects, of which only the one named `critical` is covered by GOP. The call-based optimization inserts runtime checks (colored in dark red) at the call site of `B::foo()`. When the control flow leaves the `critical` object by calling `C::bar()`, further `update()` and `check()` operations are inserted to detect errors that occur during the execution of `C::bar()`. If `B::foo()` was declared as `const`, both `update()` operations would be omitted.

gram. In this example, GOP is applied only to the class `B`, whereas the other two classes remain unprotected. At first, an unnamed instance of the class `A` invokes the member function `B::foo()`, which triggers GOP to insert a `B::check()` operation (shown in dark red). After that, `B::foo()` continues executing as usual until it invokes `C::bar()`. At this point in time, GOP inserts a `B::update()` operation to reflect any modification of the object. As soon as `C::bar()` returns, GOP inserts another `B::check()` operation to detect errors that occurred in the meantime. Finally, `B::update()` is carried out after `B::foo()` has returned.

By checking at the call site, individual `check()` and `update()` operations on data access within member functions of the same class can be omitted. In other words, the sketched approach exploits the object-oriented program structure to relax the rather stringent insertion of checks at every data access. However, access to data members from outside of the respective class still needs individual checking, such as access to `public` data members from another class.

6.5.2.1 *Call-Site Analysis*

The call-based GOP approach avoids repetitive checks within member functions. At any point in time, the implicit `this` pointer of a member function refers to the current object that has been checked already.

Therefore, data access using the implicit this pointer does not need any further checking. Moreover, the same applies to calls of member functions using the implicit this pointer: There is no need to insert any check() or update() operation if the *caller* and *callee* objects are identical.

The uniform join-point API of AspectC++ (see Table 4.2 on page 68) provides the necessary information for comparing the *caller* and *callee* objects. For instance, tjp->that() yields the current this pointer at the place of function invocation or data access, and JoinPoint::That refers to the respective class data type. Likewise, tjp->target() yields a pointer to the class instance whose member function is called or whose data member is accessed, and JoinPoint::Target describes the respective class data type.

C++ type traits

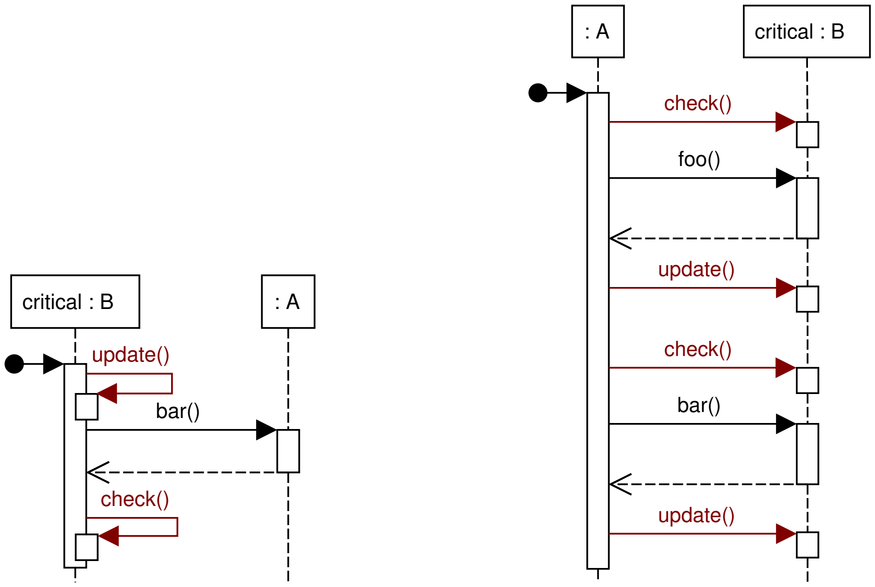
Both class data types can be tested on sameness by the type trait std::is_same<> at compile time. Only if both types are identical, the actual pointers to the caller and callee objects need to be compared. Optimizing compilers, such as the GCC, resolve the pointer comparison at compile time if possible, for example, if members are accessed using the implicit this pointer. Altogether, such a *call-site analysis* improves the efficiency by omitting repetitive check() and update() operations on the current object, which has been checked already at the call site.

6.5.2.2 *Whole-Program Optimization*

The call-site analysis avoids repetitive checks of the current object, but only considers a single join point at a time. Yet, AspectC++'s framework for user-defined static whole-program analyses (see Section 5.4) facilitates even more aggressive optimization by considering multiple join points at the same time. Figure 6.13 illustrates two use cases that benefit from whole-program optimization.

1. In Figure 6.13a, the control flow temporarily leaves the critical object by calling A::bar(), which triggers GOP to invoke the update() and check() operations. This is unfavorable if the called function returns quickly, for example, if it is an inline getter or setter function that executes only for a few instructions. In general, the depicted update() and check() operations only improve fault tolerance if the execution time of the called function A::bar() exceeds the overhead of protecting the object. Otherwise, it is better the leave the object unprotected for a few instructions.

2. Figure 6.13b depicts a sequence of two member-function calls with the critical object. Instead of updating the object's redundancy right after the first call, and immediately checking the same object again before the second call, it would be more



(a) Temporarily leaving the `critical` object's scope.

(b) Call sequence with the same object.

Figure 6.13: Use cases for whole-program optimization of GOP. Without whole-program optimization, GOP inserts the `check()` and `update()` operations printed in dark-red color. (a) If the function `A::bar()` returns quickly, both the `update()` and `check()` operations should be omitted. (b) The `update()` and `check()` operations between the call sequence of `B::foo()` and `B::bar()` should be omitted, so that only the first `check()` and last `update()` operations remain.

efficient to check the object *only once* before the entire call sequence and to update afterwards. Such a call sequence can even span several calls to arbitrary functions if there is no *long-lasting* function call in between.

In both cases, the decision whether to omit a particular `check()` or `update()` operation requires information on the *future* of the program execution. Such an optimization necessitates static program analyses. The first problem (Figure 6.13a) can be reduced to the identification of short-running functions, which is an undecidable problem in general. However, Section 6.3.2 on page 109 already presented a heuristic that identifies potentially long-lasting functions: Those functions that may request a context switch – directly or indirectly – are promising candidates for long-lasting functions, because a context switch dramatically prolongs the lifetime of a function. Thus, the complement set of functions yields an approximation for the set of short-running functions, which can be computed by the static control-flow reachability analysis presented in Section 5.4.2 on page 95.

Static control-flow reachability analysis

The second problem (Figure 6.13b) is to identify call sequences with the same target object at compile time. In many call expressions, the target object is only available in form of a pointer or reference. Therefore, I extended the AspectC++ compiler by the flow-sensitive pointer-alias analysis as described in Section 5.4, which attempts to determine whether different pointer expressions – or reference expressions – refer to the same variable. The alias analysis extends the AspectC++ project repository by information on the target object of each member-function call, which is assigned the intra-procedural identifier `target_object_lid`: Two function calls with identical values of `target_object_lid` refer to the same object.

Identification of call sequences

Thus, I implemented an XQuery program that extracts the longest call sequences for each `target_object_lid` within a basic block from the AspectC++ project repository. The limitation to a single basic block, enforced by comparing the element `cfg_block_lid`, guarantees that such a call sequence is always executed as a whole, so that each element is executed equally often. In addition, such a call sequence may contain calls to other short-running functions.

Finally, the XQuery program lists all individual join points where a `check()` or `update()` operation can be optimized out. The list is represented as the AspectC++ pointcut definitions `short_functions()`, `skip_check()`, and `skip_update()`, which can be included optionally by the GOP aspect (see Figure 6.11 on page 118).

6.5.2.3 *Inheritance and Polymorphism*

On the one hand, the call-site analysis and optimization imply that the class hierarchy of an object needs to be considered, because a member function of a derived class can implicitly access the inherited data members of its base classes. Thus, when an object of a derived class is checked, all its base classes should be checked as well. The `JPTL::BaseIterator<>` from the join-point template library traverses the class hierarchy at compile time and facilitates the invocation of respective `check()` and `update()` operations for each base class as shown by the VPP aspect (see Section 6.4.2).

Traversal of the class hierarchy

On the other hand, the dynamic dispatch of virtual functions at runtime conflicts with the call-based GOP approach: At the call site of a virtual function, it is impossible to determine the polymorphic object type at compile time. Therefore, I implemented another configuration option that introduces virtual `check()` and `update()` functions into polymorphic classes. Thus, the virtual `check()` and `update()` functions are dispatched to the most derived class, so that the complete object can be checked at any call site.

Dynamic dispatch of virtual functions

In summary, the call-based GOP approach avoids repetitive `check()` and `update()` operations by exploiting the object-oriented program structure, and further involves the class hierarchy by dynamic dispatch to derived classes and subsequent compile-time traversal of base classes.

6.5.3 *Concurrent Error Detection*

> “Critical sections are poorly suited for asynchronous, fault-tolerant systems: if a faulty process is halted or delayed in a critical section, nonfaulty processes will also be unable to progress.”
>
> – Maurice Herlihy [106, p. 124]

The concurrent modification of shared objects by multiple threads of control complicates GOP. The two basic operations `check()` and `update()`, which examine an object for memory errors and store its redundancy as described in the previous sections, can be disturbed by concurrent execution in subtle ways. For example, verifying a checksum of a shared object that is being concurrently modified certainly fails.

Introducing critical sections that enforce sequential execution of the `check()` and `update()` operations would restore program correctness at the expense of the undesired properties of locking, such as convoying and priority inversion [107]. To avoid these drawbacks, I developed a wait-free synchronization algorithm that enables excellent scalability of GOP on multiprocessor systems. I implemented the algorithm as the separate abstract aspect *`Synchronizer`* that is illustrated in Figure 6.11 on page 118. That aspect provides the pure virtual pointcut *`synchronized()`* that allows a user to specify in a fine-grained way those classes that are used concurrently. Thus, the aspect makes sure that the wait-free synchronization algorithm for GOP is applied automatically on access to concurrent data structures.

Some of the following passages have been quoted verbatim with permission[5] from an IEEE journal publication [32, p. 26–30]; these passages have been written exclusively by myself.

6.5.3.1 *Wait-free Synchronization*

“A method is *wait-free* if it guarantees that every call finishes its execution in a finite number of steps.” [107, p. 59] This means that a wait-free algorithm is necessarily lock free[6], ruling out the use of critical sections that delay other threads of control. We can exploit a particular insight to address wait freedom for GOP: If an object is at some time being modified by another thread, we can skip any further `check()` and `update()` operations on that object at the same time. The thread that *entered* the object *first* has already verified the object. The other way around, the thread that *leaves* an object *last* is committed to update the object’s redundancy. Thus, a consensus on *“which thread was first?”* (or last, respectively) must be found.

[5] ©2017 IEEE. Reprinted, with permission, from Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generic soft-error detection and correction for concurrent data structures. *IEEE Transactions on Dependable and Secure Computing*, 14(1):22–36, January 2017. doi: 10.1109/TDSC.2015.2427832

[6] The *lock-free* condition only guarantees system-wide progress and allows for individual threads to starve [107, p. 60].

To identify whether an object is being used, the aspect `Synchronizer` introduces a per-object *thread counter* into each shared class instance. This counter is incremented atomically[7] when a thread calls a member function of such an object; the counter is decremented on function return. Hence, a zero counter indicates an unused object that needs verification before usage. Likewise, a counter value of one causes an update of the object's redundancy before returning from the current member function.

However, a running `check()` or `update()` operation can be preempted, so that other threads could modify the object concurrently. To track such race conditions, the aspect `Synchronizer` additionally introduces a *dirty flag* into each shared object. Each thread marks its presence by overwriting the dirty flag with a thread-unique[8] value. A preempted thread checks for a *lost* race condition by examining whether the dirty flag has been overwritten. If so, the preempted thread aborts its checksum computation and continues without retry.

The following section specifies the sketched algorithm more precisely by a formal model that allows proving its correctness.

6.5.3.2 *Formal Model and Verification*

This section describes the wait-free synchronization algorithm in *Promela* [111], which is a specification language targeted to abstract models of concurrent programs. This allows focusing on the interaction and synchronization of concurrent threads, and further enables a tool-based verification of correctness properties.

Promela permits a limited set of language features in a C-like syntax. Figure 6.14 shows the complete abstract model of the wait-free algorithm. Line 1 defines the thread-unique values (1...N) by using the predefined variable `_pid` that identifies each Promela process, starting with zero. Lines 3 to 7 describe the data structure used in the model. The global, shared `object` defined in line 8 instantiates the data type `CriticalClass`, which consists of two ordinary member variables, a `checksum`, and three synchronization variables: the `dirty` flag and thread `counter` as described in Section 6.5.3.1, and a `version` tag. The checksum exemplifies the redundancy introduced by GOP. Lines 10 to 12 define a macro[9] for the checksum computation. Likewise, the semantics of an atomic *compare-and-swap*[10] instruction is defined in lines 14 to 18: Only if the memory `location` (first

[7] Atomic incrementing and decrementing can be implemented by the *fetch-and-add* CPU instruction that is provided by the x86, x86_64, and ARMv8.1 instruction-set architectures. If such an instruction is not available, the wait-free counter can be implemented using multiple memory locations as described by Raynal [198, p. 190ff].

[8] For example, the address of a thread's current stack frame sufficiently identifies a thread.

[9] Promela does not support callable functions; reusable code fragments must be specified as `inline` macros.

[10] If the instruction-set architecture does not support the *compare-and-swap* CPU instruction, a pair of *load-linked* and *store-conditional* instructions can be used [107, p. 480f].

```
#define thread_ID() (_pid+1) /* thread-unique value {1...N} */

typedef CriticalClass {
  int member1 = 5, member2 = 2; /* ordinary members */
  int checksum = 7; /* introduced by the aspect */
  int dirty = 0, counter = 0, version = 0 /* wait-free sync */
}
CriticalClass object; /* global shared object */

inline compute_checksum(obj, chksum) {
  chksum = obj.member1; /* non-atomic computation */
  chksum = chksum + obj.member2 }

inline compare_and_swap(location, oldval, newval) {
  d_step { /* one single indivisible statement */
    if
    :: (location == oldval) -> location = newval
    :: else fi } }

inline enter(obj) {
  if
  :: (obj.counter == 0) -> /* object not in use */
    int version = obj.version; /* remember version */
    int checksum_tmp;
    compute_checksum(obj, checksum_tmp)
    if
    :: (checksum_tmp != obj.checksum) -> /* bit error */
      if
      :: (obj.dirty == 0) -> /* check for race condition */
         assert(version != obj.version) /* false positive */
      :: else fi
    :: else fi
  :: else fi;
  obj.counter = obj.counter + 1; /* atomic fetch-and-add */
  obj.dirty = thread_ID() }

inline leave(obj) {
  obj.dirty = thread_ID();
  /* hardware memory barrier (MFENCE) needed for TSO */
  if
  :: (obj.counter == 1) -> /* the last thread leaving */
     compute_checksum(obj, obj.checksum) /* update checksum */
     obj.version = obj.version + 1;
     compare_and_swap(obj.dirty, thread_ID(), 0) /* atomic */
  :: else fi;
  obj.counter = obj.counter - 1 /* atomic fetch-and-add */ }

active[4] proctype threads() { /* start 4 threads */
  enter(object);
  object.member1 = object.member1 + thread_ID()*3; /* modify */
  object.member2 = object.member2 - thread_ID();   /*  ...   */
  leave(object) }

/* global invariant specified in linear temporal logic */
ltl { always ((object.dirty != 0) ||
          (object.checksum == object.member1 + object.member2)) }
```

Figure 6.14: Executable abstract model of the wait-free synchronization algorithm, specified in Promela. Correctness properties are printed on highlighted background.

argument) contains the value oldval, then newval is written to that memory location. In Promela, an arrow (->) denotes the *then* condition of a preceding if statement. The value comparison and potential exchange are implemented indivisibly.

Pairwise execution of enter and leave

The wait-free synchronization algorithm is split into the two procedures enter and leave, which are executed pairwise: enter is invoked *before* a thread uses an object, and leave is executed *after* the object usage. The procedure enter works as follows: First, we check whether the object is already being used by testing the thread counter. Only if unused, we proceed with lines 23 to 32, which copy the object's version tag to a local memory location (line 23), and compute the object's checksum (lines 24 to 25). If the checksum mismatches, we first check for a *lost* race condition to avoid false positives, indicated by a nonzero dirty flag (line 29) or by a differing version tag (line 30). Otherwise, there would be a hardware memory error, which is not part of the abstract model. Finally, as we either have successfully verified the object or skipped the check due to concurrent modification, we increment the object's thread counter atomically and store the thread-unique value in the dirty flag (lines 34 to 35). From that point in time, the object is marked as *in use*: Further checksum verifications are skipped. Since the counter and dirty variables are *not* written *before* the checksum verification step has been completed, concurrent attempts to verify the checksum proceed until the fastest thread has succeeded. This procedure guarantees that there is always one thread that does not skip the check.

The procedure enter

After object usage, the procedure leave is executed, which overwrites the object's dirty flag at first (line 38). If the current thread is the only thread using the object, indicated by a counter of one, the object's checksum gets updated (line 42). Further on, the version tag is incremented, and we try to reset the dirty flag to zero by the atomic compare_and_swap instruction (line 44). This succeeds only if the dirty flag still contains the thread-unique value stored in line 38, meaning that there had not been any concurrent modification of the object. Otherwise, the compare_and_swap fails and the dirty flag remains nonzero, because the object is used concurrently by another thread. Finally, the thread counter is atomically decremented (line 46).

The procedure leave

The role of the version tag becomes evident once the dirty flag is reset to zero (line 44). Consider a thread that is preempted while verifying the object's checksum (line 25). In the meantime, other threads could update the object and reset the dirty flag. When the suspended thread is resumed, the pending checksum verification fails certainly, as old data, originating from before the preemption, go into the checksum computation. This is an instance of the *ABA* problem [107, p. 223], which occurs when a shared variable switches unnoticedly from state *A* to state *B*, and back to *A* again. In our case, *A* denotes an *unused object* and *B* means the opposite. The common solution is a version tag, incremented on each state transition. A sufficiently large integer variable will not wrap around during the time a thread is preempted.

Avoiding the ABA problem

Even if the same object was modified a billion times a second, a 64-bit version tag would overflow after 585 years of preemption.

The remaining lines 48 to 56 of Figure 6.14 are needed for formal verification with the *SPIN* [111] model checker. Four threads are started (lines 48 to 52) that concurrently invoke the procedure `enter`, modify the shared object, invoke the procedure `leave`, and exit afterwards. SPIN evaluates all possible execution sequences; multiple invocations of the procedures per thread are covered by a specific, non-interleaving execution of different threads. The primary verification property is specified in *linear temporal logic* (`ltl`), claiming that – `always` – the object's `checksum` is valid or the `dirty` flag is nonzero. Line 30 asserts that there are no false positives caused by race conditions.

Model checking

6.5.3.3 *Correctness Proof*

A limitation of model checking is that only *finite* models can be verified, as the model checker exhaustively analyzes the model's state space. Therefore, this section gives a proof by complete induction that holds for any number of threads. The *induction basis* is already proven by model checking for 1 to 4 threads. For simplicity, we assume the thread-unique IDs to be defined as $\{1, \ldots, N\}$ for N threads and an unbounded `version` tag. In the following *inductive step*, N+1 concurrent threads execute the procedures `enter` and `leave` as in Figure 6.14.

THEOREM 1: If a thread with ID X is verifying the checksum, then every checksum mismatch caused by a race condition (false positive) is detected by the `assert` statement in line 30.

Proof. Assume not. A race condition is not detected by the assert statement only in the following state: $dirty = 0$ and $version = version_X$ where $version_X$ denotes the value of `obj.version` at the time when thread X reads it for the first time (line 23). For N+1 threads that run to completion, the value of $version_X$ cannot exceed N, because thread X has not finished, yet. We differentiate between two cases:
1) $version_X \in \{1, \ldots, N\} \Rightarrow$ At least one thread has already incremented the `version` variable before thread X reads it for the first time. Such threads cannot modify the object's data members and checksum anymore, so that at most N out of N+1 threads can contribute to a race condition. Applying the *induction hypothesis*, we know that every race condition is detected for up to N threads.
2) $version_X = 0 \Rightarrow$ The initial value of the `version` variable indicates that no thread has updated the object's checksum, yet. Before the object could be modified by another thread, the `dirty` flag must be overwritten (line 35), yielding a nonzero value. The only way to reset the `dirty` variable to zero is a prior increment of the `version` variable (lines 43 to 44). Hence, a modifying race condition causes a nonzero `dirty` flag or nonzero `version` variable, contradicting the proof assumption. □

THEOREM 2: If a thread with ID X updates the object's checksum and resets the dirty variable to zero (lines 42 to 44), then the checksum is valid. (This is equivalent to the ltl claim in lines 54 to 56).

Proof. The `dirty` variable can only be reset if `obj.dirty=X` holds when thread X executes line 44 (`compare_and_swap`). That line is only reached if thread X had exclusive access to the object at a previous point in time (when evaluating line 41). In between, any other thread that modifies the object overwrites the `dirty` variable with its own thread ID unequal to X (line 35). Thus, when the `compare_and_swap` succeeds, thread X had exclusive access to the object while computing the checksum (line 42). Hence, the checksum is valid. □

6.5.3.4 *Relaxed Memory Consistency*

The previous sections implicitly assumed *sequentially consistent* shared memory for verifying the formal model. Sequential consistency requires that all memory accesses of one processor are instantly visible to the other processors, and that the memory accesses are ordered with respect to the executed programs [3]. However, most contemporary multiprocessors implement *relaxed* consistency guarantees for performance optimization [3], which allows the hardware to reorder memory accesses. For instance, a store operation could be delayed to hide memory latency. Unfortunately, such a reordering breaks the wait-free synchronization algorithm. Consider swapping lines 34 and 35 in Figure 6.14 – the correctness properties would be violated. Thus, every access to the synchronization variables (`dirty`, `counter`, and `version`) must appear strictly in the specified order.

Total Store Ordering

The predominant consistency relaxation, implemented by the x86 and SPARC architectures, is *Total Store Ordering (TSO)* [217]. The only reordering allowed in TSO concerns store instructions, which can be delayed *after* a subsequent load instruction, given that the load instruction accesses a different memory location. Other instruction pairs, such as two store operations, are never reordered. TSO is attributed to per-processor store buffers, which cache recent memory writes until they are committed to memory. The store buffer is flushed implicitly by an *atomic* CPU instruction or explicitly by an `MFENCE` instruction on x86, enforcing all pending memory operations to complete. Considering the formal model in Figure 6.14, there is only one store–load instruction pair that accesses the synchronization variables (line 38 and 41). This instruction pair must be serialized explicitly by an `MFENCE` CPU instruction for TSO architectures (line 39).

Hardware memory barriers

Another source of memory-access reordering is an optimizing compiler. Therefore, additional compiler memory barriers[11] are required around every access to the synchronization variables to prevent instruction reordering at compile time. Furthermore, this disables the caching of values in CPU registers, which would otherwise have an effect like store buffers.

6.5.3.5 *Fault-tolerant Synchronization*

Recalling the initial motivation for wait-free synchronization to enable concurrent detection of memory errors, the algorithm itself needs to be resilient against memory errors as well. If memory errors corrupt the three synchronization variables `dirty`, `counter`, and `version` (see Section 6.5.3.2), the running program must not fail. The two variables `dirty` and `version` get overwritten regularly and are solely used to skip or invalidate the `check()` and `update()` operations of the protected object. Bit errors affecting `dirty` and `version` are harmless and can be safely ignored.

On the other hand, a corrupted value in the `counter` variable could cause an undesired `update()` operation while the object is being used by another thread. Furthermore, as the `counter` is only incremented and decremented, bit errors are never overwritten. Therefore, an *arithmetic code*, or rather *AN code* (see Section 2.4.1.2), protects the `counter` variable: Instead of incrementing the `counter` by one, a large[12] odd constant value A is added. Thus, a valid `counter` always contains a multiple of A. Since bit errors turn likely that value into a non-multiple of A, most errors can be detected, such as all single bit flips [37]. Likewise, decrementing the `counter` variable is implemented by subtracting A.

Arithmetic encoding of the variable `counter`

To reduce the complexity of the abstract model in Figure 6.14, only error detection is addressed. If we additionally consider error correction, it becomes evident that a short critical section, guarding the error-correcting instructions, is unavoidable: If an object is going to be corrected, it must not be modified concurrently by other threads. Hence, for error correction, the synchronization algorithm remains wait-free with the exception that only in the event of a hardware error, a lock is used until the error is resolved. Such a lock cannot introduce a deadlock, because the second *Coffman condition* (hold and *wait for* resources) [56] is not satisfied: The error-correcting instructions do not need to wait for additional resources but run to completion.

[11] For example, the GCC implements a compiler memory barrier by the following inline assembler statement: `asm volatile("":::"memory");`

[12] In this thesis, I choose A to be 127, as suggested by Chang and associates [45].

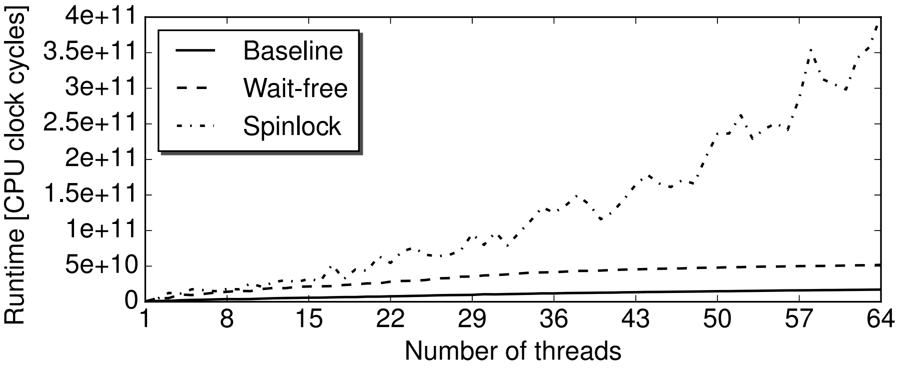


Figure 6.15: Slowdown caused by synchronization: The spinlock becomes a bottleneck (less runtime is better).

6.5.3.6 *Scalability to Many Cores*

This section evaluates how the wait-free synchronization performs compared to a locking-based approach. The locking-based solution uses a per-object spinlock[13] during the `check()` and `update()` operations. I implemented a micro benchmark that allocates one shared object containing an array of 16 integers (64 bytes). A thread first computes the sum of the integer array, and then stores that sum to each array element. This procedure is repeated one million times per thread, while the CRC-32 variant of GOP covers the shared object (see Section 6.5.1). The micro benchmark runs on a 32-core Intel Xeon E5-4650 system, supporting 64 hardware threads by hyper-threading.

Figure 6.15 shows the slowdown caused by synchronization for 1 to 64 threads, operating concurrently on the shared object. The *Baseline* curve denotes the runtime *without* any error detection, increasing slowly with the number of threads due to memory contention. The *Wait-free* curve shows a similar pattern despite the overhead caused by frequent CRC computations. In contrast, the *Spinlock* variant slows down dramatically for more than 16 concurrent threads. For 64 threads, the runtime differs by almost an order of magnitude between the wait-free and spinlock variants. Even for a single thread, the wait-free implementation is slightly faster. This benchmark quantifies the advantage of the wait-free synchronization algorithm, which scales much better as the number of threads increases. This feature could be essential for future many-core systems with possibly hundreds of processors.

[13] The spinlock variant uses `boost::detail::spinlock` (`BOOST_SP_HAS_SYNC` variant) from the Boost C++ libraries to avoid further bus contention by giving up a thread's remaining time slice if acquiring fails too often.

6.6 CHAPTER SUMMARY

This chapter presented a software library of reusable *dependability aspects*. These dependability aspects are highly generic software modules that use the AspectC++ 2.0 technology to implement transparent fault-tolerance mechanisms. In summary, I developed the following dependability aspects in the course of this thesis:

- Symptom detectors, such as checking of integer overflows
- Return-Address Protection (RAP)
- Virtual-Function Pointer Protection (VPP)
- Generic Object Protection (GOP)

All dependability aspects provide a regular and coherent programming interface that enables a selective placement and user-defined error handling.

Each aspect, except the symptom detectors, supports detection *and* correction of memory errors. Furthermore, each aspect operates in a wait-free manner when applied to multi-threaded programs. Therefore, the presented library advances the state-of-the-art in the domain of software-implement fault tolerance (see Section 2.4.3).

Finally, the next chapter evaluates the effectiveness and efficiency of these dependability aspects when applied to several systems, such as eCos and the L4/Fiasco.OC microkernel.

7 EVALUATION

> “A little redundancy, thoughtfully deployed and exploited, can yield significant benefits for fault tolerance; however, excessive or inappropriately applied redundancy is pointless.”
>
> – David Taylor and associates [244, p. 586]

The goal of this chapter is to evaluate the library of reusable dependability aspects that has been presented in the previous chapter. To this end, this chapter comprises three case studies that evaluate the individual dependability aspects when applied to large-scale software systems. Section 7.1 considers the real-time operating system eCos. Subsequently, Section 7.2 covers the L4/Fiasco.OC microkernel. In addition to these operating systems, Section 7.3 presents quantitative results of applying the dependability aspects to the user-level program *Memcached* [84].

First and foremost, this chapter evaluates the fault tolerance of the three software systems with and without dependability aspects using the methodology of fault injection as described in Chapter 3. The fundamental reliability metric for comparing program susceptibility to transient hardware faults is the *total probability of failure,* which is directly proportional to the number of failed program runs obtained by independent fault-injection experiments (see Equation 3.4 on page 43). That metric takes into account that a slower program is vulnerable to transient hardware faults for a longer period of time. Thus, a dependability aspect is effective only if it compensates for the incurred runtime overhead.

Second, each of the following case studies assesses the efficiency of the individual dependability aspects in terms of runtime overhead and memory footprint. Finally, Section 7.4 summarizes the resulting trade-off between fault tolerance and efficiency.

RELATED PUBLICATIONS

The findings presented in this chapter have partly been published in:

[160] Arthur Martens, Christoph Borchert, Manuel Nieke, Olaf Spinczyk, and Rüdiger Kapitza. CrossCheck: A holistic approach for tolerating crash-faults and arbitrary failures. In *Proceedings of the 12th European Dependable Computing Conference (EDCC '16).* IEEE Press, September 2016. doi: 10.1109/EDCC.2016.29

Acceptance rate: 46 %

7.1 CASE STUDY: HARDENING ECOS

The first case study applies the dependability aspects to the *embedded configurable operating system (eCos)*, which represents an off-the-shelf operating system for real-time applications (see Section 3.2 on page 46 for details on eCos). In short, eCos is designed as a static library, which is linked with the applications to produce an executable image. Both the kernel and the applications run in the privileged processor mode.

eCos 3.0 is bundled with a kernel test suite, which contains various benchmark applications that exercise the kernel's functionalities. This case study uses these benchmark programs to generate load on the eCos kernel during the evaluation. Table 7.2 briefly describes the 17 benchmarks programs, which run up to 15 threads per benchmark. In addition to these artificial kernel tests, this case study uses the flight-control application of the I4COPTER quadrotor helicopter (see Section 4.2.2 on page 68), which represents a real-world safety-critical application. The I4COPTER application runs 14 threads that interact in a strict execution order.

Initially, this section evaluates the dependability aspects one by one: symptom detection in Section 7.1.1, Return-Address Protection in Section 7.1.2, and Generic Object Protection in Section 7.1.3. Virtual-Function Pointer Protection is irrelevant because eCos does not use any virtual functions. Afterwards, Section 7.1.4 analyzes these dependability aspects in combination.

Experimental setup

I use the same experimental setup of the fault-injection framework FAIL* as described in Section 3.2.1 on page 47. In brief, the setup includes an x86-hardware emulator and the fault model of independent single bit flips in the whole data memory (RAM). The only variation is that I disable both serial and VGA output of eCos, as the kernel benchmarks report on success or failure before finishing, and such a time-consuming output would completely mask any protection's runtime overhead. As an alternative, the x86 emulator records invocations of the eCos function `cyg_test_output(int, ...)`, which indicates success or failure by its first argument.

Altogether, I use *exhaustive* fault-space scans of eCos running all 18 benchmarks except KILL. That particular benchmark executes about 160 million CPU instructions, which take a very long time to emulate. Therefore, the fault tolerance of eCos running the benchmark KILL is estimated by a sample of 20,000 fault injections, which yields an estimate with an overall maximum relative standard error of 3.7 percent. In total, this section presents the results of 857 million fault-injection experiments after applying FAIL*'s trace-based fault-space optimization (see Section 3.1.3). These experiments consumed 31 CPU years in the computing cluster LiDOng at TU Dortmund.

benchmark	description of the testing domain	threads
bin_sem1	Binary semaphore functionality	2
bin_sem2	Dining philosophers	15
bin_sem3	Binary semaphore timeout functionality	2
cnt_sem1	Counting semaphore functionality	2
except1	Exception handling	1
flag1	Flag functionality	3
kill	Thread `kill()` and `reinitalize()`	3
mbox1	Message box functionality	2
mqueue1	Message queue functionality	2
mutex1	Mutex functionality (see Section 3.2.2.2)	3
mutex2	Mutex `release()` functionality	4
release	Release threads from waiting	2
sched1	Scheduler functionality	2
sync2	Different locking mechanisms	4
sync3	Priority inheritance	3
thread0	Thread constructors and destructors	1
thread1	Interleaved thread execution (see Section 3.2.2.1)	2

Table 7.2: Benchmark programs of the eCos kernel test suite. These benchmarks are part of the official eCos repository and implement functional tests of the kernel (second column). The third column shows the number of threads that are started by each benchmark.

7.1.1 *Symptom Detection*

First, this section evaluates the four aspect-oriented symptom detectors that have been presented in the previous chapters. The following abbreviations denote each of them:

function: Range checking of function pointers (see Section 5.1.2)

array: Checking of array bounds (see Section 5.2.3)

type: Run-time type checking (see Section 5.3.2)

overflow: Checking of integer overflows (see Section 6.2.2)

Each symptom detector is applied to eCos as broadly as possible – that is, to *all* function pointers, to *all* arrays, and so on. A *baseline* variant of eCos, which runs without any protection, serves as a reference for the following comparisons.

7.1.1.1 *Effectiveness: Error Detection*

Figure 7.1a exemplarily presents the fault-injection results of exhaustive fault-space scans of eCos running the benchmarks bin_sem1 and I4COPTER. The diagrams show the accumulated number of failed program runs on the y-axis and the respective variants of eCos on the x-axis. Colors differentiate between the failure modes SDC, timeout, and CPU exception (see Section 3.2.1 on page 47).

When running the benchmark BIN_SEM1, the FUNCTION variant of eCos detects 11.7 percent of the faults that led to a failure in the baseline variant; the TYPE variant detects 29.4 percent. However, the number of failures increases by 11.4 percent in the OVERFLOW variant, and even by 18.1 percent in the ARRAY variant. Thus, only the FUNCTION and TYPE variants are effective when running BIN_SEM1, whereas the other two symptom detectors are counterproductive.

On the other hand, when running the benchmark I4COPTER, each symptom detector is effective: The FUNCTION variant of eCos detects 10.3 percent, the TYPE variant 49.0 percent, the OVERFLOW variant 1.9 percent, and the ARRAY variant 1.4 percent of the faults that led to a failure in the baseline variant.

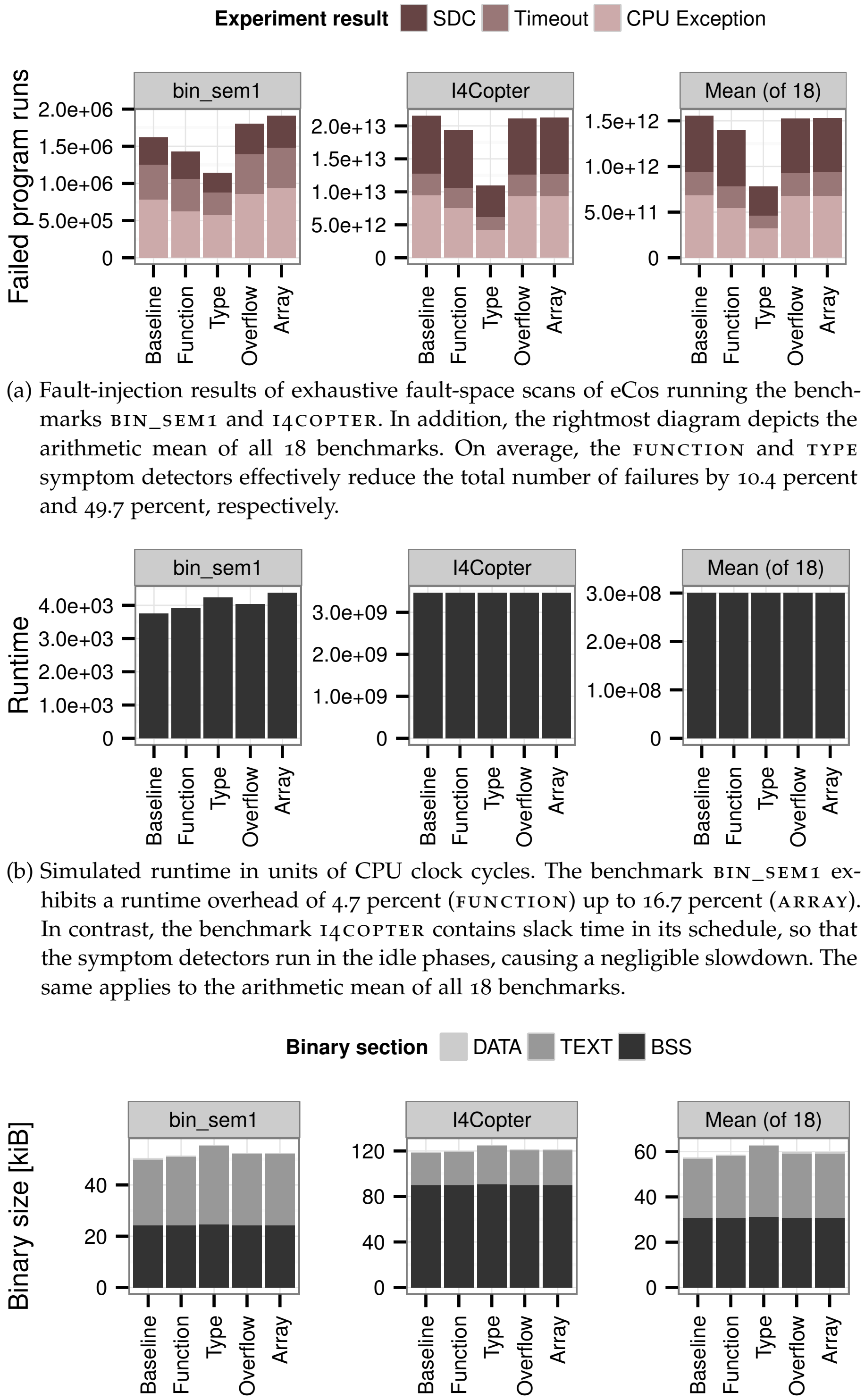
Figure B.1 in Appendix B on page 190 shows the fault-injection results of all 18 benchmarks. In brief, the individual results are similar to those of BIN_SEM1 or I4COPTER. Thus, and for the sake of clarity, the rightmost diagram of Figure 7.1a summarizes the results by presenting the arithmetic mean of all 18 benchmarks. On average, the FUNCTION and TYPE variants are effective in detecting 10.4 percent and 49.7 percent, respectively, of *all* faults that led to a failure in the baseline variant. OVERFLOW and ARRAY detect less than 2 percent each.

In summary, the effectiveness of the symptom detectors OVERFLOW and ARRAY depends on the workload of the operating system. Range checking of function pointers (FUNCTION), however, is quite effective in most cases. Finally, run-time type checking (TYPE) is extraordinarily effective regardless of the benchmark program.

7.1.1.2 *Efficiency: Runtime*

Figure 7.1b shows the simulated runtime in units of CPU clock cycles of the benchmarks BIN_SEM1 and I4COPTER running on the different eCos variants. BIN_SEM1 runs for only 3,750 clock cycles in the baseline variant. The FUNCTION variant of eCos increases the runtime by 4.7 percent, and the TYPE variant by 13.1 percent. Likewise, the ineffective variants OVERFLOW and ARRAY cause a slowdown of 7.6 percent and 16.7 percent, respectively. That slowdown explains their ineffectiveness, because the slowdown notably increases the *attack surface* – or rather, fault space – but both variants detect only a few faults. Thus, the number of failures increases to a similar extent (see Figure 7.1a).

The I4COPTER benchmark exhibits a different behavior, as the slowdown caused by all four symptom detectors is way below 0.1 percent. The reason is that the I4COPTER benchmark contains slack time in



(a) Fault-injection results of exhaustive fault-space scans of eCos running the benchmarks BIN_SEM1 and I4COPTER. In addition, the rightmost diagram depicts the arithmetic mean of all 18 benchmarks. On average, the FUNCTION and TYPE symptom detectors effectively reduce the total number of failures by 10.4 percent and 49.7 percent, respectively.

(b) Simulated runtime in units of CPU clock cycles. The benchmark BIN_SEM1 exhibits a runtime overhead of 4.7 percent (FUNCTION) up to 16.7 percent (ARRAY). In contrast, the benchmark I4COPTER contains slack time in its schedule, so that the symptom detectors run in the idle phases, causing a negligible slowdown. The same applies to the arithmetic mean of all 18 benchmarks.

(c) Memory size of eCos linked with the benchmark applications. On average, the read-only text section grows by 1.2 KiB (FUNCTION) up to 5.2 KiB (TYPE). Only TYPE consumes RAM and increases the data and BSS sections by an average of 627 bytes.

Figure 7.1: Quantitative evaluation of the four symptom detectors applied to eCos based on three metrics: (a) fault tolerance, (b) runtime, and (c) memory footprint. Appendix B on pages 190 to 192 shows the individual results of all 18 benchmarks.

its schedule, so that the symptom detectors run in the idle phases. After three hyper periods of the main control loop, the real-world application I4COPTER is stopped (see Section 4.2.2 on page 68).

Figure B.2 in Appendix B on page 191 shows the runtime of all 18 benchmarks, yet, the individual results are similar to those of BIN_SEM1 or I4COPTER. Therefore, the rightmost diagram of Figure 7.1b presents the arithmetic mean of all benchmarks. In total, the slowdown of all symptom detectors is below 0.1 percent.

7.1.1.3 *Memory Footprint: Static Binary Size*

Figure 7.1c depicts the static binary size of eCos linked with the respective benchmark applications. Once again, the rightmost diagram shows the arithmetic mean of all executable images files (Figure B.3 on page 192 presents the individual sizes). Each symptom detector increases the text section of read-only memory: On average, FUNCTION adds 1.2 KiB of program instructions (+ 4.7 percent), TYPE requires 5.2 KiB (+ 19.7 percent), and OVERFLOW and ARRAY add 2.2 KiB (+ 8.7 percent) each.

Only the symptom detector TYPE increases the data and BSS sections by an average of 627 bytes (+ 2.0 percent). The other symptom detectors are *stateless* and do not require any amount of RAM.

7.1.2 *Return-Address Protection*

This section turns towards the Return-Address Protection (RAP) dependability aspect (see Section 6.3 on page 105) now. In short, RAP extends a function's stack frame by a checksum of both the return address and frame pointer to enable error detection. In addition, error correction is implemented by optional copies of the return address and frame pointer. Moreover, the dependability aspect provides a whole-program optimization heuristic that excludes short-running functions from protection (see Section 6.3.2)

Therefore, this section considers three different configurations of the RAP aspect applied to eCos:

RAP DET.: Only error *detection*, excluding short-running functions

RAP COR.: Error *correction*, excluding short-running functions

RAP D/ALL: Only error detection, protecting *all* non-inline functions

These configurations are compared to a *baseline* variant in the following evaluation. Figure 7.2 summarizes the results of all 18 benchmarks by taking three examples and presenting the arithmetic mean of all results. Appendix B on pages 193 to 195 shows the individual results.

7.1.2.1 *Effectiveness: Error Detection and Correction*

Figure 7.2a exemplarily shows the fault-injection results of exhaustive fault-space scans of eCos running the benchmarks BIN_SEM1, EXCEPT1, and I4COPTER. The diagrams show the accumulated number of failed program runs on the y-axis, differentiated between SDC, timeout, and CPU exception.

No matter which RAP variant is applied to eCos, the benchmark BIN_SEM1 exhibits more failures than the baseline variant. This is because the accompanying runtime overhead increases the attack surface of the other, unprotected eCos components. We shall see in Section 7.1.4 that, once these other components are protected as well, the benchmark BIN_SEM1 indeed benefits from RAP. For now, however, BIN_SEM1 performs best on the baseline variant of eCos.

When running the benchmark EXCEPT1, the detection variant of RAP catches 21.9 percent of the faults that led to a failure in the baseline variant, whereas the correction variant avoids 19.4 percent. On the contrary, the unoptimized D/ALL variant is counterproductive to fault tolerance and increases the number of failures by 29.8 percent.

Ineffectiveness of RAP without whole-program optimization

The fault-injection results of I4COPTER are similar to the arithmetic mean of all 18 benchmarks, which is presented in the rightmost diagram of Figure 7.2a. On average, the detection and correction variants achieve a failure reduction of 3.3 percent and 2.9 percent, respectively. The D/ALL variant, which is applied to *all* non-inline functions, even reduces the failure count by 7.2 percent on average.

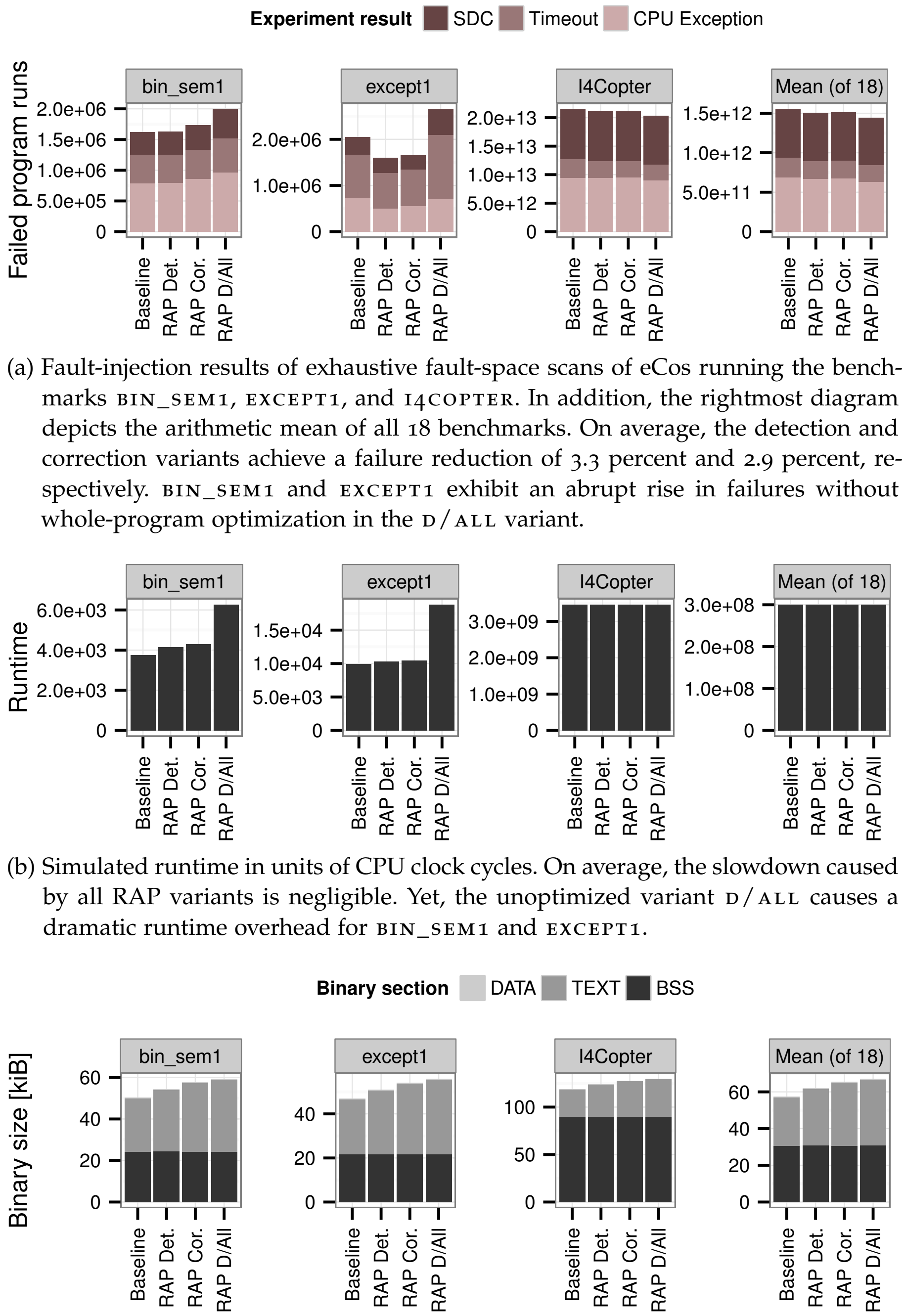
In summary, the RAP dependability aspect is generally effective if the whole-program optimization heuristic that excludes short-running functions is enabled. The protection of all non-inline functions, as implemented by the D/ALL variant, achieves a better average but suffers from pathological cases that exhibit an abrupt rise in failures.

7.1.2.2 *Efficiency: Runtime*

Figure 7.2b depicts the simulated runtime in units of CPU clock cycles of the respective benchmark applications. The detection variant of RAP prolongs the runtime of BIN_SEM1 by 10.5 percent, compared to 14.6 percent that incur in the correction variant. That difference in efficiency explains the higher failure counts of the correction variant: Both variants protect the same data, but error correction causes more runtime overhead. Likewise, the ineffectiveness of the D/ALL variant results from its excessive runtime overhead of 67.3 percent.

EXCEPT1 is most robust in the detection and correction variants, which cause a runtime overhead of 4.2 percent and 5.5 percent, respectively. However, the D/ALL variant causes 89.9 percent runtime overhead, which explains its poor fault tolerance.

The runtime of the real-world application I4COPTER exceeds the runtime of the other two benchmarks by five orders of magnitude.



(a) Fault-injection results of exhaustive fault-space scans of eCos running the benchmarks BIN_SEM1, EXCEPT1, and I4COPTER. In addition, the rightmost diagram depicts the arithmetic mean of all 18 benchmarks. On average, the detection and correction variants achieve a failure reduction of 3.3 percent and 2.9 percent, respectively. BIN_SEM1 and EXCEPT1 exhibit an abrupt rise in failures without whole-program optimization in the D/ALL variant.

(b) Simulated runtime in units of CPU clock cycles. On average, the slowdown caused by all RAP variants is negligible. Yet, the unoptimized variant D/ALL causes a dramatic runtime overhead for BIN_SEM1 and EXCEPT1.

(c) Memory size of eCos linked with the benchmark applications. On average, the read-only text section grows by 4.7 KiB (error detection) and 8.2 KiB (error correction). Without whole-program optimization (D/ALL), error detection requires even 9.8 KiB.

Figure 7.2: Quantitative evaluation of RAP applied to eCos based on three metrics: (a) fault tolerance, (b) runtime, and (c) memory footprint. Appendix B on pages 193 to 195 shows the individual results of all 18 benchmarks.

As mentioned before, 14COPTER contains slack time in its schedule, so that the RAP aspect runs in the idle phases. Thus, the slowdown caused by all RAP variants stays below 0.1 percent. Likewise, the arithmetic mean of all 18 benchmarks shows a negligible slowdown, which is dominated by the long-lasting benchmarks. This may seem odd, but Smith [228] argues that the arithmetic mean is the only accurate measure for characterizing the performance of a benchmark suite.

7.1.2.3 *Memory Footprint: Static Binary Size*

Figure 7.2c illustrates the static binary size of eCos linked with the respective benchmark applications. The RAP variants considerably increase the text section of read-only memory. On average, error detection adds 4.7 KiB of program instructions (+ 18.5 percent); error correction requires 8.2 KiB (+ 31.3 percent). The unoptimized D/ALL variant even adds 9.8 KiB (+ 37.7 percent) but still provides only error detection. Thus, the whole-program optimization effectively reduces the memory overhead by 51 percent.

7.1.3 *Generic Object Protection*

This section proceeds with evaluating the Generic Object Protection (GOP) dependability aspect (see Section 6.5 on page 115). In brief, GOP inserts information redundancy into data structures to detect and correct memory errors transparently. Furthermore, GOP is highly configurable with respect to the set of data structures to be protected. First, this section explores the costs and gains of protecting a single kernel data structure versus protecting the whole eCos kernel. Section 7.1.3.1 discusses the resulting optimization problem before evaluating selected configurations in Section 7.1.3.2.

In this section, the overall GOP configuration includes static whole-program optimization (see Section 6.5.2.2) and wait-free synchronization (see Section 6.5.3). The latter is needed because some parts of the eCos kernel are lock-free, for example, the scheduler function `get_current_thread()` is executed without acquiring a kernel lock. In addition, GOP is configured without support for polymorphism (see Section 6.5.2.3), as the eCos kernel does not use any virtual functions.

7.1.3.1 *Optimizing the Generic Object Protection*

The eCos kernel contains about 20 C++ classes that implement the essential kernel data structures. Because GOP can be configured to protect any subset of these classes, there are 2^{20} possible GOP configurations (the power set). It is impractical to evaluate all of them, yet, there are certainly optimal configurations in terms of fault tolerance and runtime overhead.

To explore the resulting optimization problem, this section first considers a series of cumulative subsets that protect the kernel classes one by one: The series starts with the empty set, followed by a subset that contains only one class, to be specific, the class that causes the lowest individual runtime overhead when covered by GOP. The third subset includes two classes – those with the lowest and second-lowest runtime overhead. This series continues until it converges to the set of all classes.

Figure 7.3b exemplarily shows the simulated runtime of the benchmarks MBOX1, MUTEX2, SYNC3, and I4COPTER that run on eCos with 0 to 13 classes being protected by the CRC variant of GOP. Note that the maximum number of protected classes varies with each benchmark, as only those classes are shown that are actually used. The benchmarks can be divided into two categories:

1. *Long runtime (more than 10 million cycles)*. For any subset of kernel classes, the slowdown caused by protection is negligible (for example, MBOX1). The reason is that these benchmarks spend a significant amount of time in calculations on the application level or contain idle phases.

2. *Short runtime (less than 10 million cycles)*. The runtime overhead increases considerably with each additional kernel class being protected. These benchmarks primarily execute kernel code.

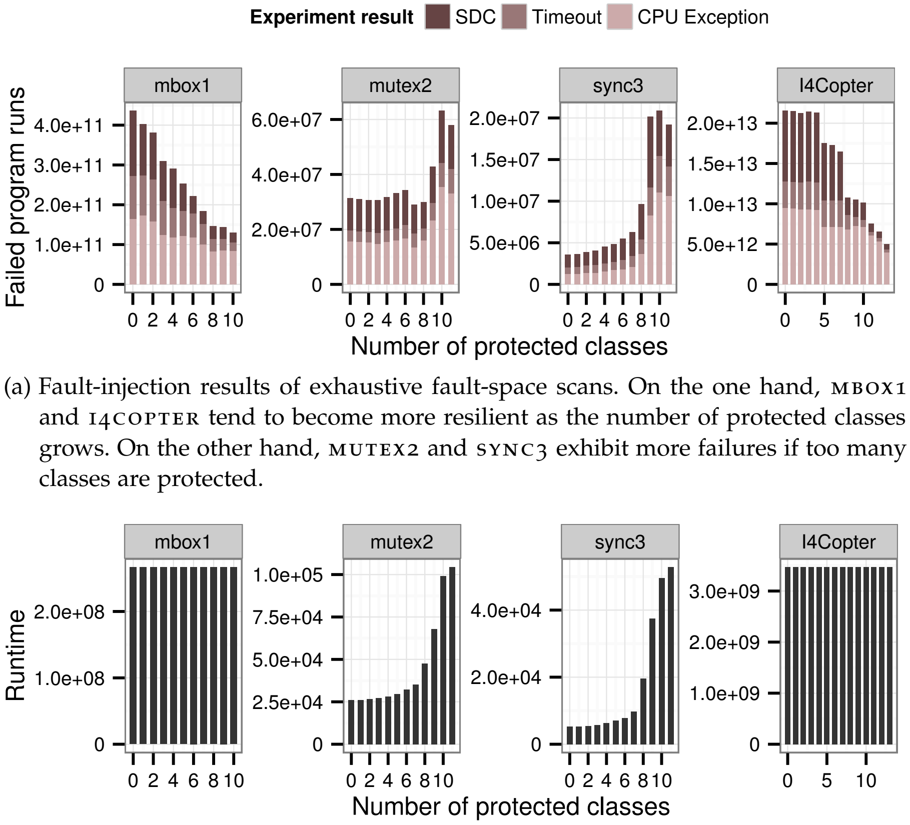
Figure B.8 in Appendix B on page 197 confirms these findings by presenting the simulated runtime of all 18 benchmarks.

In the next step, Figure 7.3a depicts the fault-injection results obtained by exhaustive fault-space scans of the same variants. The results indicate that the benchmark classification can be applied to the fault-injection results as well:

1. *Long-running* benchmarks, such as MBOX1 and I4COPTER, tend to become *more* resilient (lower failure count) with each additional class being protected. The accompanying slowdown stays below one percent in all cases. This calls for protection of all classes.

2. *Short-running* benchmarks, such as MUTEX2 and SYNC3, benefit from GOP if the runtime overhead stays below a certain threshold. For example, configurations that cause less than one percent runtime overhead typically exhibit lower failure counts. However, as shown by MUTEX2, such a configuration does not necessarily represent the minimal failure count.

Hence, for the considered set of benchmarks, the following heuristic yields a good trade-off between slowdown and fault tolerance:



(a) Fault-injection results of exhaustive fault-space scans. On the one hand, mbox1 and i4copter tend to become more resilient as the number of protected classes grows. On the other hand, mutex2 and sync3 exhibit more failures if too many classes are protected.

(b) Simulated runtime in units of CPU clock cycles. The runtime of mbox1 and i4copter stays almost constant as the number of protected classes grows, whereas mutex2 and sync3 exhibit a considerable runtime overhead for certain configurations.

Figure 7.3: Exploring the potential for optimization of GOP. Each benchmark runs on several eCos variants with *increasingly more* kernel data structures being protected by a CRC code. Appendix B on pages 196 and 197 presents the results for all 18 benchmarks.

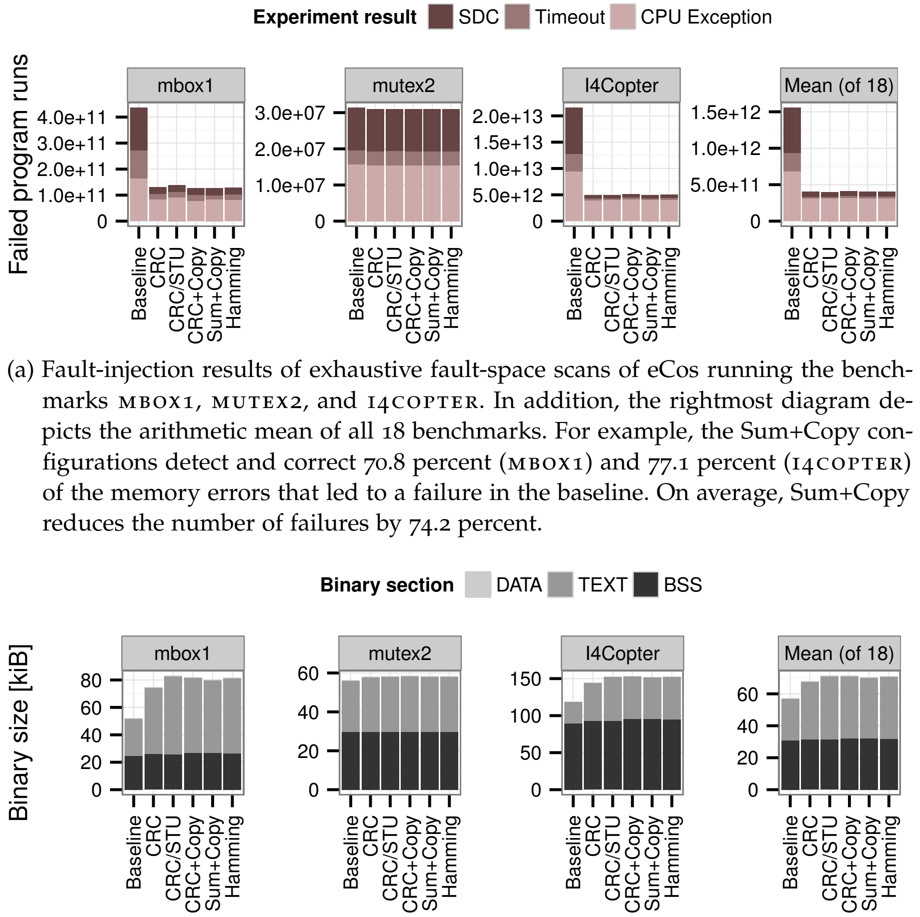
Rule of thumb

If the individual protection of a particular class results in less than one percent slowdown, then the class should be protected by GOP.

Using this rule of thumb can massively reduce the efforts spent on choosing a good configuration, because the average runtime is easily measurable without any time-consuming fault-injection experiments. Future work needs to address this multi-objective optimization problem in more detail, yet, this thesis carries on with the rule of thumb.

7.1.3.2 *Effectiveness and Memory Footprint*

The aforementioned rule of thumb (slowdown limit of one percent) produces application-specific configurations that selectively protect certain kernel data structures by GOP. This section evaluates the effectiveness of such configurations and compares the different GOP options for redundancy as described in Table 6.2 on page 119: CRC,



(a) Fault-injection results of exhaustive fault-space scans of eCos running the benchmarks MBOX1, MUTEX2, and I4COPTER. In addition, the rightmost diagram depicts the arithmetic mean of all 18 benchmarks. For example, the Sum+Copy configurations detect and correct 70.8 percent (MBOX1) and 77.1 percent (I4COPTER) of the memory errors that led to a failure in the baseline. On average, Sum+Copy reduces the number of failures by 74.2 percent.

(b) Memory size of eCos linked with the benchmark applications. For example, the Sum+Copy configurations add an average of 11.8 KiB (+45.1 percent) to the read-only text section; the data and BSS sections grow by 1,500 bytes (+4.7 percent).

Figure 7.4: Quantitative evaluation of GOP applied to eCos based on two metrics: (a) fault tolerance, and (b) memory footprint. GOP is tailored for each benchmark to keep the slowdown below one percent. Appendix B on pages 198 and 199 shows the individual results of all 18 benchmarks.

CRC+Copy, Sum+Copy, and Hamming. In addition, CRC/STU denotes the CRC option without whole-program optimization.

Figure 7.4a exemplarily shows the fault-injection results of exhaustive fault-space scans of eCos with application-specific GOP configurations. MBOX1 and I4COPTER exhibit significant improvements in fault tolerance, because their GOP configurations cover the whole eCos kernel. MUTEX2, however, uses a GOP configuration that covers only one class, resulting in a mere failure reduction of 1.3 percent for the CRC option.

GOP avoids 74 percent of the failures.

The rightmost diagram of Figure 7.4a shows the arithmetic mean of all 18 benchmarks (see Figure B.9 on page 198 for details). On average, the CRC+Copy, Sum+Copy, and Hamming options transparently detect and correct about 74 percent of the memory errors that led to a failure in the baseline. The CRC and CRC/STU options are similarly effective but provide only error detection. As shown in the leftmost diagram of Figure 7.4a, however, the benchmark MBOX1 exhibits 5.8 percent more failures for CRC/STU compared to CRC with whole-program optimization.

Figure 7.4b depicts the static binary size of eCos with application-specific GOP configurations. On the one hand, the configuration used by MUTEX2, which protects only a single class, adds about 2.0 KiB of program instructions to the read-only text section (+ 7.6 percent). On the other hand, the extensive CRC configuration used by MBOX1 enlarges the text section by 21.2 KiB (+ 78.0 percent). In this configuration, Sum+Copy adds 25.8 KiB (+ 94.7 percent), whereas CRC+Copy and Hamming require about 27.4 KiB (+ 100.7 percent).

ROM usage

The data and BSS sections increase by only 80 bytes for MUTEX2, but considerably grow with the number of protected classes in the other configurations. On average, CRC increases these memory sections by 923 bytes (+ 2.9 percent), CRC+Copy and Sum+Copy add 1,500 bytes (+ 4.7 percent), whereas the Hamming code requires only 1,307 bytes (+ 4.1 percent).

RAM usage

In summary, the GOP dependability aspect is highly effective in reducing the total number of failed program runs transparently by about 74 percent. By application-specific tailoring, the slowdown can be kept below one percent. eCos requires roughly twice the amount of read-only memory for protection of all kernel data structures, whereas the RAM sections grow by less than five percent on average.

7.1.4 *Combining the Dependability Aspects*

The previous evaluation of the individual dependability aspects allows drawing the following conclusions:

- Symptom detection in form of run-time type checking and range checking of function pointers is highly effective, whereas checking of array bounds and integer overflows is poorly suited for the detection of memory errors.
- Return-Address Protection with whole-program optimization is typically effective but protects only few memory locations.
- Generic Object Protection is extraordinarily effective when configured to meet a slowdown limit of one percent. However, short-running benchmarks do not improve because the slowdown limit rules out the protection of many kernel data structures.

Therefore, this section studies the dependability aspects in combination – that is, the three aspects are applied to eCos at the same time. Altogether, this section compares the following variants:

BASELINE: eCos without any protection serves as a reference.

GOP: Application-specific configurations of GOP to meet the slowdown limit of one percent, supporting error detection and error correction by the Sum+Copy option (same configurations and results as in Section 7.1.3.2).

GOP+S: GOP configuration as above, plus *symptom detection* in form of run-time type checking (not applied to classes already protected by GOP) and range checking of function pointers.

G+S+R: GOP and symptom detection as above, plus *Return-Address Protection* (error correction) with whole-program optimization.

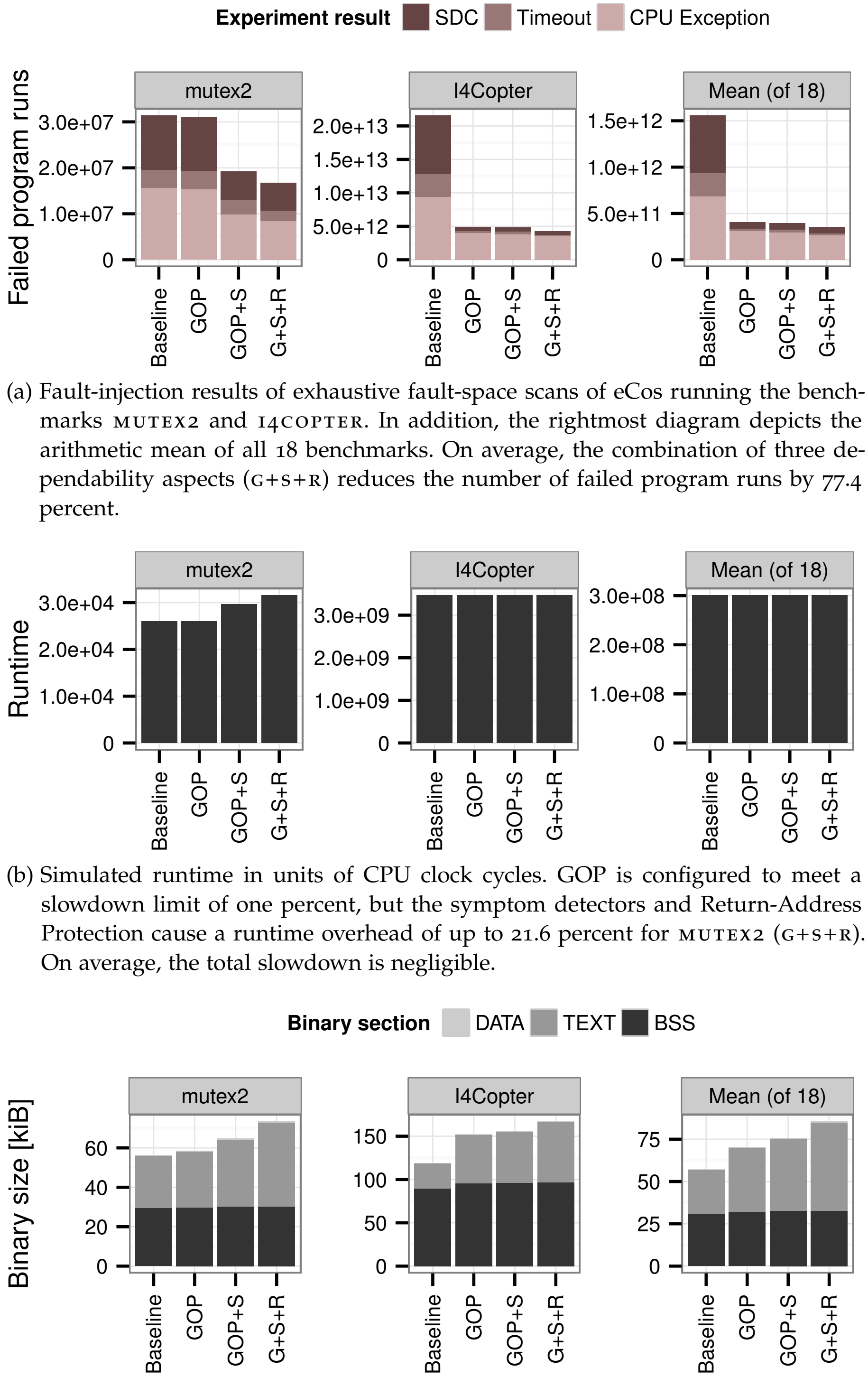
7.1.4.1 *Effectiveness: Error Detection and Correction*

Figure 7.5a exemplarily shows the fault-injection results of exhaustive fault-space scans of eCos running the benchmarks MUTEX2 and I4COPTER. The other 16 benchmarks resemble either of them in behavior, as shown in Figure B.11 on page 200 in Appendix B.

MUTEX2 represents the short-running benchmarks that cannot be protected effectively by GOP, which reduces the number of failures by only 1.2 percent in that case. Applying the two symptom detectors (GOP+S), however, reduces the number of failures by 38.8 percent. Furthermore, the combination of all dependability aspects (G+S+R) achieves the highest failure reduction of 46.5 percent. The latter improvement by 7.7 percent points is remarkable, considering that Return-Address Protection alone avoids only 2.9 percent (see Figure B.4 on page 193). The reason is that, once a large share of kernel data is protected, the runtime overhead of Return-Address Protection becomes less adverse. This phenomenon even goes as far as that the four benchmarks BIN_SEM1, CNT_SEM1, MUTEX1, and SYNC3 suddenly benefit from Return-Address Protection, whereas they initially did not (see Section 7.1.2).

I4COPTER, on the other hand, represents the long-running benchmarks that improve considerably by GOP and, thus, exhibits 77.1 percent fewer failures. In that case, the two symptom detectors (GOP+S) further improve the fault tolerance by only 0.5 percent points. Finally, adding Return-Address Protection (G+S+R) reduces the number of failures by a total of 80.1 percent.

In conclusion, the combination of the three dependability aspects achieves the highest failure reduction in both cases. Furthermore, Figure B.11 on page 200 confirms these findings by showing that the combination G+S+R significantly improves the fault tolerance of all 18 benchmarks over the baseline. On average, GOP alone reduces the failures by 74.2 percent, GOP+S by 74.7 percent, and G+S+R by 77.4 percent.



(a) Fault-injection results of exhaustive fault-space scans of eCos running the benchmarks mutex2 and i4copter. In addition, the rightmost diagram depicts the arithmetic mean of all 18 benchmarks. On average, the combination of three dependability aspects (g+s+r) reduces the number of failed program runs by 77.4 percent.

(b) Simulated runtime in units of CPU clock cycles. GOP is configured to meet a slowdown limit of one percent, but the symptom detectors and Return-Address Protection cause a runtime overhead of up to 21.6 percent for mutex2 (g+s+r). On average, the total slowdown is negligible.

(c) Memory size of eCos linked with the benchmark applications. On average, the combination of three dependability aspects (g+s+r) requires twice the amount of ROM compared to the baseline. The data and BSS sections, however, increase by an average of only 2.0 KiB (+ 6.6 percent).

Figure 7.5: Quantitative evaluation of the combined dependability aspects applied to eCos based on three metrics: (a) fault tolerance, (b) runtime, and (c) memory footprint. The abbreviation gop+s means GOP plus symptom detection, whereas g+s+r denotes a combination of GOP, symptom detection, and RAP. Appendix B on pages 200 to 202 shows the individual results of all 18 benchmarks.

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
stack	cyg_uint64[]	5,088	$2.61 \cdot 10^9$	56.9 %
thread[0].stack_ptr	cyg_uint32*	4	$8.53 \cdot 10^8$	18.6 %
hal_interrupt_handlers	cyg_uint32 (*[])()	896	$7.46 \cdot 10^8$	16.3 %
comm_channels	int (*[][])()	96	$3.47 \cdot 10^8$	7.6 %
thread (w/o .stack_ptr)	Cyg_Thread[]	260	$2.80 \cdot 10^7$	0.6 %

Table 7.3: Remaining failures of eCos, protected by all dependability aspects (G+S+R), running the THREAD1 benchmark. The table shows the five most failure-prone symbols (continuous memory regions) identified by an exhaustive fault-space scan. Most failures originate from the application stacks and data used by assembler routines (highlighted on colored background).

Analysis of the remaining failures

The remaining 22.6 percent of failures mostly originate from unprotected data, such as the application stacks. For example, Table 7.3 lists the five most failure-prone symbols of eCos, protected by all dependability aspects (G+S+R), running the THREAD1 benchmark. In other words, the table shows those faults that go undetected. 56.9 percent of the remaining failures originate from the program symbol stack, which implements two application stacks that are – strictly speaking – not part of the eCos kernel. In addition, the data member .stack_ptr of the first thread instance accounts for another 18.6 percent of the remaining failures. These failures occur within the assembler routine hal_thread_switch_context, which just cannot be protected by AspectC++. Likewise, the symbol hal_interrupt_handlers causes 16.3 percent of the remaining failures and is used only within the low-level assembler routine that handles hardware interrupts. Finally, 7.6 percent of the remaining failures concern errors in low-order bits that escape the range check of function pointers.

7.1.4.2 *Efficiency: Runtime*

Figure 7.5b depicts the simulated runtime in units of CPU clock cycles of the benchmarks MUTEX2 and I4COPTER running on the different eCos variants. GOP is configured to meet a slowdown limit of one percent, but applying the two symptom detectors (GOP+S) slows down the benchmark MUTEX2 by 14.4 percent. In that case, the G+S+R variant causes a runtime overhead of 21.6 percent.

Figure B.12 on page 201 in Appendix B shows the results of all 18 benchmarks, of which RELEASE represents the worst case: GOP+S causes a slowdown of 23.5 percent, and G+S+R reaches the maximum of 50.7 percent runtime overhead.

I4COPTER contains – as mentioned earlier – slack time in its schedule, so that the dependability aspects run in the idle phases, resulting in a negligible slowdown. In conformity with Amdahl's law, the total slowdown of all 18 benchmarks is below one percent, as shown in the rightmost diagram of Figure 7.5b. The short-running benchmarks are of little significance compared to the long-lasting ones when calculating the arithmetic mean, which is yet the only accurate measure for characterizing the performance of a benchmark suite [228]. In addition to measuring the simulated runtime, I deployed the different eCos variants on an Intel Core i7-M620 machine and verified the mean slowdown to account for less than one percent on real x86 hardware [32].

7.1.4.3 *Memory Footprint: Static Binary Size*

Figure 7.5c illustrates the static binary size of eCos linked with the respective benchmark applications. On average, the read-only text section grows by 11.8 KiB (+ 45.1 percent) of program instructions for GOP alone. The combinations GOP+S and G+S+R enlarge the text section by an average of 16.5 KiB (+ 63.0 percent) and 26.2 KiB (+ 100.3 percent), respectively.

As discussed in Section 7.1.3.2, GOP alone increases the data and BSS section by an average of 1,500 bytes (+ 4.7 percent). The combination with symptom detection (GOP+S) requires a total of 2,098 bytes (+ 6.6 percent). Finally, Return-Address Protection (G+S+R) does not further increase the RAM allocation.

7.2 CASE STUDY: HARDENING L4/FIASCO.OC

The second case study applies the dependability aspects to the operating system L4/Fiasco.OC, which represents a state-of-the-art microkernel as described in Section 3.2 on page 46. In brief, L4/Fiasco.OC provides multiple address spaces with hardware-based isolation; the microkernel occupies a reserved address space and is the only piece of software that runs in the privileged processor mode, so that faults in user-level programs cannot corrupt the kernel.

To detect and correct errors in the kernel address space, Generic Object Protection (GOP) and Virtual-Function Pointer Protection (VPP) are applied to the microkernel. Return-Address Protection is not applied because the user-level programs allocate the stack memories in their respective address spaces and not in the kernel address space. Furthermore, the lifetime of return addresses on the kernel stack is short, because the control flow does not block within the kernel. This case study focuses on error correction and, thus, omits symptom detection. Altogether, this case study compares the following four variants of the L4/Fiasco.OC microkernel:

BASELINE: The unprotected microkernel serves as a reference.

VPP: Virtual-Function Pointer Protection (error detection and correction) is applied to 26 kernel data types.

BENCHMARK	DESCRIPTION OF THE TESTING DOMAIN	THREADS
CLNTSRV	Input/output stream IPC mechanism	2
IPC	L4 IPC mechanism (3 round trips)	2
MAP_IRQ	Exchange of capabilities (access control)	2
SHARED_DS	Shared memory using `L4Re::Dataspace`	2
STREAMMAP	Exchange of virtual memory pages	2
UIRQ	Interrupt functionality (see Section 3.2.3)	2
UTCB-IPC	Low-level IPC mechanism	2

Table 7.4: Example and test programs of the L4 Runtime Environment (L4Re). These user-level programs test the essential microkernel features. The third column shows the number of threads that are started by each program.

VGOP/STU: This variant extends the VPP variant by Generic Object Protection, applied to 23 kernel data types. Whole-program optimization is disabled.

VPP+GOP: Same as VGOP/STU, but with whole-program optimization.

In detail, both GOP variants include support for polymorphism (see Section 6.5.2.3) and use the adaptive Hamming code for error detection and error correction (see Section 6.5.1.2). The VGOP/STU variant allows evaluating the effect of whole-program optimization (see Section 6.5.2.2).

This case study uses seven user-level test programs of the *L4 Runtime Environment (L4Re)* to generate load on the microkernel during the evaluation. Table 7.4 briefly describes these programs, which test the essential microkernel features, such as Inter-Process Communication (IPC), interrupts, management of shared memory, and capability-based access control. Each program runs two threads and reports its status on the serial port. Therefore, I disable the exact timing of the driver for the serial device, because waiting for a fixed baud rate would completely mask any protection's runtime overhead.

The following evaluation uses the same experimental setup of the fault-injection framework FAIL* as described in Section 3.2.1. In short, the setup includes an x86-hardware emulator and the fault model of independent single bit flips in the whole data memory of the kernel address space.

7.2.1 *Effectiveness: Error Detection and Correction*

Section 3.2.3 and Appendix A.2 already present the results of exhaustive fault-space scans of the baseline variant of L4/Fiasco.OC running the benchmarks CLNTSRV, UIRQ, and STREAMMAP. This case study reuses these results and estimates the fault tolerance of the other variants and benchmarks by random samples of $N = 100,000$ fault injections each. Equation 3.4 on page 43 describes the statistical estimator that extrapolates the total number of failed program runs from the samples by taking the benchmark runtime and memory usage into account. This extrapolation is necessary because the protected variants run slower and, thus, are vulnerable to transient hardware faults for a longer period of time. Altogether, the estimate has a maximum relative standard error of 1.13 percent.

Figure 7.6a omits the hardly visible confidence intervals because of the tiny relative standard error.

Figure 7.6a shows the fault-injection results as extrapolated failure counts for each benchmark and variant combination. The baseline variant of the microkernel is the most susceptible to memory errors and exhibits a total of $11.44 \cdot 10^{13}$ failures for all benchmarks. By applying the VPP aspect, these failures are reduced by 11.9 percent. In addition, the combination of VPP and GOP without whole-program optimization (VGOP/STU) achieves a total failure reduction of 58.7 percent. Finally, the optimized VPP+GOP variant is always the most robust and reduces the total number of failures by 59.9 percent.

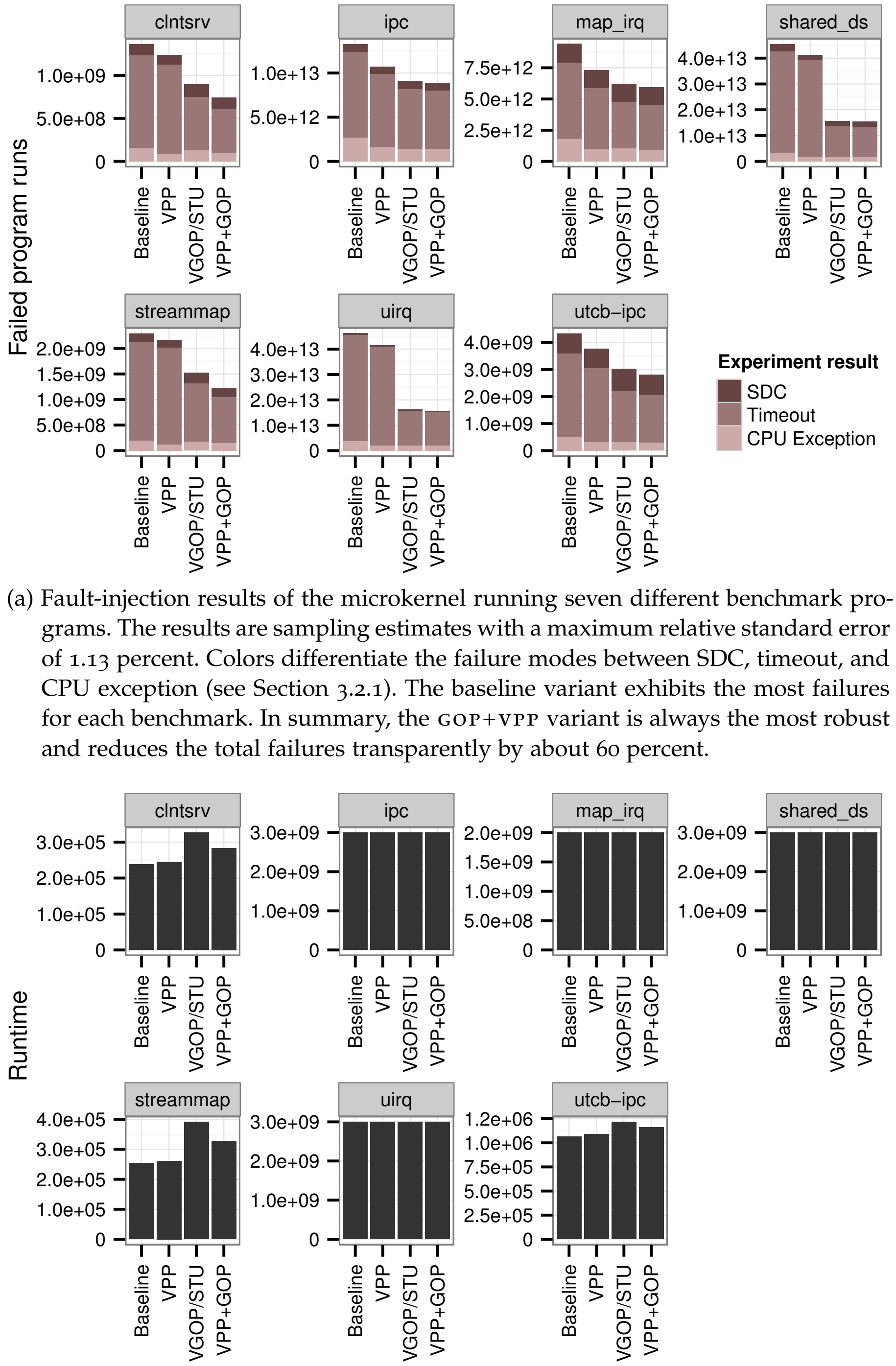
VPP and GOP avoid 60 percent of the failures.

The remaining $4.59 \cdot 10^{13}$ failures are mostly caused by unprotected data that has not been covered by VPP and GOP, yet. For example, the baseline evaluation in Section 3.2.3 on page 53 identified two large heap memories as reasons for many failures. Future work needs to analyze these heaps in more detail, so that the dependability aspects can be applied to the data structures found there.

7.2.2 *Efficiency: Runtime*

Figure 7.6b depicts the simulated runtime in units of CPU clock cycles of the seven benchmark programs. On the one hand, the runtime of the three benchmarks CLNTSRV, STREAMMAP, and UTCB-IPC increases notably with the degree of protection. STREAMMAP represents the worst case and exhibits the highest runtime overhead, which amounts to 2.7 percent for VPP. Without whole-program optimization, the VGOP/STU variant even prolongs the benchmark's runtime by 53.7 percent. However, the optimized VPP+GOP variant reduces that runtime overhead to 28.4 percent. Thus, the whole-program optimization of GOP considerably improves the efficiency, which in turn leads to increased effectiveness: Figure 7.6a shows that the VPP+GOP variant always exhibits the lowest failure counts.

On the other hand, the remaining four benchmarks have an almost constant runtime regardless of any protection. These four benchmarks contain idle phases that provide slack time for the VPP and GOP aspects. Thus, the slowdown caused by the dependability aspects is negligible in such cases.



(a) Fault-injection results of the microkernel running seven different benchmark programs. The results are sampling estimates with a maximum relative standard error of 1.13 percent. Colors differentiate the failure modes between SDC, timeout, and CPU exception (see Section 3.2.1). The baseline variant exhibits the most failures for each benchmark. In summary, the gop+vpp variant is always the most robust and reduces the total failures transparently by about 60 percent.

(b) Simulated runtime in units of CPU clock cycles. The benchmarks clntsrv, streammap, and utcb-ipc exhibit a considerable runtime overhead in the GOP variants; however, the whole-program optimization of the vpp+gop variant reduces that overhead to a maximum of 28.4 percent (streammap). The benchmarks ipc, map_irq, shared_ds, and uirq include slack time in their schedule, so that the dependability aspects cause only a negligible slowdown.

Figure 7.6: Quantitative evaluation of the VPP and GOP dependability aspects applied to the L4/Fiasco.OC microkernel based on two metrics: (a) fault tolerance, and (b) runtime. The abbreviation vgop/stu denotes a combination of VPP and GOP, just like vpp+gop, but without whole-program optimization.

KERNEL VARIANT	TEXT SECTION	DATA SECTION	BSS SECTION
BASELINE	420.9	19.5	600.0
VPP	445.8	21.1	600.5
VGOP/STU	507.8	27.0	600.6
VPP+GOP	501.1	24.4	600.6

Table 7.5: Static binary size in units of kibibytes (KiB) of the L4/Fiasco.OC kernel images. For instance, the most effective variant VPP+GOP requires 80.2 KiB of ROM and 5.5 KiB of RAM in addition to the baseline.

7.2.3 *Memory Footprint: Static Binary Size*

The dependability aspects introduce information redundancy into the microkernel to detect and correct memory errors transparently. Thus, the memory usage of the kernel image increases as shown in Table 7.5. In contrast to the previous case study, the L4/Fiasco.OC microkernel is compiled and linked separately from the benchmark applications, which are loaded individually at runtime. Therefore, the size of the kernel image is independent of the benchmark applications.

The VPP aspect alone increases the size of the read-only text section by 24.9 KiB (+ 5.9 percent) of program instructions. Furthermore, combining both dependability aspects and using whole-program optimization (VPP+GOP) enlarges the text section by a total of 80.2 KiB (+ 19.1 percent).

In addition, the RAM allocation – that is, the sum of the data and BSS sections – increases only insignificantly: VPP adds 2.1 KiB (+ 0.3 percent), whereas VPP+GOP requires a total of 5.5 KiB (+ 0.9 percent) of additional RAM. The reason for this rather low demand on RAM is the adaptive Hamming code (see Section 6.5.1.2), which uses information redundancy that grows only logarithmically with the protected object's size.

In summary, the combination of the VPP and GOP dependability aspects provides an excellent trade-off between memory overhead, slowdown, and improvement in fault tolerance by about 60 percent.

7.3 CASE STUDY: HARDENING MEMCACHED

Applying the dependability aspects to a user-level program

The third case study leaves the domain of dependable operating systems and moves on to the user-level program *Memcached* [84]. Because the dependability aspects are designed as a highly generic and reusable library, there is no reason to limit their field of application to operating systems. Thus, this case study exemplarily applies the dependability aspects to Memcached and evaluates the effectiveness and efficiency.

In brief, Memcached is a multi-threaded server program that implements an in-memory key-value store for small data. That is, the Memcached server stores user-defined data in RAM to provide low-latency access. Clients populate the server's key-value store by transmitting data over a network connection; afterwards, the clients can fetch that data quickly. Memcached is commonly used for speeding up dynamic websites by reducing database load. For example, Wikipedia [84] and Facebook [180] deploy hundreds of Memcached servers, processing over a billion accesses per second and storing trillions of data items [180, p. 385]. Thus, dependability is of particular concern.

This case study applies the dependability aspects to a C++ version [160, p. 72] of the original C-based implementation of Memcached. Generic Object Protection (GOP) is applied to the payload data that are stored in RAM to avoid silent data corruption. In addition, GOP enables error detection and correction in several bookkeeping data structures, such as linked lists that store pointers to the payload data. Furthermore, this case study evaluates whether the other dependability aspects avoid program crashes effectively.

7.3.1 *Experimental Setup*

The experimental setup of this case study differs from the previous case studies, because the software under test is not an operating system. Thus, there is no benefit from using a hardware emulator for fault injection – a crash of a user-level program after injecting a fault does not require a full hardware reset; it just suffices to restart the program. Moreover, an exhaustive fault-space scan is infeasible for a memory-intensive application such as Memcached, even using the trace-based fault-space optimization described in Section 3.1.3. Therefore, fault sampling is the only viable method.

For these reasons, I decided to develop another fault-injection tool that implements fault sampling in Linux processes that run natively on real hardware. Such a tool yields accurate timings of super-scalar and pipelined multiprocessors and their underlying memory hierarchy. The following section briefly describes the design and implementation of the tool that is used to evaluate the dependability aspects applied to Memcached.

7.3.1.1 *Fault Injection into Linux Processes*

The Unix system call `ptrace`, commonly used for program debugging, technically allows injecting memory faults into a user-level process at runtime [123, p. 250]. As affirmed by Section 3.1.1 on page 41, hardware memory faults follow a uniform random distribution in both spatial and temporal dimensions. Thus, the injected faults must be consistent with such a probability distribution.

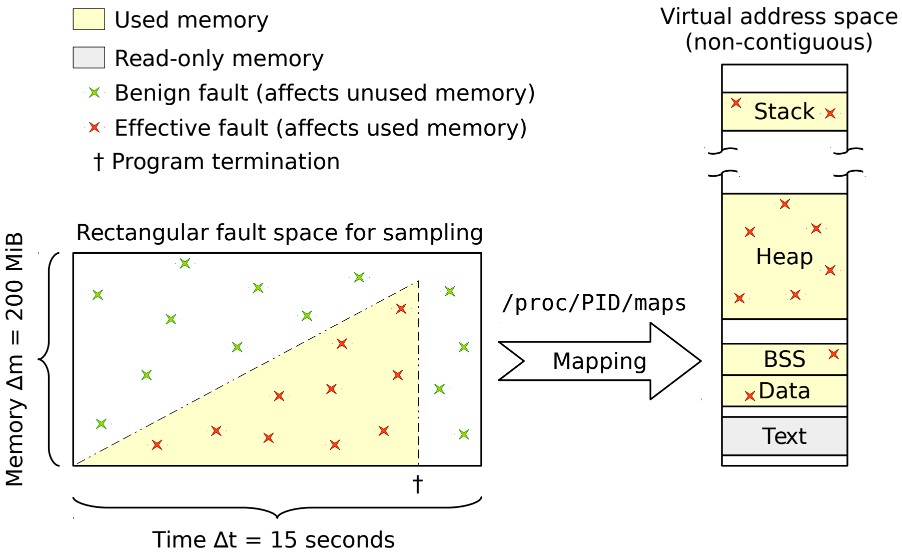


Figure 7.7: Fault injection into the virtual memory of a Linux process. To implement an unbiased yet efficient sampling strategy, the fault coordinates are chosen randomly from a rectangular fault space of fixed dimensions $\Delta t \times \Delta m$. Afterwards, the fault coordinates are mapped to the process' virtual memory addresses based on information provided by the Linux *proc* file system. If a fault coordinate exceeds the amount of virtual memory used by the process, the fault is considered as benign (green cross). Otherwise, the fault affects an actually used memory address (red cross) and is injected via the system call `ptrace`. Read-only memory is excluded from fault injection as motivated by Section 3.2.1.

On the one hand, sampling uniformly from the whole 64-bit virtual address space of a Linux program is inefficient, because the virtual address space is typically sparsely populated: Most virtual addresses are not mapped to any physical address, so that an exceedingly large number of samples would be required to hit an actually used memory address. On the other hand, the amount of used memory typically varies over time due to dynamic memory allocation, which biases a sample that was picked only from the subset of used memory addresses: Rarely used memory addresses would be underrepresented in the sample.

Uniform sampling

Therefore, I propose to pick an unbiased sample uniformly from a small fault space of fixed dimensions and to map the sample to the process' virtual address space afterwards. Figure 7.7 illustrates such an approach, assuming fault-space dimensions of $\Delta t = 15$ seconds and $\Delta m = 200$ MiB. These dimensions define a rectangular fault space $\Delta t \times \Delta m$, from which each fault coordinate (nanosecond, bit) is chosen randomly with the same probability. In a second step, these fault coordinates are mapped to the used memory addresses of the target process at runtime. The Linux *proc* file system [59, p. 419] provides information on the virtual memory addresses that are in use at `/proc/PID/maps`, where `PID` refers to the process id. Based on that

Sampling from a rectangular fault space of fixed dimensions information, each fault coordinate is mapped to a virtual memory address, unless the coordinate's value exceeds the amount of used virtual memory at the respective point in time. In such a case, the fault is benign as it affects unused memory.

For example, consider a program that repeatedly allocates dynamic memory and never releases it until program termination. Thus, the amount of used memory steadily grows, as indicated by the triangular shape on the left-hand side of Figure 7.7. Only the ten fault coordinates that are located within this triangle are effective, because they can be mapped to a virtual memory address. Therefore, if the fault-injection tool finds a coordinate that cannot be mapped to a virtual address, the tool just records the fault-injection result as benign and proceeds with the next coordinate. Effective faults are eventually injected via the system call `ptrace` – one fault per independent program run.

In summary, the described approach implements an unbiased sampling strategy consistent with the uniform random distribution. Yet, it is essential to choose the fault-space dimensions Δt and Δm in such a way that the target process never exceeds them. In other words, the process must terminate before Δt elapses, and it must not use more than Δm of memory at a time. If we keep the fault-space dimensions constant for evaluating different programs, we can even directly compare the fault-injection results of the different programs without extrapolation to the fault space (see Equation 3.4 on page 43).

7.3.1.2 *Workload, Hardware, and Protection Variants*

To evaluate the Memcached server under load, I use the load generation and benchmark tool *memaslap*, which is part of *libmemcached*[1]. The tool memaslap populates the key-value store of the Memcached server with random data and, in turn, fetches the stored data. In brief, memaslap repeats a procedure of storing one key-value pair and subsequently fetching nine random key-value pairs that have been stored previously. In other words, memaslap implements a test sequence of get–set requests with a ratio of nine to one. The test sequence executes a total of one million requests, involving 100,000 distinct key-value pairs with a fixed key size of 64 bytes and 1,024 bytes of payload data.

Software setup

In addition to the default settings, memaslap is configured to examine the fetched data for silent data corruption (`--verify=1.0`) and to use two threads (`--threads=2`), which establish a total of 16 concurrent connections to the Memcached server. Likewise, the Memcached server runs in the default configuration of four threads, but the heap memory is limited to 150 MiB.

In this case study, both the Memcached server and memaslap run on the same machine and communicate via the loopback interface

[1] libmemcached is available online at: `http://libmemcached.org`

to avoid any network latency. The hardware setup consists of one machine with two quad-core Intel Xeon X5470 processors; each processor has 12 MiB of second-level cache memory. Debian GNU/Linux 8.5 operates this machine in 64-bit mode.

Hardware setup (x86-64)

Based on the aforementioned workload and hardware setup, this case study compares the following three variants of the Memcached server:

BASELINE: An unprotected variant serves as a reference.

GOP+VPP: Both the Generic Object Protection (GOP) and Virtual-Function Pointer Protection (VPP) are applied to the payload data and several bookkeeping data structures of Memcached. The GOP configuration includes support for wait-free synchronization (see Section 6.5.3), whole-program optimization (see Section 6.5.2.2), and uses the adaptive Hamming code for error detection and error correction (see Section 6.5.1.2). Likewise, VPP is configured to detect and correct errors.

GOP+VPP+RAP: In addition to the aforementioned variant, Return-Address Protection (RAP) is applied to detect and correct errors that affect the control flow. The RAP configuration includes whole-program optimization (see Section 6.3.2).

7.3.2 *Effectiveness: Error Detection and Correction*

This section evaluates the fault tolerance of the Memcached server by fault injection. During each independent run of the benchmark memaslap, one bit flip is injected into the data memory of the Memcached server process. The fault locations are picked at random from the rectangular fault space defined by $\Delta t = 15$ seconds and $\Delta m = 200\,\text{MiB}$. After fault injection, the benchmark memaslap runs to completion; the Memcached server is restarted once the memaslap run finishes.

Each of the three Memcached variants is evaluated by a sample of 100,000 fault injections, which corresponds to a total of two weeks of continuous operation. The outcome of a single fault-injection experiment is either classified as benign, as silent data corruption (SDC) reported by memaslap, as timeout, or as CPU exception, such as invalid memory access or execution of an illegal instruction.

Figure 7.8 shows the absolute frequency of the three failure modes that occur after fault injection into the three variants of Memcached. The error bars represent the 99-percent confidence intervals obtained by applying the Central Limit Theorem; that is, 99 percent of all samples of the same size would yield results covered by these error bars.

The leftmost diagram depicts the absolute frequency of SDC, which certainly represents the most severe failure mode. About 10 percent

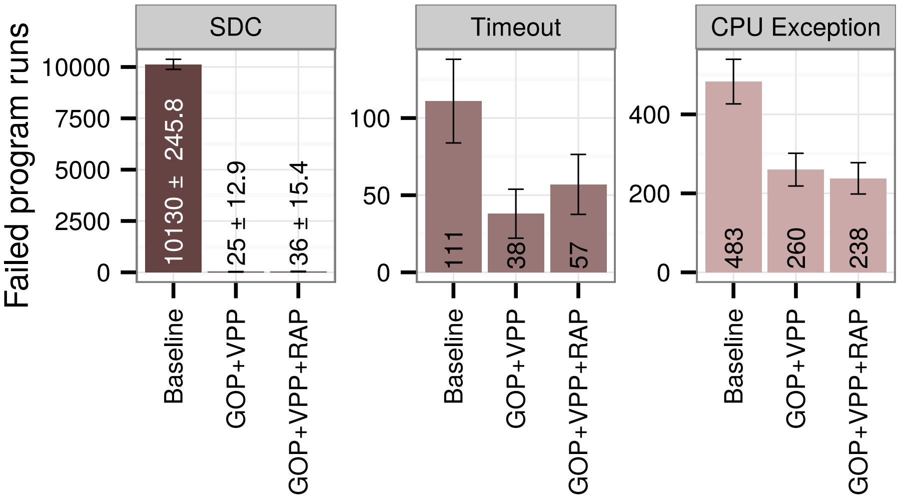


Figure 7.8: Quantitative fault-injection results of Memcached under workload generated by the benchmark tool memaslap. A random sample of 100,000 single bit flips is injected into the data memory of each Memcached variant – BASELINE, GOP+VPP, and GOP+VPP+RAP – one fault per independent benchmark run. The individual diagrams show the absolute frequencies of the failure modes SDC, timeout, and CPU exception on the y-axis. The error bars, and numbers prefixed by the ± symbol, represent the 99-percent confidence intervals to estimate the sampling error. In summary, the protected variants exhibit three orders of magnitude fewer SDCs and about 50 percent fewer crashes.

of the injected faults cause an SDC in the baseline variant of Memcached, which exhibits a total of 10,130 SDCs. The dependability aspects GOP+VPP are highly effective and reduce the number of SDCs by three orders of magnitude to just 25 occurrences. In other words, about 99.8 percent of the faults that cause an SDC are detected and corrected transparently by the two dependability aspects. Applying the RAP aspect, however, does not further improve the SDC rate.

SDC reduction

The other two diagrams in Figure 7.8 illustrate the absolute frequencies of timeouts and CPU exceptions, which both occur less often than SDCs in the baseline variant. About of 50 percent of these crash failures are avoided by the dependability aspects. The difference between the variants GOP+VPP and GOP+VPP+RAP is not statistically significant at the 99-percent confidence level, because the depicted confidence intervals overlap. However, neither confidence interval overlaps with the baseline variant, suggesting a statistically significant improvement over the baseline in both aspect combinations.

Crash avoidance

In conclusion, the RAP aspect turns out to be virtually ineffective when applied to Memcached. The GOP and VPP aspects, however, significantly improve the data integrity by three orders of magnitude, and further improve the availability of the Memcached server by avoiding about 50 percent of the crash failures.

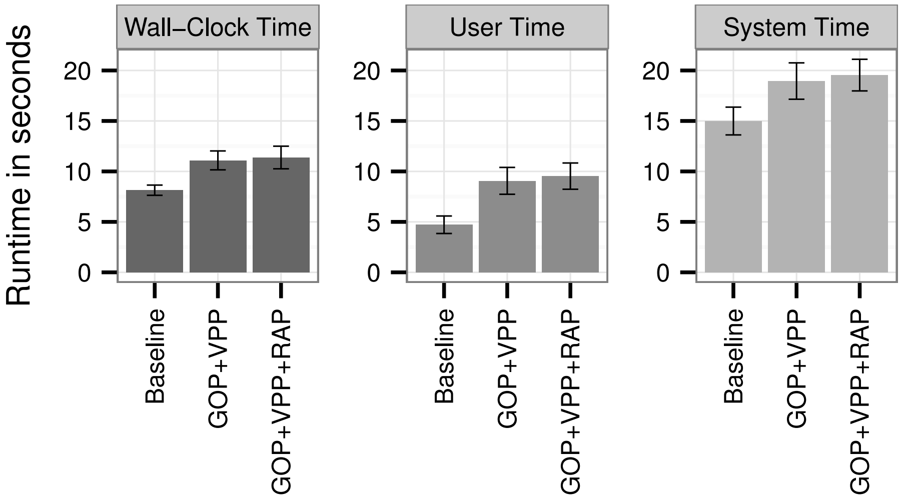


Figure 7.9: Native runtime of the Memcached server under workload generated by the benchmark tool memaslap. The individual diagrams depict the arithmetic mean of 100 benchmarks runs; the error bars represent one standard deviation of the sample. From left to right, the diagrams show the elapsed real time (wall-clock time), the actual computation time (user time), and the time spent in the Linux kernel (system time). The gop+vpp variant exhibits a total slowdown of 36.3 percent in wall-clock time, an overhead of 92.3 percent in user time, and an increase of 26.4 percent in system time.

7.3.3 *Efficiency: Runtime*

The improved dependability comes at the cost of increased runtime required for processing the workload generated by the benchmark tool memaslap. Figure 7.9 shows the arithmetic mean and one standard deviation of the runtimes of 100 benchmark runs using the different variants of the Memcached server. The leftmost diagram depicts the elapsed real time (wall-clock time): On average, the baseline variant handles the workload in 8.1 seconds, whereas the variants gop+vpp and gop+vpp+rap require 11.1 seconds and 11.4 seconds, respectively. Thus, gop+vpp adds a latency of three seconds, which corresponds to an increase of 36.3 percent.

Wall-clock time

In addition to the wall-clock time, Figure 7.9 presents a breakdown of time into user time and system time. The former amounts to the actual computation time spent within the Memcached process without waiting for I/O. On average, the baseline variant takes 4.7 seconds of computation time, which is increased to 9.1 seconds by gop+vpp. This increase amounts to an overhead of 92.3 percent, caused by the computation-intensive GOP applied to the roughly 150 MiB of payload data. In addition, the RAP aspect adds another 0.5 seconds of user time.

User time

The rightmost diagram depicts the system time – that is, the time spent in the Linux kernel for carrying out I/O operations and context

System time switches. About 15.0 seconds are consumed by the baseline variant; GOP+VPP requires 19.0 seconds and, again, the RAP aspect adds another 0.5 seconds on top. The increase of system time is an indirect consequence of the increased user time, as the protected variants exhibit roughly twice the context switches of the baseline variant. For example, if the runtime overhead occurs within a critical section, other threads of control are potentially blocked that would otherwise have not been blocked.

As a final note, the wall-clock time is much lower than the sum of user and system time, because Memcached uses four threads of control that concurrently contribute to the user and system time by utilizing multiple processors.

7.3.4 *Memory Footprint*

The memory footprint of Memcached is dominated by dynamic heap memory that stores the payload data. As described in Section 7.3.1.2, the benchmark memaslap generates random key-value pairs with a fixed key size of 64 bytes and 1,024 bytes of payload data. In total, such a data item requires 1,224 bytes of memory in the linked data structures of the baseline variant of Memcached. The GOP+VPP variant adds 208 bytes of information redundancy to each data item, which amounts to an increase of 17.0 percent. However, this increase is completely hidden by the internal memory alignment of Memcached, which aligns the data items at powers of 1.25 to avoid memory fragmentation. For example, data items are stored in chunks of either 1,184 bytes or 1,480 bytes, but not in intermediate sizes. In the end, each key-value pair generated by memaslap requires 1,480 bytes of memory – no matter whether the dependability aspects are applied or not.

In comparison to the 150 MiB of dynamic heap memory, the read-only text section of the baseline variant amounts to only 149.6 KiB of program instructions. The GOP+VPP variant increases that text section by 114.1 KiB (+ 76.3 percent), and the RAP aspect adds another 20.3 KiB. In summary, the read-only text section grows considerably, but the total memory footprint remains virtually identical because of memory alignment.

7.4 CHAPTER SUMMARY

The goal of this chapter was to evaluate the library of dependability aspects and, thus, the general approach of this thesis. First, this chapter highlights the reusability of the individual dependability aspects, which are so generic that they can be applied transparently to three large-scale, off-the-shelf software systems. The main results of this chapter are:

- *eCos*: A combination of three dependability aspects reduces the total number of failures by 77 percent.

- *L4/Fiasco.OC*: The dependability aspects GOP and VPP detect and correct 60 percent of the faults that cause a failure of the unprotected microkernel.

- *Memcached*: The dependability aspects GOP and VPP improve the data integrity by three orders of magnitude and avoid about 50 percent of the crash failures.

The kernels of the eCos and L4/Fiasco.OC operating systems turn out to be particularly suited for being hardened by software-implemented fault-tolerance mechanisms, because the total slowdown caused by these mechanisms stays below one percent in most cases. The reason is that the kernel is not active all the time, as user-level applications typically get most of the CPU time. Thus, runtime overhead within the kernel increases the total runtime by only a small fraction. However, the evaluation of the user-level program Memcached shows that the computation-time overhead of the dependability aspects is up to 92.3 percent on real x86-64 hardware. The bottom line is that the runtime overhead depends on how frequently the protected data are accessed.

In summary, this chapter suggests that the approach of using aspect-oriented programming with AspectC++ 2.0 is both efficient and effective in improving the dependability of operating-systems kernels and user-level applications.

8 DISCUSSION

The previous chapter suggests that the approach of this thesis is effective in improving the dependability of operating systems and user-level applications. Yet, the previous evaluation is based on the *fault model* of uniformly distributed single bit flips, as motivated by Section 3.1.1 on page 41. It is paramount to recapitulate that this fault model is a simplification of the real world. Thus, it is essential to validate the findings of the previous chapter. To this end, Section 8.1 confirms the findings by neutron-beam testing at Los Alamos National Laboratory, USA.

Subsequently, this chapter examines the software maintainability of the approach taken in this thesis. Section 8.2 revisits the problems of code scattering and code tangling, and how they are solved by the dependability aspects. After all, dependability implies proper maintainability [18, p. 11].

Furthermore, Section 8.3 discusses the fundamental limitations of the aspect-oriented approach to software-implemented fault tolerance. Finally, Section 8.4 outlines directions for future work.

RELATED PUBLICATIONS

The findings presented in this chapter have partly been published in:

[211] Thiago Santini, Christoph Borchert, Christian Dietrich, Horst Schirmeier, Martin Hoffmann, Olaf Spinczyk, Daniel Lohmann, Flávio Rech Wagner, and Paolo Rech. Effectiveness of Software-Based Hardening for Radiation-Induced Soft Errors in Real-Time Operating Systems. In *Proceedings of the 30th International Conference on Architecture of Computing Systems (ARCS '17)*, Springer, April 2017. doi: 10.1007/978-3-319-54999-6_1

Acceptance rate: 45 %

8.1 REALITY CHECK: NEUTRON-BEAM TESTING

The fundamental working hypothesis of the preceding chapters is based on the *fault model* of independent, uniformly distributed single bit flips in memory. In particular, Section 3.1.1 on page 41 assumes that software vulnerability under this specific fault model implies vulnerability to radiation-induced hardware faults. Based on this assumption, Section 3.1.2 provides a comparison metric for evaluating program reliability by means of fault injection, which is used extensively in the previous chapter.

However, as the fault model is a simplification of the physical reality, the fault-injection results of the previous chapter might be inaccurate. Therefore, this section performs the reality check:

> *Is there evidence that the approach of this thesis improves the dependability of operating systems?*

Accelerated testing in compliance with JEDEC 89A

This research question can be answered by testing a computer system under normal operating conditions in the terrestrial radiation environment. Yet, it takes decades to obtain statistically significant results unless a huge array of computer devices is tested in parallel [118, p. 15]. As an alternative, the JEDEC standard 89A [118] approves accelerated testing procedures using high-energy particle accelerators. In particular, the standard recommends the Los Alamos National Laboratory (LANL), USA, which provides a neutron beam with an energy spectrum that closely matches the terrestrial neutron flux at approximately 10^8 times the intensity [118, p. 38f]. In brief, the required time is reduced by the same factor. Thus, this section validates the findings of the previous chapter at LANL.

8.1.1 *Experimental Methodology*

Due to economic constraints on the duration of the neutron-beam testing, this section considers only parts of the case study on eCos (see Section 7.1) for validation. For that purpose, the operating system eCos is deployed on a state-of-the-art embedded hardware device, which is placed in the focus of the high-intensity neutron beam.

8.1.1.1 *Hardware Setup*

The device under test is a Xilinx Zynq-7000 system-on-chip manufactured in 28 nm CMOS technology. It contains an ARM Cortex-A9 dual-core microprocessor operating at 667 MHz, of which only one core is used during irradiation.

Setup of the SRAM caches

The on-chip memory consists of separate first-level caches for data memory and program instructions of each 32 KiB. In addition, there is a unified second-level cache of 512 KiB, configured to cache data memory solely. These caches support only error detection by parity codes; thus, parity checking is disabled to allow *error correction* by the dependability aspects. Furthermore, the instruction cache is invalidated at the regular interval of one timer interrupt to reduce corruption of program instructions in the cache, as proposed by Sridharan and colleagues [233, p. 363].

Two external DDR-3 DRAM devices are connected to the internal memory controller of the system-on-chip. Yet, both DRAM devices are *not* placed in the focus of the neutron beam, which targets only the system-on-chip, as the data memory of eCos and its benchmark suite already fits into the second-level cache.

8.1.1.2 *Software Setup*

A minimal configuration of eCos 3.0 without any unneeded device drivers runs on the aforementioned hardware platform[1]. That configuration includes the ignoring of spurious device interrupts.

The workload of eCos consists of two parameterizable benchmarks from its test suite – bin_sem2 and timeslice2 – because they support a parameterizable number of threads. This number is increased to fill up the second-level cache in order to *maximize the attack surface* of the neutron beam.

bin_sem2: The benchmark bin_sem2 implements the classical synchronization problem of the Dining Philosophers. That is, 400 threads (*philosophers*) use 400 semaphore objects (*forks*) for mutual exclusion (*eating* with two forks). Once a philosopher acquires both neighboring forks, it asserts that its neighboring philosophers are not in the *eating* state. The philosopher releases both forks after a pseudo-random delay; that procedure continues for 25,000 iterations.

timeslice2: This benchmark tests the per-thread time-slice distribution under preemption. For that purpose, 800 threads of low priority continuously increment a per-thread counter variable, while being interrupted by one thread of high priority at regular intervals. The benchmark runs for 1.6 seconds, so that each of the 800 threads receives two time slices, which is finally checked by an assertion.

timeslice2 *is not considered in Section 7.1 as it executes so many CPU instructions that the hardware emulator becomes too slow.*

Both benchmarks run on the following two variants of eCos:

baseline: The unprotected eCos serves as a reference.

gop+sp: Generic Object Protection (GOP) is applied to all kernel data structures as in Section 7.1.3, using the adaptive Hamming code for error detection and error correction (see Section 6.5.1.2). As pointed out in Section 7.1.4.1, most remaining failures originate from the application stacks. Thus, complementary to kernel hardening by GOP, a coarse-grained Stack Protection (SP) covers each piece of used stack memory; a 32-bit checksum is computed when a thread is preempted or yields the CPU, and it gets checked once a thread is resumed.

8.1.1.3 *Experimental Setup*

The device under test is placed in the focus of the high-intensity neutron beam at LANL in the Los Alamos Neutron Science Center (LANSCE) Irradiation of Chips and Electronics House II (ICE

[1] A port of eCos 3.0 for the Xilinx Zynq-7000 hardware platform is available online at: `https://github.com/antmicro/ecos-mars-zx3/`

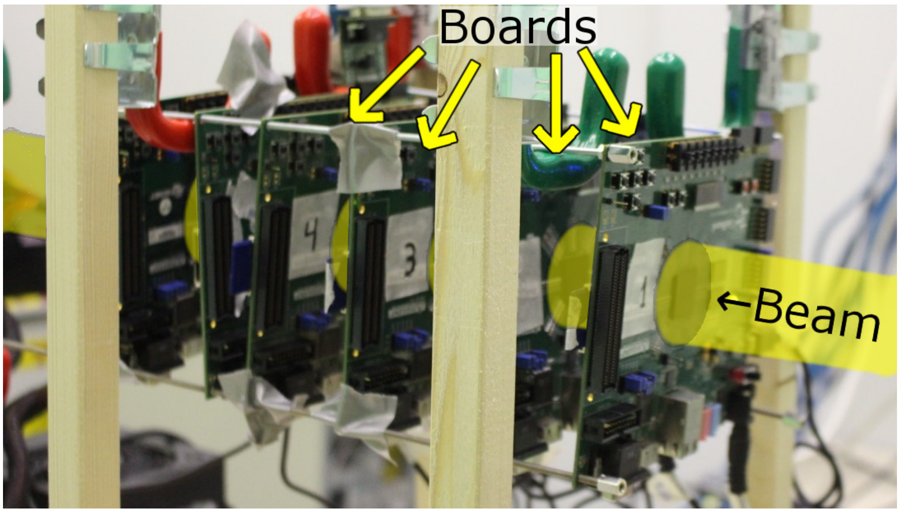


Figure 8.1: Experimental setup mounted at ICE House II. The photo shows four devices under test that are placed in the (indicated) neutron beam at once – one behind the other. The fifth and sixth boards are not used in this study.

House II). The ICE House II beam line provides a neutron flux of approximately 10^6 neutrons per cm² per second, which is focused on a spot with a diameter of 5 cm to irradiate the system-on-chip uniformly without affecting the power control circuity and DRAM.

Four devices under test are placed in the neutron beam at once – one behind the other – because the emitted neutrons are not strongly absorbed by the devices. Figure 8.1 shows the exact setup of the four boards; a flux derating factor is applied to each board depending on its distance from the neutron source. Each benchmark is executed alternately on all four devices to avoid any bias on the particular board and distance.

The testing procedure is carried out for about one day, during which the boards receive a total radiation dose (*fluence*) of $5.1 \cdot 10^{11}$ neutrons per cm². This corresponds to $4.5 \cdot 10^6$ years of exposure to the natural radiation environment at New York City, which exhibits a neutron flux ϕ_{NYC} of 13 neutrons per cm² per hour (see Section 2.1.1).

8.1.2 *Experimental Results*

Table 8.2 lists the raw result data of the neutron-beam testing. In total, the neutron beam causes 567 failures during the testing period. The failure modes are differentiated between silent data corruption (SDC), timeout, and CPU exception. In addition, the table shows the total number of benchmark runs per variant (sample), and the fluence Φ received during these runs. In short, fluence is the neutron flux integrated over the exposure time – that is, the accumulated number of neutrons that pass through the device under test during the bench-

BENCHMARK	VARIANT	SDC	TIME-OUT	CPU EXCEPTION	SAMPLE	FLUENCE Φ [cm^{-2}]	SLOW-DOWN
BIN_SEM2	BASELINE	35	87	72	29,873	$7.6 \cdot 10^{10}$	1.0
	GOP+SP	5	30	32	38,605	$12.0 \cdot 10^{10}$	1.047
TIMESLICE2	BASELINE	65	92	146	59,131	$14.5 \cdot 10^{10}$	1.0
	GOP+SP	3	43	24	100,535	$16.9 \cdot 10^{10}$	1.029

Table 8.2: Raw result data of the neutron-beam testing. The absolute number of failures (SDC, timeout, and CPU exception) *must not be compared* directly because of different sample sizes (benchmark runs) and neutron fluences per variant. A run of the benchmark BIN_SEM2 lasts for 1.981 seconds in the baseline variant, whereas TIMESLICE2 finishes in 1.603 seconds. The rightmost column lists the slowdown factor of the GOP+SP variant relative to the respective baseline.

mark runs. Finally, the last column of Table 8.2 shows the slowdown factor of the protected variant GOP+SP relative to the respective baseline variant.

However, the raw result data must not be compared directly because of different sample sizes and fluences. Thus, it is necessary to calculate a *failure rate* under a common neutron flux, such as the flux ϕ_{NYC} of 13 neutrons per cm² per hour at New York City. According to the JEDEC standard 89A [118, p. 38–42], the failure rate can be calculated as follows:

$$\textit{Failure rate}_{NYC} = \frac{\phi_{NYC}}{\Phi} \cdot F \cdot S \qquad (8.1)$$

The symbol Φ denotes the neutron fluence as listed in Table 8.2; F represents the absolute number of failures per benchmark and eCos variant. S is the respective slowdown factor of the protected variant GOP+SP (last column of Table 8.2), which needs to be taken into account as argued by Santini and colleagues [210, p. 6]. In brief, the protected variant of eCos requires more time than the baseline variant for processing the same workload and, thus, is exposed to radiation for a longer period of time.

The resulting *failure rate* is commonly expressed in units of *Failures In Time (FIT)*, which is the expected number of failures per 10^9 hours of operation. Figure 8.2 compares the extrapolated FIT rates of the two eCos variants running the benchmarks BIN_SEM2 and TIMESLICE2. All failure modes count toward the absolute number of failures F in Equation 8.1. The error bars represent the 95-percent confidence intervals to estimate the sampling error and the uncertainty of 8 percent in the measured neutron fluence.

The baseline variant of eCos exhibits a failure rate of 33.1 FIT and 27.1 FIT when running the benchmarks BIN_SEM2 and TIMESLICE2,

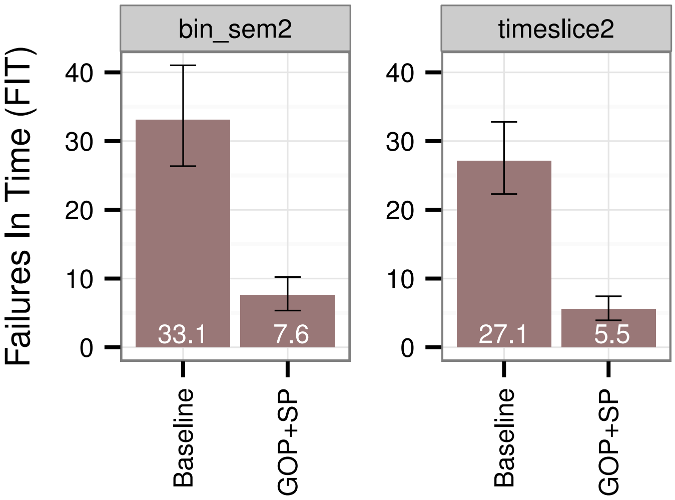


Figure 8.2: Failure rates at sea level of the baseline and protected (GOP+SP) variants of eCos, running the two benchmarks BIN_SEM2 and TIMESLICE2. The y-axis shows the expected number of failures per 10^9 hours of operation; the error bars represent the 95-percent confidence intervals. On average, the GOP+SP variant avoids 78.2 percent of the radiation-induced failures that occur in the baseline variant of eCos.

respectively. These failure rates are reduced effectively to 7.6 FIT and 5.5 FIT, respectively, when using the GOP+SP variant of eCos. Thus, the total failure reduction amounts to 78.2 percent. Furthermore, this reduction is statistically significant as neither confidence interval of GOP+SP overlaps with the baseline variant. In summary, the FIT rates shown in Figure 8.2 indicate a significant improvement in dependability with respect to radiation-induced hardware faults.

8.1.3 *Interpretation of the Results*

The neutron-beam testing results show that the baseline variant of eCos exhibits a failure rate of 27.1 to 33.1 FIT. This failure rate would satisfy an IEC 61508 Safety Integrity Level 3 (SIL 3) in continuous operation mode at sea level – that is, the failure rate is lower than 10^{-7} failures per hour [115]. Moreover, the protected eCos variant GOP+SP exhibits a failure rate of only 5.5 to 7.6 FIT, which is equivalent to less than 10^{-8} failures per hour. Therefore, the protected variant would even attain SIL 4 – the highest SIL. Thus, the GOP+SP variant considerably improves the dependability of eCos by one SIL. However, I do not claim the device under test to achieve these SILs yet, because neither hazard nor risk assessment is carried out.

IEC 61508 Safety Integrity Level (SIL) 3 → SIL 4

In summary, the *reality check* based on neutron-beam testing clearly confirms the findings of the previous chapter. As shown in Figure 8.2, the radiation-induced failure rate is reduced effectively by about 78.2 percent. Likewise, the emulation-based fault-injection results of eCos (see Section 7.1.4.1 on page 150) suggested a failure reduction of 77.4 percent on average. These results are so consistent that they differ by only 0.8 percent points despite the different hardware and software setup. Thus, both results indicate a definite trend: The dependability aspect GOP considerably improves the fault tolerance of the operating system eCos. However, the reality check neither covers all the dependability aspects evaluated in the previous chapter nor the L4/Fiasco.OC microkernel nor Memcached. Therefore, more evidence is needed to generalize the findings of this thesis.

8.2 SOFTWARE MAINTAINABILITY

Avižienis and associates [18, p. 11] argue that *maintainability* is essential for achieving true dependability. Yet, software-implemented fault tolerance represents a crosscutting concern, which is hard to implement as independent module by using a general-purpose programming language as pointed out in Chapter 4. AOP with AspectC++ 2.0, however, addresses this maintainability problem by providing extra language support for the separation of concerns at the implementation level. The AspectC++ compiler automatically applies the dependability aspects as specified by the programmer. This section quantifies to which extent the approach of this thesis improves the software maintainability compared to a manual implementation and compared to a pure compiler-based solution.

8.2.1 *Scattering and Tangling in the Implementation*

The goal of AOP is to avoid code scattering and tangling (see Section 4.1.1 on page 60), because both symptoms impair software maintainability – but to which extent?

To answer this question, Figure 8.3 illustrates the source code of eCos after applying the dependability aspects, which are *woven* by the AspectC++ compiler into the depicted source-code files, represented as horizontal bars. A colored marker indicates a line of code at which the AspectC++ compiler automatically inserts pieces of code as specified by a dependability aspect: Blue markers denote code insertions due to the GOP aspect; red markers show insertions of the RAP aspect; the symptom-detection aspects that implement run-time type checking and checking of function pointers are represented by green and black markers, respectively. Thus, Figure 8.3 shows the combination of dependability aspects evaluated in Section 7.1.4.

In total, the four dependability aspects affect 2,206 lines of code scattered over 105 files of eCos. The vast scattering shows that the dependability aspects are indeed exceedingly crosscutting. A manual implementation without AOP would cause such a dramatic tangling of the source code as illustrated in Figure 8.3, so that it would proba-

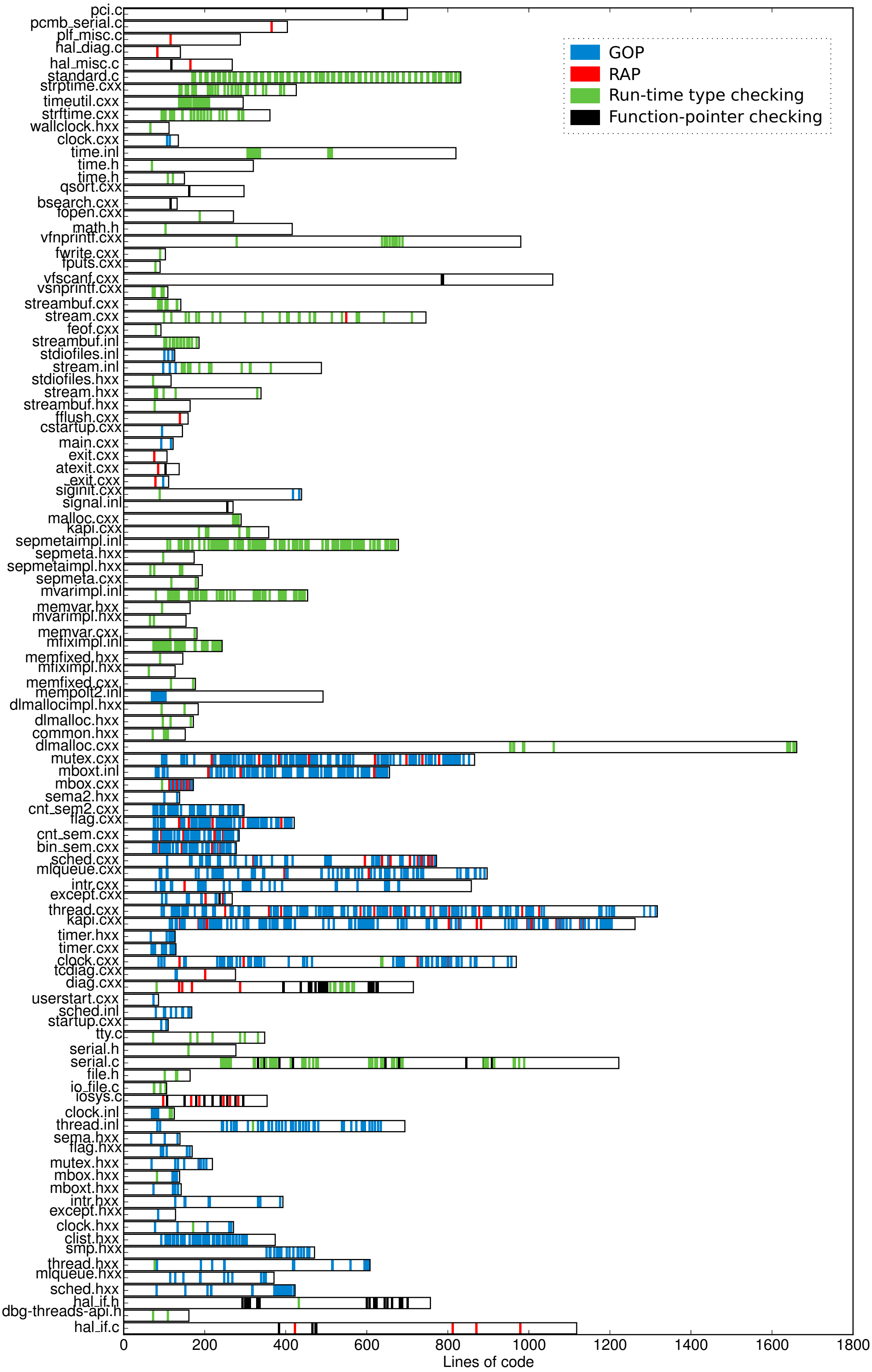


Figure 8.3: Scattering and tangling as avoided by AOP. Each horizontal bar represents a source-code file of eCos. A colored marker highlights a line of code at which the AspectC++ compiler inserts code of a dependability aspect automatically.

bly end up in being completely *unreadable*. Thus, the aspect-oriented approach of this thesis considerably improves the software maintainability compared to a manual implementation. The source code of eCos, L4/Fiasco.OC, and Memcached is kept as it is by strict modularization of the dependability aspects. Furthermore, the library of dependability aspects is even reusable in these three software systems and potentially many more.

8.2.2 *Separation of Concerns in the Compiler*

The previous section shows that the library of dependability aspects achieves a clean separation of concerns by avoiding code scattering and tangling in eCos. Moreover, AspectC++ is a general-purpose programming language that is not specifically targeted at the domain of fault tolerance. Thus, the source code of the AspectC++ compiler is also not tangled with any concerns of fault tolerance. For example, consider the aspect that checks for integer overflows as described in Section 6.2.2 on page 103; that aspect is implemented by a single file of 130 lines of AspectC++ code. To put this into perspective, a comparable extension of the Clang compiler requires more than 1,600 lines of code to implement the same functionality [67, p. 763]. In the latter case, the source code of the Clang compiler becomes tangled with the implementation of integer-overflow checking. This comparison shows that the aspect-oriented approach of this thesis also achieves a better separation of concerns at the compiler level.

Compilers should not become tangled with the concern of fault tolerance.

Furthermore, the AspectC++ programming language is also suitable for protecting low-level mechanisms, such as return addresses of procedures and virtual-function pointers, which are generally inaccessible from a high-level programming language. To this end, the programming language should provide information on all the low-level mechanisms implemented by the compiler. For example, the GNU language extensions expose the return addresses by the intrinsic function `__builtin_return_address()`. Therefore, the dependability aspect RAP (see Section 6.3) can protect the return addresses in a portable way. Compilers do not need to implement any fault-tolerance mechanism by themselves – it suffices to pass the necessary information on to the AOP language. Thus, compilers are kept comparatively simple, whereas the aspect programmer can exploit all the knowledge on the application to implement selective fault tolerance. A lesson learned is that a rich interface to low-level mechanisms allows separating the concern of fault tolerance from the compiler.

Language extensions should provide information on low-level compiler mechanisms.

In summary, the aspect-oriented approach of this thesis affects neither the maintainability of the target software nor the maintainability of the compiler, because AspectC++ is a general-purpose language that enables strict modularization of crosscutting concerns.

8.3 LIMITATIONS

This section summarizes the fundamental limitations that apply to the aspect-oriented approach of thesis. In particular, the improvement of software dependability by using AspectC++ is subject to the following two limitations:

1. *Only the C++ programming language is considered.* AspectC++ is, per definition, an extension of C++ and, thus, inherits the restrictions of that language. For instance, C++ is not fully compatible with pure C, so that AspectC++ cannot be applied out-of-the-box to software written in C. Furthermore, operating systems typically include pieces of assembler code, which cannot be protected by AspectC++ at all (see Section 7.1.4.1). The same applies to external libraries whose source code is not available. In short, the approach of this thesis applies only to C++ code.

2. *Perfect fault tolerance is impossible*. This limitation applies to software-implemented fault tolerance in general (see Section 2.4.4.1 on page 36). For example, if the microprocessor crashes completely because of a hardware fault, there are no means to recover from such a failure by software mechanisms. Thus, the intended purpose of this thesis is to reduce the *probability* of failure to an acceptable level, deliberately accepting imperfect fault tolerance.

The bottom line is that the aspect-oriented approach of this thesis is limited to one programming language and specific hardware faults. Therefore, the approach does not supersede hardware-implemented fault tolerance (see Section 2.3) in safety-critical domains such as avionics, but it can provide an additional level of software dependability. However, domains with less demanding reliability requirements can adopt the approach of this thesis as a sole method for mitigating radiation-induced hardware faults as shown in Section 8.1.3.

8.4 FUTURE WORK

The previously described limitations are so fundamental that they cannot be resolved at all. However, this thesis forms the foundation for studying further problems that can be solved to advance the approach of this thesis. There are five directions for future work:

1. *Multi-objective optimization:* The evaluation in Chapter 7 shows that applying the dependability aspects offers a trade-off between fault tolerance, runtime overhead, and memory footprint. For example, Section 7.1.3.1 formulates a rule of thumb that excludes those aspect configurations that cause a slowdown of more than one percent. Such a heuristic is reasonable, yet not

optimal. Thus, there is still potential for improving the effectiveness and efficiency of the dependability aspects.

2. *Run-time system for GOP:* Section 6.5 declares that the dependability aspect GOP prohibits aliasing of member variables by a compile-time assertion. This effectively prevents indirect access to the covered member variables via pointers or C++ references. AspectC++ 2.0 includes preliminary support for capturing indirect memory accesses by the experimental pointcut function `alias()`. On this basis, a run-time system can be implemented that identifies the target object of an indirect memory access so that the respective object can be checked.

3. *Whole-program analysis in AspectC++:* The framework for user-defined whole-program analysis, described in Section 5.4, uses the XML query language *XQuery* to post-process the whole-program information. This step could be integrated into the pointcut description language of AspectC++ as pointcut functions that capture call sequences and control-flow reachability properties [26, p. 5]. Such an extension would not improve the evaluation results of this thesis but would simplify the compilation process (see Figure 5.6 on page 95).

4. *Addressing software bugs:* As pointed out in Section 5.3.2 on page 89, the dependability aspects already detect a variety of software bugs, such as dangling pointers (use-after-free), dereference of uninitialized or null pointers, double frees, and incompatible type casts. Moreover, the checking of array bounds and integer overflows (see Section 6.2 on page 102), which turned out to be poorly suited for the detection of memory errors, address typical software problems. Hence, the approach of this thesis could be extended to a *unified approach* to mitigation of hardware faults and software faults.

5. *Design for dependability aspects:* Finally, after-the-fact hardening of existing operating systems does not tap the full potential of the dependability aspects. For example, the operating system eCos uses global array variables of function-pointer type, whose low-order bits escape the range check of function pointers (see Section 7.1.4.1 on page 150). If these arrays were (static) member variables of a class, they could be covered by GOP. Thus, an operating-system design *aware* of dependability aspects would explore their full potential.

8.5 CHAPTER SUMMARY

This chapter discussed the findings of this thesis and addressed potential threats to validity. First and foremost, Section 8.1 provides evidence for the effectiveness of the dependability aspects by neutron-beam testing at Los Alamos National Laboratory, USA. In fact, the radiation-induced failure rate of eCos is reduced by about 78.2 percent. This improvement leads to a hypothetical increase of one IEC 61508 Safety Integrity Level (SIL). Therefore, Section 8.1 confirms the emulation-based fault-injection results of the previous chapter, which suggests similar effectiveness.

Second, Section 8.2 affirms the excellent maintainability of the aspect-oriented approach. The dependability aspects turn out to be exceedingly crosscutting, yet, the AspectC++ language and compiler avoid any code scattering and tangling. Thus, the library of dependability aspects is highly generic and reusable.

Finally, Section 8.3 recapitulates the two fundamental limitations of the approach: Only the C++ programming language is considered and, in general, perfect fault tolerance is impossible.

9 SUMMARY AND CONCLUSIONS

The reliability of digital semiconductor devices is confronted with transient faults that result from neutron strikes of cosmic origin. There is no effective shielding from neutron radiation – except by several meters of concrete – so that dependable computer systems must incorporate methods of fault tolerance to cope with transient hardware faults. Software-implemented fault tolerance is a particularly promising approach, because no monetary costs for hardware redundancy are involved. Yet, the state-of-the-art in that domain focuses on the application level and neglects the operating system as pointed out in Section 2.4.3. An unreliable operating system, however, can cause catastrophic consequences in safety-critical systems, such as the initially mentioned cases of unintended acceleration of Toyota vehicles (see Chapter 1).

The objective of this thesis is to eliminate the single point of failure represented by the operating system. To this end, a thorough analysis of the eCos and L4/Fiasco.OC operating systems in Chapter 3 identified two central problems:

PROBLEM 1: Several kernel data structures, such as instances of C++ classes, stacks, and pointers, exhibit an exceeding vulnerability to transient memory errors. Integrity of these kernel data structures is crucial for the reliability of the operating systems.

PROBLEM 2: The individual criticality of the kernel data structures depends on the application profile. The lifetime of many kernel data structures is bound to the application programs that use them. As applications change, the individual criticality of the kernel data structures varies dramatically.

A secondary finding of this problem analysis is that runtime overhead has a negative impact on reliability and, therefore, I propose a selective, application-specific placement of fault-tolerance mechanisms to avoid unnecessary overhead. Such an approach is impractical with traditional techniques and tools, raising the primary research question *whether and to which extent Aspect-Oriented Programming (AOP) is a suitable technology for improving the dependability of operating systems*.

This thesis provides extensive answers to that research question and offers insights into the resulting effectiveness and efficiency of the aspect-oriented approach. Thus, this thesis makes three scientific contributions, which are summarized in the following sections.

9.1 CONTRIBUTION 1: ASPECTC++ 2.0 – LANGUAGE EXTENSIONS

My research approach focuses on the AspectC++ programming language, because it has its roots in the development of operating systems. However, the problem analysis in Chapter 3 and my review of prior work in Section 4.5 show that AspectC++ 1.0 is not expressive enough for the implementation of truly generic fault-tolerance mechanisms that cover the critical data structures of both eCos and L4/Fiasco.OC. Thus, I devise four distinct language extensions that advance the degree of expressiveness:

1. Advice for built-in operators
2. Advice for access to variables
3. Generic introductions
4. A framework for user-defined whole-program analysis

Chapter 5 specifies these language extensions in detail, which are implemented the open-source AspectC++ 2.0 compiler that was released in 2016. Thus, the AspectC++ 2.0 language and compiler are available to the public without any restrictions. This enhanced programming language is the first contribution of this thesis. The new language features are completely *general purpose* and, thus, not limited to the domain of software-implemented fault tolerance. As such, this contribution goes beyond the scope of this thesis.

9.2 CONTRIBUTION 2: LIBRARY OF DEPENDABILITY ASPECTS

My second contribution is the presentation of the design and implementation of highly generic and transparent fault-tolerance mechanisms based on the AspectC++ 2.0 technology. I refer to these mechanisms as *dependability aspects*, which constitute a library of the following reusable modules:

1. Symptom detection, such as range checking of function pointers, checking of array bounds, run-time type checking, and checking of integer overflows
2. Return-Address Protection (RAP)
3. Virtual-Function Pointer Protection (VPP)
4. Generic Object Protection (GOP)

For instance, the dependability aspect GOP allows choosing from an extensible toolbox of easily pluggable error-detection and error-correction schemes, such as the adaptive Hamming code and the CRC code that leverages Intel's SSE4.2 instructions.

Furthermore, the second contribution includes the specification and correctness proof of a wait-free synchronization algorithm that enables concurrent error detection in multi-threaded programs. In summary, the dependability aspects show that fine-grained configurability of software-implemented fault tolerance is feasible without affecting the software maintainability: The AspectC++ compiler automatically inserts the fault-tolerance mechanisms into the relevant source-code locations of an operating system as specified by the programmer.

9.3 CONTRIBUTION 3: EVIDENCE FOR EFFECTIVENESS

My third contribution is the thorough quantitative evaluation of the aspect-oriented approach taken in this thesis. The evaluation allows drawing conclusions on AOP in general, on the AspectC++ 2.0 technology in particular, and on the library of dependability aspects. For that purpose, this thesis considers three pieces of large-scale software that are used in a broad range of production systems. Extensive fault-injection experiments indicate that the approach improves the dependability of all three software systems by reducing crashes and silent data corruptions. In summary, the evaluation shows a trade-off between fault tolerance and slowdown, which can be adjusted by a selective placement and combination of the dependability aspects.

1. *eCos*: A combination of three dependability aspects reduces the total number of failures by 77 percent. At the same time, the application-specific configuration limits the total slowdown to one percent.

2. *L4/Fiasco.OC*: The dependability aspects GOP and VPP detect and correct 60 percent of the faults that cause a failure of the unprotected microkernel. Again, the total slowdown is negligible.

3. *Memcached*: The dependability aspects GOP and VPP improve the data integrity by three orders of magnitude and avoid about 50 percent of the crash failures. The slowdown of the Memcached application amounts to 36.3 percent.

The reason for the negligible slowdown of the eCos and L4/Fiasco.OC operating systems is that the kernel is not active all the time, as user-level applications typically get most of the CPU time. Thus, runtime overhead within the kernel increases the total runtime by only a small fraction. On the contrary, the Memcached application shows that the computational overhead is much higher and has a considerable impact on user-level applications.

In addition, the approach of this thesis is validated by accelerated neutron-beam testing that provides evidence for the effectiveness. In fact, the irradiation confirms a failure reduction of eCos by 78.2 percent. This improvement leads to a hypothetical increase of one IEC 61508 Safety Integrity Level (SIL). In conclusion, my third contribution is providing evidence for the hypothesis that AOP is an effective technology for improving the dependability of operating systems.

9.4 FINAL REMARKS

The conclusion of this thesis is that AOP with AspectC++ represents a suitable technology for improving the dependability of operating systems. AspectC++ 2.0 turns out to be effective and efficient in this regard and, furthermore, does not impair the software maintainability. In addition to mitigation of hardware faults, AspectC++ also facilitates the detection of software bugs. This programming language enables the implementation of highly generic software modules for fault tolerance that can be applied automatically to the kernel of an operating system and user-level applications.

This thesis advances thereby the state-of-the-art in the domain of dependable operating systems, documented by 14 peer-reviewed publications in international journals [27, 32, 218], proceedings of conferences [28, 30, 109, 160, 211, 213, 214], and workshops [26, 29, 31, 159].

Although AOP has become a controversial subject of software engineering, this thesis shows that AOP is still poorly understood in other domains of computer science, such as dependability. In this sense, Steimann speculated ten years ago:

> “ I wouldn’t be surprised if AOP ended up being used for something quite different from what it is thought to be good for today. ”
>
> – Friedrich Steimann [237, p. 492]

A

APPENDIX: BASELINE DEPENDABILITY ASSESSMENT

This appendix provides supplemental data of the baseline dependability assessment in Section 3.2 on page 46. That section identifies the most vulnerable memory regions of the eCos and L4/Fiasco.OC operating systems by exhaustive fault-space scans using the fault model of independent single bit flips in memory. Section 3.2.1 on page 47 describes the exact experimental setup of the fault-injection experiments.

The following section presents the fault-injection results of eCos running 13 additional benchmark programs that are not covered in Section 3.2.2 on page 48. Subsequently, the fault-injection results of L4/Fiasco.OC running two additional benchmark programs that are not covered in Section 3.2.3 on page 53 are shown.

Each of the following tables lists the ten most failure-prone symbols, which are identified by exhaustive fault-space scans of the operating systems running the respective benchmark programs. Beside the symbol name and its data type, the tables list the accumulated number of failures per symbol next to the percentage of all failures. Symbols that account for more than ten percent of the failures are exceedingly critical and thus highlighted.

A.1 ECOS

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
`stack`	`cyg_uint64[]`	5,088	$4.82 \cdot 10^5$	30.5 %
`thread_obj`	`Cyg_Thread[]`	288	$3.18 \cdot 10^5$	20.2 %
`Cyg_Scheduler::scheduler`	`Cyg_Scheduler`	132	$1.42 \cdot 10^5$	9.0 %
`comm_channels`	`int (*[][])()`	96	$1.19 \cdot 10^5$	7.5 %
`s2`	`Cyg_Binary_Semaphore`	8	$1.10 \cdot 10^5$	7.0 %
`q`	`cyg_ucount8`	4	$6.62 \cdot 10^4$	4.2 %
`Cyg_Interrupt::dsr_list`	`Cyg_Interrupt*`	4	$6.61 \cdot 10^4$	4.2 %
`s0`	`Cyg_Binary_Semaphore`	8	$6.30 \cdot 10^4$	4.0 %
`cyg_interrupt_stack_base`	`cyg_uint64[]`	4,096	$5.91 \cdot 10^4$	3.7 %
`s1`	`Cyg_Binary_Semaphore`	8	$4.91 \cdot 10^4$	3.1 %

Table A.1: Quantitative fault-injection results of eCos running the benchmark BIN_SEM1, showing the ten most failure-prone symbols

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
stack	cyg_uint64[]	38,176	$2.03 \cdot 10^{12}$	45.8 %
thread_obj	Cyg_Thread[]	2,048	$1.65 \cdot 10^{12}$	37.1 %
chopstick	Cyg_Binary_Semaphore[]	120	$3.07 \cdot 10^{11}$	6.9 %
Cyg_RealTimeClock::rtc	Cyg_RealTimeClock	52	$1.53 \cdot 10^{11}$	3.5 %
Cyg_Scheduler::scheduler	Cyg_Scheduler	132	$5.90 \cdot 10^{10}$	1.3 %
pstate_mutex	Cyg_Mutex	20	$2.47 \cdot 10^{10}$	0.6 %
idle_thread	Cyg_IdleThread	132	$2.25 \cdot 10^{10}$	0.5 %
comm_channels	int (*[][])()	96	$2.05 \cdot 10^{10}$	0.5 %
hal_interrupt_objects	cyg_uint32*[]	896	$2.05 \cdot 10^{10}$	0.5 %
hal_interrupt_handlers	cyg_uint32 (*[])()	896	$2.05 \cdot 10^{10}$	0.5 %

Table A.2: Quantitative fault-injection results of eCos running the benchmark bin_sem2, showing the ten most failure-prone symbols

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
thread_obj	Cyg_Thread[]	288	$1.40 \cdot 10^{11}$	33.1 %
stack	cyg_uint64[]	5,088	$9.09 \cdot 10^{10}$	21.5 %
Cyg_RealTimeClock::rtc	Cyg_RealTimeClock	52	$6.28 \cdot 10^{10}$	14.9 %
Cyg_Scheduler::scheduler	Cyg_Scheduler	132	$2.23 \cdot 10^{10}$	5.3 %
s2	Cyg_Binary_Semaphore	16	$1.71 \cdot 10^{10}$	4.0 %
comm_channels	int (*[][])()	96	$8.53 \cdot 10^{9}$	2.0 %
q	cyg_ucount8	4	$8.53 \cdot 10^{9}$	2.0 %
s1	Cyg_Binary_Semaphore	8	$8.53 \cdot 10^{9}$	2.0 %
s0	Cyg_Binary_Semaphore	8	$8.53 \cdot 10^{9}$	2.0 %
hal_interrupt_objects	cyg_uint32*[]	896	$8.53 \cdot 10^{9}$	2.0 %

Table A.3: Quantitative fault-injection results of eCos running the benchmark bin_sem3, showing the ten most failure-prone symbols

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
stack	cyg_uint64[]	5,088	$7.55 \cdot 10^{5}$	31.8 %
thread_obj	Cyg_Thread[]	288	$5.81 \cdot 10^{5}$	24.4 %
s0	Cyg_Counting_Semaphore	8	$1.77 \cdot 10^{5}$	7.4 %
s2	Cyg_Counting_Semaphore	8	$1.56 \cdot 10^{5}$	6.6 %
comm_channels	int (*[][])()	96	$1.49 \cdot 10^{5}$	6.3 %
s1	Cyg_Counting_Semaphore	8	$1.23 \cdot 10^{5}$	5.2 %
q	cyg_ucount8	4	$9.38 \cdot 10^{4}$	3.9 %
Cyg_Interrupt::dsr_list	Cyg_Interrupt*	4	$9.38 \cdot 10^{4}$	3.9 %
Cyg_Scheduler::scheduler	Cyg_Scheduler	132	$9.09 \cdot 10^{4}$	3.8 %
Cyg_Scheduler::current_thread	Cyg_Thread*[]	4	$6.72 \cdot 10^{4}$	2.8 %

Table A.4: Quantitative fault-injection results of eCos running the benchmark cnt_sem1, showing the ten most failure-prone symbols

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
`stack`	`cyg_uint64[]`	2,544	$7.41 \cdot 10^5$	34.0 %
`comm_channels`	`int (*[][])()`	96	$3.29 \cdot 10^5$	15.1 %
`cyg_interrupt_stack_base`	`cyg_uint64[]`	4,096	$2.96 \cdot 10^5$	13.6 %
`Cyg_Interrupt::dsr_list`	`Cyg_Interrupt*`	4	$1.12 \cdot 10^5$	5.1 %
`Cyg_Scheduler::scheduler`	`Cyg_Scheduler`	132	$1.10 \cdot 10^5$	5.0 %
`idle_thread`	`Cyg_IdleThread`	132	$1.07 \cdot 10^5$	4.9 %
`Cyg_Thread::thread_list`	`Cyg_Thread*`	4	$1.06 \cdot 10^5$	4.9 %
`cyg_libc_main_thread`	`Cyg_Thread`	160	$1.03 \cdot 10^5$	4.7 %
`Cyg_Thread::exception_control`	`Cyg_Exception_Control`	8	$1.02 \cdot 10^5$	4.7 %
`nthreads`	`int`	4	$9.03 \cdot 10^4$	4.1 %

Table A.5: Quantitative fault-injection results of eCos running the benchmark except1, showing the ten most failure-prone symbols

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
`thread_obj`	`Cyg_Thread[]`	416	$3.85 \cdot 10^{11}$	40.9 %
`stack`	`cyg_uint64[]`	7,632	$2.12 \cdot 10^{11}$	22.5 %
`Cyg_RealTimeClock::rtc`	`Cyg_RealTimeClock`	52	$1.23 \cdot 10^{11}$	13.0 %
`Cyg_Scheduler::scheduler`	`Cyg_Scheduler`	132	$4.19 \cdot 10^{10}$	4.4 %
`idle_thread`	`Cyg_IdleThread`	132	$1.83 \cdot 10^{10}$	1.9 %
`hal_interrupt_objects`	`cyg_uint32*[]`	896	$1.71 \cdot 10^{10}$	1.8 %
`hal_interrupt_handlers`	`cyg_uint32 (*[])()`	896	$1.71 \cdot 10^{10}$	1.8 %
`Cyg_Interrupt::dsr_list_tail`	`Cyg_Interrupt*`	4	$1.71 \cdot 10^{10}$	1.8 %
`Cyg_Scheduler::sched_lock`	`cyg_ucount32`	4	$1.71 \cdot 10^{10}$	1.8 %
`comm_channels`	`int (*[][])()`	96	$1.65 \cdot 10^{10}$	1.8 %

Table A.6: Quantitative fault-injection results of eCos running the benchmark flag1, showing the ten most failure-prone symbols

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
`thread_obj`	`Cyg_Thread[]`	288	$1.44 \cdot 10^{11}$	33.0 %
`stack`	`cyg_uint64[]`	5,088	$8.74 \cdot 10^{10}$	20.0 %
`Cyg_RealTimeClock::rtc`	`Cyg_RealTimeClock`	52	$6.75 \cdot 10^{10}$	15.4 %
`m0`	`Cyg_Mbox`	64	$3.25 \cdot 10^{10}$	7.4 %
`Cyg_Scheduler::scheduler`	`Cyg_Scheduler`	132	$2.23 \cdot 10^{10}$	5.1 %
`m2`	`Cyg_Mbox`	64	$1.71 \cdot 10^{10}$	3.9 %
`comm_channels`	`int (*[][])()`	96	$8.53 \cdot 10^9$	1.9 %
`hal_interrupt_objects`	`cyg_uint32*[]`	896	$8.53 \cdot 10^9$	1.9 %
`hal_interrupt_handlers`	`cyg_uint32 (*[])()`	896	$8.53 \cdot 10^9$	1.9 %
`Cyg_Interrupt::dsr_list_tail`	`Cyg_Interrupt*`	4	$8.53 \cdot 10^9$	1.9 %

Table A.7: Quantitative fault-injection results of eCos running the benchmark mbox1, showing the ten most failure-prone symbols

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
stack	cyg_uint64[]	5,088	$1.23 \cdot 10^7$	26.9 %
the_mq	Cyg_Mqueue	44	$8.50 \cdot 10^6$	18.7 %
mempool	char[]	500	$7.32 \cdot 10^6$	16.1 %
thread_obj	Cyg_Thread[]	288	$5.80 \cdot 10^6$	12.7 %
Cyg_Scheduler::scheduler	Cyg_Scheduler	132	$2.52 \cdot 10^6$	5.5 %
t0sem	Cyg_Binary_Semaphore	24	$1.29 \cdot 10^6$	2.8 %
comm_channels	int (*[][])()	96	$1.22 \cdot 10^6$	2.7 %
Cyg_Interrupt::dsr_list	Cyg_Interrupt*	4	$1.15 \cdot 10^6$	2.5 %
Cyg_Scheduler::current_thread	Cyg_Thread*[]	4	$1.06 \cdot 10^6$	2.3 %
storedmempoollen	size_t	4	$1.03 \cdot 10^6$	2.3 %

Table A.8: Quantitative fault-injection results of eCos running the benchmark MQUEUE1, showing the ten most failure-prone symbols

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
stack	cyg_uint64[]	10,176	$1.28 \cdot 10^7$	39.0 %
thread_obj	Cyg_Thread[]	544	$9.52 \cdot 10^6$	29.0 %
Cyg_Scheduler::scheduler	Cyg_Scheduler	132	$3.05 \cdot 10^6$	9.3 %
thread_state	int[]	16	$1.23 \cdot 10^6$	3.7 %
m0	Cyg_Mutex	20	$1.20 \cdot 10^6$	3.7 %
comm_channels	int (*[][])()	96	$8.62 \cdot 10^5$	2.6 %
m1	Cyg_Mutex	20	$8.18 \cdot 10^5$	2.5 %
Cyg_Interrupt::dsr_list	Cyg_Interrupt*	4	$8.13 \cdot 10^5$	2.5 %
cvar2	Cyg_Condition_Variable	8	$7.44 \cdot 10^5$	2.3 %
Cyg_Scheduler::current_thread	Cyg_Thread*[]	24	$7.26 \cdot 10^5$	2.2 %

Table A.9: Quantitative fault-injection results of eCos running the benchmark MUTEX2, showing the ten most failure-prone symbols

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
stack	cyg_uint64[]	5,088	$1.75 \cdot 10^6$	36.6 %
thread_obj	Cyg_Thread[]	288	$1.30 \cdot 10^6$	27.1 %
Cyg_Scheduler::scheduler	Cyg_Scheduler	132	$4.89 \cdot 10^5$	10.3 %
comm_channels	int (*[][])()	96	$2.67 \cdot 10^5$	5.6 %
Cyg_Interrupt::dsr_list	Cyg_Interrupt*	4	$2.28 \cdot 10^5$	4.8 %
s1	Cyg_Binary_Semaphore	8	$2.22 \cdot 10^5$	4.6 %
thread	Cyg_Thread*[]	8	$2.15 \cdot 10^5$	4.5 %
Cyg_Scheduler::current_thread	Cyg_Thread*[]	4	$1.41 \cdot 10^5$	3.0 %
cyg_interrupt_stack_base	cyg_uint64[]	4,096	$5.88 \cdot 10^4$	1.2 %
s0	Cyg_Binary_Semaphore	8	$5.25 \cdot 10^4$	1.1 %

Table A.10: Quantitative fault-injection results of eCos running the benchmark RELEASE, showing the ten most failure-prone symbols

symbol name	data type	size	failures	percentage
stack	cyg_uint64[]	5,088	$1.36 \cdot 10^{5}$	37.2 %
comm_channels	int (*[][])()	96	$5.78 \cdot 10^{4}$	15.8 %
thread_obj	Cyg_Thread[]	288	$5.20 \cdot 10^{4}$	14.2 %
cyg_interrupt_stack_base	cyg_uint64[]	4,096	$3.85 \cdot 10^{4}$	10.6 %
Cyg_Interrupt::dsr_list	Cyg_Interrupt*	4	$1.85 \cdot 10^{4}$	5.1 %
Cyg_Scheduler::scheduler	Cyg_Scheduler	132	$1.43 \cdot 10^{4}$	3.9 %
idle_thread	Cyg_IdleThread	132	$1.06 \cdot 10^{4}$	2.9 %
Cyg_Scheduler::sched_lock	cyg_ucount32	4	$1.04 \cdot 10^{4}$	2.8 %
cyg_libc_main_thread	Cyg_Thread	160	$1.03 \cdot 10^{4}$	2.8 %
Cyg_Thread::thread_list	Cyg_Thread*	4	$9.60 \cdot 10^{3}$	2.6 %

Table A.11: Quantitative fault-injection results of eCos running the benchmark sched1, showing the ten most failure-prone symbols

symbol name	data type	size	failures	percentage
stack	cyg_uint64[]	10,176	$1.72 \cdot 10^{8}$	39.6 %
thread_obj	Cyg_Thread[]	544	$1.11 \cdot 10^{8}$	25.5 %
cs3	Cyg_Counting_Semaphore	8	$1.55 \cdot 10^{7}$	3.6 %
cs1	Cyg_Counting_Semaphore	8	$1.55 \cdot 10^{7}$	3.6 %
cs0	Cyg_Counting_Semaphore	8	$1.54 \cdot 10^{7}$	3.6 %
cs2	Cyg_Counting_Semaphore	8	$1.54 \cdot 10^{7}$	3.5 %
m0	Cyg_Mutex	20	$1.01 \cdot 10^{7}$	2.3 %
s1	Cyg_Binary_Semaphore	8	$9.37 \cdot 10^{6}$	2.2 %
s2	Cyg_Binary_Semaphore	8	$8.59 \cdot 10^{6}$	2.0 %
s0	Cyg_Binary_Semaphore	8	$7.89 \cdot 10^{6}$	1.8 %

Table A.12: Quantitative fault-injection results of eCos running the benchmark sync2, showing the ten most failure-prone symbols

symbol name	data type	size	failures	percentage
thread_obj	Cyg_Thread[]	416	$1.16 \cdot 10^{6}$	30.8 %
stack	cyg_uint64[]	7,632	$8.37 \cdot 10^{5}$	22.2 %
Cyg_Scheduler::scheduler	Cyg_Scheduler	132	$2.83 \cdot 10^{5}$	7.5 %
thread	Cyg_Thread*[]	32	$2.42 \cdot 10^{5}$	6.4 %
m0	Cyg_Mutex	20	$2.17 \cdot 10^{5}$	5.7 %
comm_channels	int (*[][])()	96	$1.68 \cdot 10^{5}$	4.5 %
s1	Cyg_Binary_Semaphore	8	$1.15 \cdot 10^{5}$	3.0 %
Cyg_Interrupt::dsr_list	Cyg_Interrupt*	4	$9.23 \cdot 10^{4}$	2.4 %
cyg_interrupt_stack_base	cyg_uint64[]	4,096	$8.99 \cdot 10^{4}$	2.4 %
s0	Cyg_Binary_Semaphore	8	$8.87 \cdot 10^{4}$	2.4 %

Table A.13: Quantitative fault-injection results of eCos running the benchmark sync3, showing the ten most failure-prone symbols

A.2 L4/FIASCO.OC

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
Physmem	*Heap*	62,914,560	$6.00 \cdot 10^8$	44.0 %
Sched_context::rq	Ready_queue	1,036	$5.58 \cdot 10^8$	40.9 %
Kconsole::_c	Kconsole	56	$3.88 \cdot 10^7$	2.8 %
vga	Vga_console	72	$3.32 \cdot 10^7$	2.4 %
Capabilities	*Heap*	41,943,040	$1.70 \cdot 10^7$	1.2 %
Cpu::cpus	Cpu	224	$1.45 \cdot 10^7$	1.1 %
_fcon	Filter_console	80	$1.34 \cdot 10^7$	1.0 %
the_timeslice_timeout	Timeslice_timeout	28	$1.14 \cdot 10^7$	0.8 %
_kernel_uart	Kernel_uart	16	$9.88 \cdot 10^6$	0.7 %
Timeout_q::timeout_queue	Timeout_q	48	$7.71 \cdot 10^6$	0.6 %

Table A.14: Quantitative fault-injection results of L4/Fiasco.OC running the benchmark CLNTSRV, showing the ten most failure-prone symbols

SYMBOL NAME	DATA TYPE	SIZE	FAILURES	PERCENTAGE
Sched_context::rq	Ready_queue	1,036	$1.11 \cdot 10^9$	48.2 %
Physmem	*Heap*	62,914,560	$9.40 \cdot 10^8$	40.9 %
Kconsole::_c	Kconsole	56	$4.16 \cdot 10^7$	1.8 %
vga	Vga_console	72	$3.58 \cdot 10^7$	1.6 %
Capabilities	*Heap*	41,943,040	$1.93 \cdot 10^7$	0.8 %
Cpu::cpus	Cpu	224	$1.56 \cdot 10^7$	0.7 %
_fcon	Filter_console	80	$1.45 \cdot 10^7$	0.6 %
the_timeslice_timeout	Timeslice_timeout	28	$1.25 \cdot 10^7$	0.5 %
_kernel_uart	Kernel_uart	16	$1.07 \cdot 10^7$	0.5 %
Timeout_q::timeout_queue	Timeout_q	48	$8.43 \cdot 10^6$	0.4 %

Table A.15: Quantitative fault-injection results of L4/Fiasco.OC running the benchmark STREAMMAP, showing the ten most failure-prone symbols

B

APPENDIX: HARDENING ECOS

The second appendix shows the complete fault-injection results of all 18 benchmarks (see Table 7.2 on page 139) used in the case study on *hardening eCos* in Section 7.1 on page 138. In addition, this appendix illustrates the simulated runtime and memory size of each benchmark linked with each protected variant of eCos. In total, this appendix provides the results of 431 combinations of benchmarks and eCos configurations. These configurations include the dependability aspects in form of *symptom detection* (range checking of function pointers, checking of array bounds, run-time type checking, and checking of integer overflows), *Return-Address Protection (RAP)*, *Generic Object Protection (GOP)*, and combinations thereof. A *baseline* variant of eCos, which runs without any protection, serves as a reference in the figures on the following pages.

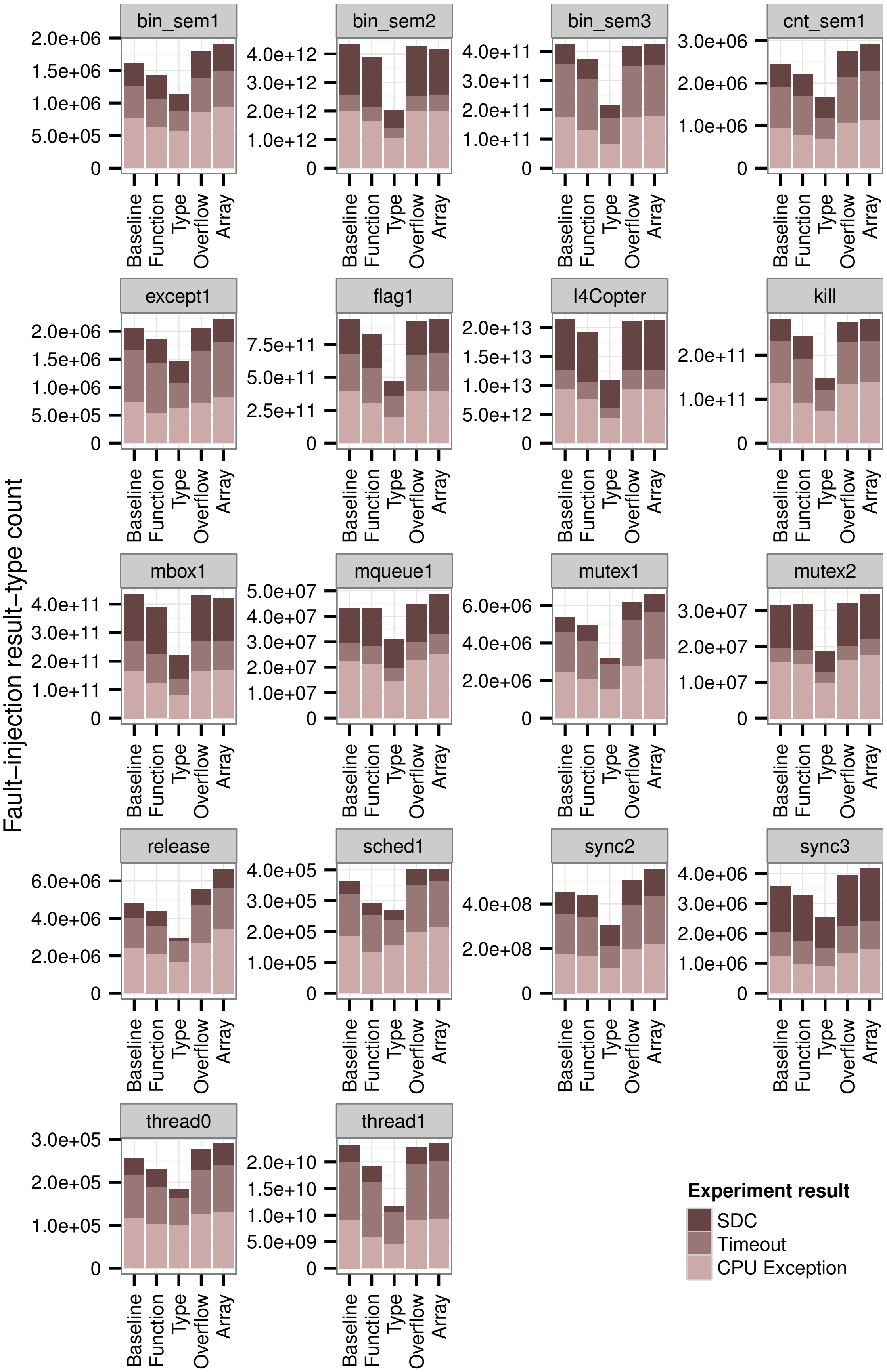


Figure B.1: Fault-injection results of the four *symptom detectors* FUNCTION, TYPE, OVERFLOW, and ARRAY applied to eCos running all 18 benchmarks. These results complement the data shown in Figure 7.1a on page 141.

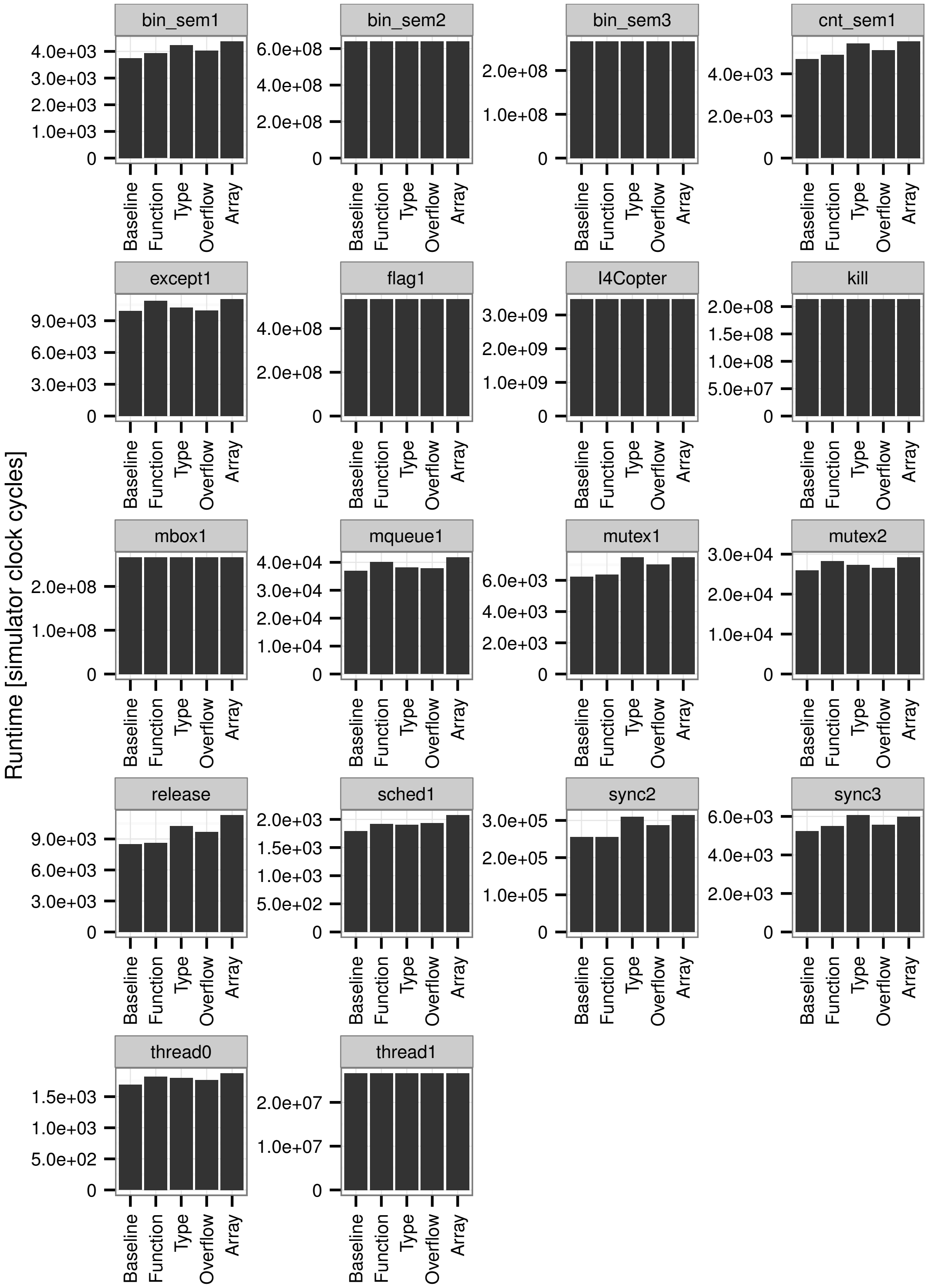


Figure B.2: Simulated runtime of all 18 benchmarks programs running on eCos provided with the four *symptom detectors* FUNCTION, TYPE, OVERFLOW, and ARRAY. These results complement the data shown in Figure 7.1b on page 141.

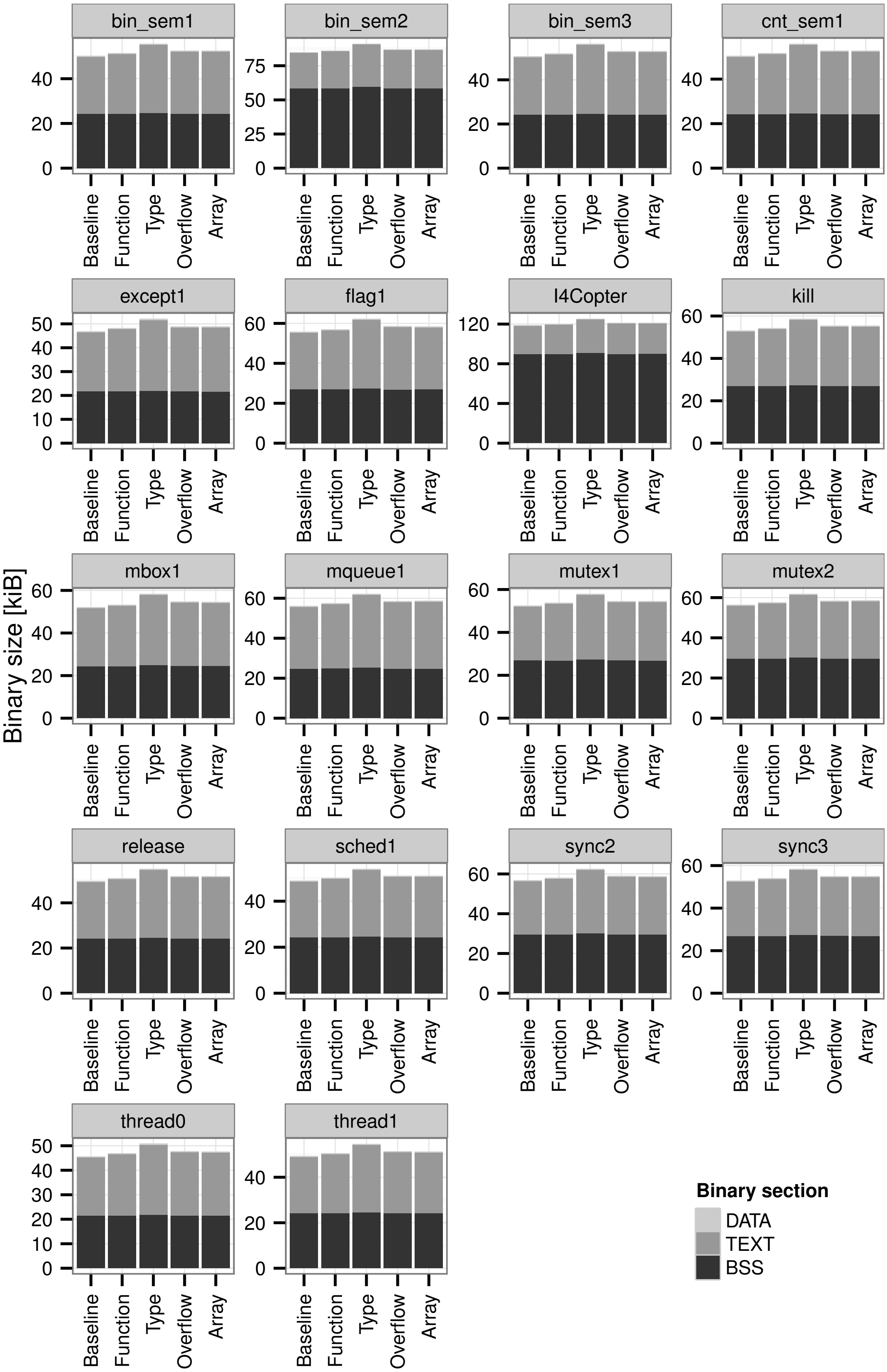


Figure B.3: Memory size of eCos linked with all 18 benchmark applications, provided with the four *symptom detectors* FUNCTION, TYPE, OVERFLOW, and ARRAY. These results complement the data shown in Figure 7.1c on page 141.

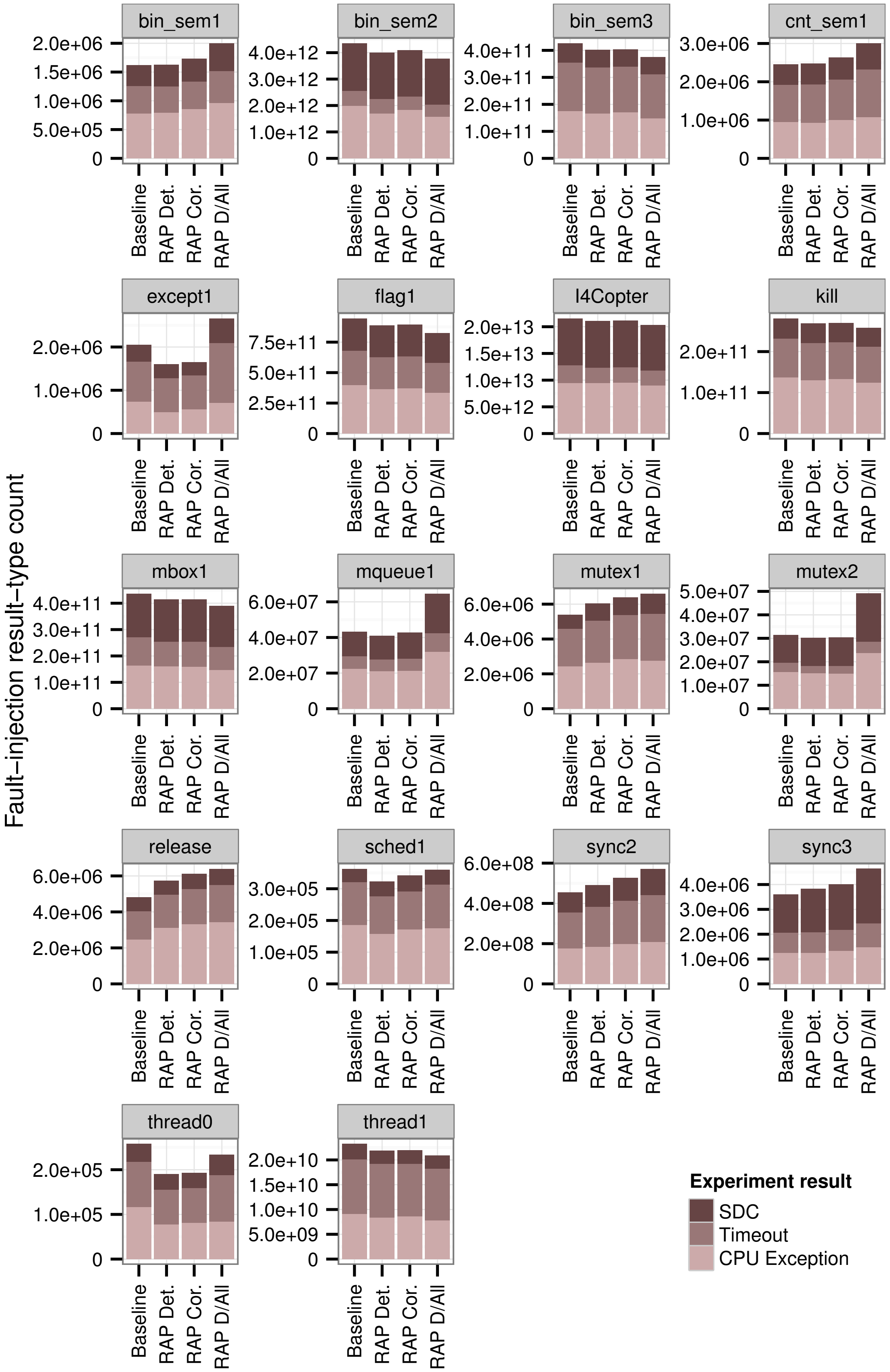


Figure B.4: Fault-injection results of *Return-Address Protection (RAP)* applied to eCos running all 18 benchmarks. The abbreviations *Det.* and *Cor.* denote error detection and error correction, respectively. D/ALL refers to a configuration of error detection without whole-program optimization. These results complement the data shown in Figure 7.2a on page 144.

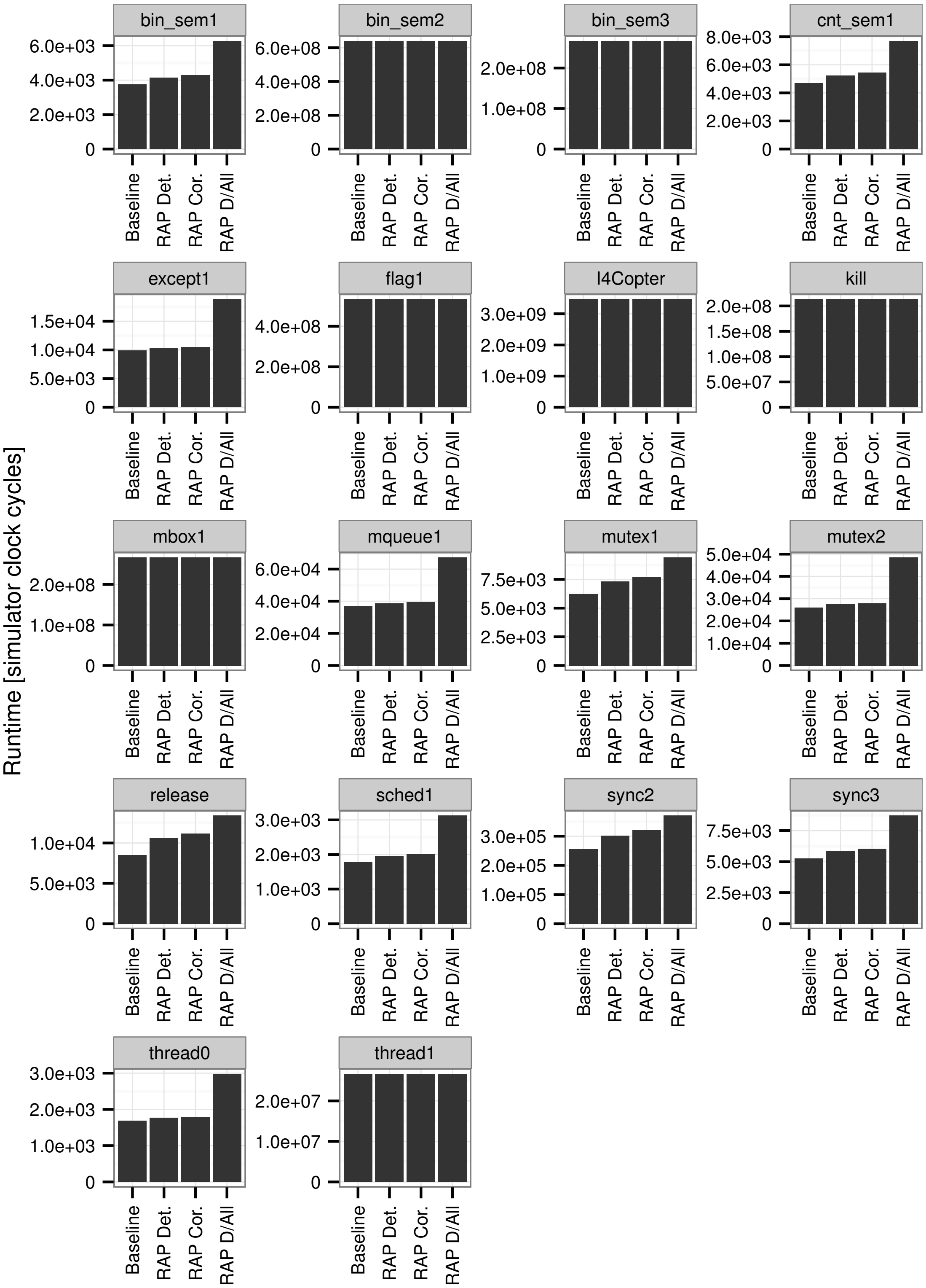


Figure B.5: Simulated runtime of all 18 benchmarks programs running on eCos provided with *Return-Address Protection (RAP)*. The abbreviations *Det.* and *Cor.* denote error detection and error correction, respectively. d/all refers to a configuration of error detection without whole-program optimization. These results complement the data shown in Figure 7.2b on page 144.

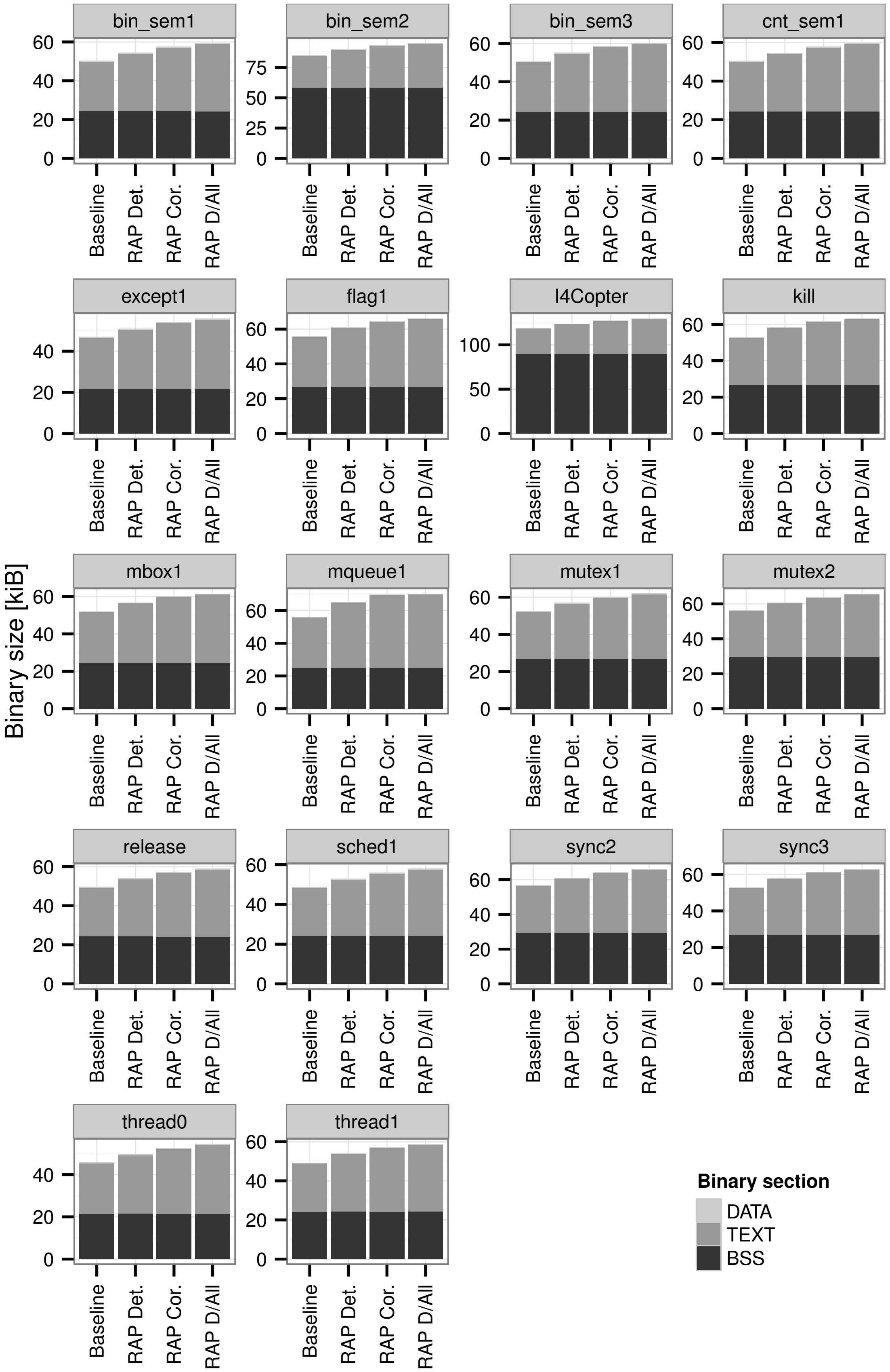


Figure B.6: Memory size of eCos linked with all 18 benchmark applications, provided with *Return-Address Protection (RAP)*. The abbreviations *Det.* and *Cor.* denote error detection and error correction, respectively. D/ALL refers to a configuration of error detection without whole-program optimization. These results complement the data shown in Figure 7.2c on page 144.

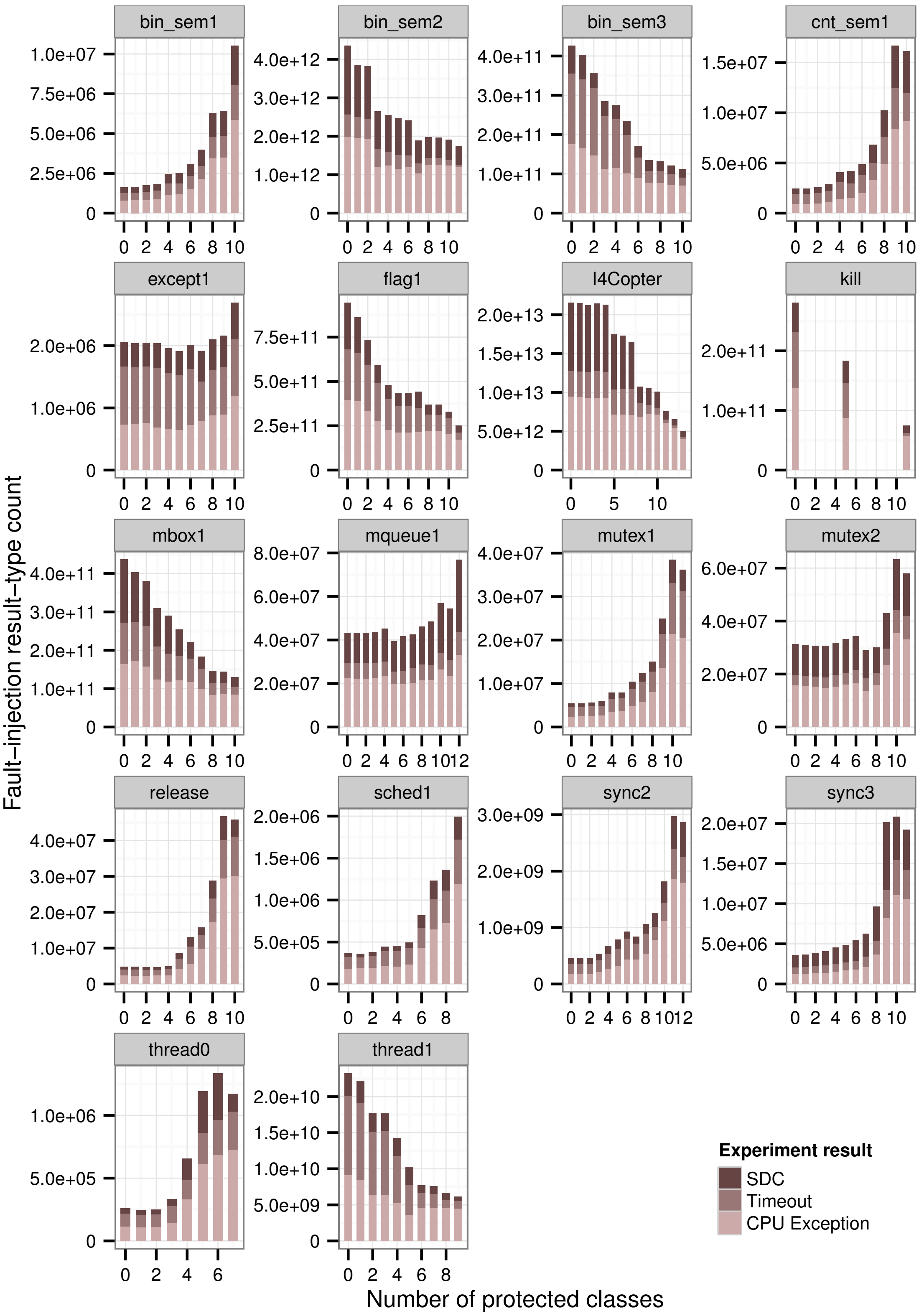


Figure B.7: Fault-injection results of the CRC variant of *Generic Object Protection (GOP)* applied to *increasingly more* kernel data structures of eCos running all 18 benchmarks. These results explore the potential for optimization of GOP and complement the data shown in Figure 7.3a on page 147.

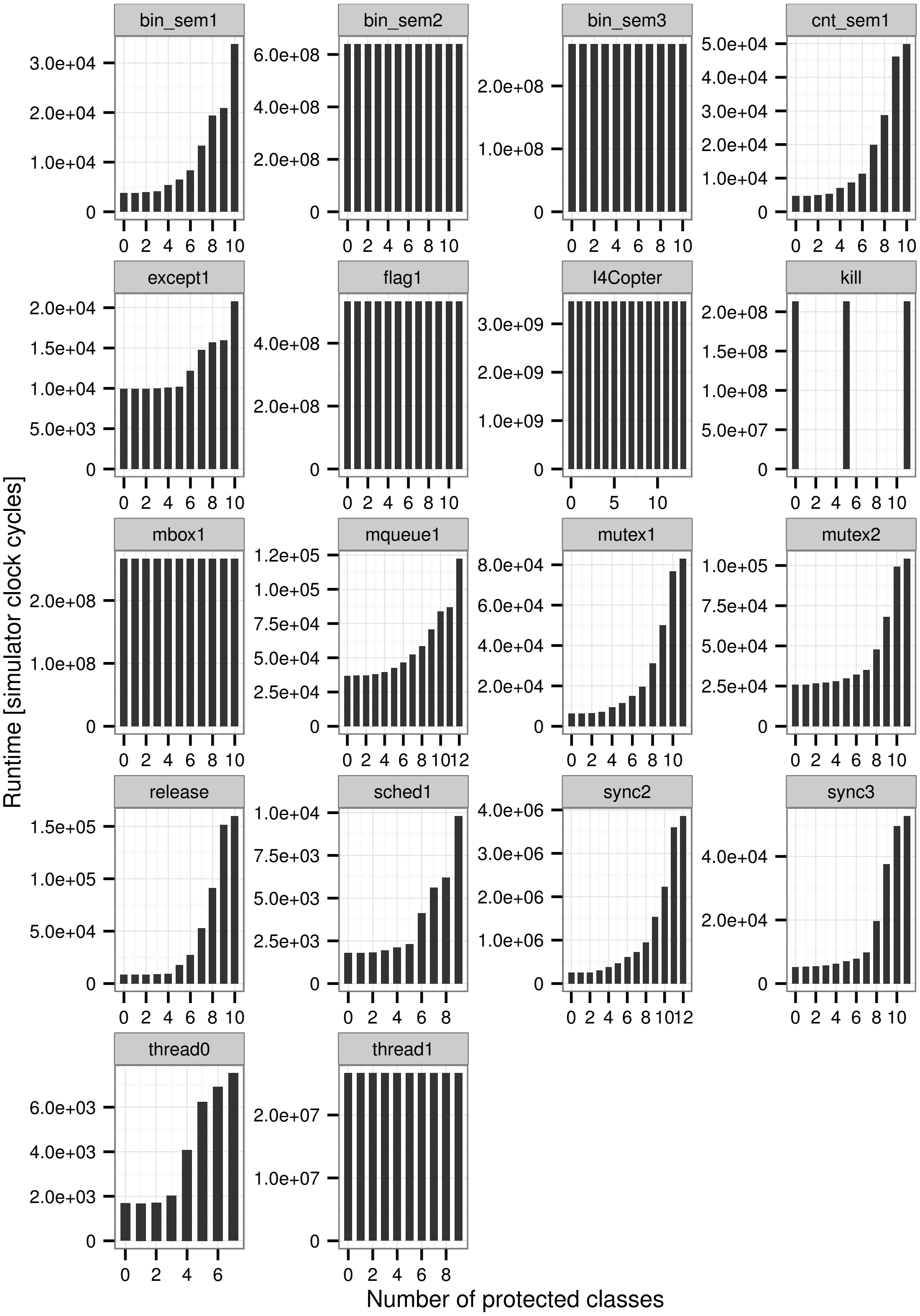


Figure B.8: Simulated runtime of all 18 benchmarks programs running on eCos provided with the CRC variant of *Generic Object Protection (GOP)* applied to *increasingly more* kernel data structures. These results explore the potential for optimization of GOP and complement the data shown in Figure 7.3b on page 147.

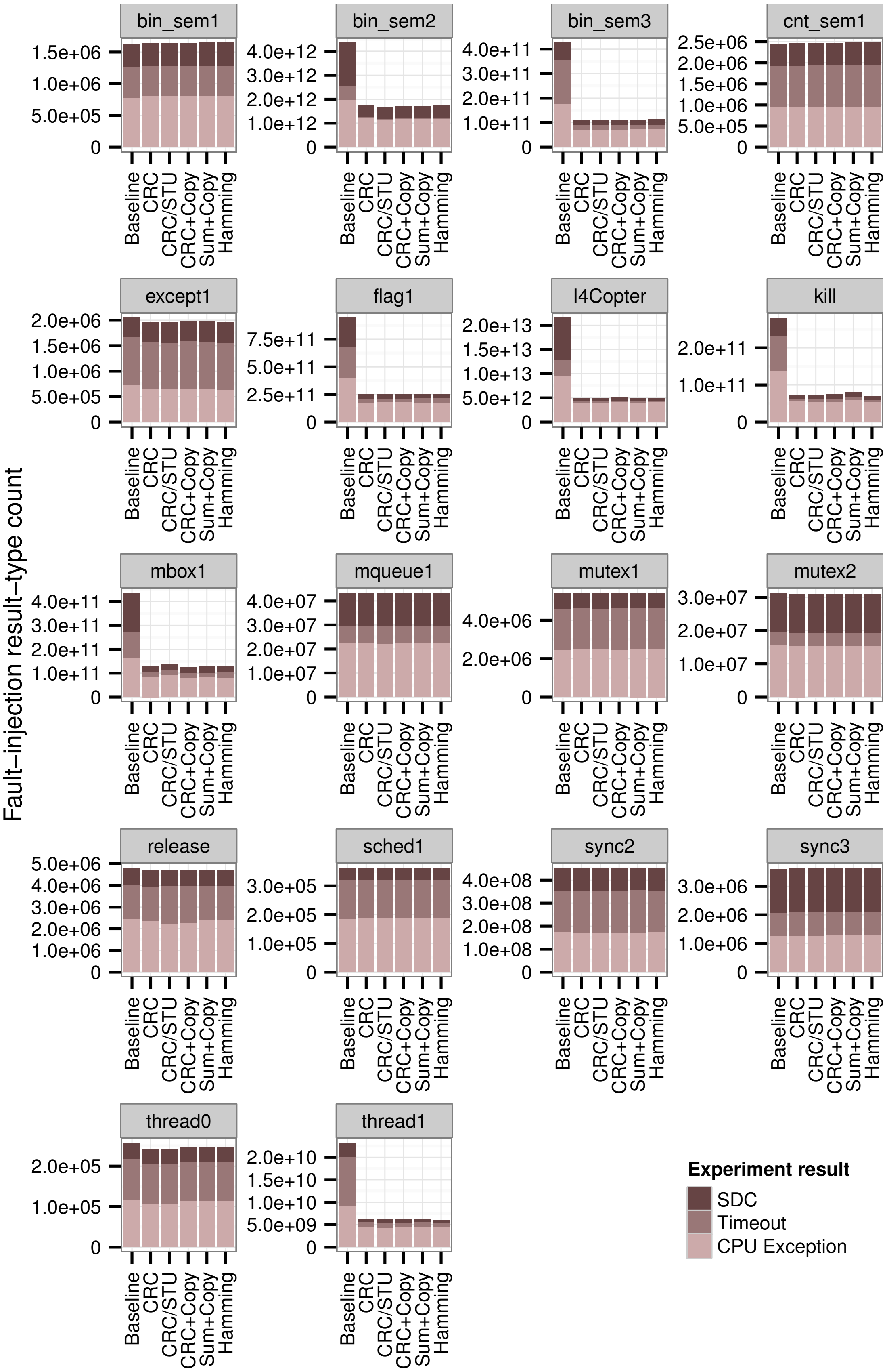


Figure B.9: Fault-injection results of GOP tailored for each benchmark of eCos to keep the slowdown below one percent. The different GOP options for redundancy are described in Table 6.2 on page 119: CRC, CRC+Copy, Sum+Copy, and Hamming. These results complement the data shown in Figure 7.4a on page 148.

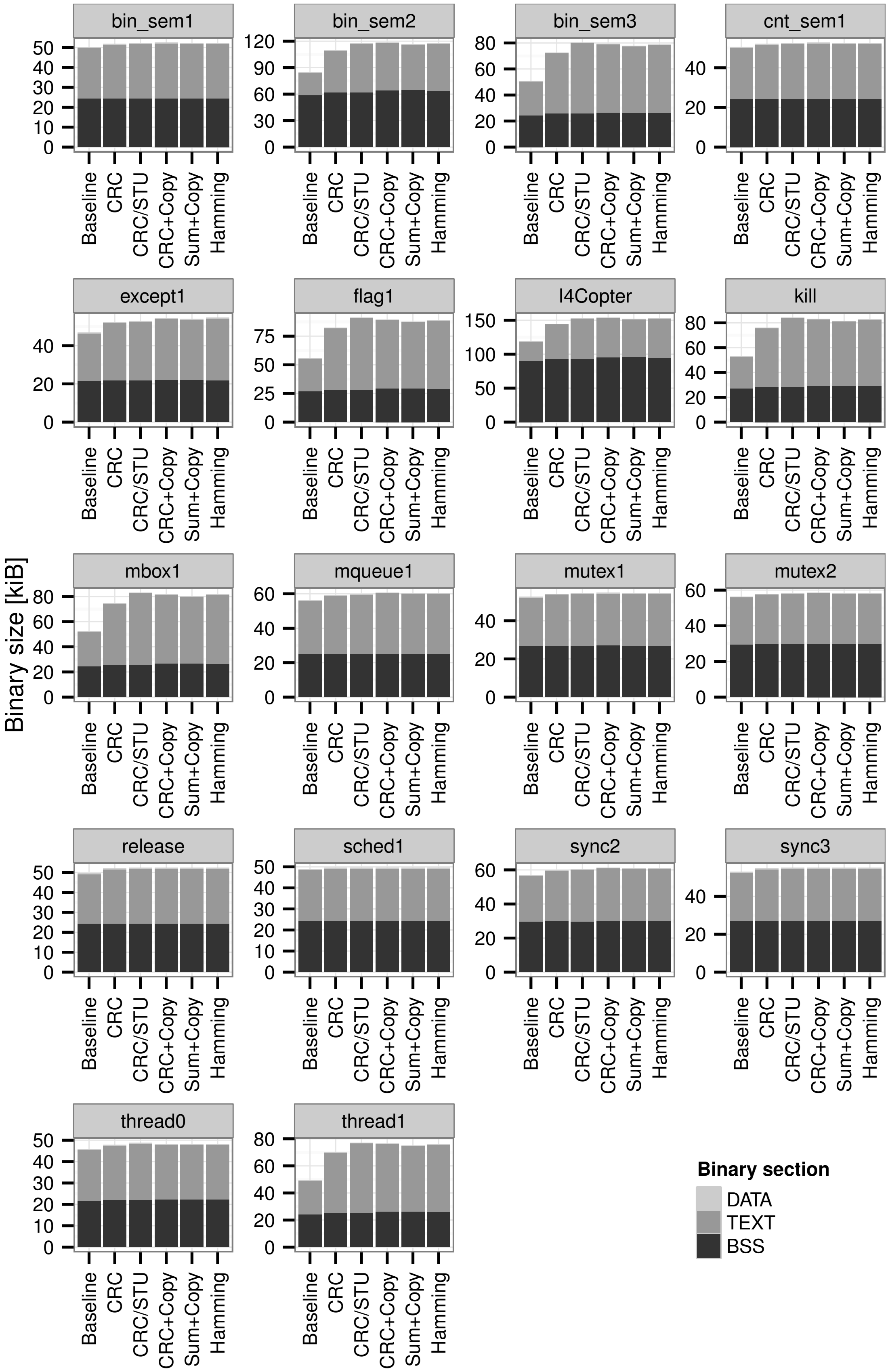


Figure B.10: Memory size of eCos linked with all 18 benchmark applications, provided with GOP tailored for each benchmark to keep the slowdown below one percent. The different GOP options for redundancy are described in Table 6.2 on page 119: CRC, CRC+Copy, Sum+Copy, and Hamming. These results complement the data shown in Figure 7.4b on page 148.

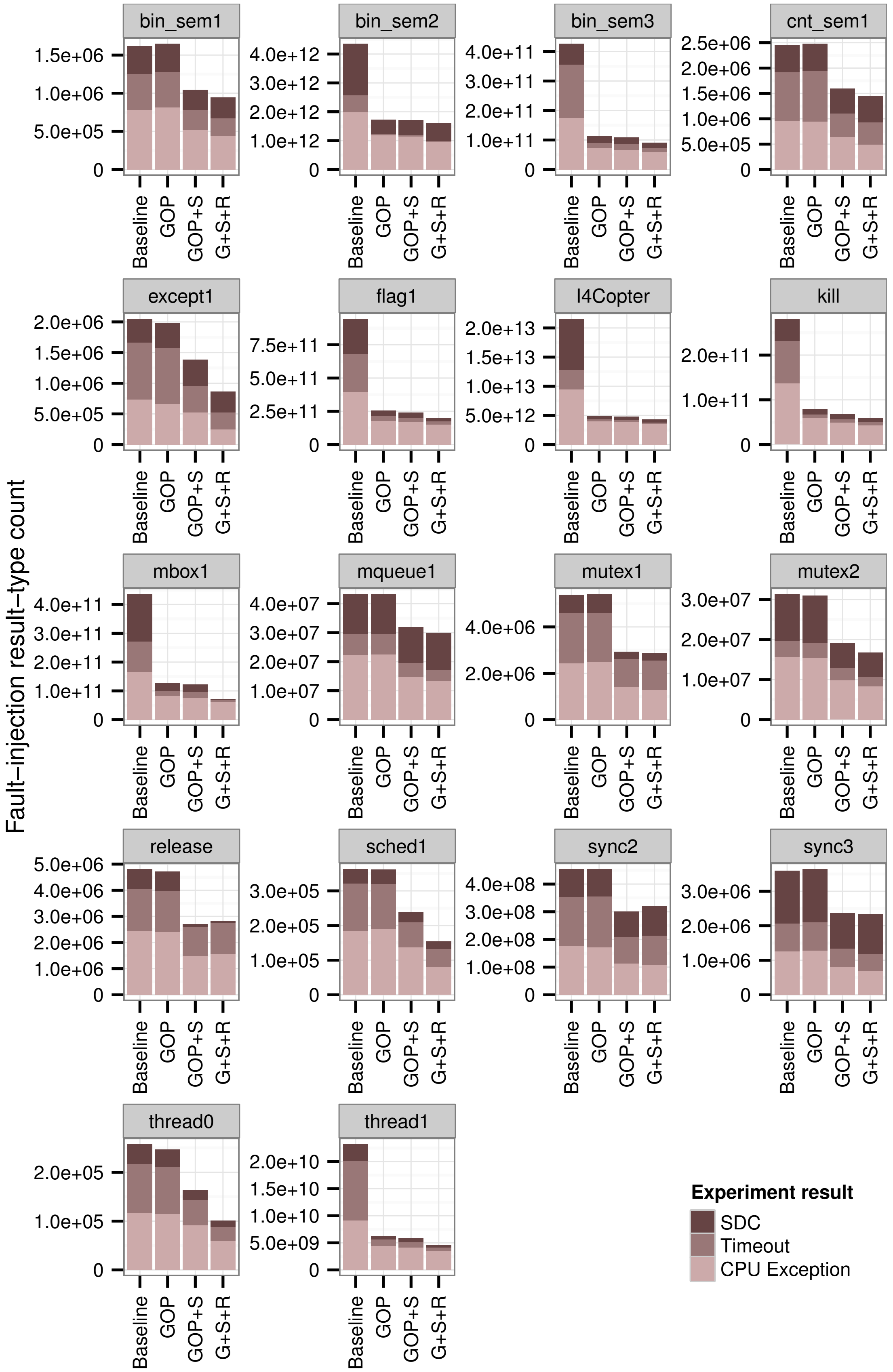


Figure B.11: Fault-injection results of the combined dependability aspects applied to eCos running all 18 benchmark programs. The abbreviation GOP+S means GOP plus symptom detection, whereas G+S+R denotes a combination of GOP, symptom detection, and RAP. These results complement the data shown in Figure 7.5a on page 151.

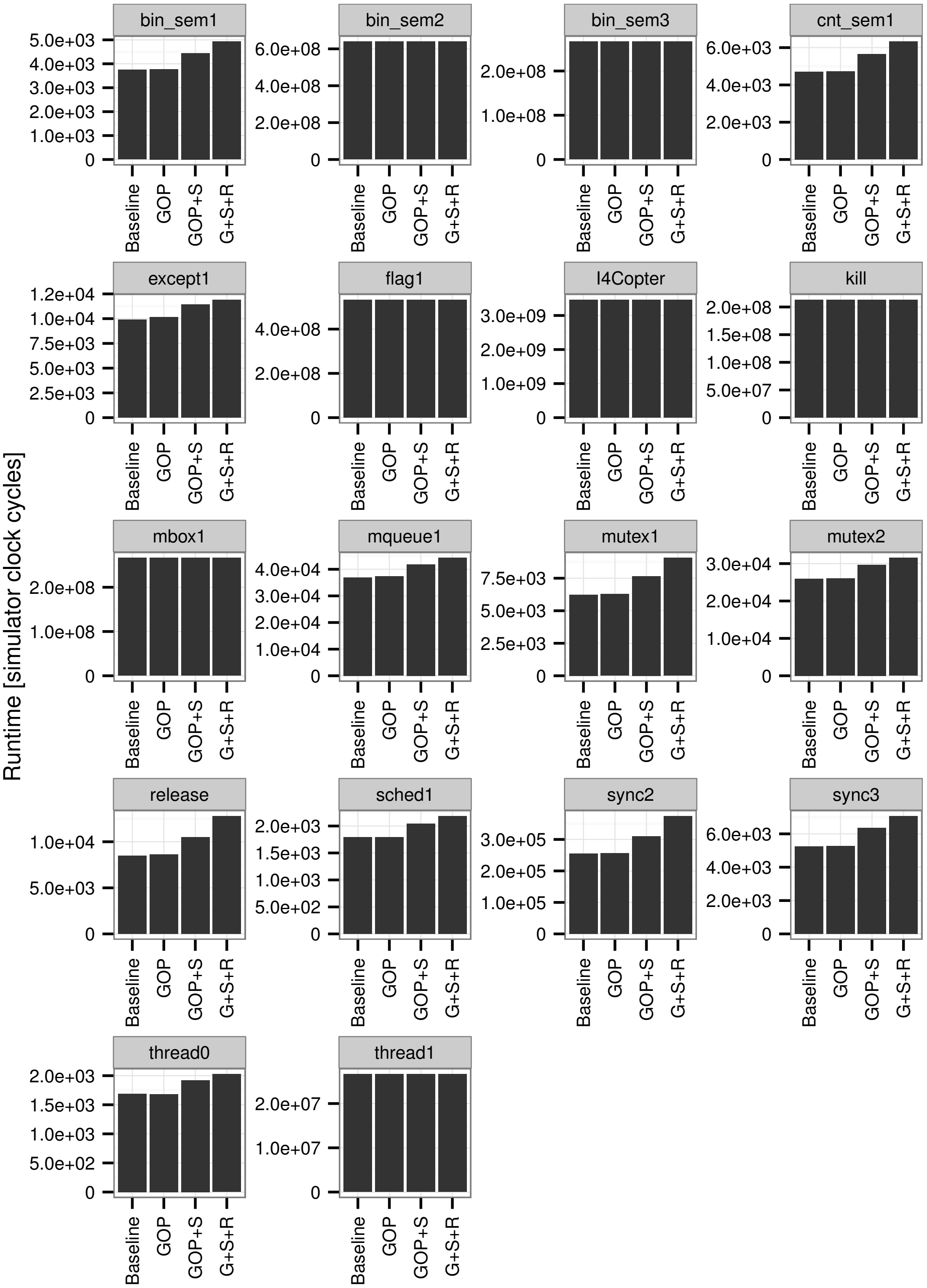


Figure B.12: Simulated runtime of all 18 benchmarks programs running on eCos protected by the combined dependability aspects. The abbreviation GOP+S means GOP plus symptom detection, whereas G+S+R denotes a combination of GOP, symptom detection, and RAP. These results complement the data shown in Figure 7.5b on page 151.

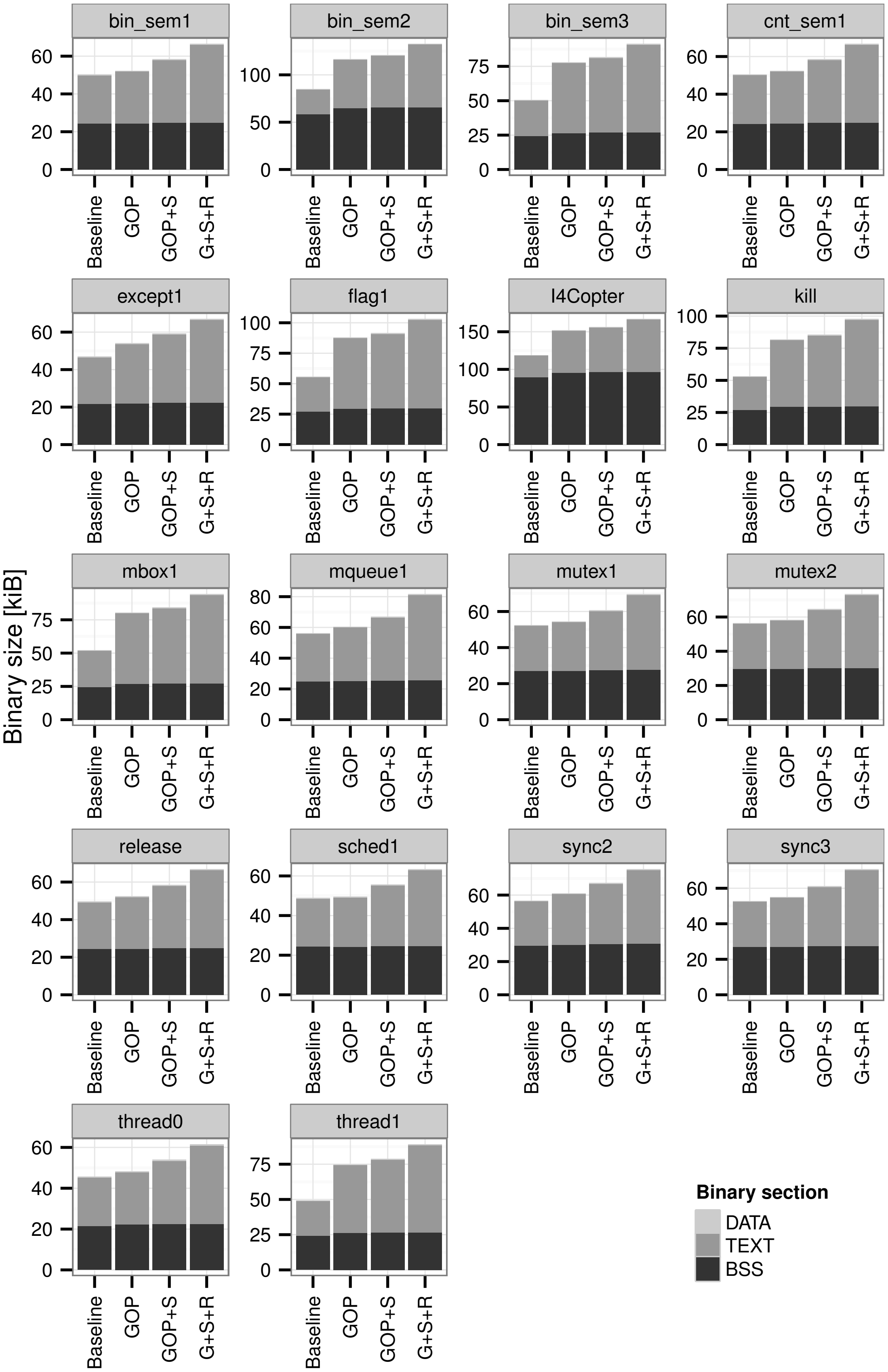


Figure B.13: Memory size of eCos linked with all 18 benchmark applications, protected by the combined dependability aspects. The abbreviation GOP+S means GOP plus symptom detection, whereas G+S+R denotes a combination of GOP, symptom detection, and RAP. These results complement the data shown in Figure 7.5c on page 151.

LIST OF FIGURES

Figure 2.1 Cross section through a MOSFET 11

Figure 2.2 Particle strike at a MOSFET 12

Figure 2.3 Equivalent circuit diagram for a DRAM cell 14

Figure 2.4 Pathology of failure 18

Figure 2.5 Taxonomy of software-implemented hardware fault tolerance 25

Figure 3.1 Fault space of a program 45

Figure 3.2 Excerpt of the fault space spanned by eCos running the thread1 benchmark 49

Figure 3.3 Fault space spanned by eCos running the mutex1 benchmark 52

Figure 3.4 Memory layout of C++ objects 54

Figure 4.1 Scattering and tangling in a linked list 61

Figure 4.2 Schematic of aspect weaving 63

Figure 4.3 Implementation of a linked list with error detection using AspectC++ 65

Figure 4.4 I4Copter software architecture 69

Figure 5.1 Range check of function pointers 80

Figure 5.2 Bounds check of array variables 85

Figure 5.3 Run-time type checking 90

Figure 5.4 Counting the number of virtual functions 92

Figure 5.5 AspectC++ metamodel of a translation unit 93

Figure 5.6 Whole-program analysis using AspectC++ 95

Figure 5.7 Control-flow reachability analysis in XQuery 96

Figure 6.1 Interface of the dependability aspects 101

Figure 6.2 Integer-overflow checking 104

Figure 6.3 Assembler instructions for overflow checking 105

Figure 6.4 Layout of stack frames 107

Figure 6.5 Return-Address Protection 108

Figure 6.6 Call-stack histogram of eCos running the benchmark program mutex1 110

Figure 6.7 Virtual-function pointers and virtual-function tables 112

Figure 6.8 Virtual-Function Pointer Protection 113

Figure 6.9 Call graph initiated by a virtual-function call 114

Figure 6.10 Generic Object Protection 117

Figure 6.11 Design of Generic Object Protection 118

Figure 6.12 Call-based optimization of GOP 123

Figure 6.13 Whole-program optimization of GOP 125

Figure 6.14 Wait-free synchronization algorithm 129

Figure 6.15 Slowdown caused by synchronization 134

Figure 7.1 Quantitative evaluation of the symptom detectors applied to eCos 141
Figure 7.2 Quantitative evaluation of RAP applied to eCos 144
Figure 7.3 Exploring the potential for optimization of GOP 147
Figure 7.4 Quantitative evaluation of GOP applied to eCos 148
Figure 7.5 Quantitative evaluation of the combined dependability aspects applied to eCos 151
Figure 7.6 Quantitative evaluation of the VPP and GOP dependability aspects applied to L4/Fiasco.OC 156
Figure 7.7 Fault injection into the virtual memory of a Linux process 159
Figure 7.8 Quantitative fault-injection results of Memcached 162
Figure 7.9 Runtime of the Memcached server 163
Figure 8.1 Experimental setup mounted at ICE House II 170
Figure 8.2 Failure rates of eCos at sea level 172
Figure 8.3 Scattering and tangling as avoided by AOP 174
Figure B.1 Fault-injection results of the *symptom detectors* applied to eCos running all 18 benchmarks 190
Figure B.2 Simulated runtime of all 18 benchmarks programs running on eCos provided with the *symptom detectors* 191
Figure B.3 Memory size of eCos linked with all 18 benchmark applications, provided with the *symptom detectors* 192
Figure B.4 Fault-injection results of RAP applied to eCos running all 18 benchmarks 193
Figure B.5 Simulated runtime of all 18 benchmarks programs running on eCos provided with RAP 194
Figure B.6 Memory size of eCos linked with all 18 benchmark applications, provided with RAP 195
Figure B.7 Fault-injection results of the CRC variant of GOP applied to *increasingly more* kernel data structures of eCos running all 18 benchmarks 196
Figure B.8 Simulated runtime of all 18 benchmarks programs running on eCos provided with the CRC variant of GOP applied to *increasingly more* kernel data structures 197
Figure B.9 Fault-injection results of GOP tailored for each benchmark of eCos 198
Figure B.10 Memory size of eCos linked with all 18 benchmark applications, provided with GOP tailored for each benchmark 199
Figure B.11 Fault-injection results of the combined dependability aspects applied to eCos running all 18 benchmark programs 200

Figure B.12 Simulated runtime of all 18 benchmarks programs running on eCos protected by the combined dependability aspects 201
Figure B.13 Memory size of eCos linked with all 18 benchmark applications, protected by the combined dependability aspects 202

LIST OF TABLES

Table 2.2 Overview of fault rates caused by atmospheric radiation at sea level 16
Table 2.3 Comparison of approaches to software-implemented hardware fault tolerance 35
Table 3.2 Binomial probabilities 42
Table 3.3 Quantitative fault-injection results of eCos running the THREAD1 benchmark 50
Table 3.4 Quantitative fault-injection results of eCos running the MUTEX1 benchmark 51
Table 3.5 Quantitative fault-injection results of L4/Fiasco.OC running the UIRQ program 54
Table 4.2 Excerpt from the AspectC++ join-point API 68
Table 4.3 Quantitative fault-injection results of eCos and CiAO 70
Table 5.3 Specification of the pointcut function `builtin` 79
Table 5.4 Specification of the pointcut functions `get`, `set`, and `ref` 82
Table 5.5 Extension to the join-point API of AspectC++ 83
Table 5.6 Excerpt from the join-point API of introductions 88
Table 6.2 GOP configuration options for redundancy 119
Table 7.2 Benchmark programs of the eCos kernel test suite 139
Table 7.3 Remaining failures of eCos 152
Table 7.4 Example and test programs of L4Re 154
Table 7.5 Static binary size of the L4/Fiasco.OC kernel images 157
Table 8.2 Raw result data of the neutron-beam testing 171
Table A.1 Quantitative fault-injection results of eCos running the benchmark BIN_SEM1 183
Table A.2 Quantitative fault-injection results of eCos running the benchmark BIN_SEM2 184
Table A.3 Quantitative fault-injection results of eCos running the benchmark BIN_SEM3 184

Table A.4 Quantitative fault-injection results of eCos running the benchmark CNT_SEM1 184
Table A.5 Quantitative fault-injection results of eCos running the benchmark EXCEPT1 185
Table A.6 Quantitative fault-injection results of eCos running the benchmark FLAG1 185
Table A.7 Quantitative fault-injection results of eCos running the benchmark MBOX1 185
Table A.8 Quantitative fault-injection results of eCos running the benchmark MQUEUE1 186
Table A.9 Quantitative fault-injection results of eCos running the benchmark MUTEX2 186
Table A.10 Quantitative fault-injection results of eCos running the benchmark RELEASE 186
Table A.11 Quantitative fault-injection results of eCos running the benchmark SCHED1 187
Table A.12 Quantitative fault-injection results of eCos running the benchmark SYNC2 187
Table A.13 Quantitative fault-injection results of eCos running the benchmark SYNC3 187
Table A.14 Quantitative fault-injection results of L4/Fiasco.OC running the benchmark CLNTSRV 188
Table A.15 Quantitative fault-injection results of L4/Fiasco.OC running the benchmark STREAMMAP 188

BIBLIOGRAPHY

[1] R. A. Åberg, J. L. Lawall, M. Südholt, G. Muller, and A. L. Meur. On the automatic evolution of an OS kernel using temporal logic and AOP. In *Proceedings of the 18th IEEE International Conference on Automated Software Engineering (ASE '03)*, pages 196–204, Piscataway, NJ, USA, Oct. 2003. IEEE Press. doi: 10.1109/ASE.2003.1240307. (Cited on page 72.)

[2] B. Adams. *Co-evolution of Source Code and the Build System: Impact on the Introduction of AOSD in Legacy Systems*. PhD thesis, Ghent University, Ghent, Belgium, May 2008. (Cited on page 72.)

[3] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, Dec. 1996. doi: 10.1109/2.546611. (Cited on page 132.)

[4] F. Afonso, C. Silva, S. Montenegro, and A. Tavares. Applying aspects to a real-time embedded operating system. In *Proceedings of the 6th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '07)*, New York, NY, USA, Mar. 2007. ACM Press. doi: 10.1145/1233901.1233902. (Cited on pages 3 and 75.)

[5] F. Afonso, C. Silva, N. Brito, S. Montenegro, and A. Tavares. Aspect-oriented fault tolerance for real-time embedded systems. In *Proceedings of the 7th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (AOSD-ACP4IS '08)*, pages 2:1–2:8, New York, NY, USA, Mar. 2008. ACM Press. doi: 10.1145/1404891.1404893. (Cited on pages 3 and 75.)

[6] J. Aidemark, J. Vinter, P. Folkesson, and J. Karlsson. Experimental evaluation of time-redundant execution for a brake-by-wire application. In *Proceedings of the 32nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '02)*, pages 210–218, Piscataway, NJ, USA, June 2002. IEEE Press. doi: 10.1109/DSN.2002.1028902. (Cited on pages 30 and 35.)

[7] R. Alexandersson and J. Karlsson. Fault injection-based assessment of aspect-oriented implementation of fault tolerance. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11)*, pages 303–314, Piscataway, NJ, USA, June 2011. IEEE Press. doi: 10.1109/DSN.2011.5958244. (Cited on pages 3, 28, 30, 35, and 75.)

[8] R. Alexandersson and P. Öhman. Implementing fault tolerance using aspect oriented programming. In *Proceedings of the 3rd Latin-American Conference on Dependable Computing (LADC '07)*, pages 57–74, Berlin, Germany, Sept. 2007. Springer. doi: 10.1007/978-3-540-75294-3_6. (Cited on pages 3 and 75.)

[9] Z. Alkhalifa, V. S. S. Nair, N. Krishnamurthy, and J. A. Abraham. Design and evaluation of system-level checks for on-line control flow error detection. *IEEE Transactions on Parallel and Distributed Systems*, 10(6):627–641, June 1999. doi: 10.1109/71.774911. (Cited on pages 28 and 35.)

[10] F. E. Allen. Control flow analysis. In *Proceedings of a Symposium on Compiler Optimization*, pages 1–19, New York, NY, USA, July 1970. ACM Press. doi: 10.1145/800028.808479. (Cited on page 94.)

[11] I. Aracic, V. Gasiunas, M. Mezini, and K. Ostermann. Overview of CaesarJ. In A. Rashid and M. Akşit, editors, *Transactions on AOSD I*, volume 3880 of *Lecture Notes in Computer Science*, pages 135–173. Springer, Berlin, Germany, 2006. doi: 10.1007/11687061_5. (Cited on page 71.)

[12] J. Arlat, M. Aguera, L. Amat, Y. Crouzet, J.-C. Fabre, J.-C. Laprie, E. Martins, and D. Powell. Fault injection for dependability validation: A methodology and some applications. *IEEE Transactions on Software Engineering*, 16(2):166–182, Feb. 1990. doi: 10.1109/32.44380. (Cited on page 40.)

[13] J. Arlat, Y. Crouzet, J. Karlsson, P. Folkesson, E. Fuchs, and G. H. Leber. Comparison of physical and software-implemented fault injection techniques. *IEEE Transactions on Computers*, 52(9):1115–1133, Sept. 2003. doi: 10.1109/TC.2003.1228509. (Cited on page 40.)

[14] G. Attardi and A. Cisternino. Reflection support by means of template metaprogramming. In *Proceedings of the 3rd International Conference on Generative and Component-Based Software Engineering (GCSE '01)*, pages 118–127, London, UK, Sept. 2001. Springer. doi: 10.1007/3-540-44800-4_11. (Cited on page 89.)

[15] Y. Aumann and M. A. Bender. Fault tolerant data structures. In *Proceedings of the 37th Annual Symposium on Foundations of Computer Science (FOCS '96)*, pages 580–589, Piscataway, NJ, USA, Oct. 1996. IEEE Press. doi: 10.1109/SFCS.1996.548517. (Cited on pages 33 and 35.)

[16] J. L. Autran, P. Roche, S. Sauze, G. Gasiot, D. Munteanu, P. Loaiza, M. Zampaolo, and J. Borel. Altitude and underground real-time SER characterization of CMOS 65 nm SRAM. *IEEE Transactions on Nuclear Science*, 56(4):2258–2266, Aug. 2009. doi: 10.1109/TNS.2009.2012426. (Cited on pages 13 and 41.)

[17] A. Avižienis. The N-version approach to fault-tolerant software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, Dec. 1985. doi: 10.1109/TSE.1985.231893. (Cited on page 30.)

[18] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic concepts and taxonomy of dependable and secure computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, Jan. 2004. doi: 10.1109/TDSC.2004.2. (Cited on pages 17, 18, 19, 99, 167, and 173.)

[19] D. F. Bacon, S. L. Graham, and O. J. Sharp. Compiler transformations for high-performance computing. *ACM Computing Surveys*, 26(4):345–420, Dec. 1994. doi: 10.1145/197405.197406. (Cited on pages 86, 105, and 109.)

[20] P. Banerjee, J. T. Rahmeh, C. Stunkel, V. S. Nair, K. Roy, V. Balasubramanian, and J. A. Abraham. Algorithm-based fault tolerance on a hypercube multiprocessor. *IEEE Transactions on Computers*, 39(9):1132–1145, Sept. 1990. doi: 10.1109/12.57055. (Cited on pages 33 and 35.)

[21] J. G. P. Barnes. Ada. In C. R. Spitzer, editor, *Avionics: Elements, Software and Functions*, The Avionics Handbook, chapter 15, pages 15:1–15:50. CRC Press, Boca Raton, FL, USA, second edition, Dec. 2006. (Cited on page 73.)

[22] M. Barr. *Programming Embedded Systems in C and C++*. O'Reilly, Sebastopol, CA, USA, 1999. (Cited on pages 27 and 47.)

[23] R. C. Baumann. Radiation-induced soft errors in advanced semiconductor technologies. *IEEE Transactions on Device and Materials Reliability*, 5(3):305–316, Sept. 2005. doi: 10.1109/TDMR.2005.853449. (Cited on pages 10 and 11.)

[24] R. C. Baumann. Soft errors in advanced computer systems. *IEEE Design & Test of Computers*, 22(3):258–266, May 2005. doi: 10.1109/MDT.2005.69. (Cited on pages 9, 10, and 12.)

[25] A. Benso, S. Chiusano, P. Prinetto, and L. Tagliaferri. A C/C++ source-to-source compiler for dependable applications. In *Proceedings of the 30th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '00)*, pages 71–78, Piscataway, NJ, USA, June 2000. IEEE Press. doi: 10.1109/ICDSN.2000.857517. (Cited on pages 2, 26, 35, and 41.)

[26] C. Borchert and O. Spinczyk. Hardening an L4 microkernel against soft errors by aspect-oriented programming and whole-program analysis. In *Proceedings of the 8th Workshop on Programming Languages and Operating Systems (PLOS '15)*, pages 1–7, New York, NY, USA, Oct. 2015. ACM Press. doi: 10.1145/2818302.2818304. (Cited on pages v, 7, 77, 177, and 182.)

[27] C. Borchert and O. Spinczyk. Hardening an L4 microkernel against soft errors by aspect-oriented programming and whole-program analysis. *ACM Operating Systems Review*, 49(2):37–43, Jan. 2016. doi: 10.1145/2883591.2883600. (Cited on pages v, 77, and 182.)

[28] C. Borchert, D. Lohmann, and O. Spinczyk. CiAO/IP: A highly configurable aspect-oriented IP stack. In *Proceedings of the 10th International Conference on Mobile Systems, Applications, and Services (MobiSys '12)*, pages 435–448, New York, NY, USA, June 2012. ACM Press. doi: 10.1145/2307636.2307676. (Cited on pages v, 7, 59, 69, and 182.)

[29] C. Borchert, H. Schirmeier, and O. Spinczyk. Protecting the dynamic dispatch in C++ by dependability aspects. In *Proceedings of the 1st GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*, Lecture Notes in Informatics, pages 521–535, Bonn, Germany, Sept. 2012. German Society of Informatics. URL: `http://subs.emis.de/LNI/Proceedings/Proceedings208/521.pdf`. (Cited on pages v, 7, 100, and 182.)

[30] C. Borchert, H. Schirmeier, and O. Spinczyk. Generative software-based memory error detection and correction for operating system data structures. In *Proceedings of the 43rd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '13)*, Piscataway, NJ, USA, June 2013. IEEE Press. doi: 10.1109/DSN.2013.6575308. (Cited on pages v, 6, 7, 39, and 182.)

[31] C. Borchert, H. Schirmeier, and O. Spinczyk. Return-address protection in C/C++ code by dependability aspects. In *Proceedings of the 2nd GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*, Lecture Notes in Informatics, pages 2519–2533, Bonn, Germany, Sept. 2013. German Society of Informatics. URL: `http://subs.emis.de/LNI/Proceedings/Proceedings220/2519.pdf`. (Cited on pages v, 6, 7, 100, and 182.)

[32] C. Borchert, H. Schirmeier, and O. Spinczyk. Generic soft-error detection and correction for concurrent data structures. *IEEE Transactions on Dependable and Secure Computing*, 14(1):22–36, Jan. 2017. doi: 10.1109/TDSC.2015.2427832. (Cited on pages v, 6, 7, 100, 127, 153, and 182.)

[33] S. Y. Borkar. Designing reliable systems from unreliable components: The challenges of transistor variability and degradation. *IEEE Micro*, 25(6):10–16, Nov. 2005. doi: 10.1109/MM.2005.110. (Cited on pages 9 and 10.)

[34] L. Borucki, G. Schindlbeck, and C. Slayman. Comparison of accelerated DRAM soft error rates measured at component and system level. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS '08)*, pages 482–487, Piscataway, NJ, USA, Apr. 2008. IEEE Press. doi: 10.1109/RELPHY.2008.4558933. (Cited on page 14.)

[35] T. C. Bressoud and F. B. Schneider. Hypervisor-based fault tolerance. In *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pages 1–11, New York, NY, USA, Dec. 1995. ACM Press. doi: 10.1145/224056.224058. (Cited on page 26.)

[36] J. D. Bright, G. F. Sullivan, and G. M. Masson. Checking the integrity of trees. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing (FTCS-25)*, pages 402–411, Piscataway, NJ, USA, June 1995. IEEE Press. doi: 10.1109/FTCS.1995.466959. (Cited on page 27.)

[37] D. T. Brown. Error detecting and correcting binary codes for arithmetic operations. *IRE Transactions on Electronic Computers*, EC-9(3):333–337, Sept. 1960. doi: 10.1109/TEC.1960.5219855. (Cited on pages 27 and 133.)

[38] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall, and G. Muller. A foundation for flow-based program matching: Using temporal logic and model checking. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '09)*, pages 114–126, New York, NY, USA, Jan. 2009. ACM Press. doi: 10.1145/1480881.1480897. (Cited on page 72.)

[39] N. Z. Butt and M. Alam. Modeling single event upsets in floating gate memory cells. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS '08)*, pages 547–555, Piscataway, NJ, USA, Apr. 2008. IEEE Press. doi: 10.1109/RELPHY.2008.4558944. (Cited on page 15.)

[40] T. Calin, M. Nicolaidis, and R. Velazco. Upset hardened memory design for submicron CMOS technology. *IEEE Transactions on Nuclear Science*, 43(6):2874–2878, Dec. 1996. doi: 10.1109/23.556880. (Cited on pages 2 and 22.)

[41] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox. Microreboot — a technique for cheap recovery. In *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI '04)*, pages 1–14, Berkeley, CA, USA, Dec. 2004. USENIX Association. (Cited on pages 32 and 35.)

[42] P. Cappelletti, C. Golla, P. Olivo, and E. Zanoni. *Flash Memories*. Springer, New York, NY, USA, 1999. doi: 10.1007/978-1-4615-5015-0. (Cited on page 15.)

[43] G. Castagnoli, S. Brauer, and M. Herrmann. Optimization of cyclic redundancy-check codes with 24 and 32 parity bits. *IEEE Transactions on Communications*, 41(6): 883–892, June 1993. doi: 10.1109/26.231911. (Cited on page 118.)

[44] G. Cellere, S. Gerardin, M. Bagatin, A. Paccagnella, A. Visconti, M. Bonanomi, S. Beltrami, P. Roche, G. Gasiot, R. Harboe Sorensen, A. Virtanen, C. Frost, P. Fuochi, C. Andreani, G. Gorini, A. Pietropaolo, and S. Platt. Neutron-induced soft errors in advanced flash memories. In *Proceedings of the IEEE International Electron Devices Meeting (IEDM '08)*, pages 1–4, Piscataway, NJ, USA, Dec. 2008. IEEE Press. doi: 10.1109/IEDM.2008.4796693. (Cited on page 15.)

[45] J. Chang, G. A. Reis, and D. I. August. Automatic instruction-level software-only recovery. In *Proceedings of the 36th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '06)*, pages 83–92, Piscataway, NJ, USA, June 2006. IEEE Press. doi: 10.1109/DSN.2006.15. (Cited on pages 30, 31, 35, and 133.)

[46] D. Chen, A. Messer, P. Bernadat, G. Fu, Z. Dimitrijevic, D. J. F. Lie, D. Mannaru, A. Riska, and D. Milojicic. JVM susceptibility to memory errors. In *Proceedings of the 1st Java Virtual Machine Research and Technology Symposium (JVM '01)*, Berkeley, CA, USA, Apr. 2001. USENIX Association. (Cited on pages 27 and 35.)

[47] G. Chen and M. Kandemir. Improving java virtual machine reliability for memory-constrained embedded systems. In *Proceedings of the 42nd Design Automation Conference (DAC '05)*, pages 690–695, New York, NY, USA, June 2005. ACM Press. doi: 10.1145/1065579.1065761. (Cited on pages 31 and 35.)

[48] G. Chen, M. Kandemir, N. Vijaykrishnan, A. Sivasubramaniam, and M. J. Irwin. Analyzing heap error behavior in embedded JVM environments. In *Proceedings of the 2nd IEEE/ACM International Conference on Hardware/Software Codesign and System Synthesis (CODES+ISSS '04)*, pages 230–235, New York, NY, USA, Sept. 2004. ACM Press. doi: 10.1145/1016720.1016775. (Cited on pages 27, 35, and 41.)

[49] G. Chen, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin. Object duplication for improving reliability. In *Proceedings of the 2006 Asia and South Pacific Design Automation Conference (ASP-DAC '06)*, pages 140–145, Piscataway, NJ, USA, Jan. 2006. IEEE Press. doi: 10.1145/1118299.1118343. (Cited on pages 32, 35, and 41.)

[50] S. Chiba. A metaobject protocol for C++. In *Proceedings of the 10th ACM Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA '95)*, pages 285–299, New York, NY, USA, Oct. 1995. ACM Press. doi: 10.1145/217838.217868. (Cited on page 31.)

[51] S. Chiba and K. Nakagawa. Josh: An open AspectJ-like language. In *Proceedings of the 3rd International Conference on Aspect-Oriented Software Development (AOSD '04)*, pages 102–111, New York, NY, USA, Mar. 2004. ACM Press. doi: 10.1145/976270.976284. (Cited on page 71.)

[52] T.-C. Chiueh and F.-H. Hsu. RAD: A compile-time solution to buffer overflow attacks. In *Proceedings of the 21st International Conference on Distributed Computing Systems (ICDCS '01)*, pages 409–417, Piscataway, NJ, USA, Apr. 2001. IEEE Press. doi: 10.1109/ICDSC.2001.918971. (Cited on pages 29 and 35.)

[53] H. Cho, S. Mirkhani, C.-Y. Cher, J. A. Abraham, and S. Mitra. Quantitative evaluation of soft error injection techniques for robust system design. In *Proceedings of the 50th Design Automation Conference (DAC '13)*, pages 1–10, New York, NY, USA, May 2013. ACM Press. doi: 10.1145/2463209.2488859. (Cited on pages 16 and 41.)

[54] Y. Coady and G. Kiczales. Back to the future: A retroactive study of aspect evolution in operating system code. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 50–59, New York, NY, USA, Mar. 2003. ACM Press. doi: 10.1145/643603.643609. (Cited on page 72.)

[55] Y. Coady, G. Kiczales, M. Feeley, and G. Smolyn. Using AspectC to improve the modularity of path-specific customization in operating system code. In *Proceedings of the 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE-9)*, pages 88–98, New York, NY, USA, Sept. 2001. ACM Press. doi: 10.1145/503209.503223. (Cited on page 72.)

[56] E. G. Coffman, M. J. Elphick, and A. Shoshani. System deadlocks. *ACM Computing Surveys*, 3(2):67–78, June 1971. doi: 10.1145/356586.356588. (Cited on page 133.)

[57] C. Constantinescu. Trends and challenges in VLSI circuit reliability. *IEEE Micro*, 23 (4):14–19, July 2003. doi: 10.1109/MM.2003.1225959. (Cited on pages 9 and 10.)

[58] C. Constantinides, T. Skotiniotis, and M. Störzer. AOP considered harmful. In *Proceedings of the 1st European Interactive Workshop on Aspects in Software (EIWAS '04)*, Berlin, Germany, Aug. 2004. (Cited on page 73.)

[59] J. Corbet, A. Rubini, and G. Kroah-Hartman. *Linux Device Drivers*. O'Reilly, Sebastopol, CA, USA, third edition, Feb. 2005. (Cited on page 159.)

[60] C. Cowan, C. Pu, D. Maier, H. Hintony, J. Walpole, P. Bakke, S. Beattie, A. Grier, P. Wagle, and Q. Zhang. StackGuard: Automatic adaptive detection and prevention of buffer-overflow attacks. In *Proceedings of the 7th conference on USENIX Security Symposium (SSYM '98)*, Berkeley, CA, USA, Jan. 1998. USENIX Association. (Cited on pages 29 and 35.)

[61] B. Cully, G. Lefebvre, D. Meyer, M. Feeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. In *Proceedings of the 5th USENIX Symposium on Networked Systems Design and Implementation (NSDI' 08)*, pages 161–174, Berkeley, CA, USA, Apr. 2008. USENIX Association. (Cited on pages 31 and 35.)

[62] K. Czarnecki and U. W. Eisenecker. *Generative Programming: Methods, Tools and Applications*. Addison-Wesley, Boston, MA, USA, May 2000. (Cited on pages 86 and 89.)

[63] D. S. Dantas and D. Walker. Harmless advice. In *Proceedings of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL '06)*, pages 383–396, New York, NY, USA, Jan. 2006. ACM Press. doi: 10.1145/1111037.1111071. (Cited on page 74.)

[64] F. M. David. *Building a Reliable Operating System*. PhD thesis, University of Illinois at Urbana-Champaign, Urbana, IL, USA, 2008. Retrieved from ProQuest Digital Dissertations. (Cited on page 34.)

[65] F. M. David, E. M. Chan, J. C. Carlyle, and R. H. Campbell. CuriOS: Improving reliability through operating system structure. In *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation (OSDI '08)*, pages 59–72, Berkeley, CA, USA, Dec. 2008. USENIX Association. (Cited on page 37.)

[66] B. Demsky and M. Rinard. Automatic detection and repair of errors in data structures. In *Proceedings of the 18th ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '03)*, pages 78–95, New York, NY, USA, Oct. 2003. ACM Press. doi: 10.1145/949305.949314. (Cited on pages 33 and 35.)

[67] W. Dietz, P. Li, J. Regehr, and V. Adve. Understanding integer overflow in C/C++. In *Proceedings of the 34th International Conference on Software Engineering (ICSE '12)*, pages 760–770, Piscataway, NJ, USA, June 2012. IEEE Press. doi: 10.1109/ICSE.2012.6227142. (Cited on pages 104 and 175.)

[68] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, New Jersey, NJ, USA, 1976. (Cited on page 59.)

[69] A. Dixit and A. Wood. The impact of new technology on soft error rates. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS '11)*, pages 5B.4.1–5B.4.7, Piscataway, NJ, USA, Apr. 2011. IEEE Press. doi: 10.1109/IRPS.2011.5784522. (Cited on pages 1, 13, and 15.)

[70] B. Döbel, H. Härtig, and M. Engel. Operating system support for redundant multithreading. In *Proceedings of the 10th ACM International Conference on Embedded Software (EMSOFT '12)*, pages 83–92, New York, NY, USA, Oct. 2012. ACM Press. doi: 10.1145/2380356.2380375. (Cited on pages 2, 30, and 35.)

[71] P. E. Dodd and L. W. Massengill. Basic mechanisms and modeling of single-event upset in digital microelectronics. *IEEE Transactions on Nuclear Science*, 50(3):583–602, June 2003. doi: 10.1109/TNS.2003.813129. (Cited on pages 13 and 14.)

[72] P. E. Dodd, M. R. Shaneyfelt, J. R. Schwank, and J. A. Felix. Current and future challenges in radiation effects on CMOS electronics. *IEEE Transactions on Nuclear Science*, 57(4):1747–1763, Aug. 2010. doi: 10.1109/TNS.2010.2042613. (Cited on page 22.)

[73] E. Dubrova. *Fault-Tolerant Design*. Springer, New York, NY, USA, 2013. doi: 10.1007/978-1-4614-2113-9. (Cited on page 21.)

[74] K. Echtle. *Fehlertoleranzverfahren*. Studienreihe Informatik. Springer, Berlin, Germany, 1990. doi: 10.1007/978-3-642-75765-5. (Cited on pages 17, 19, 34, and 36.)

[75] M. Eichberg, M. Mezini, and K. Ostermann. Pointcuts as functional queries. In W.-N. Chin, editor, *Proceedings of the 2nd Asian Symposium on Programming Languages and Systems (APLAS '04)*, pages 366–381, Berlin, Germany, Nov. 2004. Springer. doi: 10.1007/978-3-540-30477-7_25. (Cited on page 94.)

[76] M. Engel and B. Döbel. The reliable computing base – a paradigm for software-based reliability. In *Proceedings of the 1st GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*, Lecture Notes in Informatics, pages 480–493, Bonn, Germany, Sept. 2012. German Society of Informatics. (Cited on page 37.)

[77] M. Engel and B. Freisleben. Supporting Autonomic Computing Functionality via Dynamic Operating System Kernel Aspects. In *Proceedings of the 4th International Conference on Aspect-Oriented Software Development (AOSD '05)*, pages 51–62, New York, NY, USA, Mar. 2005. ACM Press. doi: 10.1145/1052898.1052903. (Cited on page 64.)

[78] J.-C. Fabre, F. Salles, M. Rodriguez-Moreno, and J. Arlat. Assessment of COTS microkernels by fault injection. In *Proceedings of the Conference on Dependable Computing for Critical Applications 7 (DCCA '99)*, pages 25–44, Piscataway, NJ, USA, Jan. 1999. IEEE Press. doi: 10.1109/DCFTS.1999.814288. (Cited on page 41.)

[79] S. Feng, S. Gupta, A. Ansari, S. A. Mahlke, and D. I. August. Encore: Low-cost, fine-grained transient fault recovery. In *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO-44)*, pages 398–409, New York, NY, USA, Dec. 2011. ACM Press. doi: 10.1145/2155620.2155667. (Cited on pages 23, 31, and 35.)

[80] C. Fetzer, U. Schiffel, and M. Süßkraut. AN-encoding compiler: Building safety-critical systems with commodity hardware. In *Proceedings of the 28th International Conference on Computer Safety, Reliability, and Security (SAFECOMP '09)*, pages 283–296, Berlin, Germany, Sept. 2009. Springer. doi: 10.1007/978-3-642-04468-7_23. (Cited on pages 27 and 35.)

[81] D. Fiala, K. B. Ferreira, F. Mueller, and C. Engelmann. A tunable, software-based DRAM error detection and correction library for HPC. In *Proceedings of the 2011 International Conference on Parallel Processing – Volume 2 (Euro-Par '11)*, pages 251–261, Berlin, Germany, Aug. 2011. Springer. doi: 10.1007/978-3-642-29740-3_29. (Cited on pages 32 and 35.)

[82] R. E. Filman and D. P. Friedman. Aspect-oriented programming is quantification and obliviousness. In R. Filman, T. Elrad, S. Clarke, and M. Akşit, editors, *Aspect-Oriented Software Development*, chapter 2, pages 21–31. Addison-Wesley, Boston, MA, USA, first edition, 2004. (Cited on page 62.)

[83] I. Finocchi and G. F. Italiano. Sorting and searching in the presence of memory faults (without redundancy). In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing (STOC '04)*, pages 101–110, New York, NY, USA, June 2004. ACM Press. doi: 10.1145/1007352.1007375. (Cited on pages 33 and 35.)

[84] B. Fitzpatrick. Distributed caching with Memcached. *Linux Journal*, 2004(124):5, Aug. 2004. (Cited on pages 137, 157, and 158.)

[85] M. E. Fiuczynski, R. Grimm, Y. Coady, and D. Walker. patch(1) considered harmful. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS '05)*, Berkeley, CA, USA, June 2005. USENIX Association. (Cited on page 72.)

[86] A. D. Fogle, D. Darling, R. C. Blish, and E. Daszko. Flash memory under cosmic and alpha irradiation. *IEEE Transactions on Device and Materials Reliability*, 4(3):371–376, Sept. 2004. doi: 10.1109/TDMR.2004.834054. (Cited on page 15.)

[87] I. R. Forman and N. Forman. *Java Reflection in Action*. Manning Publications Company, Greenwich, CT, USA, 2004. (Cited on page 89.)

[88] E. Fuchs. An evaluation of the error detection mechanisms in MARS using software-implemented fault injection. In A. Hlawiczka, J. G. Silva, and L. Simoncini, editors, *Proceedings of the 2nd European Dependable Computing Conference (EDCC '96)*, volume 1150 of *Lecture Notes in Computer Science*, pages 73–90, Berlin, Germany, Oct. 1996. Springer. doi: 10.1007/3-540-61772-8_31. (Cited on page 41.)

[89] D. E. Fulkerson, D. K. Nelson, and R. M. Carlson. Boxes: An engineering methodology for calculating soft error rates in SOI integrated circuits. *IEEE Transactions on Nuclear Science*, 53(6):3329–3335, Dec. 2006. doi: 10.1109/TNS.2006.886150. (Cited on pages 2 and 22.)

[90] A. Gal, W. Schröder-Preikschat, and O. Spinczyk. On aspect-orientation in distributed real-time dependable systems. In *Proceedings of the 7th IEEE International Workshop on Object-oriented Real-time Dependable Systems (WORDS '02)*, pages 230–236, Piscataway, NJ, USA, Jan. 2002. IEEE Press. doi: 10.1109/WORDS.2002.1000061. (Cited on pages 3 and 75.)

[91] S. Gerardin, M. Bagatin, A. Paccagnella, G. Cellere, A. Visconti, S. Beltrami, C. Andreani, G. Gorini, and C. D. Frost. Scaling trends of neutron effects in MLC NAND flash memories. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS '10)*, pages 400–406, Piscataway, NJ, USA, May 2010. IEEE Press. doi: 10.1109/IRPS.2010.5488797. (Cited on page 15.)

[92] B. Gill, N. Seifert, and V. Zia. Comparison of alpha-particle and neutron-induced combinational and sequential logic error rates at the 32nm technology node. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS '09)*, pages 199–205, Piscataway, NJ, USA, Apr. 2009. IEEE Press. doi: 10.1109/IRPS.2009.5173251. (Cited on page 16.)

[93] R. Gioiosa, J. C. Sancho, S. Jiang, and F. Petrini. Transparent, incremental checkpointing at kernel level: A foundation for fault tolerance for parallel computers. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*, Piscataway, NJ, USA, Nov. 2005. IEEE Press. doi: 10.1109/SC.2005.76. (Cited on pages 31 and 35.)

[94] O. Goloubeva, M. Rebaudengo, M. S. Reorda, and M. Violante. *Software-Implemented Hardware Fault Tolerance*. Springer, New York, NY, USA, 2006. doi: 10.1007/0-387-32937-4. (Cited on pages 24 and 28.)

[95] M. D. Groves. *AOP in .NET: Practical Aspect-oriented Programming*. Manning Publications Company, Shelter Island, NY, USA, June 2013. (Cited on page 73.)

[96] C. S. Guenzer, E. A. Wolicki, and R. G. Allas. Single event upset of dynamic RAMs by neutrons and protons. *IEEE Transactions on Nuclear Science*, 26(6):5048–5052, Dec. 1979. doi: 10.1109/TNS.1979.4330270. (Cited on page 12.)

[97] J. Güthoff and V. Sieh. Combining software-implemented and simulation-based fault injection into a single fault injection method. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing (FTCS-25)*, pages 196–206, Piscataway, NJ, USA, June 1995. IEEE Press. doi: 10.1109/FTCS.1995.466978. (Cited on page 44.)

[98] R. W. Hamming. Error detecting and error correcting codes. *Bell System Technical Journal*, 29(2):147–160, Apr. 1950. (Cited on page 19.)

[99] S. Hanenberg and R. Unland. Parametric introductions. In *Proceedings of the 2nd International Conference on Aspect-Oriented Software Development (AOSD '03)*, pages 80–89, New York, NY, USA, Mar. 2003. ACM Press. doi: 10.1145/643603.643612. (Cited on pages 71 and 87.)

[100] S. Hangal and M. S. Lam. Tracking down software bugs using automatic anomaly detection. In *Proceedings of the 24th International Conference on Software Engineering (ICSE '02)*, pages 291–301, New York, NY, USA, May 2002. ACM Press. doi: 10.1145/581339.581377. (Cited on page 29.)

[101] O. Hannius and J. Karlsson. Impact of soft errors in a jet engine controller. In *Proceedings of the 31st International Conference on Computer Safety, Reliability, and Security (SAFECOMP '12)*, pages 223–234, Berlin, Germany, Sept. 2012. Springer. doi: 10.1007/978-3-642-33678-2_19. (Cited on pages 29 and 35.)

[102] S. K. S. Hari, S. V. Adve, and H. Naeimi. Low-cost program-level detectors for reducing silent data corruptions. In *Proceedings of the 42nd IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '12)*, pages 1–12, Piscataway, NJ, USA, June 2012. IEEE Press. doi: 10.1109/DSN.2012.6263960. (Cited on pages 29 and 35.)

[103] P. Hazucha, T. Karnik, S. Walstra, B. A. Bloechel, J. W. Tschanz, J. Maiz, K. Soumyanath, G. E. Dermer, S. Narendra, V. De, and S. Borkar. Measurements and analysis of SER-tolerant latch in a 90-nm dual-v_T CMOS process. *IEEE Journal of Solid-State Circuits*, 39(9):1536–1543, Sept. 2004. doi: 10.1109/JSSC.2004.831449. (Cited on pages 2 and 22.)

[104] J. Henkel, L. Bauer, N. Dutt, P. Gupta, S. Nassif, M. Shafique, M. Tahoori, and N. Wehn. Reliable on-chip systems in the nano-era: Lessons learnt and future trends. In *Proceedings of the 50th Design Automation Conference (DAC '13)*, pages 99:1–99:10, New York, NY, USA, May 2013. ACM Press. doi: 10.1145/2463209.2488857. (Cited on page 1.)

[105] J. N. Herder, H. Bos, B. Gras, P. Homburg, and A. S. Tanenbaum. Construction of a highly dependable operating system. In *Proceedings of the 6th European Dependable Computing Conference (EDCC '06)*, pages 3–12, Piscataway, NJ, USA, Oct. 2006. IEEE Press. doi: 10.1109/EDCC.2006.7. (Cited on page 37.)

[106] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, Jan. 1991. doi: 10.1145/114005.102808. (Cited on page 127.)

[107] M. Herlihy and N. Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. (Cited on pages 127, 128, and 130.)

[108] M. Hind. Pointer analysis: Haven't we solved this problem yet? In *Proceedings of the 2001 ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering (PASTE '01)*, pages 54–61, New York, NY, USA, June 2001. ACM Press. doi: 10.1145/379605.379665. (Cited on page 94.)

[109] M. Hoffmann, C. Borchert, C. Dietrich, H. Schirmeier, R. Kapitza, O. Spinczyk, and D. Lohmann. Effectiveness of fault detection mechanisms in static and dynamic operating system designs. In *Proceedings of the 17th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC '14)*, pages 230–237, Piscataway, NJ, USA, June 2014. IEEE Press. doi: 10.1109/ISORC.2014.26. (Cited on pages vi, 7, 60, 69, and 182.)

[110] M. Hoffmann, F. Lukas, C. Dietrich, and D. Lohmann. dOSEK: The design and implementation of a dependability-oriented static embedded kernel. In *Proceedings of the 21st IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS '15)*, pages 259–270, Piscataway, NJ, USA, Apr. 2015. IEEE Press. doi: 10.1109/R-TAS.2015.7108449. (Cited on pages 3, 27, 34, and 35.)

[111] G. J. Holzmann. *The Spin Model Checker: Primer and Reference Manual*. Addison-Wesley, Boston, MA, USA, first edition, 2003. (Cited on pages 128 and 131.)

[112] K.-H. Huang and J. A. Abraham. Algorithm-based fault tolerance for matrix operations. *IEEE Transactions on Computers*, 33(6):518–528, June 1984. doi: 10.1109/TC.1984.1676475. (Cited on pages 32 and 35.)

[113] Y. Huang and C. Kintala. Software implemented fault tolerance: Technologies and experience. In *Proceedings of the 23rd Annual International Symposium on Fault-Tolerant Computing (FTCS-23)*, pages 2–9, Piscataway, NJ, USA, June 1993. IEEE Press. doi: 10.1109/FTCS.1993.627302. (Cited on pages 31 and 35.)

[114] G. C. Hunt and J. R. Larus. Singularity: Rethinking the software stack. *ACM Operating Systems Review*, 41(2):37–49, Apr. 2007. doi: 10.1145/1243418.1243424. (Cited on page 73.)

[115] IEC. *IEC 61508 – Functional safety of electrical/electronic/programmable electronic safety-related systems*. International Electrotechnical Commission, Geneva, Switzerland, Dec. 1998. (Cited on page 172.)

[116] ISO. *International Standard ISO/IEC 14882:2011(E) – Programming Language C++*. International Organization for Standardization, Geneva, Switzerland, 2011. (Cited on pages 78, 79, 82, 83, 84, 90, 103, 104, 116, 119, and 120.)

[117] B. Jacob, S. W. Ng, and D. T. Wang. *Memory Systems: Cache, DRAM, Disk*. Morgan Kaufmann Publishers Inc., Burlington, MA, USA, 2008. (Cited on pages 2, 13, 14, 16, 23, and 41.)

[118] JEDEC. *JEDEC Standard JESD89A – Measurement and Reporting of Alpha Particle and Terrestrial Cosmic Ray-Induced Soft Errors in Semiconductor Devices*. JEDEC Solid State Technology Association, Arlington, VA, USA, Oct. 2006. (Cited on pages 10, 168, and 171.)

[119] C. M. Jeffery and R. J. O. Figueiredo. A flexible approach to improving system reliability with virtual lockstep. *IEEE Transactions on Dependable and Secure Computing*, 9(1):2–15, Jan. 2012. doi: 10.1109/TDSC.2010.53. (Cited on pages 26 and 35.)

[120] R. Johansen, P. Sestoft, and S. Spangenberg. Zero-overhead composable aspects for .NET. In E. Börger and A. Cisternino, editors, *Advances in Software Engineering*, pages 185–215. Springer, Berlin, Germany, 2008. doi: 10.1007/978-3-540-89762-0_7. (Cited on page 73.)

[121] A. G. Jørgensen, G. Moruz, and T. Mølhave. Priority queues resilient to memory faults. In *Proceedings of the 10th International Workshop on Algorithms and Data Structures (WADS '07)*, pages 127–138, Berlin, Germany, Aug. 2007. Springer. doi: 10.1007/978-3-540-73951-7_12. (Cited on pages 33 and 35.)

[122] G. Just, J. L. Autran, S. Serre, D. Munteanu, S. Sauze, A. Regnier, J. L. Ogier, P. Roche, and G. Gasiot. Soft errors induced by natural radiation at ground level in floating gate flash memories. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS '13)*, pages 3D.4.1–3D.4.8, Piscataway, NJ, USA, Apr. 2013. IEEE Press. doi: 10.1109/IRPS.2013.6531992. (Cited on page 15.)

[123] G. A. Kanawati, N. A. Kanawati, and J. A. Abraham. FERRARI: A flexible software-based fault and error injection system. *IEEE Transactions on Computers*, 44(2):248–260, Feb. 1995. doi: 10.1109/12.364536. (Cited on page 158.)

[124] W. Kao, R. K. Iyer, and D. Tang. FINE: a fault injection and monitoring environment for tracing the UNIX system behavior under faults. *IEEE Transactions on Software Engineering*, 19(11):1105–1118, Nov. 1993. doi: 10.1109/32.256857. (Cited on page 41.)

[125] T. Karnik, P. Hazucha, and J. Patel. Characterization of soft errors caused by single event upsets in CMOS processes. *IEEE Transactions on Dependable and Secure Computing*, 1(2):128–143, Apr. 2004. doi: 10.1109/TDSC.2004.14. (Cited on pages 10, 11, and 57.)

[126] S. Karol, N. A. Rink, B. Gyapjas, and J. Castrillon. Fault tolerance with aspects: A feasibility study. In *Proceedings of the 15th International Conference on Modularity (MODULARITY '16)*, pages 66–69, New York, NY, USA, Mar. 2016. ACM Press. doi: 10.1145/2889443.2889453. (Cited on pages 3 and 75.)

[127] M. Kasbekar, C. R. Das, S. Yajnik, R. Klemm, and Y. Huang. Issues in the design of a reflective library for checkpointing C++ objects. In *Proceedings of the 18th IEEE Symposium on Reliable Distributed Systems (SRDS '99)*, pages 224–233, Piscataway, NJ, USA, Oct. 1999. IEEE Press. doi: 10.1109/RELDIS.1999.805098. (Cited on pages 31 and 35.)

[128] C. N. Keltcher, K. J. McGrath, A. Ahmed, and P. Conway. The AMD opteron processor for multiprocessor servers. *IEEE Micro*, 23(2):66–76, Mar. 2003. doi: 10.1109/MM.2003.1196116. (Cited on pages 2 and 23.)

[129] D. S. Khudia and S. Mahlke. Low cost control flow protection using abstract control signatures. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '13)*, pages 3–12, New York, NY, USA, June 2013. ACM Press. doi: 10.1145/2499369.2465568. (Cited on pages 28 and 35.)

[130] G. Kiczales and M. Mezini. Aspect-oriented programming and modular reasoning. In *Proceedings of the 27th International Conference on Software Engineering (ICSE '05)*, pages 49–58, New York, NY, USA, May 2005. ACM Press. doi: 10.1145/1062455.1062482. (Cited on page 73.)

[131] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin. Aspect-oriented programming. In M. Akşit and S. Matsuoka, editors, *Proceedings of the 11th European Conference on Object-Oriented Programming (ECOOP '97)*, volume 1241 of *Lecture Notes in Computer Science*, pages 220–242, Berlin, Germany, June 1997. Springer. doi: 10.1007/BFb0053381. (Cited on pages 3, 59, 60, and 63.)

[132] G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In J. L. Knudsen, editor, *Proceedings of the 15th European Conference on Object-Oriented Programming (ECOOP '01)*, volume 2072 of *Lecture Notes in Computer Science*, pages 327–354, Berlin, Germany, June 2001. Springer. doi: 10.1007/3-540-45337-7_18. (Cited on pages 61, 62, and 71.)

[133] G. Kniesel and T. Rho. A definition, overview and taxonomy of generic aspect languages. *L'Objet, Special Issue on Aspect-Oriented Software Development*, 12(2–3):9–39, Sept. 2006. doi: 10.3166/objet.12.2-3.9-39. (Cited on pages 71 and 87.)

[134] H. Kobayashi, N. Kawamoto, J. Kase, and K. Shiraish. Alpha particle and neutron-induced soft error rates and scaling trends in SRAM. In *Proceedings of the IEEE International Reliability Physics Symposium (IRPS '09)*, pages 206–211, Piscataway, NJ, USA, Apr. 2009. IEEE Press. doi: 10.1109/IRPS.2009.5173252. (Cited on page 10.)

[135] H. Kopetz, H. Kantz, G. Grunsteidl, P. Puschner, and J. Reisinger. Tolerating transient faults in MARS. In *Proceedings of the 20th Annual International Symposium on Fault-Tolerant Computing (FTCS-20)*, pages 466–473, Piscataway, NJ, USA, June 1990. IEEE Press. doi: 10.1109/FTCS.1990.89384. (Cited on pages 26 and 34.)

[136] I. Koren and C. M. Krishna. *Fault-Tolerant Systems*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Apr. 2007. (Cited on pages 19, 23, 27, 30, 31, and 32.)

[137] B. A. Kuperman, C. E. Brodley, H. Ozdoganoglu, T. N. Vijaykumar, and A. Jalote. Detection and prevention of stack buffer overflow attacks. *Communications of the ACM*, 48(11):50–56, Nov. 2005. doi: 10.1145/1096000.1096004. (Cited on pages 29 and 35.)

[138] D. Kuvaiskii and C. Fetzer. Δ-encoding: Practical encoded processing. In *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, pages 13–24, Piscataway, NJ, USA, June 2015. IEEE Press. doi: 10.1109/DSN.2015.20. (Cited on pages 2, 27, and 35.)

[139] A. Lackorzynski and A. Warg. Taming subsystems: Capabilities as universal resource access control in L4. In *Proceedings of the 2nd Workshop on Isolation and Integration in Embedded Systems (IIES '09)*, pages 25–30, New York, NY, USA, Mar. 2009. ACM Press. doi: 10.1145/1519130.1519135. (Cited on pages 4 and 46.)

[140] R. Laddad. *AspectJ in Action: Practical Aspect-Oriented Programming*. Manning Publications Company, Greenwich, CT, USA, July 2003. (Cited on pages 71, 81, and 83.)

[141] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, Dec. 1992. doi: 10.1145/161494.161501. (Cited on page 83.)

[142] C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the 2nd International Symposium on Code Generation and Optimization (CGO'04)*, Piscataway, NJ, USA, Mar. 2004. IEEE Press. (Cited on page 93.)

[143] J. L. Lawall and G. Muller. Efficient incremental checkpointing of java programs. In *Proceedings of the 30th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '00)*, pages 61–70, Piscataway, NJ, USA, June 2000. IEEE Press. doi: 10.1109/ICDSN.2000.857515. (Cited on pages 31 and 35.)

[144] K. P. Lawton. Bochs: A portable PC emulator for Unix/X. *Linux Journal*, 1996(29):7, Sept. 1996. (Cited on page 47.)

[145] M. Leeke and A. Jhumka. An automated wrapper-based approach to the design of dependable software. In *Proceedings of the 4th International Conference on Dependability (DEPEND '11)*, pages 43–50. IARIA, Aug. 2011. (Cited on pages 32 and 35.)

[146] J. Lemieux. *Programming in the OSEK/VDX Environment*. CMP Books, Lawrence, KS, USA, Jan. 2001. (Cited on page 68.)

[147] C.-C. J. Li and W. K. Fuchs. CATCH – compiler-assisted techniques for checkpointing. In *Proceedings of the 20th Annual International Symposium on Fault-Tolerant Computing (FTCS-20)*, pages 74–81, Piscataway, NJ, USA, June 1990. IEEE Press. doi: 10.1109/FTCS.1990.89337. (Cited on pages 31 and 35.)

[148] M.-L. Li, P. Ramachandran, S. K. Sahoo, S. V. Adve, V. S. Adve, and Y. Zhou. Understanding the propagation of hard errors to software and implications for resilient system design. In *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '08)*, pages 265–276, New York, NY, USA, Mar. 2008. ACM Press. doi: 10.1145/1346281.1346315. (Cited on pages 28, 33, 35, and 37.)

[149] X. Li, S. V. Adve, P. Bose, and J. A. Rivers. Architecture-level soft error analysis: Examining the limits of common assumptions. In *Proceedings of the 37th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '07)*, pages 266–275, Piscataway, NJ, USA, June 2007. IEEE Press. doi: 10.1109/DSN.2007.15. (Cited on page 41.)

[150] X. Li, M. C. Huang, K. Shen, and L. Chu. A realistic evaluation of memory hardware errors and software system susceptibility. In *Proceedings of the 2010 USENIX Annual Technical Conference (ATC '10)*, pages 75–88, Berkeley, CA, USA, June 2010. USENIX Association. (Cited on page 14.)

[151] P. Liden, P. Dahlgren, R. Johansson, and J. Karlsson. On latching probability of particle induced transients in combinational networks. In *Proceedings of the 24th Annual International Symposium on Fault-Tolerant Computing (FTCS-24)*, pages 340–349, Piscataway, NJ, USA, June 1994. IEEE Press. doi: 10.1109/FTCS.1994.315626. (Cited on page 16.)

[152] R. Lidl and H. Niederreiter. *Introduction to Finite Fields and Their Applications*. Cambridge University Press, Cambridge, UK, 1994. (Cited on page 20.)

[153] S. T. Liu, W. C. Jenkins, and H. L. Hughes. Total dose radiation hard 0.35 μm SOI CMOS technology. *IEEE Transactions on Nuclear Science*, 45(6):2442–2449, Dec. 1998. doi: 10.1109/23.736484. (Cited on pages 2 and 22.)

[154] D. Lohmann. *Aspect Awareness in the Development of Configurable System Software*. PhD thesis, Friedrich-Alexander University Erlangen-Nuremberg, Erlangen-Nuremberg, Germany, 2009. (Cited on page 68.)

[155] D. Lohmann, G. Blaschke, and O. Spinczyk. Generic advice: On the combination of AOP with generative programming in AspectC++. In G. Karsai and E. Visser, editors, *Proceedings of the 3rd International Conference on Generative Programming and Component Engineering (GPCE '04)*, volume 3286 of *Lecture Notes in Computer Science*, pages 55–74, Berlin, Germany, Oct. 2004. Springer. doi: 10.1007/978-3-540-30175-2_4. (Cited on pages 68, 86, and 87.)

[156] D. Lohmann, W. Hofer, W. Schröder-Preikschat, J. Streicher, and O. Spinczyk. CiAO: An aspect-oriented operating-system family for resource-constrained embedded systems. In *Proceedings of the 2009 USENIX Annual Technical Conference (ATC '09)*, pages 215–228, Berkeley, CA, USA, June 2009. USENIX Association. (Cited on pages 68 and 69.)

[157] D. Lohmann, W. Hofer, W. Schröder-Preikschat, and O. Spinczyk. Aspect-aware operating system development. In *Proceedings of the 10th International Conference on Aspect-Oriented Software Development (AOSD '11)*, pages 69–80, New York, NY, USA, Mar. 2011. ACM Press. doi: 10.1145/1960275.1960285. (Cited on page 68.)

[158] E. Maricau and G. Gielen. *Analog IC Reliability in Nanometer CMOS*. Analog Circuits and Signal Processing. Springer, New York, NY, USA, 2013. doi: 10.1007/978-1-4614-6163-0. (Cited on page 10.)

[159] A. Martens, C. Borchert, T. O. Geißler, D. Lohmann, O. Spinczyk, and R. Kapitza. Crosscheck: Hardening replicated multithreaded services. In *Proceedings of the 4th International Workshop on Dependability of Clouds, Data Centers and Virtual Machine Technology (DCDV '14)*, pages 648–653, Piscataway, NJ, USA, June 2014. IEEE Press. doi: 10.1109/DSN.2014.98. (Cited on pages vi and 182.)

[160] A. Martens, C. Borchert, M. Nieke, O. Spinczyk, and R. Kapitza. CrossCheck: A holistic approach for tolerating crash-faults and arbitrary failures. In *Proceedings of the 12th European Dependable Computing Conference (EDCC '16)*, pages 65–76, Piscataway, NJ, USA, Sept. 2016. IEEE Press. doi: 10.1109/EDCC.2016.29. (Cited on pages vi, 7, 137, 158, and 182.)

[161] A. Martinez-Alvarez, F. Restrepo-Calle, S. Cuenca-Asensi, L. M. Reyneri, A. Lindoso, and L. Entrena. A hardware-software approach for on-line soft error mitigation in interrupt-driven applications. *IEEE Transactions on Dependable and Secure Computing*, 13(4):502–508, July 2016. doi: 10.1109/TDSC.2014.2382593. (Cited on pages 32 and 35.)

[162] A. Massa. *Embedded Software Development with eCos*. Prentice Hall, Upper Saddle River, NJ, USA, 2002. (Cited on pages 4 and 46.)

[163] T. C. Maxino and P. J. Koopman. The effectiveness of checksums for embedded control networks. *IEEE Transactions on Dependable and Secure Computing*, 6(1):59–72, Jan. 2009. doi: 10.1109/TDSC.2007.70216. (Cited on pages 27, 106, 118, and 119.)

[164] T. C. May and M. H. Woods. Alpha-particle-induced soft errors in dynamic memories. *IEEE Transactions on Electron Devices*, 26(1):2–9, Jan. 1979. doi: 10.1109/T-ED.1979.19370. (Cited on page 14.)

[165] S. A. McDermott, L. G. Jordán, and K. Anderson. The use of overloaded software operators for error detection and correction. In *Proceedings of the 18th AIAA/USU Conference on Small Satellites (SSC04-IV-7)*, pages 1–15, 2004. URL: `http://digitalcommons.usu.edu/smallsat/2004/All2004/22/`. (Cited on pages 32 and 35.)

[166] J. Melton and S. Buxton. *Querying XML: XQuery, XPath, and SQL/XML in Context*. The Morgan Kaufmann series in data management systems. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, Apr. 2006. (Cited on page 93.)

[167] A. Messer, P. Bernadat, G. Fu, D. Chen, Z. Dimitrijevic, D. Lie, D. D. Mannaru, A. Riska, and D. Milojicic. Susceptibility of commodity systems and software to memory soft errors. *IEEE Transactions on Computers*, 53(12):1557–1568, Dec. 2004. doi: 10.1109/TC.2004.119. (Cited on page 41.)

[168] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Upper Saddle River, NJ, USA, second edition, 1997. (Cited on page 100.)

[169] S. Meyers. *Effective C++ Digital Collection: 140 Ways to Improve Your Programming*. Addison-Wesley, Upper Saddle River, NJ, USA, 2012. (Cited on page 122.)

[170] J. Meza, Q. Wu, S. Kumar, and O. Mutlu. Revisiting memory errors in large-scale production data centers: Analysis and modeling of new trends from the field. In *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, pages 415–426, Piscataway, NJ, USA, June 2015. IEEE Press. doi: 10.1109/DSN.2015.57. (Cited on pages 2 and 23.)

[171] G. Miremadi, J. Karlsson, U. Gunneflo, and J. Torin. Two software techniques for on-line error detection. In *Proceedings of the 22nd Annual International Symposium on Fault-Tolerant Computing (FTCS-22)*, pages 328–335, Piscataway, NJ, USA, July 1992. IEEE Press. doi: 10.1109/FTCS.1992.243568. (Cited on pages 28 and 35.)

[172] S. Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008. (Cited on pages 2, 10, and 22.)

[173] G. Muller and U. P. Schultz. Harissa: A hybrid approach to Java execution. *IEEE Software*, 16(2):44–51, Mar. 1999. doi: 10.1109/52.754052. (Cited on page 72.)

[174] I. Nagy, R. van Engelen, and D. van der Ploeg. An overview of Mirjam and WeaveC. In R. van Engelen and J. Voeten, editors, *Ideals: evolvability of software-intensive high-tech systems*, pages 69–86. Embedded Systems Institute, Eindhoven, NL, 2007. (Cited on page 72.)

[175] H. G. Naik, R. Gupta, and P. Beckman. Analyzing checkpointing trends for applications on the IBM blue Gene/P system. In *Proceedings of the International Conference on Parallel Processing Workshops (ICPPW '09)*, pages 81–88, Piscataway, NJ, USA, Sept. 2009. IEEE Press. doi: 10.1109/ICPPW.2009.42. (Cited on pages 31 and 35.)

[176] B. Nicolescu and R. Velazco. Detecting soft errors by a purely software approach: method, tools and experimental results. In *Proceedings of the Design, Automation and Test in Europe Conference and Exhibition (DATE '03)*, pages 57–62, Piscataway, NJ, USA, Mar. 2003. IEEE Press. doi: 10.1109/DATE.2003.1253806. (Cited on pages 2 and 26.)

[177] B. Nicolescu, R. Velazco, and M. S. Reorda. Effectiveness and limitations of various software techniques for "soft error" detection: a comparative study. In *Proceedings of the 7th International on On-Line Testing Workshop (IOLTW '01)*, pages 172–177, Piscataway, NJ, USA, July 2001. IEEE Press. doi: 10.1109/OLT.2001.937838. (Cited on pages 27, 32, 35, 41, and 47.)

[178] B. Nicolescu, R. Velazco, M. Sonza-Reorda, M. Rebaudengo, and M. Violante. A software fault tolerance method for safety-critical systems: effectiveness and drawbacks. In *Proceedings of the 15th Symposium on Integrated Circuits and Systems Design (SBCCI '02)*, pages 101–106, Piscataway, NJ, USA, Sept. 2002. IEEE Press. doi: 10.1109/SBCCI.2002.1137644. (Cited on page 106.)

[179] E. B. Nightingale, J. R. Douceur, and V. Orgovan. Cycles, cells and platters: An empirical analysis of hardware failures on a million consumer PCs. In *Proceedings of the 6th ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '11)*, pages 343–356, New York, NY, USA, Apr. 2011. ACM Press. doi: 10.1145/1966445.1966477. (Cited on pages 38 and 39.)

[180] R. Nishtala, H. Fugal, S. Grimm, M. Kwiatkowski, H. Lee, H. C. Li, R. McElroy, M. Paleczny, D. Peek, P. Saab, D. Stafford, T. Tung, and V. Venkataramani. Scaling Memcache at Facebook. In *Proceedings of the 10th USENIX Symposium on Networked Systems Design and Implementation (NSDI '13)*, pages 385–398, Berkeley, CA, USA, Apr. 2013. USENIX Association. (Cited on page 158.)

[181] N. Oh and E. J. McCluskey. Error detection by selective procedure call duplication for low energy consumption. *IEEE Transactions on Reliability*, 51(4):392–402, Dec. 2002. doi: 10.1109/TR.2002.804735. (Cited on pages 26 and 35.)

[182] N. Oh, S. Mitra, and E. J. McCluskey. Ed4i: Error detection by diverse data and duplicated instructions. *IEEE Transactions on Computers*, 51(2):180–199, Feb. 2002. doi: 10.1109/12.980007. (Cited on pages 2, 27, and 35.)

[183] N. Oh, P. P. Shirvani, and E. J. McCluskey. Control-flow checking by software signatures. *IEEE Transactions on Reliability*, 51(1):111–122, Mar. 2002. doi: 10.1109/24.994926. (Cited on pages 28 and 35.)

[184] N. Oh, P. P. Shirvani, and E. J. McCluskey. Error detection by duplicated instructions in super-scalar processors. *IEEE Transactions on Reliability*, 51(1):63–75, Mar. 2002. doi: 10.1109/24.994913. (Cited on pages 2, 26, and 35.)

[185] J. Ohlsson and M. Rimen. Implicit signature checking. In *Proceedings of the 25th Annual International Symposium on Fault-Tolerant Computing (FTCS-25)*, pages 218–227, Piscataway, NJ, USA, June 1995. IEEE Press. doi: 10.1109/FTCS.1995.466976. (Cited on pages 28 and 35.)

[186] R. H. L. Ong and M. J. Pont. Empirical comparison of software-based error detection and correction techniques for embedded systems. In *Proceedings of the 9th International Symposium on Hardware/Software Codesign (CODES '01)*, pages 230–235, New York, NY, USA, Apr. 2001. ACM Press. doi: 10.1145/371636.371739. (Cited on pages 28 and 35.)

[187] Y. Padioleau, J. Lawall, R. R. Hansen, and G. Muller. Documenting and Automating Collateral Evolutions in Linux Device Drivers. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008 (Eurosys '08)*, pages 247–260, New York, New York, USA, 2008. ACM Press. doi: \url {10.1145/1352592.1352618}. (Cited on page 72.)

[188] D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, Dec. 1972. doi: 10.1145/361598.361623. (Cited on page 74.)

[189] K. Pattabiraman, V. Grover, and B. G. Zorn. Samurai: Protecting critical data in unsafe languages. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys '08)*, pages 219–232, New York, NY, USA, Apr. 2008. ACM Press. doi: 10.1145/1352592.1352616. (Cited on pages 32 and 35.)

[190] K. Pattabiraman, G. P. Saggese, D. Chen, Z. Kalbarczyk, and R. Iyer. Automated derivation of application-specific error detectors using dynamic analysis. *IEEE Transactions on Dependable and Secure Computing*, 8(5):640–655, Sept. 2011. doi: 10.1109/TDSC.2010.19. (Cited on pages 29 and 35.)

[191] K. H. Pedersen and C. Constantinides. AspectAda: Aspect oriented programming for Ada95. In *Proceedings of the 2005 Annual ACM SIGAda International Conference on Ada: The Engineering of Correct and Reliable Software for Real-Time & Distributed Systems Using Ada and Related Technologies (SigAda '05)*, pages 79–92, New York, NY, USA, Nov. 2005. ACM Press. doi: 10.1145/1103846.1103858. (Cited on page 73.)

[192] F. Perry, L. Mackey, G. A. Reis, J. Ligatti, D. I. August, and D. Walker. Fault-tolerant typed assembly language. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI '07)*, pages 42–53, New York, NY, USA, June 2007. ACM Press. doi: 10.1145/1250734.1250741. (Cited on page 36.)

[193] M. Peter, M. Petschick, J. Vetter, J. Nordholz, J. Danisevskis, and J.-P. Seifert. Undermining isolation through covert channels in the Fiasco.OC microkernel. In H. O. Abdelrahman, E. Gelenbe, G. Gorbil, and R. Lent, editors, *Information Sciences and Systems 2015: 30th International Symposium on Computer and Information Sciences (ISCIS '15)*, volume 363 of *Lecture Notes in Electrical Engineering*, pages 147–156. Springer, Cham, Switzerland, Sept. 2015. doi: 10.1007/978-3-319-22635-4_13. (Cited on page 46.)

[194] W. W. Peterson and D. T. Brown. Cyclic codes for error detection. *Proceedings of the IRE*, 49(1):228–235, Jan. 1961. doi: 10.1109/JRPROC.1961.287814. (Cited on page 27.)

[195] W. W. Peterson and E. J. Weldon. *Error-correcting Codes*. M.I.T. Press, Cambridge, MA, USA, second edition, 1972. (Cited on pages 21 and 32.)

[196] A. Popovici, T. Gross, and G. Alonso. Dynamic weaving for aspect-oriented programming. In *Proceedings of the 1st International Conference on Aspect-Oriented Software Development (AOSD '02)*, pages 141–147, New York, NY, USA, Apr. 2002. ACM Press. doi: 10.1145/508386.508404. (Cited on page 64.)

[197] P. Racunas, K. Constantinides, S. Manne, and S. S. Mukherjee. Perturbation-based fault screening. In *Proceedings of the IEEE 13th International Symposium on High Performance Computer Architecture (HPCA '07)*, pages 169–180, Piscataway, NJ, USA, Feb. 2007. IEEE Press. doi: 10.1109/HPCA.2007.346195. (Cited on pages 29 and 35.)

[198] M. Raynal. *Concurrent Programming: Algorithms, Principles, and Foundations*. Springer, Berlin, Germany, 2013. doi: 10.1007/978-3-642-32027-9. (Cited on page 128.)

[199] M. Rebaudengo, M. S. Reorda, M. Torchiano, and M. Violante. Soft-error detection through software fault-tolerance techniques. In *Proceedings of the International Symposium on Defect and Fault Tolerance in VLSI Systems (DFT '99)*, pages 210–218, Piscataway, NJ, USA, Nov. 1999. IEEE Press. doi: 10.1109/DFTVS.1999.802887. (Cited on pages 26, 28, 35, and 41.)

[200] M. Rebaudengo, M. S. Reorda, M. Violante, and M. Torchiano. A source-to-source compiler for generating dependable software. In *Proceedings of the 1st IEEE International Workshop on Source Code Analysis and Manipulation (SCAM '01)*, pages 33–42, Piscataway, NJ, USA, Nov. 2001. IEEE Press. doi: 10.1109/SCAM.2001.972664. (Cited on pages 2, 26, and 35.)

[201] M. Rebaudengo, M. S. Reorda, and M. Violante. A new software-based technique for low-cost fault-tolerant application. In *Proceedings of the Annual Reliability and Maintainability Symposium (RAMS '03)*, pages 25–28, Piscataway, NJ, USA, Jan. 2003. IEEE Press. doi: 10.1109/RAMS.2003.1181897. (Cited on page 41.)

[202] G. A. Reis, J. Chang, N. Vachharajani, R. Rangan, and D. I. August. SWIFT: Software implemented fault tolerance. In *Proceedings of the 3rd International Symposium on Code Generation and Optimization (CGO '05)*, pages 243–254, Piscataway, NJ, USA, Mar. 2005. IEEE Press. doi: 10.1109/CGO.2005.34. (Cited on pages 27 and 35.)

[203] M. Z. Rela, H. Madeira, and J. G. Silva. Experimental evaluation of the fail-silent behaviour in programs with consistency checks. In *Proceedings of the 26th Annual International Symposium on Fault-Tolerant Computing (FTCS-26)*, pages 394–403, Piscataway, NJ, USA, June 1996. IEEE Press. doi: 10.1109/FTCS.1996.534625. (Cited on pages 29, 35, and 36.)

[204] M. A. Rivas and M. G. Harbour. Evaluation of new POSIX real-time operating systems services for small embedded platforms. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems*, pages 161–168, Piscataway, NJ, USA, July 2003. IEEE Press. doi: 10.1109/EMRTS.2003.1212739. (Cited on page 73.)

[205] O. Rohlik, A. Pasetti, V. Cechticky, and I. Birrer. Implementing adaptability in embedded software through aspect oriented programming. In *Proceedings of Mechatronics & Robotics (MechRob '04)*, pages 85–90, Piscataway, NJ, USA, Sept. 2004. IEEE Press. (Cited on pages 73 and 94.)

[206] A. Roy-Chowdhury and P. Banerjee. Algorithm-based fault location and recovery for matrix computations on multiprocessor systems. *IEEE Transactions on Computers*, 45(11):1239–1247, Nov. 1996. doi: 10.1109/12.544480. (Cited on pages 33 and 35.)

[207] A. Roy-Chowdhury, N. Bellas, and P. Banerjee. Algorithm-based error-detection schemes for iterative solution of partial differential equations. *IEEE Transactions on Computers*, 45(4):394–407, Apr. 1996. doi: 10.1109/12.494098. (Cited on pages 33 and 35.)

[208] G. P. Saggese, A. Vetteth, Z. Kalbarczyk, and R. Iyer. Microprocessor sensitivity to failures: control vs. execution and combinational vs. sequential logic. In *Proceedings of the 35th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '05)*, pages 760–769, Piscataway, NJ, USA, June 2005. IEEE Press. doi: 10.1109/DSN.2005.63. (Cited on page 16.)

[209] S. K. Sahoo, M.-L. Li, P. Ramachandran, S. V. Adve, V. S. Adve, and Y. Zhou. Using likely program invariants to detect hardware errors. In *Proceedings of the 38th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '08)*, pages 70–79, Piscataway, NJ, USA, June 2008. IEEE Press. doi: 10.1109/DSN.2008.4630072. (Cited on pages 29 and 35.)

[210] T. Santini, P. Rech, G. L. Nazar, and F. R. Wagner. Beyond cross-section: Spatio-temporal reliability analysis. *ACM Transactions on Embedded Computing Systems*, 15 (1):3:1–3:16, Feb. 2016. doi: 10.1145/2794148. (Cited on page 171.)

[211] T. Santini, C. Borchert, C. Dietrich, H. Schirmeier, M. Hoffmann, O. Spinczyk, D. Lohmann, F. R. Wagner, and P. Rech. Effectiveness of software-based hardening for radiation-induced soft errors in real-time operating systems. In *Proceedings of the 30th International Conference on Architecture of Computing Systems (ARCS '17)*, pages 3–15, Cham, Switzerland, Apr. 2017. Springer. doi: 10.1007/978-3-319-54999-6_1. (Cited on pages vi, 7, 167, and 182.)

[212] U. Schiffel. *Hardware Error Detection Using AN-Codes*. PhD thesis, Technische Universität Dresden, Dresden, Germany, June 2011. (Cited on page 27.)

[213] H. Schirmeier, C. Borchert, and O. Spinczyk. Rapid fault-space exploration by evolutionary pruning. In *Proceedings of the 33rd International Conference on Computer Safety, Reliability and Security (SAFECOMP '14)*, pages 17–32, Cham, Switzerland, Sept. 2014. Springer. doi: 10.1007/978-3-319-10506-2_2. (Cited on pages vi, 40, and 182.)

[214] H. Schirmeier, C. Borchert, and O. Spinczyk. Avoiding pitfalls in fault-injection based comparison of program susceptibility to soft errors. In *Proceedings of the 45th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '15)*, pages 319–330, Piscataway, NJ, USA, June 2015. IEEE Press. doi: 10.1109/DSN.2015.44. (Cited on pages vi, 6, 40, 42, and 182.)

[215] H. Schirmeier, M. Hoffmann, C. Dietrich, M. Lenz, D. Lohmann, and O. Spinczyk. FAIL*: An open and versatile fault-injection framework for the assessment of software-implemented hardware fault tolerance. In *Proceedings of the 11th European Dependable Computing Conference (EDCC '15)*, pages 245–255, Piscataway, NJ, USA, Sept. 2015. IEEE Press. doi: 10.1109/EDCC.2015.28. (Cited on page 47.)

[216] K. Serebryany, D. Bruening, A. Potapenko, and D. Vyukov. AddressSanitizer: A fast address sanity checker. In *Proceedings of the 2012 USENIX Annual Technical Conference (ATC '12)*, pages 28–28, Berkeley, CA, USA, June 2012. USENIX Association. (Cited on pages 29 and 35.)

[217] P. Sewell, S. Sarkar, S. Owens, F. Z. Nardelli, and M. O. Myreen. x86-TSO: A rigorous and usable programmer's model for x86 multiprocessors. *Communications of the ACM*, 53(7):89–97, July 2010. doi: 10.1145/1785414.1785443. (Cited on page 132.)

[218] M. Shafique, P. Axer, C. Borchert, J. Chen, K. Chen, B. Döbel, R. Ernst, H. Härtig, A. Heinig, R. Kapitza, F. Kriebel, D. Lohmann, P. Marwedel, S. Rehman, F. Schmoll, and O. Spinczyk. Multi-layer software reliability for unreliable hardware. *it - Information Technology*, 57(3):170–180, June 2015. doi: 10.1515/itit-2014-1081. (Cited on pages vi and 182.)

[219] P. P. Shirvani, N. R. Saxena, and E. J. McCluskey. Software-implemented EDAC protection against SEUs. *IEEE Transactions on Reliability*, 49(3):273–284, Sept. 2000. doi: 10.1109/24.914544. (Cited on pages 32, 35, 47, and 120.)

[220] A. Shrivastava, A. Rhisheekesan, R. Jeyapaul, and C.-J. Wu. Quantitative analysis of control flow checking mechanisms for soft errors. In *Proceedings of the 51st Design Automation Conference (DAC '14)*, pages 13:1–13:6, New York, NY, USA, June 2014. ACM Press. doi: 10.1145/2593069.2593195. (Cited on page 28.)

[221] A. Shye, J. Blomstedt, T. Moseley, V. J. Reddi, and D. A. Connors. PLR: A software approach to transient fault tolerance for multicore architectures. *IEEE Transactions on Dependable and Secure Computing*, 6(2):135–148, Apr. 2009. doi: 10.1109/TDSC.2008.62. (Cited on pages 2, 30, and 35.)

[222] A. Silberschatz, P. B. Galvin, and G. Gagne. *Operating System Concepts*. Wiley & Sons, Hoboken, NJ, USA, eighth edition, 2009. (Cited on page 48.)

[223] D. Skarin and J. Karlsson. Software implemented detection and recovery of soft errors in a brake-by-wire system. In *Proceedings of the 7th European Dependable Computing Conference (EDCC '08)*, pages 145–154, Piscataway, NJ, USA, May 2008. IEEE Press. doi: 10.1109/EDCC-7.2008.24. (Cited on pages 10, 29, and 35.)

[224] C. W. Slayman. Cache and memory error detection, correction, and reduction techniques for terrestrial servers and workstations. *IEEE Transactions on Device and Materials Reliability*, 5(3):397–404, Sept. 2005. doi: 10.1109/TDMR.2005.856487. (Cited on page 23.)

[225] C. W. Slayman. Soft error trends and mitigation techniques in memory devices. In *Proceedings of the Annual Reliability and Maintainability Symposium (RAMS '11)*, pages 1–5, Piscataway, NJ, USA, Jan. 2011. IEEE Press. doi: 10.1109/RAMS.2011.5754515. (Cited on pages 13, 15, and 16.)

[226] T. J. Slegel, I. Averill, R. M., M. A. Check, B. C. Giamei, B. W. Krumm, C. A. Krygowski, W. H. Li, J. S. Liptay, J. D. MacDougall, T. J. McPherson, J. A. Navarro, E. M. Schwarz, K. Shum, and C. F. Webb. IBM's S/390 G5 microprocessor design. *IEEE Micro*, 19(2):12–23, Mar. 1999. doi: 10.1109/40.755464. (Cited on pages 2 and 23.)

[227] D. T. Smith, B. W. Johnson, J. A. Profeta, III, and D. G. Bozzolo. A method to determine equivalent fault classes for permanent and transient faults. In *Proceedings of the Annual Reliability and Maintainability Symposium (RAMS '95)*, pages 418–424, Piscataway, NJ, USA, Jan. 1995. IEEE Press. doi: 10.1109/RAMS.1995.513278. (Cited on page 44.)

[228] J. E. Smith. Characterizing computer performance with a single number. *Communications of the ACM*, 31(10):1202–1206, Oct. 1988. doi: 10.1145/63039.63043. (Cited on pages 145 and 153.)

[229] J. Song, J. Wittrock, and G. Parmer. Predictable, efficient system-level fault tolerance in C^3. In *Proceedings of the 34th Real-Time Systems Symposium (RTSS '13)*, pages 21–32, Piscataway, NJ, USA, Dec. 2013. IEEE Press. doi: 10.1109/RTSS.2013.11. (Cited on pages 32 and 35.)

[230] D. J. Sorin. *Fault Tolerant Computer Architecture.* Synthesis Lectures on Computer Architecture. Morgan & Claypool Publishers, 2009. doi: 10.2200/S00192ED1V01Y200904CAC005. (Cited on page 10.)

[231] O. Spinczyk and D. Lohmann. The design and implementation of AspectC++. *Knowledge-Based Systems, Special Issue on Techniques to Produce Intelligent Secure Software*, 20(7):636–651, Oct. 2007. doi: 10.1016/j.knosys.2007.05.004. (Cited on pages 4, 63, 64, and 69.)

[232] V. Sridharan and D. Liberty. A study of DRAM failures in the field. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '12)*, pages 76:1–76:11, Piscataway, NJ, USA, Nov. 2012. IEEE Press. doi: 10.1109/SC.2012.13. (Cited on pages 14 and 17.)

[233] V. Sridharan, H. Asadi, M. B. Tahoori, and D. Kaeli. Reducing data cache susceptibility to soft errors. *IEEE Transactions on Dependable and Secure Computing*, 3(4): 353–364, Oct. 2006. doi: 10.1109/TDSC.2006.55. (Cited on page 168.)

[234] V. Sridharan, J. Stearley, N. DeBardeleben, S. Blanchard, and S. Gurumurthi. Feng shui of supercomputer memory: Positional effects in DRAM and SRAM faults. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis (SC '13)*, pages 22:1–22:11, New York, NY, USA, Nov. 2013. ACM Press. doi: 10.1145/2503210.2503257. (Cited on pages 14 and 41.)

[235] V. Sridharan, N. DeBardeleben, S. Blanchard, K. B. Ferreira, J. Stearley, J. Shalf, and S. Gurumurthi. Memory errors in modern systems: The good, the bad, and the ugly. In *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '15)*, pages 297–310, New York, NY, USA, Mar. 2015. ACM Press. doi: 10.1145/2694344.2694348. (Cited on pages 13 and 14.)

[236] F. Steimann. Domain models are aspect free. In *Proceedings of the 8th International Conference on Model-Driven Engineering, Languages, and Systems (MoDELS '05)*, pages 171–185, Berlin, Germany, Oct. 2005. Springer. doi: 10.1007/11557432_13. (Cited on page 74.)

[237] F. Steimann. The paradoxical success of aspect-oriented programming. In *Proceedings of the 21st ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '06)*, pages 481–497, New York, NY, USA, Oct. 2006. ACM Press. doi: 10.1145/1167515.1167514. (Cited on pages 73, 74, 75, and 182.)

[238] I. Stilkerich, M. Strotz, C. Erhardt, M. Hoffmann, D. Lohmann, F. Scheler, and W. Schröder-Preikschat. A JVM for soft-error-prone embedded systems. In *Proceedings of the 14th ACM SIGPLAN/SIGBED Conference on Languages, Compilers and Tools for Embedded Systems (LCTES '13)*, pages 21–32, New York, NY, USA, June 2013. ACM Press. doi: 10.1145/2499369.2465571. (Cited on pages 27, 35, and 41.)

[239] M. Stoerzer and J. Graf. Using pointcut delta analysis to support evolution of aspect-oriented software. In *Proceedings of the 21st IEEE International Conference on Software Maintenance (ICSM '05)*, pages 653–656, Piscataway, NJ, USA, Sept. 2005. IEEE Press. doi: 10.1109/ICSM.2005.99. (Cited on page 74.)

[240] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Upper Saddle River, NJ, USA, fourth edition, May 2013. (Cited on pages 53, 109, and 111.)

[241] M. M. Swift, B. N. Bershad, and H. M. Levy. Improving the reliability of commodity operating systems. *ACM Transactions on Computer Systems*, 23(1):77–110, Feb. 2005. doi: 10.1145/1047915.1047919. (Cited on pages 32 and 35.)

[242] A. S. Tanenbaum. Lessons learned from 30 years of MINIX. *Communications of the ACM*, 59(3):70–78, Mar. 2016. doi: 10.1145/2795228. (Cited on page 37.)

[243] R. Tartler, D. Lohmann, F. Scheler, and O. Spinczyk. AspectC++: An integrated approach for static and dynamic adaptation of system software. *Knowledge-Based Systems*, 23(7):704–720, Oct. 2010. doi: 10.1016/j.knosys.2010.03.002. Special issue on Intelligent Formal Techniques for Software Design: IFTSD. (Cited on page 64.)

[244] D. J. Taylor, D. E. Morgan, and J. P. Black. Redundancy in data structures: Improving software fault tolerance. *IEEE Transactions on Software Engineering*, SE-6(6):585–594, Nov. 1980. doi: 10.1109/TSE.1980.234507. (Cited on pages 33, 35, and 137.)

[245] C. Tice, T. Roeder, P. Collingbourne, S. Checkoway, U. Erlingsson, L. Lozano, and G. Pike. Enforcing forward-edge control-flow integrity in GCC & LLVM. In *Proceedings of the 23rd USENIX Conference on Security Symposium (USENIX Security '14)*, pages 941–955, Berkeley, CA, USA, Aug. 2014. USENIX Association. (Cited on pages 29 and 35.)

[246] K. Twyford, reporter. *Bookout v. Toyota Motor Corporation*. Case No. CJ-2008-7969. Transcript of morning trial proceedings had on the 14th day of October, 2013. District Court of Oklahoma County, Oklahoma City, OK, USA, Oct. 2013. (Cited on pages 1 and 2.)

[247] P. Ulbrich, R. Kapitza, C. Harkort, R. Schmid, and W. Schröder-Preikschat. I4Copter: An adaptable and modular quadrotor platform. In *Proceedings of the 2011 ACM Symposium on Applied Computing (SAC '11)*, pages 380–386, New York, NY, USA, Mar. 2011. ACM Press. doi: 10.1145/1982185.1982267. (Cited on page 69.)

[248] P. Ulbrich, M. Hoffmann, R. Kapitza, D. Lohmann, W. Schröder-Preikschat, and R. Schmid. Eliminating single points of failure in software-based redundancy. In *Proceedings of the 9th European Dependable Computing Conference (EDCC '12)*, pages 49–60, Piscataway, NJ, USA, May 2012. IEEE Press. doi: 10.1109/EDCC.2012.21. (Cited on pages 27 and 35.)

[249] E. Van Der Kouwe, C. Giuffrida, and A. S. Tanenbaum. On the soundness of silence: Investigating silent failures using fault injection experiments. In *Proceedings of the 10th European Dependable Computing Conference (EDCC '14)*, pages 118–129, Piscataway, NJ, USA, May 2014. IEEE Press. doi: 10.1109/EDCC.2014.16. (Cited on page 40.)

[250] D. Vandevoorde and N. M. Josuttis. *C++ Templates: The Complete Guide*. Addison-Wesley, Boston, MA, USA, 2003. (Cited on page 67.)

[251] C. Wang, H.-s. Kim, Y. Wu, and V. Ying. Compiler-managed software-based redundant multi-threading for transient fault detection. In *Proceedings of the 5th International Symposium on Code Generation and Optimization (CGO '07)*, pages 244–258, Piscataway, NJ, USA, Mar. 2007. IEEE Press. doi: 10.1109/CGO.2007.7. (Cited on pages 2, 26, and 35.)

[252] N. J. Wang and S. J. Patel. ReStore: Symptom based soft error detection in microprocessors. In *Proceedings of the 35th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '05)*, pages 30–39, Piscataway, NJ, USA, June 2005. IEEE Press. doi: 10.1109/DSN.2005.82. (Cited on pages 28, 35, 102, and 103.)

[253] U. Wappler and C. Fetzer. Software encoded processing: Building dependable systems with commodity hardware. In *Proceedings of the 26th International Conference on Computer Safety, Reliability, and Security (SAFECOMP '07)*, pages 356–369, Berlin, Germany, Sept. 2007. Springer. doi: 10.1007/978-3-540-75101-4_34. (Cited on pages 28 and 35.)

[254] J. H. Wensley, M. W. Green, K. N. Levitt, and R. E. Shostak. The design, analysis, and verification of the SIFT fault tolerant system. In *Proceedings of the 2nd International Conference on Software Engineering (ICSE '76)*, pages 458–469, Piscataway, NJ, USA, Oct. 1976. IEEE Press. (Cited on pages 30 and 35.)

[255] J. H. Wensley, L. Lamport, J. Goldberg, M. W. Green, K. N. Levitt, P. M. Melliar-Smith, R. E. Shostak, and C. B. Weinstock. SIFT: Design and analysis of a fault-tolerant computer for aircraft control. *Proceedings of the IEEE*, 66(10):1240–1255, Oct. 1978. doi: 10.1109/PROC.1978.11114. (Cited on page 24.)

[256] K. S. Yim, Z. Kalbarczyk, and R. K. Iyer. Measurement-based analysis of fault and error sensitivities of dynamic memory. In *Proceedings of the 40th IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '10)*, pages 431–436, Piscataway, NJ, USA, June 2010. IEEE Press. doi: 10.1109/DSN.2010.5544287. (Cited on page 41.)

[257] D. H. Yoon and M. Erez. Virtualized and flexible ECC for main memory. In *Proceedings of the 15th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '10)*, pages 397–408, New York, NY, USA, Mar. 2010. ACM Press. doi: 10.1145/1736020.1736064. (Cited on page 23.)

[258] J. F. Ziegler. Terrestrial cosmic rays. *IBM Journal of Research and Development*, 40(1): 19–39, Jan. 1996. doi: 10.1147/rd.401.0019. (Cited on page 10.)

[259] J. F. Ziegler and W. A. Lanford. Effect of cosmic rays on computer memories. *AAAS Science*, 206(4420):776–788, Nov. 1979. doi: 10.1126/science.206.4420.776. (Cited on page 10.)