Operating System Support for Redundant Multithreading

Dissertation zur Erlangung des akademischen Grades
Doktoringenieur (Dr.-Ing.)

Vorgelegt an der
Technischen Universität Dresden
Fakultät Informatik

Eingereicht von
Dipl.-Inf. Björn Döbel
geboren am 17. Dezember 1980
in Lauchhammer

Betreuender Hochschullehrer:	Prof. Dr. Hermann Härtig Technische Universität Dresden
Gutachter:	Prof. Frank Mueller, Ph.D. North Carolina State University
Fachreferent:	Prof. Dr. Christof Fetzer Technische Universität Dresden
Statusvortrag:	29.02.2012
Eingereicht am:	21.08.2014
Verteidigt am:	25.11.2014

FÜR JAKOB

*† 15. Februar 2013

Contents

1 Introduction 7

1.1 Hardware meets Soft Errors 8

1.2 An Operating System for Tolerating Soft Errors 9

1.3 Whom can you Rely on? 12

2 Why Do Transistors Fail And What Can Be Done About It? 15

2.1 Hardware Faults at the Transistor Level 15

2.2 Faults, Errors, and Failures – A Taxonomy 18

2.3 Manifestation of Hardware Faults 20

2.4 Existing Approaches to Tolerating Faults 25

2.5 Thesis Goals and Design Decisions 36

3 Redundant Multithreading as an Operating System Service 39

3.1 Architectural Overview 39

3.2 Process Replication 41

3.3 Tracking Externalization Events 42

3.4 Handling Replica System Calls 45

3.5 Managing Replica Memory 49

3.6 Managing Memory Shared with External Applications 57

3.7 Hardware-Induced Non-Determinism 63

3.8 Error Detection and Recovery 65

4 Can We Put the Concurrency Back Into Redundant Multithreading? 71

4.1 What is the Problem with Multithreaded Replication? 71

4.2 Can we make Multithreading Deterministic? 74

4.3 Replication Using Lock-Based Determinism 79

4.4 Reliability Implications of Multithreaded Replication 92

5 Evaluation 97

5.1 Methodology 97

5.2 Error Coverage and Detection Latency 98

5.3 Runtime and Resource Overhead 109

5.4 Implementation Complexity 121

5.5 Comparison with Related Work 122

6 Who Watches the Watchmen? 127

6.1 The Reliable Computing Base 127

6.2 Case Study #1: How Vulnerable is the Operating System? 131

6.3 Case Study #2: Mixed-Reliability Hardware Platforms 139

6.4 Case Study #3: Compiler-Assisted RCB Protection 146

7 Conclusions and Future Work 149

7.1 OS-Assisted Replication 149

7.2 Directions for Future Research 151

8 Bibliography 163

1
Introduction

Computer systems fail every day, destroying personal data, causing economic loss and even threatening users' lives. The reasons for such failures are manifold, ranging from environmental hazards (e.g., a fire breaking out in a data center) to programming errors (e.g., invalid pointer dereferences in unsafe programming languages), as well as defective hardware components (e.g., a hard disk returning erroneous data when reading).

A large fraction of these failures results from programming mistakes.[1] This observation triggered a large body of research related to improving software quality, ranging from static code analysis[2] to formally verified operating system kernels.[3] However, even if the combined solutions at some point lead to a situation where software is fault-free, their assumptions only hold if we can trust in hardware to function correctly.

From a hardware development perspective, Moore's Law[4] indicates a doubling of transistor counts in modern microprocessors roughly every two years. While the law started off as a prediction, it is today used by hardware vendors as a means to establish product release cycles, research, and development goals. This has turned Moore's Law into a self-fulfilling prophecy. Increasing the number of transistors enables to integrate more and more components into a single chip. This permits the addition of more processors, larger caches, as well as specialized functional units, such as on-chip graphics processors.

While transistor counts increase, the available chip area does not. Emerging technologies, such as three-dimensional stacking of transistors try to address this problem.[5] However, the standard way of solving this problem in practical systems is to decrease the size of individual transistors by applying more fine-grained production processes.

Unfortunately, hardware components are exposed to energetic stress caused by radiation, thermal effects, energy fluctuations, and mechanical force. As I will outline in Chapter 2, hardware vendors spend significant effort in keeping the failure probability of their components below certain thresholds. However, this effort is becoming increasingly difficult as hardware structure sizes shrink, because smaller transistors are more vulnerable to the effects previously mentioned.[6]

This increased vulnerability accounts for an increased amount of *intermittent (or soft) hardware faults*.[7] In contrast to *permanent faults*, which constitute a constant malfunction of a component, intermittent faults occur and vanish seemingly randomly during execution time. This randomness stems from the physical effects mentioned above.[8] Soft errors are often *tran-*

[1] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, Redmond, WA, 2 edition, 2004

[2] Dawson Engler and David Yu et al. Chen. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Symposium on Operating Systems Principles*, SOSP'01, pages 57–72, Banff, Alberta, Canada, 2001. ACM

[3] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Symposium on Operating Systems Principles*, SOSP'09, pages 207–220, Big Sky, MT, USA, October 2009. ACM

[4] Gordon E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8), 1965

[5] Dae Hyun Kim et al. 3D-MAPS: 3D Massively Parallel Processor With Stacked Memory. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 188–190, 2012

[6] Sherkar Borkar. Designing Reliable Systems From Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10 – 16, 2005

[7] Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. Reliable On-chip Systems in the Nano-Era: Lessons Learnt and Future Trends. In *Annual Design Automation Conference*, DAC '13, pages 99:1–99:10, Austin, Texas, 2013. ACM

[8] James F. Ziegler and William A. Lanford. Effect of Cosmic Rays on Computer Memories. *Science*, 206(4420):776–788, 1979

sient, which means they are only visible for a limited period of time before the affected component returns to a correct state. As an example, consider a bit in memory that flips due to a cosmic ray strike: Reading this bit will deliver an erroneous value until the containing memory word is later overwritten with a new datum.

Fault tolerance mechanisms to deal with hardware errors have been devised at both hardware and software levels. While lower-level hardware components are suitable to detect and correct a large number of these errors, they do not possess the necessary system knowledge to do so efficiently. For instance, IBM's S390 processors provide redundancy in the form of lockstepping.[9] However, this effort may not be necessary for all applications, because some software can protect itself, for instance using resilient programming techniques.[10] In such cases, the system could better use the additional hardware for computing purposes.

Existing software-level solutions often make assumptions about how developers write software. For instance, they may require use of specific programming models, such as process pairs.[11] Other solutions apply specific compiler techniques for fault tolerance.[12] These software techniques trade general applicability for fault tolerance.

The operating system bridges the gap between hardware and software. In this thesis I therefore evaluate whether we can strike a balance between flexibility and generality by implementing fault tolerance as an operating system service.

[9] Timothy J. Slegel, Robert M. Averill III, Mark A. Check, Bruce C. Giamei, Barry W. Krumm, Christopher A. Krygowski, Wen H. Li, John S. Liptay, John D. MacDougall, Thomas J. McPherson, Jennifer A. Navarro, Eric M. Schwarz, Kevin Shum, and Charles F. Webb. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, 19(2):12–23, 1999

[10] Andrew M. Tyrrell. Recovery Blocks and Algorithm-Based Fault Tolerance. In *EUROMICRO 96. Beyond 2000: Hardware and Software Design Strategies*, pages 292–299, 1996

[11] Jim Gray. Why Do Computers Stop and What Can Be Done About It? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986

[12] Semeen Rehman, Muhammad Shafique, and Jörg Henkel. Instruction Scheduling for Reliability-Aware Compilation. In *Annual Design Automation Conference*, DAC '12, pages 1292–1300, San Francisco, California, 2012. ACM

1.1 Hardware meets Soft Errors

Sun acknowledged soft errors to be the cause for server crashes in 2000,[13] reportedly costing the company millions of dollars to replace the affected components. Anecdotal evidence of soft errors affecting today's systems can be found across the internet.[14] In Section 2.1 I will give a more thorough overview of scientific studies investigating causes and consequences of such hardware errors.

Not all consequences of a soft error may become immediately visible as a fault. As I will describe in Chapter 2, these errors can also lead to erratic failures: the affected system continues to provide a service and simply generates wrong results. Such behavior also opens up new windows of security vulnerabilities. Dinaburg presented such a vulnerability as an experiment at BlackHat 2011: The authors registered 30 domain names that were one bit off popular domains, but that were not susceptible to be plain typing errors made by users. Examples for such domains were the registration of `mic2osoft.com` (as opposed to `microsoft.com`) and `ikamai.net` (as opposed to `akamai.net`). In a time frame of roughly 6 months, the author observed more than 50,000 accesses from about 12,000 unique clients going to these domains.[15] This experiment shows that a) soft errors today are a real issue for consumer electronic devices, and b) new attack vectors may arise if these errors are not taken care of.

So how do system and hardware vendors deal with soft errors? I will show in Section 2.4 that approaches to detect and correct soft errors exist at both the hardware and software levels. Hardware-level solutions are usually

[13] Daniel Lyons. Sun Screen. Forbes Magazine, November 2000, accessed on April 22nd 2013, mirror: `http://tudos.org/~doebel/phd/forbes2000sun`

[14] Nelson Elhage. Attack of the Cosmic Rays! KSPlice Blog, 2010, `https://blogs.oracle.com/ksplice/entry/attack_of_the_cosmic_rays1`, accessed on April 22nd 2013

[15] Artem Dinaburg. Bitsquatting: DNS Hijacking Without Exploitation. *BlackHat Conference*, 2011

expensive in terms of production cost, chip area, resource consumption, or runtime overhead. Hence, radiation-hardened processors are only used in highly-specific environments: NASA's Mars Rover "Curiosity" for instance employs a radiation-hardened processor that is rumored to cost about 200,000 US$ a piece.[16]

In contrast, the majority of computing systems from servers to personal computers to mobile phones are built from *commercial off-the-shelf (COTS)* hardware components. These components are designed to provide good overall performance at the minimum possible cost. Customized hardware is expensive unless there is a large market and therefore COTS systems are the last to see wide-spread deployment of hardware defenses against soft errors. This in turn calls for the use of software-level fault tolerance methods.

Software-implemented hardware fault tolerance can be categorized into two classes: compiler-level techniques aim to generate resilient and self-validating machine code,[17] whereas replication-based solutions run multiple instances of a program and compare their outputs.[18]

CLAIM: Observing the need for software-level fault tolerance, I develop ASTEROID, an operating system (OS) design that protects applications against both permanent and transient hardware faults on COTS hardware platforms. To achieve fault tolerance I implement replication as an operating system service. The system uses modern multi-core CPUs to parallelize replicated execution and thereby reduce runtime overheads. The architecture detects errors by comparing replicas' states at synchronization points. Once detected, the system corrects errors by performing majority voting or by incorporating alternative recovery strategies, such as application-level checkpointing.

1.2 An Operating System for Tolerating Soft Errors

Practical systems today consist of a complex web of interacting software components, which a fault-tolerant operating system needs to accommodate. Some of these components are small enough to be rewritten for fault tolerance. However, the majority of existing software is too large and too complex to be rewritten from scratch. Hence, my solution needs to provide fault tolerance to real-world applications that are not optimized for dealing with hardware faults.

If those applications are available as open-source software, compiler-level transformations can improve fault tolerance without requiring expensive hardware extensions. Fault-tolerant compilers can generate machine code that performs operations multiple times using different hardware resources.[17] Other tools extend the data domain a program operates on and transform operands using arithmetic encoding. This mechanism allows to detect whether a value can be a valid result of a preceding arithmetic operation.[19]

Unfortunately, there is a substantial amount of software that we cannot protect by using the mechanisms described above. These programs are provided in their binary form only. As this form of distribution is the business model of major companies such as Microsoft, Apple, and Oracle, we can safely assume

[16] John Rhea. BAE Systems Moves Into Third Generation RAD-hard Processors. Military & Aerospace Electronics, 2002, accessed on April 22nd 2013, mirror: `http://tudos.org/~doebel/phd/bae2002/`

[17] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization*, CGO '05, pages 243–254, 2005

[18] 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, 2009

[19] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *International Conference on Computer Safety, Reliability and Security*, Safecomp'10, Vienna, Austria, 2010

it to be the major way of supplying proprietary software to customers. The rising sales numbers of software distributors for mobile platforms, such as Apple's *AppStore* or *Google Play* indicate that this fact is likely to remain a reality.

If this problem only affected end-users, we might avoid it by trusting software developers to use the proper tools to protect their applications. However, the end-user cannot validate that such tools were used when downloading software through the internet. Furthermore, commercial libraries (for instance Intel's PIN instrumentation library[20]) are often shipped as binaries only, in which case a developer cannot apply compiler-based protection anymore. Lastly, existing software tends to be used for a long time and it might not be possible to replace such legacy software with newer versions immediately. For these three reasons I aim to find mechanisms that provide fault tolerance to binary-only applications.

[20] Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools With Dynamic Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM

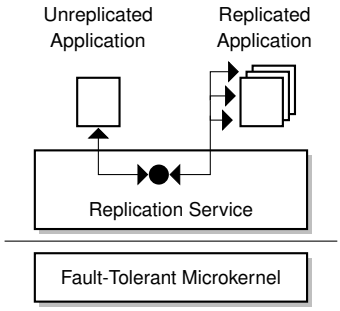

Figure 1.1: ASTEROID Resilient OS Architecture

CLAIM: The operating system is responsible for merging the different requirements of all applications and providing them with a fault-tolerant execution environment. The ASTEROID OS architecture introduced in this thesis and depicted in Figure 1.1 does not enforce a specific model of protection. Instead, different types of applications are supported:

- Binary applications built without specific fault tolerance mechanisms are transparently replicated in order to protect them against soft errors.
- Applications that are protected using fault tolerant compiler techniques or algorithms can be run unmodified without incurring replication-related runtime or resource overheads.
- Applications integrate with each other regardless of the fault tolerance model that is chosen to protect them. Unreplicated applications can interact with replicated applications without having to be aware of this fact as depicted in Figure 1.1.

ASTEROID is based on the FIASCO.OC microkernel. Microkernels provide strong isolation between small sets of software components. This design principle restricts the effects of hardware errors to single software components[21] and allows for fast recovery once an error is detected.[22]

In the field of distributed systems, *state-machine replication* has long been used to achieve fault tolerance.[23] Potentially faulty server nodes are considered black boxes that implement state machines delivering identical outputs when presented with the same sequence of inputs. This property allows to instantiate multiple such server nodes, let them process inputs and detect a failing node either as it delivers an output different from the majority of nodes or because it delivers no output at all.

Redundant Multithreading (RMT) is a technique similar to state-machine replication. In this case the black box is a thread either at the hardware[24] or software level.[25] Multiple copies (replicas) of a thread are instantiated and execute identical code. The RMT system may run those threads independently on different hardware resources as long as they only process internal state.

[21] Jorrit N. Herder. *Building a Dependable Operating System: Fault Tolerance in MINIX3*. Dissertation, Vrije Universiteit Amsterdam, 2010

[22] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot: A Technique For Cheap Recovery. In *Symposium on Operating Systems Design & Implementation*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association

[23] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990

[24] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. *SIGARCH Comput. Archit. News*, 28:25–36, May 2000

[25] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed Software-based Redundant Multithreading for Transient Fault Detection. In *International Symposium on Code Generation and Optimization*, CGO '07, pages 244–258, 2007

Once a thread's state is made externally visible (e.g., written to memory or used as parameter to a system call), the replicas' states are compared to detect potential errors.

Replication-based fault tolerance allows to treat applications as black boxes, not requiring any knowledge about program internals. By distributing replicas across the available compute cores and minimizing the required state comparisons, replication can achieve low runtime overheads. These two properties replication an attractive fault tolerance technique. The main disadvantage of replication-based fault tolerance is the increase in required resources (e.g., N replicas will roughly consume N times the amount of CPU time and N times the amount of memory compared to a single application instance). However, modern servers and even laptop computers provide users with an abundance of processing elements and memory, which are underutilized in the common case.[26] Therefore, replication becomes a feasible alternative if the user is willing to trade resources for low runtime overhead and fast error recovery times.

[26] James Glanz. Power, Pollution and the Internet. The New York Times, accessed on July 1st 2013, mirror: `http://os.inf.tu-dresden.de/~doebel/phd/nyt2012util/article.html`, September 2012

CLAIM: The main contribution of my thesis is ROMAIN, an operating system service that uses redundant multithreading to protect unmodified binary applications from hardware errors. The service's structure is depicted in Figure 1.2 and solves the following problems:

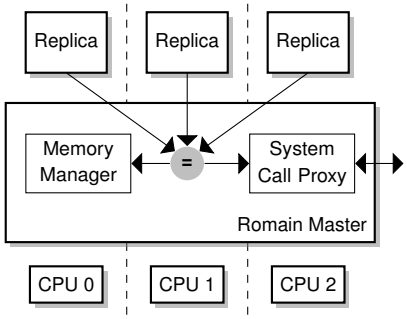

Figure 1.2: Replicated Application

1. Instead of implementing expensive binary recompilation techniques, ROMAIN reuses existing features provided by the FIASCO.OC microkernel to implement redundant multithreading. A *master process* manages replication for a single program. The master maps replicas to OS threads and runs them in isolated address spaces to prevent undetected fault propagation. To obtain low runtime overheads, replicas are distributed across the available physical CPU cores. ROMAIN's general architecture is introduced in Section 3.2

2. ROMAIN transparently manages replicas' resources, such as memory and kernel objects. Applications do not need to be aware of the replication framework and can be implemented using any programming language and development model. Sections 3.4 and 3.5 provide details on replica resource management.

3. Applications do not run isolated, but interact with the rest of the system through system calls and shared-memory communication channels. ROMAIN allows replicated applications to use both mechanisms. Shared memory requires special handling, because such channels may constitute potential input sources influencing replicated program execution. Therefore shared-memory access must not happen without involving the replication service. I will describe system call handling in Section 3.3 and discuss problems related to shared memory in Section 3.6.

4. Multithreaded applications cannot easily be replicated, because scheduling-induced non-determinism may lead to differing behavior between replicas. The ROMAIN master would detect this behavioral divergence. In the best case this would merely cost unnecessary time for correcting such a false positive error. However, in an even worse case all replicas of a program may have diverged in a way that does no longer allow for error correction at all. ROMAIN implements two ways of enforcing deterministic behavior across multithreaded replicas, which I will explain in Chapter 4.

1.3 Whom can you Rely on?

The ASTEROID operating system architecture allows user-level applications to detect and recover from soft errors. However, ASTEROID relies on a subset of hardware and software components to always function correctly. This set comprises the ROMAIN replication service and the underlying OS kernel. Other software-based fault tolerance methods share this problem, although the concrete set of required components varies: Some methods rely on a fully-functioning Linux kernel.[27] Others additionally rely on the correct operation of system libraries, such as the thread library.[28] I refer to such sets of required components as the *Reliable Computing Base* (RCB).

As ROMAIN does not protect the RCB, ASTEROID needs to employ alternative mechanisms to make the RCB reliable. These additional mechanisms come at an additional cost. The type of cost depends on what exact mechanism is applied to protect the RCB: Using fault-tolerant algorithms to implement the RCB will require additional development effort. Protecting the RCB using compiler-based fault tolerance may increase its runtime overhead. Integrating specially hardened, non-COTS hardware components into our system will increase hardware cost.

CLAIM: I introduce the concept of the Reliable Computing Base (RCB) in Chapter 6 and identify the hardware and software components that are part of ASTEROID's RCB. I show that other software-implemented fault tolerance also possess an RCB and that the OS kernel is part of this RCB in most cases. Based on this analysis I present three studies that analyze how RCB components can be protected against the effects of hardware errors:

1. As the OS kernel constitutes a major part of any RCB, I use fault injection experiments to analyze the FIASCO.OC kernel's vulnerability against hardware faults. Based on these findings I discuss potential paths towards protecting FIASCO.OC in future work.
2. Current COTS hardware is becoming more and more heterogeneous by incorporating different types of compute nodes that vary with respect to their processing capabilities and energy requirements. Other researchers suggested that this may lead to the advent of manycore processors with mixed reliability properties.[29]

[27] 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, 2009

[28] Yun Zhang, Jae W. Lee, Nick P. Johnson, and David I. August. DAFT: Decoupled Acyclic Fault Tolerance. In *International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 87–98, Vienna, Austria, 2010. ACM

[29] L. Leem, Hyungmin Cho, J. Bau, Q.A. Jacobson, and S Mitra. ERSA: Error Resilient System Architecture for Probabilistic Applications. In *Design, Automation Test in Europe Conference Exhibition*, DATE'10, pages 1560–1565, 2010

Assuming that such hardware will become COTS at some point in time, we can map RCB software to those components with low vulnerability – such as hardware-protected CPU cores – while we can run ROMAIN's replicas on more fast, cheap, but more vulnerable processing elements.

Such an architecture will require fewer resilient than non-resilient CPUs. As non-resilient CPUs occupy less chip area, this design allows for an integration of more processing elements on a single chip while allowing fault tolerant execution. I show that the ASTEROID architecture can efficiently be implemented on top of such hardware.

3. ROMAIN's goal is to protect unmodified binary-only applications. However, we still have full control over the source code of all RCB components. Hence, applying compiler-based fault tolerance may be a feasible way to protect them. While this will increase ASTEROID's error coverage, it will also lead to an increased runtime overhead. I approximate this overhead using simulation experiments and show that a hybrid approach that protects RCB components using compiler methods while replicating user programs using ROMAIN is a promising approach towards a fully protected software stack.

2
Why Do Transistors Fail And What Can Be Done About It?

In this chapter I present the fault model I address in my thesis. I first give an overview about how cosmic radiation, aging, and thermal effects can lead to the failure of hardware components. Thereafter I introduce a taxonomy of fault tolerance that I am going to use throughout my dissertation. In the final part of this chapter I review existing fault tolerance techniques to motivate assumptions and design goals driving my operating system design.

2.1 Hardware Faults at the Transistor Level

The integrated circuits that form today's hardware components are built from *metal-oxide-semiconductor field-effect transistors (MOSFETs)*.[1] These transistors can suffer from a range of hardware faults that may cause them to fail. In this section I give an overview of what makes transistors error-prone. Unless stated otherwise, I base this summary on Mukherjee's book on fault-tolerant hardware design.[2]

Figure 2.1 shows a MOSFET model. Two semiconductors, source and drain, are separated by a bulk substrate. During production, source and drain are doped to create an n-type semiconductor. In contrast, the bulk substrate is deprived of electrons. The combination of these layers forms a p-type semiconductor. The boundaries between these regions act as diodes and prevent current flowing from source to drain.

A gate electrode sits on top of the transistor. A non-conducting oxide layer (usually silicon-dioxide SiO_2) isolates this electrode from the bulk. If we apply a voltage to the gate electrode an electric field is created. As shown in Figure 2.2, positive electron holes in the p-type substrate are repelled from the gate, whereas negative electrons are pulled towards the gate and accumulate below the oxide layer.

If the gate voltage exceeds a certain *threshold voltage* U_{thr}, the number of electrons below the oxide layer is sufficient to create a conducting channel between source and drain. The accompanying charge at the gate electrode is termed *critical charge* Q_{crit}.

Hardware vendors aim to decrease MOSFET sizes as far as physically possible. Smaller transistors consume less power and allow to integrate a larger amount of memory and processing elements into the same chip area. However, there are three groups of effects that let smaller MOSFETs fail

[1] Dawon Kahng. Electrified Field-Controlled Semiconductor Device. US Patent No. 3,102,230, `http://www.freepatentsonline.com/3102230.html`, 1963

[2] Shubhendu Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008

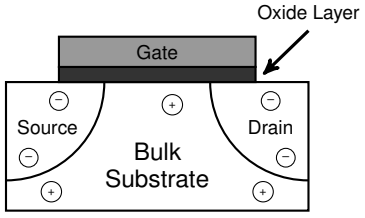

Figure 2.1: Model of a metal-oxide semiconductor field effect transistor (MOSFET)

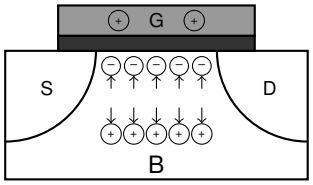

Figure 2.2: MOSFET when switched to conducting state

more often: (1) variability introduced during the manufacturing process may alter the behavior of identically designed transistors, (2) aging and thermal effects may cause a properly functioning transistor to fail, and (3) radiation may induce errors into processing and storage elements. I will now survey research into these effects in more detail.

2.1.1 Manufacturing Variability

As transistors scale down, the gate oxide layer as well as the transistor's channel size shrink, resulting in a smaller number of atoms within a single transistor. Due to this fact, the critical charge and threshold voltage to switch the transistor's state decrease. This eventually leads to a reduced energy consumption.[3] While saving energy is an advantage, manufacturing smaller transistors becomes harder.

The process of doping semiconductor material with excess electrons is far from precise. Instead, the dopant atoms are randomly distributed. For smaller transistors, the total number of atoms is small and therefore even tiny random variations between transistors can have a high impact on their electrical properties. As a consequence, transistors may exhibit large variations in terms of threshold voltage and leakage current.[4]

Reid and colleagues simulated the effects of random atom placement on a large number of 35 nm and 13 nm transistors.[5] They found that smaller structure sizes lead to a larger distribution of threshold voltages across these devices. While the majority of transistors still exhibits correct behavior, they showed that at a channel length of 13 nm a significant amount of MOSFETs falls into ranges where the threshold voltage is either very high or close to zero. Both effects prevent the transistor from switching states at all.

Even when turned off, there is a small static current flowing through a transistor.[6] Studies showed that depending on manufacturing issues, this leakage current greatly varies across chips, even if these originate from the same wafer.[7] In extreme cases this means that processors exceed their planned power budget. This is not an immediately visible malfunction, but it will render mobile devices unusable after a short amount of time if the CPU drains all battery power.

As the effects described above occur in the manufacturing phase, hardware vendors can detect them during stress-testing, which they perform before shipping their products. However, similar effects to manufacturing errors can also arise much later due to chip aging.

2.1.2 Aging and Thermal Effects

At runtime, the transistors forming a semiconductor circuit switch frequently. This switching causes voltage and temperature stress, which leads to a degradation of transistor operation over time. The three main contributors to this degradation are (1) hot-carrier injection, (2) negative-bias temperature instability (NBTI), and (3) electromigration.[8]

Electrons traveling from a MOSFET's source to drain differ with respect to the energy they carry. Highly energetic (hot) carriers sometimes do not pass from source to drain, but instead hit the oxide layer that isolates the gate electrode. Thereby they either show up as leakage current or become trapped

[3] Yuan Taur. The Incredible Shrinking Transistor. *IEEE Spectrum*, 36(7):25–29, 1999

[4] Miguel Miranda. When Every Atom Counts. *IEEE Spectrum*, 49(7):32–32, 2012

[5] Dave Reid, Campbell Millar, Gareth Roy, Scott Roy, and Asen Asenov. Analysis of Threshold Voltage Distribution Due to Random Dopants: A 100,000-Sample 3-D Simulation Study. *IEEE Transactions on Electron Devices*, 56(10):2255–2263, 2009

[6] Kaushik Roy, Saibal Mukhopadhyay, and Hamid Mahmoodi-Meimand. Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-Submicrometer CMOS Circuits. *Proceedings of the IEEE*, 91(2):305–327, 2003

[7] Lucas Wanner, Charwak Apte, Rahul Balani, Puneet Gupta, and Mani Srivastava. Hardware Variability-Aware Duty Cycling for Embedded Sensors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(6):1000–1012, 2013

[8] John Keane and Chris H. Kim. An Odomoeter for CPUs. *IEEE Spectrum*, 48(5):28–33, 2011

inside the gate oxide. The latter process is called hot-carrier injection and results in a shift in the transistor's threshold voltage U_{thr}.[9]

Switching a transistor frequently increases the temperature under which the device operates. At high temperatures, electrons passing through the interface between bulk and gate oxide may destroy the chemical bonds within the oxide and create positively charged Si^{+} ions. Thereby they increase the number of p-dopants in the transistor, which in turn means that a larger threshold voltage is required to switch the transistor. This process is called negative-bias temperature instability (NBTI)[10].

A third effect does not impact the transistor but rather the metal interconnects between transistors. Over time high-energy electrons may cause metal atoms to move out of their position. This process, called electromigration,[11] leads to voids within the wire that prevent current from flowing and therefore break the interconnect. The missing material may additionally move to other locations in the interconnect and create short circuits there.

Circuits suffering from hot-carrier injection or electromigration permanently malfunction. Vendors therefore carefully analyze their circuit's aging characteristics so that these aging effects only set in after a given age threshold. In contrast, the effects of NBTI partially[12] revert if the gate voltage is turned off and temperature decreases. These faults are therefore transient.

[9] Waisum Wong, Ali Icel, and J.J. Liou. A Model for MOS Failure Prediction due to Hot-Carriers Injection. In *Electron Devices Meeting, 1996., IEEE Hong Kong*, pages 72–76, 1996

[10] Muhammad Ashraful Alam, Haldun Kufluoglu, D. Varghese, and S. Mahapatra. A Comprehensive Model for PMOS NBTI Degradation: Recent Progress. *Microelectronics Reliability*, 47(6):853–862, 2007

[11] James R. Black. Electromigration – A Brief Survey and Some Recent Results. *IEEE Transactions on Electron Devices*, 16(4):338–347, 1969

[12] Literature distinguishes reversible short-term NBTI and irreversible long-term NBTI.

2.1.3 Radiation-Induced Effects

35 years ago, Ziegler and Lanford showed that radiation may also trigger malfunctions in semiconductors.[13] There are two main sources for these radiation effects: First, cosmic rays may penetrate earth's atmosphere and influence transistors. Second, the materials that are used for packaging circuits carry a certain amount of terrestrial radiation. The radioactive decay of these materials may therefore also influence circuit behavior.

Given a fixed hardware structure size, the probability of suffering from a packaging-induced radiation effect is fixed. Cosmic radiation in contrast becomes stronger with higher altitudes.[14]

When a single MOSFET is struck by radiation, particles may create electron-hole pairs which thereafter recombine and therefore create a flowing current. This process may increase the transistor's charge. If the charge then exceeds the critical charge Q_{crit}, the transistor's state may switch.

With smaller transistor sizes, Q_{crit} decreases and the transistors become more vulnerable against environmental radiation. However, at the same time the transistor surface that may be hit by a ray also decreases and thereby the probability of a individual transistor being struck by radiation decreases. Initially, overall transistor vulnerability dropped when moving below 135 nm technologies. However, a study by Oracle Labs indicates that transistor vulnerabilities begin to rise again as transistor sizes shrink below 40 nm.[15]

As a single radiation event can force a transistor's state to change, these events are termed *single-event upsets (SEU)*. In contrast to the previously discussed error types, SEUs are non-permanent. Any future reset of the transistor, e.g., by applying the threshold voltage to the gate electrode, will return the transistor to a valid state. Correcting SEUs is therefore easier,

[13] James F. Ziegler and William A. Lanford. Effect of Cosmic Rays on Computer Memories. *Science*, 206(4420):776–788, 1979

[14] Ziegler, James F. and Curtis, Huntington W. et al. IBM Experiments in Soft Fails in Computer Electronics (1978–1994). *IBM Journal of Research and Development*, 40(1):3–18, 1996

[15] A. Dixit and Alan Wood. The Impact of new Technology on Soft Error Rates. In *IEEE Reliability Physics Symposium*, IRPS'11, pages 5B.4.1–5B.4.7, 2011

because overwriting a faulty memory location will fix a radiation-induced error. For this reason, SEUs are also called *soft errors*.

SUMMARY: Computer architects understand that semiconductors suffer from manufacturing and aging faults as well as the influence of radiation. These faults can be permanent or transient. A reliable system needs to cope with both cases of faults.

2.2 Faults, Errors, and Failures – A Taxonomy

Before I review how hardware and software-level solutions provide fault tolerance against the types of hardware errors described above, it is necessary to introduce a terminology that we can use to talk about cause and effect of these errors. In this thesis I am going to use a terminology that was introduced by Avizienis,[16] which I summarize below.

[16] Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004

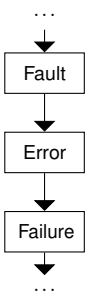

Figure 2.3: Chain of errors

The cause, effect, and consequence of component misbehavior form a chain as depicted in Figure 2.3. We call the ultimate cause of misbehavior a *fault*. In the exemplary case of radiation-induced soft errors, a cosmic ray strike hitting a transistor constitutes a fault.

Malfunction of the affected device only occurs if a fault becomes *active* and modifies the component's internal state. This is an *error*. A radiation fault is for instance activated if the particle's charge exceeds the affected transistor's critical charge Q_{crit} and the transistor is currently not in its conducting state.

If a component's state is modified due to an error, the component may provide a service that deviates from the expected service. For instance, a transistor struck by radiation may be part of a memory cell and the radiation fault may cause the memory cell's content to change. If this erroneous content is then read, it may impact further computations. This externally visible deviation is called a *failure*.

Not every error will eventually lead to a visible component failure. For instance, even if a memory cell's value is modified, this data does not impact system behavior as long as it is not used. If a fault or error does not escalate into a failure, we call this a *masked fault* or *masked error* respectively.

The terminology so far only considers a single component. Computer systems are complex networks of interacting components and therefore the distinction between faults, errors, and failures fades. One component's failure may be input to a second component. From the second component's perspective this will be a fault, which again may activate and trigger an error and potentially escalate into a failure. This domino effect is called *fault propagation*.

We can furthermore distinguish faults by their lifetime: *Permanent faults* enter the system at a certain point in time and the affected component thereafter constantly behaves incorrectly. For example, production errors in processors may lead to situations in which certain bits of a register always deliver the same value or where a broken interconnect never transmits any current.

In contrast, *transient (or soft) faults* vanish after a certain period of time. This is the case for radiation-induced faults I introduced in Section 2.1 on page 15. These faults may cause single bits of a register to change their value.

However, overwriting the register with a new value will correct these errors, because writing will recharge the affected transistors.

Literature sometimes uses the term *intermittent fault* to denote a fault that occasionally activates and then vanishes again. These faults are believed to stem from complex interactions between multiple faulty hardware components.[17]

[17] Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. Towards Understanding The Effects Of Intermittent Hardware Faults on Programs. In *Workshops on Dependable Systems and Networks*, pages 101–106, June 2010

2.2.1 Fault-Tolerant Computing

Fault-tolerance mechanisms strive to prevent faults from escalating into failures by either masking faults or detecting and correcting errors before they lead to failures. Decades of research and product development have produced guidelines on how to design computer components so they can tolerate faults efficiently.

If a device stops providing a service when encountering a failure, it is considered a *fail-stop* component.[18] Such a component cannot produce erroneous output. Therefore, a fault-tolerance mechanism can easily detect if this unit becomes unresponsive. In contrast, detecting erroneous output is a much harder task. Some publications also refer to components that stop providing output after a failure as *fail-silent* components.[19]

If the termination of service happens immediately after the service provider failed, the respective component is considered *fail-fast*.[20] Failing fast means that the propagation of an error into other components of the system is limited. Consequently, repairing the system becomes easier.

I will examine existing fault-tolerance mechanisms and their properties more closely in Section 2.4. However, to do so we need metrics that allow us to evaluate the suitability of a certain mechanism in a given scenario.

[18] Richard D. Schlichting and Fred B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-tolerant Computing Systems. *ACM Transactions on Computer Systems*, 1:222–238, 1983

[19] Francisco V. Brasileiro, Paul D. Ezhilchelvan, Santosh K. Shrivastava, Neil A. Speirs, and S. Tao. Implementing Fail-Silent Nodes for Distributed Systems. *Computers, IEEE Transactions on*, 45(11):1226–1238, 1996

[20] Jim Gray. Why Do Computers Stop and What Can Be Done About It? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986

2.2.2 Reliability Metrics

When assessing fault tolerant systems, Gray distinguishes between *reliability* and *availability*:[20]

- "Reliability is not doing the wrong thing." By this definition, a fail-stop system is considered reliable, because it never provides a wrong result.
- "Availability is doing the right thing within the specified response time." This definition adds the requirement of a timely and correct response. To be available, a fail-stop system needs to be augmented with a fast and suitable recovery mechanism.

Researchers use temporal metrics as shown in Figure 2.4 to quantify these terms. The expected time between failure events in a component is termed the component's *Mean Time Between Failures (MTBF)*. MTBF is often measured in years. Computer architects use the rate of *Failures In Time (FIT)*, which is closely related to the MTBF and defined as $FIT := 1/MTBF$. The FIT rate is usually given in failures per one billion hours of operation.

The lifetime and evolution of a fault can also be measured: the time between the occurrence of a fault and its activation is called the *Fault Latency*. The subsequent time elapsing between the manifestation of an error and its detection by a fault tolerance mechanism is termed *Error Detection Latency*.

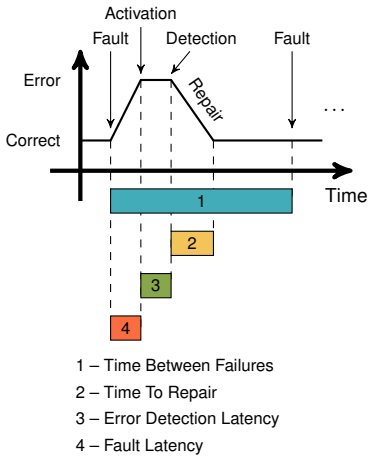

Figure 2.4: Temporal Reliability Metrics

Applying fault tolerance to a system adds *development, execution time and resource overheads*. For instance, periodically taking application checkpoints adds both additional execution time and memory consumption. These overheads are of concern for evaluating fault tolerant systems, because they are often paid regardless of whether the system is hit by a fault. Additionally, once an error has occurred, the system needs to repair this error. The time to do so is called the *Mean Time To Repair (MTTR)*. Gray uses MTBF and MTTR to define an availability metric: [20]

$$Availability := \frac{MTBF}{MTBF + MTTR}$$

Lastly, the fraction of errors covered by a fault tolerance mechanism in contrast to the overall amount of errors a system may encounter is called *error coverage*.

An ideal fault-tolerant system achieves a high error coverage while keeping overheads and repair times minimal. Unfortunately, in practice there is no free lunch. Real solutions therefore need to choose between increased execution time overheads and increased error coverage and find a sweet-spot for their purposes. Therefore, I review existing solutions to fault tolerance with respect to their cost and applicability in the next section.

SUMMARY: Component malfunctions follow a chain of events: faults affect the component and escalate to errors in the component's state. An externally visible error is considered a failure.

Fault tolerant systems are designed to either mask errors or detect and correct them in time. Reliability metrics, such as MTBF, MTTR, and error coverage, allow to evaluate the impact and usefulness of fault tolerance mechanisms.

2.3 Manifestation of Hardware Faults

The errors discussed in Section 2.1 originate from the transistor level. My thesis focuses on tolerating the software-visible effects of these errors. Hence, it is useful to understand how hardware errors manifest from an application's point of view. Therefore, I will now explore studies that analyze the rate at which errors occur in today's hardware and the impact they have on program execution.

2.3.1 Do Hardware Errors Happen in the Real World?

Hardware faults are rare events in actual systems. This makes studying their effects a tedious task, because we need to obtain data from a large set of computers over a long time. The resources required to do so are often unavailable to researchers and even to most industrial hardware and software vendors. As an example, Li and colleagues monitored a set of more than 300 computers in production use over a span of three to seven months and were only able to attribute two observed errors as being the likely effects of a soft error in memory.[21]

Researchers often study hardware error effects by looking at memory hardware. This is appropriate for two reasons: First, memory makes up a

[21] Xin Li, Kai Shen, Michael C. Huang, and Lingkun Chu. A Memory Soft Error Measurement on Production Systems. In *USENIX Annual Technical Conference*, ATC'07, pages 275–280, June 2007

large fraction of hardware circuits in a modern computer. Halfhill reports that 69% of a modern Intel CPU's chip area are used for the L3 cache.[22] Additional area is consumed off-chip by the several gigabytes of RAM in modern machines. Memory is therefore most likely to encounter a hardware fault. Second, detecting memory errors is straightforward by either monitoring complete memory ranges in software or using error information provided by the memory controller.[21]

Instead of monitoring computers for a long time, we can also expose hardware to an increased rate of radiation in a special experiment setup. In these investigations hardware is ionized at a rate much higher than cosmic radiation. This approach amplifies the number of soft errors that affect the investigated devices. Autran and colleagues used such an experiment to measure the SEU effects on 65 nm SRAM cells. They extrapolate from their measurements that at sea level we are likely to see a FIT rate of 759 bit errors per megabit of memory within one billion hours of operation.[23]

SRAM cells are used in modern CPUs for implementing on-chip caches. If we assume an 8 MB L3 cache size as is common in Intel's Core i7 series of processors, Autran's FIT/Mbit rate leads us to a cache FIT rate of $759\ FIT/Mbit * 64\ Mbit = 48{,}576\ FIT$. This number corresponds to an MTBF of about 2.3 years. This fault rate may appear negligible from the perspective of a single user's home entertainment system. However, modern large scale computing installations — such as high-performance computers and data centers powering the cloud — contain thousands of processors. Consequently, failures in these systems happen in intervals of minutes instead of weeks or months.

Focusing on such large-scale installations, Schroeder and colleagues carried out a field study of DRAM errors by monitoring the majority of Google Inc.'s server fleet for nearly two years.[24] They found that about a third of all computers experienced at least one memory error within a year. The error rates strongly correlated with hardware utilization. Furthermore, they observed that DRAMs that had already encountered previous errors were more likely to suffer from errors in the future. From these observations they concluded that memory faults are dominated by permanent faults, because otherwise errors would be distributed more evenly across the machines under observation. They then confirmed this assumption with a second study incorporating an even larger range of machines.[25]

AMD engineers performed an independent 12 month study using the Jaguar cluster at Oak Ridge National Laboratories, containing 18,688 compute nodes and about 2.69 million DRAM devices.[26] They report that while only 0.1% of all DRAMs encountered any kind of fault during their study, the large number of devices translates this number into an MTBF of roughly 6 hours.

In contrast to Schroeder's studies, the AMD study aims to attribute errors to their underlying faults. The authors argue that counting errors instead of faults leads to a bias towards permanent errors. Because of their very nature, permanent faults will produce reportable errors until the defective memory bank is replaced. In contrast, radiation-induced soft errors are only reported once and then corrected. By only considering actual faults, the study reports a transient fault rate of 28%.

[22] Tom R. Halfhill. Processor Watch: DRAM+CPU Hybrid Breaks Barriers. Linley Group, 2011, accessed on July 26th 2013, mirror: `http://tudos.org/~doebel/phd/linley11core/`

[23] 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 65nm SRAM. In *European Conference on Radiation and Its Effects on Components and Systems*, RADECS'08, pages 519–524, 2008

[24] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'09, 2009

[25] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 111–122, London, England, UK, 2012. ACM

[26] Vilas Sridharan and Dean Liberty. A Study of DRAM Failures in the Field. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 76:1–76:11, Salt Lake City, Utah, 2012. IEEE Computer Society Press

As described in Section 2.1, hardware circuitry is constantly shrinking and therefore becoming more vulnerable to both radiation and aging effects. This trend has been pointed out by studies from both industry[27] and academia.[28] As a result, the error rates described above will grow larger with future hardware generations, making the quest for efficient fault tolerance mechanisms more urgent.

[27] Sherkar Borkar. Designing Reliable Systems From Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10 – 16, 2005

[28] Robert Baumann. Soft Errors in Advanced Computer Systems. *IEEE Design Test of Computers*, 22(3):258–266, 2005

SUMMARY: Hardware faults and the resulting failures are a practical problem. Today, the probability of encountering a hardware error in an end user's computer is fairly low.

High-performance computers and data centers already amass a sufficient amount of hardware to encounter such faults on a daily basis. They therefore require protection by hardware or software mechanisms.

Future generations of computer hardware will bring these problems to the consumer market as well.

2.3.2 How do Errors Manifest in Software?

Before designing a fault tolerance mechanism, developers specify how they expect the behavior of a component to change if this component suffers from an error. This *fault model* is used as the basis for evaluating the efficiency of newly implemented mechanisms.

Software-level fault tolerance often assumes that hardware errors manifest as deviations in the state of single bits in memory or other CPU components, such as registers or caches.[29] Based on this assumption, permanent errors are often modeled as *stuck-at errors*, where reading a bit constantly returns the same value. In contrast, a commonly used fault model for transient errors is a *bit flip*, where a memory bit is inverted at a random point in time. The bit then returns erroneous data until the next write to this resource resets the bit to a correct state.

Ideally, to evaluate a fault tolerance mechanism, we would fully enumerate all potential errors according to the assumed fault model and check if these errors are detected and corrected by the mechanism. Unfortunately, this is infeasible because the total set of errors is huge: For instance, when assuming register bit flips, we would have to test the mechanism for a bit flip in every bit of every register for every dynamic instruction executed by a given workload. This leads to millions or billions of potential experiments.

Real-world studies often work around this problem by sampling the error space. Fault injection tools[30] aid in performing coverage analysis. These tools furthermore allow to determine which samples to select in order to get representative results.[31]

The total set of fault injection experiments carried out is called a *fault injection campaign*. In every experiment a single fault is injected. The system's output is then monitored and compared to that of an unmodified execution, the so-called *golden run*. The experiment results are then classified.

[29] V. B. Kleeberger, C. Gimmler-Dumont, C. Weis, A. Herkersdorf, D. Mueller-Gritschneder, S. R. Nassif, U. Schlichtmann, and N. Wehn. A Cross-Layer Technology-Based Study of how Memory Errors Impact System Resilience. *IEEE Micro*, 33(4):46–55, 2013

[30] Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. FAIL*: Towards a Versatile Fault-Injection Experiment Framework. In Gero Mühl, Jan Richling, and Andreas Herkersdorf, editors, *International Conference on Architecture of Computing Systems*, volume 200 of *ARCS'12*, pages 201–210. German Society of Informatics, March 2012

[31] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 123–134, New York, NY, USA, 2012. ACM

The names of these result classes vary throughout literature. Nevertheless, studies often distinguish between four general result classes and I will use the following distinction when evaluating ROMAIN's error coverage capabilities in Chapter 5:

- If the application continues execution without visible deviation and successfully generates the correct output, the error is considered to have *no effect*. This is also called a *benign fault*.
- An error that leads to a visible application malfunction, for instance because it drives the application to access an invalid memory address, is called a *detected error* or a *crash*.
- If the application terminates successfully in the presence of an error, but produces wrong output, the error is considered a *silent data corruption (SDC)* error.
- Sometimes the application does not terminate within a specific time frame at all, for instance because the error affects a loop condition and the program gets stuck in an infinite loop. These cases are classified as *incomplete execution*.

In the following paragraphs I survey six studies that investigated different aspects of how hardware errors affect software.

Error Propagation to Software Saggese and colleagues studied fault effects using gate-level simulators of both a DLX and an Alpha processor.[32] Using these simulators they were able to inject faults into the different functional units of the processor in operation. For logic gates they found that 85% of the injected errors were masked at the hardware level. Consequently, software-level fault tolerance should focus on detecting errors that affect stored data, such as registers or memory.

The authors furthermore analyzed whether failure distributions vary across the different functional units of a processor. I render their results in Figure 2.5. Most notably, they found that execution units (such as the instruction decoder) contribute largely to crashes. In contrast memory accesses make up a large fraction of silent data corruption failures.

Error Manifestation at the OS Level Two studies by Arlat and Madeira inspected how transient memory errors in commercial-off-the-shelf (COTS) hardware impact an operating system. Arlat focused on Chorus[33] whereas Madeira's work considered LynxOS.[34]

Both studies found that – after filtering faults masked by hardware – between 30% and 50% of all errors lead to no difference in application behavior. A fault tolerance mechanism that successfully ignores benign faults while properly handling all others will therefore cause less execution time overhead than a mechanism that tries to correct all errors. Hence, the way a mechanism deals with benign faults constitutes a major opportunity for optimizing performance.

Arlat's study focused on exercising certain kernel subsystems. A large fraction of visible errors triggered CPU exceptions or error handling inside the kernel. These errors can easily be detected by an OS-level fault tolerance mechanism.

[32] Giacinto P. Saggese, Nicholas J. Wang, Zbigniew T. Kalbarczyk, Sanjay J. Patel, and Ravishankar K. Iyer. An Experimental Study of Soft Errors in Microprocessors. *IEEE Micro*, 25:30–39, November 2005

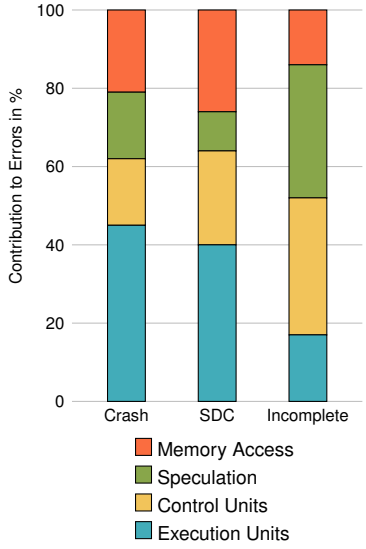

Figure 2.5: Study by Saggese[32] on how soft errors in different parts of the processor manifest as software failures

[33] Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frédéric Salles. Dependability of COTS Microkernel-Based Systems. *IEEE Transactions on Computing*, 51(2):138–163, February 2002

[34] Henrique Madeira, Raphael R. Some, Francisco Moreira, Diamantino Costa, and David Rennels. Experimental Evaluation of a COTS System for Space Applications. In *International Conference on Dependable Systems and Networks*, DSN 2002, pages 325–330, 2002

In contrast to Arlat's study, Madeira also considered applications that spent a substantial amount of time executing in user space. In these experiments, up to 50% of all errors led to silent data corruption or incomplete execution of the user program. These errors cannot easily be detected by the OS, because from the kernel's perspective such a failing application does not behave differently from a program executing normally.

Arlat and Madeira also investigated whether errors propagate through the OS kernel into other applications. They report this to happen rarely and attribute this to the fact that hardware-assisted process isolation is a suitable measure to prevent error propagation across applications. Both kernels make use of this feature. Yoshimura later confirmed that this isolation property also exists for Linux processes.[35]

[35] Takeshi Yoshimura, Hiroshi Yamada, and Kenji Kono. Is Linux Kernel Oops Useful or Not? In *Workshop on Hot Topics in System Dependability*, HotDep'12, pages 2–2, Hollywood, CA, 2012. USENIX Association

Manifestation of Permanent Errors While the previous studies focused on transient errors, Li and colleagues studied the manifestation of permanent errors on a simulated CPU running the Solaris operating system and the SPEC CPU benchmarks.[36] They found that permanent errors are only rarely masked by hardware and seldom manifest as silent data corruption. Instead, these errors often lead to hardware exceptions (such as page faults), which are detected by the OS kernel. The authors also measured that in most cases an error leads to a crash within less than 1,000 CPU cycles. However, in 65% of the cases, the kernel's internal data structures are corrupted before error detection mechanisms are triggered. Therefore, these important data structures require special protection even when a system is protected by special fault tolerance mechanisms.

[36] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 265–276, Seattle, WA, USA, 2008. ACM

Errors Meet Programming Language Constructs In addition to the previous studies, software developers investigated whether certain programming languages or development models influence the reliability of a program. One such investigation was performed by Wang and colleagues, who explored branch errors in the SPEC CPU 2000 benchmarks.[37] They forced the programs to take wrong branches and found that in up to 40% of the dynamic branches the program converged to correct execution without generating wrong data.

The authors attribute these observations to the use of certain programming constructs. For example, they point out that programs sometimes contain alternative implementations of the same feature. In such cases selecting one or the other (by taking a different branch) does not make a difference with respect to program outcome.

A different study by Borchert and colleagues shows that programming language features can also have an impact on the vulnerability of programs.[38] In the C++ programming language, inheritance hierarchies are implemented using a function pointer lookup table, the `vtable`. Borchert et al. evaluated C++ code using fault injection experiments and determined that these `vtable` pointers are especially vulnerable to memory bit flips. They then devised a mechanism to replicate these `vtable` pointers at runtime and thereby increase the program's reliability.

[37] Nicholas Wang, Michael Fertig, and Sanjay Patel. Y-Branches: When You Come to a Fork in the Road, Take it. In *International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 56–, Washington, DC, USA, 2003. IEEE Computer Society

[38] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Protecting the dynamic dispatch in C++ by dependability aspects. In *GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*, Lecture Notes in Informatics, pages 521–535. German Society of Informatics, September 2012

Limitations of the Bit Flip Model The studies discussed in this section analyzed the vulnerability of large application scenarios. Fault injection campaigns for such scenarios are computationally impossible to carry out by simulating hardware at the transistor level. Instead, the studies used simulation software that abstracts away details of the underlying hardware platform in order to gain performance and allow conducting such a vast amount of fault injection experiments.

While these studies allow analyzing the effectiveness of reliability mechanisms, a recent study by Cho points out that the *absolute* numbers produced by high-level simulation need to be taken with a grain of salt:[39] their comparison of transistor-level experiments with memory fault injections based on the bit-flip model indicates that bit flip experiments tend to over-estimate SDC errors, whereas real hardware shows a higher rate of crash errors. This means that high-level fault injection is not suitable to perform an absolute vulnerability analysis of a given system. However, these campaigns are still suitable to investigate whether a fault tolerance mechanism increases reliability with respect to a given fault model. This latter scenario only requires an apples-to-apples comparison between unprotected and protected execution within the same simulator.

[39] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A. Abraham, and Subhasish Mitra. Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–10, 2013

SUMMARY: Both permanent and transient hardware faults can propagate to the software level. They mostly manifest in storage cells, such as registers and main memory. Despite limitations, the bit-flip model is commonly used to analyze the impact these errors have on software and whether fault tolerance mechanisms detect and correct them efficiently.

Depending on the workload, up to 50% of all errors are benign and do not modify program behavior. Fault tolerance mechanisms can leverage this fact to optimize performance if they find a way to ignore benign faults and only pay execution time overhead for actual failures.

2.4 Existing Approaches to Tolerating Faults

In the previous sections I explained why hardware suffers from faults and how these faults manifest at the software level. In this section I review previous work in the field of fault-tolerant computing. I first give an overview of hardware-level solutions that try to prevent hardware faults from ever becoming visible to software.

Hardware extensions increase the required chip area for a processor and therefore make the chip more expensive to produce and consume a higher amount of energy during operation. As an alternative, software-level solutions try to address fault tolerance without relying on dedicated hardware. I explore such mechanisms in the second part of this section.

Operating systems have long tried to increase the reliability of kernel code. While many of these solutions focus on dealing with programming errors, some of their design principles can also aid in dealing with hardware faults. Hence, I review these works in the third part of this section.

2.4.1 Hardware Extensions Providing Fault Tolerance

Computer architects address the problem of failing hardware components by adding additional hardware to detect and correct these failures. These components may be simple copies of existing circuits. More lightweight approaches achieve fault tolerance using dedicated checker components to dynamically validate hardware properties. Lastly, memory cells and buses can be augmented with signature-based checkers to protect stored data.

Replicating Hardware Components One of the simplest ways to achieve fault tolerance in hardware is to *replicate* existing components. This concept is called *N-modular redundancy*.[40] The protected components are instantiated N times and carry out the same operations with identical inputs. Outputs are sent to a voter, which in turn selects the output produced by a majority of all components.

[40] W. G. Brown, J. Tierney, and R. Wasserman. Improvement of Electronic-Computer Reliability Through the Use of Redundancy. *IRE Transactions on Electronic Computers*, EC-10(3):407–416, 1961

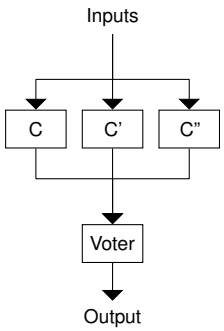

Figure 2.6: Triple modular redundancy (TMR)

Figure 2.6 shows such a system with $N = 3$. This *triple-modular redundant* (TMR) setup is able to correct single-component failures using majority voting. The major drawback of simple replication is that the multiplication of components increases production cost (due to larger chip area used) as well as runtime cost (due to increased energy consumption) of the system. Hence, TMR setups are mainly used in high-end fault tolerant systems, such as avionics[41] and spacecraft.[42]

Redundant Multithreading Modern microprocessors improve instruction-level parallelism by pipelining[43] as well as by executing multiple hardware threads in parallel. The latter technique is known as *simultaneous multithreading* (SMT) or *hyper-threading*.[44] While SMT normally increases utilization of otherwise unused functional units, researchers have also investigated whether this parallelism can be used to increase fault tolerance.

With a technique called *Redundant Multithreading* (RMT), Reinhardt and Mukherjee extended an SMT-capable microprocessor to run identical code redundantly in different hardware threads. Whenever these threads perform a memory access, the RMT hardware extension compares the data involved in this access and resets the processor in case of a mismatch.[45]

RMT's authors use the term *sphere of replication (SoR)* to express which part of the system is protected by a fault tolerance mechanism. They argue that in order to reduce execution time overhead, replicas should not be compared while they only modify state internal to their SoR. Only once this state becomes externally visible (for instance by being written to memory), RMT performs these potentially expensive comparisons. Using this deferred validation, RMT reduces execution time overhead while maintaining strict fault isolation and error coverage.

This last property motivated several software-level solutions that apply ideas similar to RMT, but use software threads. I will discuss these solutions in the following section on software-level fault tolerance mechanisms.

Mainframe Servers Practical applications of RMT's ideas can be found in highly available mainframe servers. IBM's PowerPC 750GX series allows to operate two processors in lockstep mode. The lockstep CPUs execute the

[41] Ying-Chin Yeh. Triple-Triple Redundant 777 Primary Flight Computer. In *Aerospace Applications Conference*, volume 1, pages 293–307, 1996

[42] David Ratter. FPGAs on Mars. *Xilinx Xcell Journal*, 2004

[43] John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2003

[44] Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing on-chip Parallelism. In *International Symposium on Computer Architecture*, pages 392–403, 1995

[45] Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. *SIGARCH Comput. Archit. News*, 28:25–36, May 2000

same code using identical inputs. Whenever they write data to memory, an additional validator component compares the outputs and raises an error upon mismatch.[46]

HP's NonStop Advanced Architecture[47] aims to decrease production cost by working with COTS components wherever possible. These components are cheaper because they are produced for the mass market instead of being specifically designed for highly customized use cases.

NonStop servers include COTS processors, which run replicated. To improve fault isolation, NonStop pairs processors from different physical processor slices, each using individual power supplies and memory banks. A dedicated (non-COTS) interconnect intercepts and compares the replicated processors' memory operations, flags differences, and implements recovery of failed processors. The comparator component itself can also be replicated so that it does not become a single point of failure for this system.[48]

[46] IBM. PowerPC 750GX lockstep facility. IBM Application Note, 2008

[47] HP NonStop originates from the products of Tandem Computer Inc.

[48] David Bernick, Bill Bruckert, Paul del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. NonStop: Advanced Architecture. In *International Conference on Dependable Systems and Networks*, pages 12–21, June 2005

Dedicated Checker Circuits Replicating whole hardware components requires a large amount of additional chip area. Alternative approaches add small, light-weight checker circuits that monitor the execution of a single component in order to detect invalid behavior. For instance, Austin's DIVA architecture, which is shown in Figure 2.7, extends a standard superscalar processor pipeline with an additional validation stage.[49]

The traditional, unmodified pipeline stages (shown white in the figure) fetch and decode instructions before submitting them to an out-of-order execution stage. By default, instructions remain in this stage until they along with all their dependencies have been successfully executed. Thereafter, the instructions are committed, which makes their results externally visible.

DIVA extends this pipeline by inspecting all instructions, their operands, and their results before they reach the commit stage. Additional components recompute the result (CHKComp) and revalidate data load operations (CHKComm). As the validators see all instructions in commit order, they can be less complex than a complete out-of-order stage, because there are no unresolved dependencies and no speculative execution is required. Furthermore, Austin suggests to build the validators from components with larger structure sizes, which in turn are less susceptible to hardware faults.

As a result, the DIVA pipeline can be built from standard, error-prone components and only the validators need to be carefully constructed in order to protect the remainder of the system. Performance overhead is introduced only by the additional pipeline stage. The checker cores only slightly impact performance, because they only validate instructions that are actually committed. This means, there is no overhead from checking speculatively executed instructions and the checkers never have to stall for data from memory, because this data is already available from the previous phases.

While DIVA verifies correct execution using recomputation, other architectures address the problem that the physical properties of hardware change due to faults. As explained in Section 2.1, aging and temperature-related hardware faults may modify a transistor's critical switching voltage. With this modification the time needed for state changes increases. This effect becomes visible once switch timing exceeds the processor's clock period, because then

[49] Todd M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *International Symposium on Microarchitecture*, MICRO'32, pages 196–207, Haifa, Israel, 1999. IEEE Computer Society

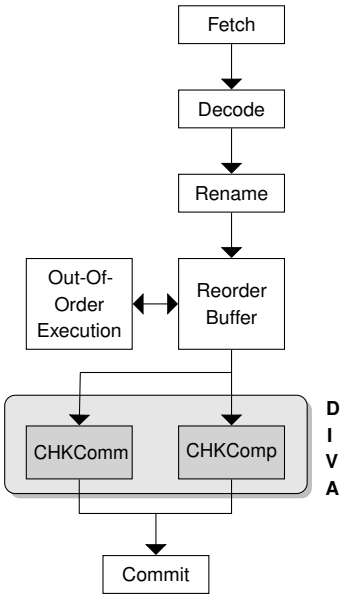

Figure 2.7: DIVA: The Dynamic Implementation Verification Architecture extends a state-of-the-art microprocessor pipeline with an additional execution validation stage.

the results of a computation may not reach other circuits fast enough to carry on computation.

Hardware vendors typically increase timing tolerance by running CPUs with a higher supply voltage than actually needed. Thereby, aging-related changes in switching times are hidden by initially switching transistors faster than required for a certain clock frequency. To cope with additional variability arising from the manufacturing process, these supply voltage margins need to be chosen conservatively. Due to these higher margins a processor may consume more energy than would otherwise be necessary.

The Razor architecture[50] tries to reduce these voltage margins by introducing additional timing delay checkers into a processor. These checkers test if the timing of the critical path through a CPU is still within the specified range. Only if the checkers detect a timing violation the supply voltage is increased to neutralize this effect.

[50] Ernst, Dan and Nam Sung Kim et al. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *International Symposium on Microarchitecture*, MICRO'36, pages 7–18, 2003

Error Detection Using Data Signatures The mechanisms discussed so far protect the actual execution of a hardware component by validating results and timing. However, as I explained in Section 2.3, the hardware components most susceptible to hardware faults are those units storing and transmitting data. Replicating all data storage in a system would significantly increase resource requirements, because memory and caches constitute a large fraction of today's chip area.

An alternative to keeping copies of all data for fault tolerance is to apply checksumming and only store checksums along with the data. This approach is used in networking, where network protocols add checksums to transmitted data in order to detect transmission failures.[51] Mainframe processors, such as the IBM Power7 series, furthermore use Cyclic Redundancy Checks (CRC) to protect data buses.[52]

[51] J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093

[52] Daniel Henderson and Jim Mitchell. POWER7 System RAS – Key Aspects of Power Systems Reliability, Availability, and Servicability. IBM Whitepaper, 2012

Hardware Memory Protection Modern memory controllers can apply *Error-Correcting Codes* (ECC) to protect data from the effects of hardware faults. ECC adds additional parity or checking bits to data words. These additional bits are updated with every write operation and can be used during a read operation to determine if the values are identical to the previously stored ones.

Memory ECC is usually based on Hamming Codes[53] as they are fast and easy to implement in hardware. Current implementations mostly use a (72,64) code, which means that 64 bits of actual data are augmented with 8 bits of parity. Such codes can correct single-bit errors and detect (but not necessarily correct) double-bit errors (SECDED – single error correct, double error detect).[54]

ECC defers detection of an erroneous memory cell until the next access. Unfortunately, data is sometimes stored without being accessed for a long period of time. In such a time frame multiple independent memory errors might affect the same cell, rendering ECC useless because SECDED codes can only recover from single-bit errors. Advanced memory controllers address this problem by periodically accessing all memory cells. This technique is called *memory scrubbing*.[55] The scrubbing period places an upper bound on each memory cell's vulnerability against multi-bit errors.

[53] Richard W. Hamming. Error Detecting And Error Correcting Codes. *Bell System Technical Journal*, 29:147–160, 1950

[54] Shubhendu Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008

[55] Shubhendu S. Mukherjee, Joel Emer, Tryggve Fossum, and Steven K. Reinhardt. Cache Scrubbing in Microprocessors: Myth or Necessity? In *Pacific Rim International Symposium on Dependable Computing*, PRDC '04, pages 37–42, Washington, DC, USA, 2004. IEEE Computer Society

Hardware research as mentioned in Section 2.1 pointed out that radiation influences are likely to not only affect a single memory cell. Depending on angle and charge of the striking particle, multiple adjacent bits may suffer from a transient error.[56] This makes memory prone to multi-bit errors, which standard ECC cannot correct. Advanced ECC schemes, such as IBM's *Chipkill* address this issue by not computing parity over physically adjacent bits. Instead bits from different memory words and even separate DIMMs are incorporated into a single ECC checksum.[57]

ECC is an effective and well-understood technique for protecting memory cells in hardware. It has become a commodity in recent years and is often considered a COTS hardware component. For this reason many of the software-level mechanisms I will introduce in the next section assume ECC-protected main memory and focus on guarding against errors in the remaining components of a CPU. However, there are two reasons why ECC is not applied in all hardware: First, ECC even in combination with scrubbing and Chipkill technologies can still miss multi-bit errors and studies have shown that this actually happens in large-scale systems.[58] Second, ECC increases the number of transistors in hardware and consequentially impacts energy demand. For the latter reasons, developers of embedded systems and low-cost consumer devices refrain from adding ECC protection to their hardware.

[56] Jose Maiz, Scott Hareland, Kevin Zhang, and Patrick Armstrong. Characterization of Multi-Bit Soft Error Events in Advanced SRAMs. In *IEEE International Electron Devices Meeting*, pages 21.4.1–21.4.4, 2003

[57] Timothy J. Dell. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory. IBM Whitepaper, 1997

[58] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 111–122, London, England, UK, 2012. ACM

SUMMARY: Hardware-level solutions exist that reduce the failure probability for the hardware error scenarios described in Section 2.1. These solutions include replication of computing components, introduction of specialized validation circuits, as well as using signatures to detect errors in data storage and buses.

The major drawback of hardware-level fault tolerance is cost. New circuitry increases chip size, which directly correlates with production cost. Furthermore, additional circuits require power at runtime and thereby increase a chip's power consumption.

2.4.2 Software-Implemented Fault Tolerance

While hardware-implemented fault tolerance is effective, platform vendors abstain from using these methods especially in COTS systems. Customers are interested in the cheapest possible gadgets and may be willing to live with occasional failures due to hardware errors.

COTS components are still vulnerable against increasing hardware fault rates, though. For this reasons it is becoming necessary to add special protection to mass-market computers as well. Software-implemented fault-tolerance mechanisms try to achieve this protection without relying on any non-COTS hardware components. These methods include compiler techniques to generate more reliable code. Alternative approaches use replication at the software level and utilize operating system processes or virtual machines to enforce fault isolation.

Manually Implemented Fault Tolerance Early research into fault tolerance investigated whether programs can be manually extended with mechanisms

that detect and correct errors. The field of *algorithm-based fault tolerance* (ABFT) focuses on finding algorithms that are able to validate the correctness of their results.[59] For ABFT, a developer inspects the data domain of a specific algorithm and augments data with checksums and other signatures, similar to encoding-based hardware techniques. Thereafter, the algorithm is redesigned to incorporate signature updates and validation. This approach is labor-intensive, but results in low-overhead solutions with high error coverage. Nevertheless, recent research in high performance computing found that other fault-tolerant techniques are less suited for future exascale compute clusters. This has lead to renewed interest in ABFT within the HPC community.[60]

Software engineering aims to formalize the development of fault-tolerant software. *Executable assertions* allow the developer to specify assumptions about the state of data structures in program code.[61] These assumptions are then checked at runtime. As assertions manifest programmer knowledge in code, they also lend themselves to detect state that is corrupted due to a hardware fault.

If a program's state validation mechanisms detect an error, the developer needs to specify how to deal with this situation. The program might choose to retry computation, use an alternative implementation of the algorithm, or simply terminate. These respective actions can be implemented using the concept of *recovery blocks*.[62] Since their inception, both assertions and recovery blocks have found their way into every software developer's toolbox.

Development Tools for Fault Tolerance Manually implementing fault-tolerant software is a labor-intensive and error-prone task. Development tools relieve programmers of this burden. Therefore, compiler writers aim to automate the process of generating fault-tolerant code. Oh proposed a compiler extension that extends a program's control flow with compiler-generated signatures.[63] These signatures are updated on every jump operation. Runtime checks can then verify that the current instruction was reached by taking a valid path. This approach detects invalid jumps that were caused by errors affecting the jump target or during instruction decoding.

Another compiler extension by Oh generates code that doubly performs all computations using different CPU resources and validates their results.[64] Such augmented code detects transient hardware faults that impact computational components. Intuitively, doubling the amount of computations would at least double the execution time of the generated code. Oh's work shows that this overhead can be lowered by relying on a state-of-the-art pipelined processor architecture.

Reis and colleagues built on Oh's idea of duplicating instructions, but optimized their compiler to further reduce execution time overhead. Their main observation was that memory is often already protected by hardware ECC and therefore duplication of memory-related instructions is no longer necessary. With this optimization their SWIFT compiler achieved the same error coverage as Oh's compiler, but at less than 50% execution time overhead.[65]

SWIFT leaves some parts of the protected code vulnerable to errors. By excluding memory operations from duplication, stores to memory may be lost. These lost updates then remain undetected. Furthermore, SWIFT's

[59] Kuang-Hua Huang and Jacob A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, C-33(6):518–528, 1984

[60] Dong Li, Zizhong Chen, Panruo Wu, and Jeffrey S. Vetter. Rethinking Algorithm-Based Fault Tolerance with a Cooperative Software-Hardware Approach. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'13, pages 44:1–44:12, Denver, Colorado, 2013. ACM

[61] Aamer Mahmood, Dorothy M. Andrews, and Edward J. McClusky. *Executable Assertions and Flight Software*. Center for Reliable Computing, Computer Systems Laboratory, Dept. of Electrical Engineering and Computer Science, Stanford University, 1984

[62] Andrew M. Tyrrell. Recovery Blocks and Algorithm-Based Fault Tolerance. In *EUROMICRO 96. Beyond 2000: Hardware and Software Design Strategies*, pages 292–299, 1996

[63] Namsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control-Flow Checking by Software Signatures. *IEEE Transactions on Reliability*, 51(1):111 –122, March 2002

[64] Namsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error Detection by Duplicated Instructions in Super-Scalar Processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002

[65] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization*, CGO '05, pages 243–254, 2005

internal validation code runs itself unprotected and is therefore susceptible to errors. Schiffel's AN-encoding compiler addresses these shortcomings by applying arithmetic encoding to all operands and operations.[66] In a closer analysis Schiffel and colleagues showed that their improvements come with an increased execution time overhead, but at the same time eliminate most of the vulnerabilities introduced by SWIFT.[67]

[66] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *International Conference on Computer Safety, Reliability and Security*, Safecomp'10, Vienna, Austria, 2010

[67] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. Software-Implemented Hardware Error Detection: Costs and Gains. In *Third International Conference on Dependability*, DEPEND'10, pages 51–57, 2010

The previously discussed compiler extensions assume all instructions to be affected by hardware errors with the same probability. Rehman and colleagues observed that different classes of instructions remain in the processor for a different amount of cycles. While a simple arithmetic instruction involving registers may be finished within a single cycle, memory operations that include a fetch may remain in the pipeline for several cycles. They therefore analyzed the instruction set and implementation of a specific SPARC v8 processor and attributed each instruction with an *instruction vulnerability index (IVI)*.[68]

The IVI represents how likely an instruction is to suffer from a hardware error relative to other instructions. Based on the IVI the authors then designed a compiler that prioritizes the protection of instructions with higher IVIs over those with lower IVIs. The compiler can then be configured to keep the overhead for generated code below a certain threshold. Developers may thereby explicitly trade error coverage for performance depending on their system's needs.

[68] Semeen Rehman, Muhammad Shafique, Florian Kriebel, and Jörg Henkel. Reliable Software for Unreliable Hardware: Embedded Code Generation Aiming at Reliability. In *International Conference on Hardware/-Software Codesign and System Synthesis*, CODES+ISSS '11, pages 237–246, Taipei, Taiwan, 2011. ACM

Borchert used an aspect-oriented compiler to protect data structures of an operating system kernel implemented in C++ from memory errors.[69] His aspects leverage additional knowledge about important data structures provided by the kernel developer. These data structures are automatically extended with checksumming and validation upon every access.

[69] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generative Software-Based Memory Error Detection and Correction for Operating System Data Structures. In *International Conference on Dependable Systems and Networks*, DSN'13. IEEE Computer Society Press, June 2013

Compiler-level fault tolerance requires all applications to be recompiled using a new compiler version. Binary-only third-party libraries as well as programs downloaded without their source code cannot be protected. Most compilers however allow interaction between protected and unprotected code. For this purpose they generate code that translates between encoded and non-encoded values as well as between duplicated and single-instance variables and function parameters.

Software-Implemented Replication Replication-based software fault tolerance works for binary-only software and thereby addresses the shortcomings of the compiler-level solutions discussed above. Similar to replication at the hardware level, software-level replication creates multiple instances of software components and compares their outputs to detect erroneous behavior.

Software-implemented approaches differ in their spheres of replication. At a large scale, replication is used to achieve fault tolerance in distributed systems. Here, whole compute nodes are replicated with dedicated hardware resources, operating system, and software stack. Kapritsos and colleagues showed with the EVE system that this kind of replication can achieve fault tolerance at low overhead, because replication can incorporate application-level knowledge and batch operations for more efficient processing.[70] My thesis

[70] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. EVE: Execute-Verify Replication for Multi-Core Servers. In *Symposium on Opearting Systems Design & Implementation*, OSDI'12, Oct 2012

explicitly focuses on hardware faults in single compute nodes. Therefore I am not going to further look into distributed systems fault tolerance from here on.

On a single compute node, Bressoud and Schneider showed that virtual machines (VMs) can be used to replicate whole operating system instances.[71] These instances are completely isolated using virtualized hardware. The underlying hypervisor makes sure that replicas receive the same inputs in terms of interrupts and I/O operations. VM states are compared after a predefined interval of VM instructions to detect errors. Unfortunately, implementing virtualization-based replication is fairly complex. Modern hypervisors, such as KVM[72], require tens of thousands of lines of code not yet including virtualized device models. Furthermore, Bressoud's work reports a significant execution time overhead of at least a factor of two.

Shye's *process-level redundancy (PLR)* moves replication to the operating system level and replicates Linux processes.[73] When launching a program, PLR instantiates multiple replicas as well as a replica manager. PLR uses a binary recompiler to rewrite the program in a way that all system calls are redirected to the manager process for error detection. Using this approach, PLR does not require source code availability. Furthermore, by distributing replicas across concurrent compute nodes, PLR achieves low execution time overheads of less than 50% on average for the SPEC CPU 2000 benchmarks in triple-modular redundant mode.

PLR's architecture motivates the ROMAIN replication service I present in this thesis. As an improvement over PLR, I implement ROMAIN as an operating system service that reuses existing OS infrastructure instead of relying on a complex binary recompiler. This approach significantly reduces the code complexity of the replication mechanism, because binary recompilers such as Valgrind[74] comprise more than 100,000 lines of C code.

Furthermore, ROMAIN supports replication of multithreaded applications. A recent work by Mushtaq shares these improvements.[75] In contrast to ROMAIN, their work however assumes ECC-protected memory and differs in recovery overhead. I will discuss these differences more thoroughly when I discuss multithreaded replication in Chapter 4.

Zhang's DAFT compiler combines compiler-assisted fault tolerance with redundant multithreading.[76] Instead of encoding data differently, DAFT uses multiple software threads for redundant execution. The compiler then only inserts additional code to exchange and compare compute results between replicas. DAFT furthermore executes code speculatively within its sphere of replication and thereby achieves a execution time overhead of less than 40%.

In the context of high-performance computing, Fiala and colleagues noted that HPC applications traditionally use checkpoint/restart mechanisms to tolerate failing nodes. They modeled the cost for such strategies in future exa-scale systems. Based on these models they claim if current hardware failure trends remain constant, the overhead involved in checkpointing will require more than 80% of the total compute time in 10,000 node systems. This observation makes replication by running replicated processes concurrently on the huge amount of available CPUs a feasible alternative. Fiala therefore

[71] Thomas C. Bressoud and Fred B. Schneider. Hypervisor-Based Fault Tolerance. *ACM Transactions on Computing Systems*, 14:80–107, February 1996

[72] Avi Kivity. KVM: The Linux Virtual Machine Monitor. In *The Ottawa Linux Symposium*, pages 225–230, July 2007

[73] 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, 2009

[74] Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM

[75] Hamid Mushtaq, Zaid Al-Ars, and Koen L. M. Bertels. Efficient Software Based Fault Tolerance Approach on Multicore Platforms. In *Design, Automation & Test in Europe Conference*, Grenoble, France, March 2013

[76] Yun Zhang, Jae W. Lee, Nick P. Johnson, and David I. August. DAFT: Decoupled Acyclic Fault Tolerance. In *International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 87–98, Vienna, Austria, 2010. ACM

proposed a replication extension for the Message Passing Interface (MPI) that replicates MPI processes as well as the communication among those programs.[77]

[77] David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Salt Lake City, Utah, 2012. IEEE Computer Society Press

SUMMARY: Software-implemented fault tolerance solutions can be categorized into two classes: compiler-assisted fault tolerance and replication-based approaches. Compiler-level solutions generate machine code that includes additional checksums and validation of results. They require the protected program's source code to be available.

In contrast, replication-based approaches protect software by running it in multiple instances and comparing these instances' results. Replication can be applied at various levels ranging from virtual machines through operating system processes down to software threads. While these solutions support binary-only software, they often come with a high resource overhead.

2.4.3 Recovery: What to do When Things go Wrong?

Detecting an error before it becomes a failure is only one half of what a fault tolerant system needs to do. In order to provide availability by the definition introduced in Section 2.2.2, the system also needs to react by correcting the error and delivering a correct result. This process is called *recovery*. Many implementations of recovery mechanisms exist. Randell categorizes them into two classes: backward and forward recovery.[78]

[78] Brian Randell, Peter A. Lee, and Philip C. Treleaven. Reliability Issues in Computing System Design. *ACM Computing Surveys*, 10(2):123–165, June 1978

Backward (or Rollback) Recovery comprises solutions that — upon detecting an error — return the system into a previous state that is assumed to be error-free. The most intuitive version of such a mechanism is to simply terminate the erroneous software component and restart it.[79] Unfortunately, with this approach all non-persistent application state is lost. To address this issue, checkpoint/rollback systems periodically create copies of important application data. Upon a restart, the most recently stored version can be recovered.[80] Still, all results computed after the last checkpoint are lost and need to be recomputed. This leads to an increased time to repair.

[79] Shubhendu Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008

[80] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009

In contrast to rollback, *forward recovery* tries to repair the erroneous component so that it can continue without the need for re-execution. Replication-based mechanisms implement forward recovery using majority voting. If state comparison detects a mismatch between replicas, the majority of replicas is assumed to be correct and the mismatching replicas are overwritten. This concept has been formalized by Schneider as *t-fault tolerance:* a t-fault-tolerant system can handle t erroneous components. In order to perform successful recovery, $2t + 1$ replicas are required.[81] As a consequence, replication requires a lower time to repair, but in turn may require a larger amount of resources and energy during normal operation.

[81] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990

While replication is able to recover from transient errors, it will not fix permanent ones, because a replica encountering a permanent error will simply suffer from it again. A forward recovery method to deal with permanently broken hardware is to adapt the running system. This may include turning

off broken CPU cores and running the respective replicas elsewhere. Alternatively, malfunctioning hardware may be worked around by using a software implementation that does not leverage the broken hardware units. As an example for that, Meixner developed Detouring, a compiler solution to deal with permanent errors in floating point hardware. Upon detecting a broken FPU, Detouring switches to program paths that use software-level floating point computations, which are slower but work solely using integer arithmetic.[82]

A different line of research argues that future hardware and software will suffer from so many errors that developers should rather try to live with them instead of trying to build correct systems. Palem suggested to use probabilistic hardware that generates results that are correct within certain thresholds. He showed that the resulting hardware may be smaller and less energy-demanding than standard processors.[83] Unfortunately, with such an approach all software needs to be redesigned to cope with hardware-level uncertainty. It remains to be shown whether this prerequisite is easier to meet than building traditional fault-tolerant systems.

SUMMARY: Error recovery mechanisms can be distinguished into backward and forward recovery. In general, these mechanisms are orthogonal to error detection. Therefore, most of the previously discussed error detection mechanisms can be combined with any suitable recovery technique.

2.4.4 Fault-Tolerant Operating Systems

Research in operating system fault tolerance is mostly concerned with software errors. Researchers and developers argue that software bugs, especially in device drivers and other hardware-related code, are the main reason for failures in today's systems.[84] In this section I have a closer look at how operating systems deal with software errors. I argue that main design principles – such as the use micro-rebootable components – may be employed for tolerating hardware faults as well.

Building Reliable Operating Systems Operating system crashes, such as the infamous Windows blue screens or Linux' kernel panics are a major annoyance for computer users. When the system reaches panic mode, it has already failed and a reboot is the only way of returning into a working state. Reboots may take several minutes and therefore have a major impact on a system's availability ratio.[85]

To improve recovery times after software crashes, Candea proposed to design systems to be *micro-rebootable*.[86] Such systems are built from many small, isolated components that do not share state. If one of these components fails, it is enough to restart this single component instead of rebooting the whole machine. Hence, service downtime is reduced drastically.

Componentization comes with another advantage: in traditional — monolithic — operating systems, device drivers are a main source of software failures.[87] Swift's Nooks system demonstrated that these drivers can be isolated into separate Linux address spaces. This approach protects unrelated

[82] Albert Meixner and Daniel J. Sorin. Detouring: Translating Software to Circumvent Hard Faults in Simple Cores. In *International Conference on Dependable Systems and Networks (DSN)*, pages 80–89, 2008

[83] Krishna V. Palem, Lakshmi N.B. Chakrapani, Zvi M. Kedem, Avinash Lingamneni, and Kirthi Krishna Muntimadugu. Sustaining Moore's Law in Embedded Computing Through Probabilistic and Approximate Design: Retrospects and Prospects. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 1–10, Grenoble, France, 2009. ACM

[84] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 305–318, Newport Beach, California, USA, 2011. ACM

[85] Alex Depoutovitch and Michael Stumm. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *European Conference on Computer Systems*, EuroSys '10, pages 181–194, Paris, France, 2010. ACM

[86] George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot: A Technique For Cheap Recovery. In *Symposium on Operating Systems Design & Implementation*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association

[87] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering Device Drivers. *ACM Transactions on Computing Systems*, 24(4):333–360, November 2006

kernel data from being overwritten by a faulty device driver and thereby increases system reliability.[87]

Operating systems focusing on the isolation of components are nothing new. These design principles have been advocated by the microkernel community since the 1980s.[88] Microkernels move traditional kernel services, such as file systems, network stacks and device drivers, out of privileged kernel mode into isolated user-level applications. Microkernel proponents argue that this improved isolation leads to an increase in scalability, portability, and security. While early microkernels were dismissed for their performance overheads, later kernel generations showed that the improved isolation properties can be gained at a low cost.[89]

Minix3 is a microkernel-based operating system that was specifically designed to tolerate software failures.[90] When a Minix3 process crashes or stops sending heartbeat messages, a system-wide manager terminates and restarts the respective program. Thereafter, all other applications are notified of this restart, so that they can adapt to this situation, for instance by resending outstanding service requests. This approach works best for stateless processes, such as device drivers. Based on Minix3, Giuffrida investigated how stateful services can be restructured to fit into this paradigm. He showed that applications can be written in a transactional manner where either all state is written to a central state storage upon commit or an operation can be rolled back if the application crashes while processing it.[91]

Formally Verified Systems Code Programming errors often lead to software failures. In the worst case, attackers can exploit these bugs to attack the system. Ryzhyk analyzed failing systems code and found that for device drivers, these errors are often not syntactic or algorithmic errors but instead stem from subtle misunderstandings regarding the hardware or operating system interface.[92] Based on this observation he proposed to formalize software development by creating well-defined models of the underlying software and hardware components and then have a compiler automatically generate code that adheres to these models.[93]

The idea to generate code from formal models to improve security is also found in safety-critical systems, such as the Partitioned Operating System Kernel (POK).[94] However, creating the respective models requires a non-negligible manual effort. The involved cost is therefore only spent for building critical systems, whereas standard consumer electronics rather live with occasional crashes to reduce development cost.

In contrast to monolithic operating systems that consist of hundreds of thousands of lines of code, microkernels have a relatively small code size.[95] It is therefore possible to model such a kernel and formally verify this model adheres to well-defined properties. This approach is prohibitive for large systems software because building these formal models requires a huge manual effort. Klein et al. were able to formally prove the correctness of the seL4 microkernel.[96]

[88] Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Technical Conference*, pages 93–112, 1986

[89] Kevin Elphinstone and Gernot Heiser. From L3 to seL4: What Have We Learnt in 20 Years of L4 Microkernels? In *Symposium on Operating Systems Principles*, SOSP'13, pages 133–150, Farminton, Pennsylvania, 2013. ACM

[90] Jorrit N. Herder. *Building a Dependable Operating System: Fault Tolerance in MINIX3*. Dissertation, Vrije Universiteit Amsterdam, 2010

[91] Cristiano Giuffrida, Lorenzo Cavallaro, and Andrew S. Tanenbaum. We Crashed, Now What? In *Workshop on Hot Topics in System Dependability*, HotDep'10, Vancouver, BC, Canada, 2010. USENIX Association

[92] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming Device Drivers. In *ACM European Conference on Computer Systems*, EuroSys '09, pages 275–288, Nuremberg, Germany, 2009. ACM

[93] Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic Device Driver Synthesis with Termite. In *Symposium on Operating Systems Principles*, SOSP '09, pages 73–86, Big Sky, Montana, USA, 2009. ACM

[94] Julian Delange and Laurent Lec. POK, an ARINC653-compliant operating system released under the BSD license. In *Realtime Linux Workshop*, RTLWS'11, 2011

[95] Kevin Elphinstone and Gernot Heiser. From L3 to seL4: What Have We Learnt in 20 Years of L4 Microkernels? In *Symposium on Operating Systems Principles*, SOSP'13, pages 133–150, Farminton, Pennsylvania, 2013. ACM

[96] Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Symposium on Operating Systems Principles*, SOSP'09, pages 207–220, Big Sky, MT, USA, October 2009. ACM

While formal verification of practical code is still in an early phase, efforts are underway to not only prove the correctness of a microkernel, but to also include other operating system components, such as file systems.[97] It is therefore safe to assume that future systems software is going to contain much fewer programming errors than today. However, all these formal efforts still assume that the hardware their software runs on is behaving correctly. As I have shown in the previous sections, this is not necessarily the case. We therefore need additional hardware or software layers that allow to deal with faulty hardware.

[97] Gabriele Keller, Toby Murray, Sidney Amani, Liam O'Connor, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. File Systems Deserve Verification Too! In *Workshop on Programming Languages and Operating Systems*, PLOS '13, pages 1:1–1:7, Farmington, Pennsylvania, 2013. ACM

SUMMARY: Operating system research has largely focused on hardening the kernel against software errors. A commonly found concept to tolerate software failures is to isolate components into separate address spaces, so that a malfunctioning component can only harm itself.

Formal modeling and verification of systems components strives to drastically reduce programming errors and the resulting failures. All these efforts assume hardware to work correctly. In order to really protect a system, we need additional measures to tolerate hardware faults.

2.5 Thesis Goals and Design Decisions

In this thesis I develop a fault tolerant operating system architecture. I will call this architecture ASTEROID from now on. ASTEROID aims to protect software from transient and permanent faults arising at the hardware level. Based on the observations presented in the previous sections I derive the following design goals, which I aim to accomplish:

1. **Support for COTS Hardware:** Building fault-tolerant hardware components incurs additional design cost, chip area, as well as execution time overhead. ASTEROID strives to avoid these costs by solely relying on features that are available in commercial-off-the-shelf hardware, so that its results are applicable to other existing COTS platforms.

   For some hardware mechanisms, such as ECC-protected memory, it is hard to say whether they are already a commodity or still considered specialized. In such situations I will conservatively assume these features to be unavailable.

2. **Exploit Hardware-Level Concurrency:** Modern hardware platforms usually contain multiple CPU cores in combination with abundant memory and complex cache hierarchies. It is likely that the number of cores will continue to grow, although not all cores might be powered at the same time anymore.[98]

   These practical realities make the use of replication-based fault tolerance feasible and ASTEROID therefore uses replication as the foundation for detecting and correcting errors. Additionally, I will in this context also have a look at the interaction between caches, memory, and CPU cores

[98] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Annual International Symposium on Computer Architecture*, ISCA'11, pages 365–376, San Jose, California, USA, 2011. ACM

in modern platforms in order to make informed decisions about resource allocation and replica placement.

3. **Efficient Error Detection and Correction:** Hardware faults are still rare enough that a system will run error-free for most of its lifetime. Error detection mechanisms should therefore aim to decrease execution time overhead. Correction needs to be fast in order to reduce its impact on system availability.

   ASTEROID replicates software on distinct physical CPU cores, allowing replicas to run independently as long as possible. I implement a replication service leveraging the redundant multithreading concept in order to minimize execution time overhead.

   Recovery mechanisms are orthogonal to error detection techniques. ASTEROID supports various ways of recovery including application restart, checkpoint/rollback, as well as majority voting. Restart and checkpointing have already been considered in previous work.[99] I will focus on majority voting as a recovery technique in this thesis.

4. **Use a Componentized Operating System:** Previous work in OS-level fault tolerance has shown that isolating OS components into separate processes is useful with respect to tolerating software faults. I believe that this assumption holds with respect to hardware errors and will therefore base ASTEROID on a microkernel. I use the FIASCO.OC microkernel[100], because this kernel is freely available and has a proven track record of working in different scenarios, such as real-time, security, and virtualization.

5. **Binary Application Support:** Modern software is often distributed in its binary form and the source code is unavailable to the general public. ASTEROID protects any existing program without relying on parsing or modifying their source code. I do not leverage compiler-based fault tolerance mechanisms in this thesis. However, I will discuss situations in which compiler support may be useful and evaluate what impact this would have on ASTEROID in general.

6. **Support Multithreaded Programs:** With the abundance of CPU cores in modern hardware, software developers speed up program execution times by parallelizing compute operations. The OS kernel typically implements threads and schedules them on the available CPU cores.

   Scheduling multi-threaded applications introduces non-determinism into the system. Determinism is however a prerequisite for replication. I will present a solution to replicate multithreaded applications.

7. **Protect the Reliable Computing Base:** Software-level fault tolerance mechanisms suffice to protect execution of user-level programs as well as high-level operating system services against hardware errors. However, most of these mechanisms implicitly rely on the fact that at least parts of the underlying hardware and software stack always function correctly. I call these components the *Reliable Computing Base (RCB)*.

   I will investigate ASTEROID's RCB in this thesis and propose ways of hardening this part of the system against hardware faults.

[99] Dirk Vogt, Björn Döbel, and Adam Lackorzynski. Stay Strong, Stay Safe: Enhancing Reliability of a Secure Operating System. In *Workshop on Isolation and Integration for Dependable Systems*, IIDS'10, Paris, France, 2010. ACM

[100] Adam Lackorzynski and Alexander Warg. Taming Subsystems: Capabilities as Universal Resource Access Control in L4. In *Workshop on Isolation and Integration in Embedded Systems*, IIES'09, pages 25–30, Nuremburg, Germany, 2009. ACM

SUMMARY: In this chapter I reviewed hardware effects that may lead to erroneous behavior and surveyed existing research that tries to mitigate these problems. From this survey I derived requirements and design goals for ASTEROID, the fault-tolerant operating system architecture I present in this thesis.

ASTEROID aims to leverage replicated execution to provide error detection and correction to binary-only programs. The architecture relies on modern multi-core COTS hardware and aims to protect all parts of the software stack against the effects of hardware errors.

In the remainder of this thesis I present how ASTEROID achieves the above goals. In Chapter 3, I give an overview of ASTEROID and introduce ROMAIN, an operating system service for replicated execution on top of FIASCO.OC. Thereafter, I describe how to extend ROMAIN to replicate multithreaded applications in Chapter 4 and evaluate ASTEROID's overhead and error detection capabilities in Chapter 5. Finally, I investigate ASTEROID's Reliable Computing Base in Chapter 6.

3
Redundant Multithreading as an Operating System Service

In the previous chapter I explained that hardware faults do occur in today's systems and that their rate is likely to increase in future hardware generations. The goal of my thesis is to develop an operating system architecture that detects errors by replicating applications and that uses majority voting to recover from errors.

The main focus of this chapter is ROMAIN, an operating-system service implementing redundant multithreading for applications running on top of FIASCO.OC. I describe how ROMAIN interposes itself between application replicas and the operating system kernel. I present mechanisms to manage replicas' resources. Furthermore, I describe how ROMAIN provides error detection and recovery based on these mechanisms. The ideas and decisions described in this chapter were originally published in EMSOFT 2012[1] and SOBRES 2013.[2]

[1] Björn Döbel, Hermann Härtig, and Michael Engel. Operating System Support for Redundant Multithreading. In *12th International Conference on Embedded Software*, EMSOFT'12, Tampere, Finland, 2012

[2] Björn Döbel and Hermann Härtig. Where Have all the Cycles Gone? – Investigating Runtime Overheads of OS-Assisted Replication. In *Workshop on Software-Based Methods for Robust Embedded Systems*, SOBRES'13, Koblenz, Germany, 2013

3.1 Architectural Overview

In order to protect platforms based on commercial-off-the-shelf (COTS) hardware components, we need to apply software-level fault tolerance techniques. A large fraction of today's software is only available in binary form and vendors do not provide access to the source code. It is therefore infeasible to protect the whole software stack solely using existing compiler-level fault tolerance methods.

The operating system lies at the boundary between hardware and software and is in control of all programs. Placing a fault tolerance mechanism at the OS level therefore protects the largest possible set of applications. However, fault tolerance is potentially expensive in terms of execution overhead and resource requirements. The OS should therefore not enforce a mechanism on all applications but allow the system designer to selectively protect programs. This approach has two advantages over full-system replication:

1. Applications that have been implemented using fault tolerant algorithms or programs that were compiled using a fault-tolerant compiler take care of fault tolerance themselves. These applications do not require additional support from the OS.

2. In resource-constrained environments, a user may be willing to accept a failure in an unimportant application while still wanting to protect another, more important program.

For the above reasons, I structured the ASTEROID fault-tolerant operating system design as shown in Figure 3.1. ASTEROID uses software-level replication to detect and correct errors that manifest as the effects of hardware faults. It runs on COTS hardware components and protects binary-only applications. This approach makes fault tolerance transparent to user-level applications and developers do not have to take specific precautions to counter hardware errors.

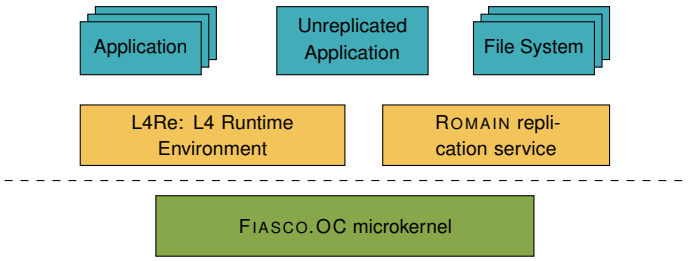

Figure 3.1: ASTEROID System Architecture: The ROMAIN replication service extends the L4 Runtime Environment. Replication on a per-process basis allows to integrate both replicated and unreplicated applications into the system. Using a microkernel architecture, replication also covers traditional OS services, such as file systems.

ASTEROID replicates on a per-application basis using an OS service named ROMAIN. This design decision allows users to selectively turn on replication for applications that require this service. ASTEROID's sphere of replication is a process. As long as a process only performs internal computation, there is no interaction with the replication service or the kernel. Only when application state becomes visible to outside observers, replicas are compared for state deviations that indicate an error. This redundant multithreading approach reduces the execution time overhead imposed by the replication service. This benefit does not come for free, because replication increases memory and CPU usage compared to native execution.

In the previous chapter we saw that splitting software into small, isolated components improves system reliability. Hence, a microkernel is a natural choice for building a fault tolerant operating system. ASTEROID is based on the FIASCO.OC microkernel developed at TU Dresden. In addition to its isolation properties, a microkernel foundation offers another advantage: as traditional operating system services – such as device drivers,[3] file systems,[4] and networking stacks[5] – run in user space, ROMAIN can replicate them and thereby provide protection against hardware faults.[6]

ASTEROID reuses L4Re, FIASCO.OC's existing user-level runtime environment.[7] In the remainder of this chapter I focus on my extension to this system, the ROMAIN replication service. I describe how ROMAIN adds replication capabilities to L4Re and how it detects and recovers from the effects of hardware errors.

[3] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Symposium on Operating Systems Design and Implementation*, SOSP'04, San Francisco, CA, December 2004

[4] Carsten Weinhold and Hermann Härtig. jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components. In *USENIX Annual Technical Conference*, ATC'11, pages 32–32, Portland, OR, 2011. USENIX Association

[5] Tomas Hruby, Dirk Vogt, Herbert Bos, and Andrew S. Tanenbaum. Keep Net Working - On a Dependable and Fast Networking Stack. In *Conference on Dependable Systems and Networks*, Boston, MA, June 2012

[6] We will see in Chapter 7 that replication of device drivers is still an open issue.

[7] `http://l4re.org`

3.2 Process Replication

ROMAIN–the **Ro**bust **M**ultithreaded **A**pplication **In**frastructure–is an extension to L4Re that allows replicated execution of single processes. It provides replication as a form of redundant multithreading as described in Section 2.4.1 on page 26 and leverages software threads provided by the FIASCO.OC kernel. Figure 3.2 shows ROMAIN's architecture for a triple-modular redundant setup.

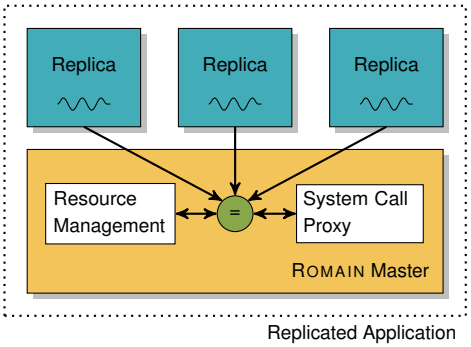

Figure 3.2: ROMAIN Architecture

Replicas execute a single instance of a protected application. They run in separate address spaces to achieve fault isolation and prevent a failing replica from overwriting the correct state of other replicas. Users can configure the number of replicas at application startup time and thereby create arbitrary n-modular redundant setups. For every replicated application, a *master* process is responsible for managing replicas, validating their states, and performing error recovery.

The ROMAIN master process has three tasks:

1. *Binary Loading:* During application startup, the master acts as a program loader. According to the configured number of replicas, the master creates address spaces and loads the respective binary code and data segments from the executable file. Thereafter, the master creates a new thread for every replica and makes these threads start executing program code within their respective address spaces.
2. *Resource Management:* Redundant multithreading executes replicas independently while they only modify their internal state. In terms of ROMAIN a replica's internal state comprises:

   - Replica-owned memory regions,
   - Each replica's view on FIASCO.OC kernel objects, and
   - Each thread's CPU register state.

   The master process maintains *full control over the above resources* for every replica. Thereby, the master ensures that the replicas always receive identical inputs. The replicas then execute identical code and will produce identical outputs as long as they are not affected by a hardware fault.

   We will see in Chapter 4 that we need to take additional precautions to handle multithreaded replicas, because these programs may suffer from scheduling-related non-determinism, which the master process needs to cope with as well. *In the scope of this chapter I will however assume single-threaded replicas.*

3. *Error Detection and Correction:* The master process *detects and corrects errors* by monitoring replica outputs. In the context of ROMAIN a replica output is any event that makes application state visible to the outside world. These events include system calls, CPU exceptions (such as page faults), as well as writes to memory regions that are shared with other applications. I will refer to these events as *externalization events* in the remainder of this thesis.

3.3 Tracking Externalization Events

To remain in control over the replicas' states, the master process needs a way to intercept externalization events. Shye's Process Level Redundancy (PLR), which I described in Section 2.4.2 on page 32, applies binary recompilation for this purpose and rewrites the replicated program so that all system calls are reflected to the PLR replication manager.

Why not use Binary Rewriting? Binary rewriting is a complex task[8] and applying it to general-purpose programs needs to solve two problems: First, we need to identify all binary instructions in order to instrument them. This is difficult for instruction set architectures with variable-length instructions, such as Intel x86. The rewriter here needs to disassemble all instructions step by step. Furthermore, dynamic branches due to function pointers and register-indirect addressing mean that not all instructions can be rewritten statically, because runtime information is needed to identify the targets of dynamic branches. The rewriting process therefore needs to be carried out incrementally.

The second issue with binary rewriting is how to instrument code. Naively, we might assume that instrumentation would simply replace the instrumented instruction with a `call` to an external instrumentation handler. Unfortunately, given x86' variable instruction lengths this does not work: instructions may be as short as a single byte, but a `call` instruction requires 5 bytes to be overwritten.

Solving these problems correctly and efficiently is difficult[9] and out of scope for this thesis. Furthermore, Kedia's efficient binary translation work reports an execution time overhead of about 10% for application workloads.[8] This is the base overhead even a very efficient replication service would have to work with. I therefore decided to avoid binary rewriting and instead rely on existing FIASCO.OC infrastructure for intercepting externalization events.

Virtual CPUs ROMAIN tracks a replicated application's externalization events using a software exception mechanism implemented by the FIASCO.OC kernel. Whenever a monitored thread performs an activity that becomes visible to the kernel – such as issuing a system call or raising a page fault – FIASCO.OC notifies a user-level exception handler of this fact.

In previous versions of FIASCO.OC developers had to distinguish between different types of exceptions (page faults, system calls, protection faults) and register specific handler threads for these events.[10] In its most recent version, FIASCO.OC unifies all types of exception handling into a single mechanism called a *virtual CPU (vCPU)*. vCPUs constitute threads

[8] Piyus Kedia and Sorav Bansal. Fast Dynamic Binary Translation for the Kernel. In *Symposium on Operating Systems Principles*, SOSP '13, pages 101–115, Farminton, Pennsylvania, 2013. ACM

[9] Derek Bruening and Qin Zhao. Practical Memory Checking with Dr. Memory. In *Symposium on Code Generation and Optimization*, CGO '11, pages 213–223, 2011

[10] Adam Lackorzynski. L^{4}Linux Porting Optimizations. Diploma thesis, TU Dresden, 2004

that execute within the bounds of an address space and are subject to the kernel's scheduling, as well as resource and access limitations. Besides being easier to program against, the vCPU exception handling model requires fewer kernel resources and is slightly faster.[11]

vCPUs differ from normal threads in the way system calls and exceptions are handled by the kernel. I illustrate this handling in Figure 3.3. Instead of directly handling a CPU trap (1), the kernel delivers this trap as an exception message to a user-level handler process (2). This *exception handler* inspects the vCPU's register state and then reacts upon the exception by modifying this state or the vCPU's resource mappings (3). Thereafter the handler instructs the kernel (4) to resume execution of the vCPU (5).

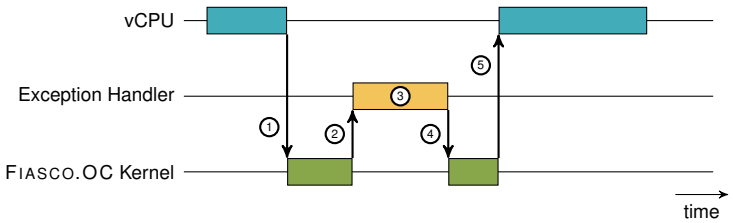

Figure 3.3: FIASCO.OC: Handling CPU Exceptions in User Space

Using the vCPU model, the external exception handler can intercept all CPU traps caused by a program. These traps include system calls, page faults, as well as all other traps defined by the x86 manual.[12] Hardware- and software-induced traps are the only way for an x86 program to break out of user-mode execution and communicate with other applications through a system call. ROMAIN uses vCPUs for executing replica code. The master process takes over the role of the external exception handler and thereby intercepts all such externalization events. We will see in Chapter 5 that this way of inspecting replicas has an execution time overhead of close to 0% for application benchmarks. It is therefore more efficient and less complex than applying binary rewriting.

Limitations of Using vCPUs There is one type of externalization event that the ROMAIN master cannot intercept using the vCPU model: reads and writes to memory regions shared with external applications. If a memory region is mapped to an application, any further accesses to it will not raise any externally visible trap that the kernel can intercept. I will describe my solution to this problem in Section 3.6 on page 57.

A second drawback of my decision to use vCPUs is that ROMAIN depends on FIASCO.OC. The implementation cannot easily be transferred to another operating system, such as Linux. This problem can be solved by retrofitting the target OS with a vCPU implementation. However, this would require a substantial effort. Alternatively, we can exploit mechanisms in the target OS that provide features similar to the vCPU model. Florian Pester showed that a ROMAIN implementation on Linux is possible[13] by leveraging Linux' kernel-level virtual machine (KVM) feature.[14]

[11] Adam Lackorzynski, Alexander Warg, and Michael Peter. Generic Virtualization with Virtual Processors. In *Proceedings of Twelfth Real-Time Linux Workshop*, Nairobi, Kenya, October 2010

[12] Intel Corp. Intel64 and IA-32 Architectures Software Developer's Manual. Technical Documentation at `http://www.intel.com`, 2013

[13] Florian Pester. ELK Herder: Replicating Linux Processes with Virtual Machines. Diploma thesis, TU Dresden, 2014

[14] Avi Kivity. KVM: The Linux Virtual Machine Monitor. In *The Ottawa Linux Symposium*, pages 225–230, July 2007

Replication using vCPUs To replicate a program, ROMAIN launches one vCPU for each replica. A replica vCPU then executes inside a dedicated address space as explained in the previous section. System calls and other CPU exceptions are handled as depicted in Figure 3.4. (I do not show involvement of the microkernel for better readability.)

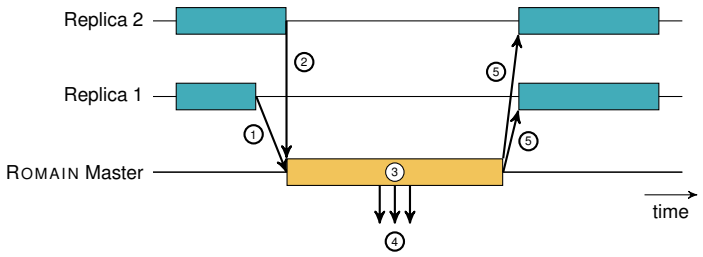

Figure 3.4: ROMAIN: Handling of externalization events

In line with the concept of redundant multithreading, replicas execute independently until their next externalization event occurs. At this point the kernel delivers information about the event along with the faulting vCPU's state to the ROMAIN master process. Externalization acts as a barrier blocking execution of all replicas that raise an event (1) until the last replica raises an event as well (2).

Once all replicas arrive at their next externalization event, the master begins processing it (3). The master first compares the replicas' states and detects and corrects potential errors. I will have a closer look at this stage in Section 3.8 on page 65. Once this phase is completed, the master is sure that all replicas agree on their state. The master then handles the actual event.

Depending on the event type, the master performs different actions. While most system calls are simply proxied to the kernel, resource management requests are handled by the master itself. I will discuss in detail how replica resources are managed in Sections 3.4–3.7. During event processing the master may perform one or more additional system calls and allocate additional resources and kernel objects (4). Once the event is handled, the master updates the replicas' states according to the event's outcome and directs the vCPUs to continue execution (5).

Replica Event Processing From a software engineering perspective, ROMAIN's event processing is implemented using the Observer design pattern.[15] Each event observer is capable of handling a specific event type. After validating replica states, the master's exception handler iterates over a list of these independent observer objects. Each observer inspects the current event and decides whether the event can be handled. If the observer handled the event, processing is stopped and the replicas resume execution. Otherwise the event is passed to the next observer in the list.

Table 3.1 lists ROMAIN's most important observer objects. Listing 3.5 on the next page gives an overview of the event processing steps described in this section as pseudo-code.

[15] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995

Observer	Purpose
Syscalls	Proxies or emulates system calls (see Section 3.4)
PageFault	Handles memory faults and shared memory access (see Sections 3.5 and 3.6)
Time	Handles timing information, such as `gettimeofday()` (see Section 3.7)
Debug	Allows to set breakpoints and debug replicas
SWIFI	Performs fault injection experiments
Lock-Observer	Implements deterministic lock acquisition for replicating multithreaded programs (see Chapter 4)

Table 3.1: ROMAIN event observers

SUMMARY: ROMAIN replicates binary applications running on top of the FIASCO.OC microkernel. To make replicas deterministic,

```
1  void Master::handle_event()
2  {
3     state_list = wait_for_all_replicas();
4
5     compare_states_and_recover(state_list); // see Section 3.8
6
7     // We know all states to be identical, so just
8     // pick the first one.
9     ReplicaState state = state_list.first();
10
11    for (Observer o : eventObservers)
12    {
13       // Event processing, see Sections 3.4 - 3.7
14       ret = o.process_event(state);
15
16       if (ret == Event::Handled) // processing successful
17       {
18          for (ReplicaState replica : state_list) {
19             // update from processed state
20             replica.update(state);
21             replica.resume();
22          }
23          return;
24       }
25    }
26
27    ERROR("Invalid event detected!");
28 }
```

Listing 3.5: ROMAIN processing of externalization events

a master process needs to remain in control of all input going into a replicated application. Outputs need to be intercepted to validate replica states before they become visible to external applications. These properties are achieved using FIASCO.OC's vCPU mechanism.

3.4 Handling Replica System Calls

Whenever programs access OS services, they interact with the underlying kernel using system calls. The kernel provides services through specific kernel objects. The actual implementation of these objects varies: Unix-like systems – such as Linux – provide kernel services through files, whereas Windows uses service handles. FIASCO.OC enables object access using an object-capability system.[16]

[16] Mark Miller, Ka-Ping Yee, Jonathan Shapiro, and Combex Inc. Capability Myths Demolished. Technical report, Johns Hopkins University, 2003

Referencing Objects Through Capabilities Before introducing system call replication, I explain FIASCO.OC's object model using Figure 3.6 as an example. The kernel manages objects (A, B, C) that represent execution abstractions (threads), address spaces (processes), and communication channels. These objects exist inside the kernel. The kernel implements access control for these objects using a per-process *capability table* that stores object references.

Programs invoke kernel functionality by issuing a system call to a kernel object. They denote the kernel object using an integer capability selector, which is interpreted by the kernel as an index into the process' capability table. In the example, Program 1 has access to objects A and C through capability

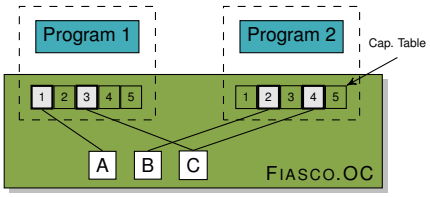

Figure 3.6: FIASCO.OC's object-capability mechanism in action

selectors 1 and 3. Program 2 has access to objects B and C through capability selectors 2 and 4.

Programs can furthermore create new kernel objects. During creation a reference to the new object will be added to the creator's capability table. The kernel does not track allocation of capability table entries – it is up to the application to decide where to place the new object reference.

System Call Types In order to perform a system call, an application needs to send parameters to the kernel. FIASCO.OC provides a per-thread memory region for this purpose, the *user-level thread control block (UTCB)*. A calling thread puts parameters into its UTCB and then invokes a specific kernel object. The kernel then interprets the UTCB's content and handles it depending on the type of invoked object.

In the context of ROMAIN, a system call constitutes an externalization event where data exits the application's sphere of replication. The master process intercepts these events and compares replica states. If they match, the event handler inspects system call parameters and distinguishes between kernel object management, messaging system calls, and resource mappings.

1. FIASCO.OC implements *kernel objects*, such as processes and threads. For the purpose of creating and managing these objects, the kernel provides specific system calls. The ROMAIN master needs to intercept these calls and replicate kernel-level objects in order to implement redundant multithreading. For instance, when a replicated application creates a new thread, the master needs to ensure that a separate instance of this thread is created inside every single replica.
2. *Messaging system calls* send a message through FIASCO.OC's Inter-Process Communication (IPC) mechanism. The kernel provides a specific object for this purpose, the IPC channel. Sending data through such a channel copies the message payload to a receiver. IPC system calls do not modify any kernel or application state. The master process therefore simply executes them using the replicas' system call parameters.
3. *Resource mappings* are an extension to the IPC mechanism. In addition to a data payload they contain resource descriptors, which are called *flexpages* in FIASCO.OC terminology. Flexpages are used to transfer access rights to kernel objects to or from the calling thread. Flexpage IPC therefore modifies the state of a replicated application and requires special handling by the master process.

On the following pages I first describe how ROMAIN handles data-only IPC messages. Thereafter I discuss the handling of object-capability mappings. I defer the discussion of memory management and related issues to Sections 3.5 and 3.6.

3.4.1 Proxying IPC Messages

Microkernel-based operating systems implement most OS functionality inside isolated user-level applications. Clients use these OS services through a kernel-provided IPC mechanism. IPC therefore constitutes the largest fraction of system calls in any microkernel-based system and is considered to be most crucial when designing such a kernel. [17]

[17] Jochen Liedtke. Improving IPC by Kernel Design. In *ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, Asheville, North Carolina, USA, 1993. ACM

FIASCO.OC provides IPC through the previously mentioned communication channel kernel object. A sender thread puts data into its UTCB and issues a system call. The kernel then copies data into the receiver thread's UTCB. Messages are limited by the UTCB payload size of 256 bytes. Aigner showed that this size is sufficient for most messaging use cases in microkernel-based systems.[18]

[18] Ronald Aigner. *Communication in Microkernel-Based Systems*. Dissertation, TU Dresden, 2011

When detecting a messaging system call, the ROMAIN master sends the respective IPC message once to the specified target thread. Conceptually, it does so by copying the message from a replica's UTCB into its own UTCB and then performing a system call to the same communication channel that was originally invoked by the replica. Replica threads are always vCPUs, which are blocked while the master is processing a system call. Therefore we can in practice avoid the UTCB-to-UTCB copy. Instead, the master reuses the trapping replica's UTCB when it proxies the IPC message.

Sending a replicated IPC message only once enables replicated and unreplicated applications to coexist on the same system as shown in Figure 3.7: Unreplicated programs will only receive messages once, regardless of the number of replicas in the sending application. They will furthermore always send their messages to the master process, which then multiplexes incoming messages to the actual replicas.

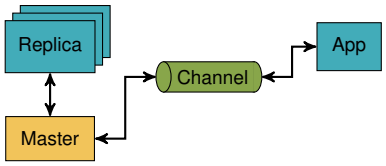

Figure 3.7: Integrating replicated and unreplicated applications

However, this design decision makes the transmission of a single IPC message vulnerable to hardware errors. The same is true for any other execution within the master process and other interactions with the kernel, because these mechanisms remain outside ROMAIN's sphere of replication and therefore form a *single point of failure* for the ASTEROID system. I will return to this problem and possible solutions in Chapter 6.

3.4.2 Managing Replica Capabilities

In order to correctly redirect IPC messages, the master needs to know which kernel objects the replicas were trying to invoke. For this purpose all the capabilities of these objects need to be mapped into the master's capability space. New kernel objects are always created through a system call and the master intercepts all of these calls. Through this mechanism the master can always add such object mappings to its own capability table.

As mentioned in Section 3.4, the layout of a capability table is managed by each user application itself. L4Re applications do so by keeping a bitmap of used and unused capability slots in memory. Modifications of this bitmap are plain memory accesses, will never lead to a system call, and can therefore not be inspected by the master process. When replicating an application this leads to the problem shown in Figure 3.8.

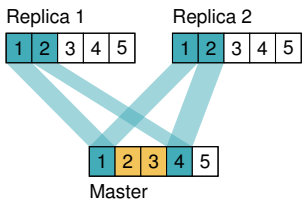

Figure 3.8: Replica capability selectors need to be translated into master capability selectors.

The figure shows the capability tables of two replicas and their master process after running for some time. As the replicas are deterministic, their capability tables are always laid out identically. Modifications to the capability table happen in the form of system calls so that any divergence would be detected by the master process. In this example, two objects are mapped into slots 1 and 2. The remaining slots are still unused. The master process also gets access to the replicas' capabilities by mapping these objects into free slots inside its own capability table. Additionally, the master may allocate objects

for private use, so that its capability table contains more object references than the replicas' tables. In the example, the master has allocated private objects into slots 2 and 3, whereas the replica-visible objects are mapped to slots 1 and 4.

A Capability Translation Mechanism If the master wants to relay IPC messages to the destination requested by the replicas, it has to translate the capability selector specified by the calling replica into a valid selector in the master's capability table. An intuitive solution to this problem would be to maintain a Selector Lookup Table (SLT). This SLT is then used in the following three cases:

1. If the replicas request to add a kernel object to their capability tables at slot R, find an empty slot M in the master's capability table. Store $R \rightarrow M$ in the SLT. Rewrite the replica system call parameters to obtain mapping into master capability slot M.[19] Then perform the system call.
2. If the replicas perform a system call using an existing capability selector R, look up the corresponding master capability selector M from the SLT. Rewrite the system call parameters to use M. Then perform the system call.
3. If the replicas remove a capability R using the `l4_fpage_unmap()` system call, look up the master capability selector M from the SLT. Rewrite the system call parameters to delete M. Perform the system call. Finally, remove $R \rightarrow M$ from the SLT.

[19] The master can rewrite replicas' system call parameters by modifying their UTCB and their register state. Both are available during vCPU exception handling.

Avoiding Capability Translation Using an SLT to translate capability selectors is feasible, but adds rewriting and lookup overhead to every replicated system call. To avoid this overhead, ROMAIN uses *partitioned capability tables* as shown in Figure 3.9. As explained in Section 3.2 on page 41, the master acts as the program loader for its replicas. During program loading the master also sets up the replicas' initial memory regions. The replicas' capability bitmap resides at a fixed location in one of these memory regions.

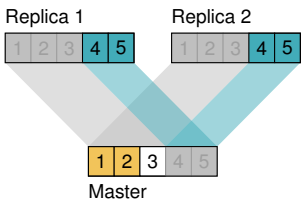

Figure 3.9: Partitioning capability tables allows the master to relay replica system calls without translation overhead.

To partition capability tables the master marks the first 16,384 entries in the replicas' capability bitmaps as reserved by setting their bits to 1 during application loading.[20] In turn, all but the first 16,384 entries in the master's capability bitmap are marked as used as well. Reserved regions are marked gray in Figure 3.9. As a result the replicas will always map kernel objects into capability slots $R \geq$16,384, whereas the master will allocate all its private objects within capability slots $M <$16,384.

Using this partitioning approach replica and master capability selectors will never overlap. Furthermore, all capability selectors used by replicas will have matching empty slots in the master's capability table. The master may therefore map copies of replicas' capabilities into the same slots in its own capability table, thereby creating a 1:1 mapping between replica and master capability selectors. For this reason the master neither has to perform any translation of capability selectors, nor does it need to rewrite replicas' system call parameters.

As a drawback, the partitioning approach reduces the number of available capability selectors for both the replicas and the master. However, this was not a problem in any of the experiments I conducted during this thesis.

[20] I chose the number 16,384 after some experimentation and it sufficed for all applications that I used throughout this thesis. The number can be adapted if necessary.

FIASCO.OC's possible capability space is much larger than the number of capabilities actually used by applications.

Partitioning furthermore only works for applications where the master can locate the fixed capability management bitmap. This is the case for all applications that link against the L4Re libraries, which in turn is the default way of building FIASCO.OC applications. If an application did not use L4Re, the master would have to fall back to using an SLT as described above. However, I did not implement an SLT yet as this feature was not necessary for any of the experiments conducted in this thesis.

SUMMARY: System calls are the main way for an application to perform input and output. ROMAIN uses FIASCO.OC's virtual CPU mechanism that allows to intercept any CPU exception raised by a replica. Upon a system call, the master compares replica states for error detection.

If the replicas' states match, the master performs the requested system call on behalf of the replicas. Afterwards, the replica vCPUs are modified as if they had done the system call themselves. Thereby the master ensures that identical inputs reach the replicas.

FIASCO.OC's object-capability system maintains a table of capability selectors for every process. These selector tables will always be identical between correct replicas. However, the master's selector table may differ. To avoid complicated translations between replica and master capabilities, the master applies a partitioning scheme that allows 1:1 translation of replica to master capabilities.

3.5 Managing Replica Memory

In the previous sections I described replication techniques that allow the ROMAIN master to execute replicas inside isolated address spaces, intercept and monitor their system calls, and maintain control over all kernel objects that are used by a replicated application. Replicas furthermore access data in memory, which therefore needs to be managed as well.

From the perspective of redundant multithreading, we can distinguish between *private* and *shared* memory regions. Private memory, which is the focus of this section, is only used by an application internally and never becomes directly accessible to an outside observer. ROMAIN provides each replica with dedicated physical copies of private memory regions. After an initial setup, the replicas can use these copies without synchronizing with the master or other replicas. Private memory regions therefore only have a low impact on replicated execution overhead.

In contrast to private memory, shared memory regions exist between multiple applications and are therefore not in full control of a single ROMAIN master process. I will show in Section 3.6 that this can lead to inconsistencies within a replicated application and that these regions therefore need to be handled in a special way by the master process.

3.5.1 FIASCO.OC *Memory Management*

Microkernel-based operating systems implement the management of memory regions inside user-level applications and only provide kernel mechanisms to enforce these management decisions. FIASCO.OC's memory management is derived from Sawmill's hierarchical memory managers.[21] I explain the concepts involved in this management using Figure 3.10 as an example.

[21] Mohit Aron, Luke Deller, Kevin Elphinstone, Trent Jaeger, Jochen Liedtke, and Yoonho Park. The SawMill Framework for Virtual Memory Diversity. In *Asia-Pacific Computer Systems Architecture Conference*, Bond University, Gold Coast, QLD, Australia, January 29–February 2 2001

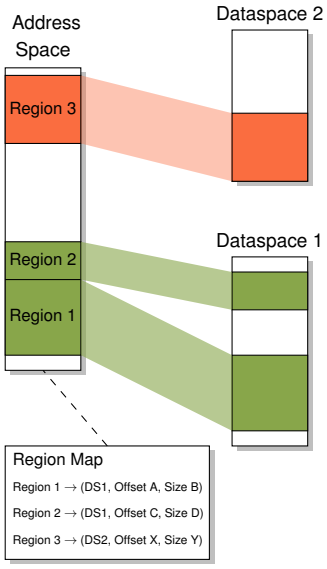

Figure 3.10: A FIASCO.OC application's address space is managed by a local region manager, which combines memory pages from different dataspaces.

Transferring Memory Mappings Making chunks of memory available to another application requires modifications to the target's hardware page table, which is only accessible in privileged processor mode. FIASCO.OC's IPC mechanism provides means to send memory mappings from one application to another. The kernel then carries out the respective page table modifications. This procedure is identical to the one used to delegate kernel object access rights described in Section 3.4.[22]

[22] The kernel distinguishes between object mappings that require modifications to the capability table and memory mappings that lead to modification of the page table.

Dataspaces as Generic Memory Objects Memory content can originate from different sources, such as anonymous physical memory, memory-mapped files, or even a hardware device. In FIASCO.OC all these sources of memory are managed by user-level servers, which provide access to the memory they own through a generic memory object, a *dataspace*.

Region Manager An application's address space consists of a combination of *regions*. Regions are parts of remote dataspaces mapped to a specific virtual address range within the local address space. For every application, a *region manager (RM)* maintains a *region map*, which stores a mapping between regions and the dataspaces that provide backing storage for them.

Page Fault Handling Whenever an application accesses a virtual address that has no corresponding entry in the hardware page table, the CPU raises a page fault exception. The kernel's page fault handler redirects this exception to the faulting application's RM. The RM then looks up the faulting address' region and asks the corresponding dataspace to receive a valid memory mapping via IPC. Using this mechanism all page faults are handled by a user-level component as well.

3.5.2 ROMAIN *Memory Management*

Before a FIASCO.OC application can successfully access an address in memory, it has to complete three actions:

1. The application needs to obtain a capability to a dataspace.
2. A new region in the application's virtual memory must be associated with this dataspace. For this purpose the application asks its RM to *attach* this region to its address space.
3. Finally, the application accesses the memory address. The RM catches the resulting page fault and asks the corresponding dataspace manager for a memory mapping to the appropriate virtual address.

The main difference with respect to memory management in a replicated environment is that ROMAIN needs to provide each replica with a dedicated copy of its respective memory regions. Then, the replica can access memory independently from other replicas with no further execution time overhead, while ROMAIN still remains in control of all input and output operations for the purpose of error detection.

Obtaining Dataspace Capabilities (Step 1) Dataspaces are implemented using the communication channel kernel object. ROMAIN does not replicate these objects, but instead attaches any incoming dataspace to the replicas' and master's capability tables as described in Section 3.4.2.

Region Management (Step 2) For the second step, the ROMAIN master process takes over the role of each replica's region manager. To achieve this, the master uses the system call interception mechanism described in Section 3.4.1 to intercept all messages replicas send to their respective RM.

The master then performs replicated region management as shown in Figure 3.11. Upon interception of a dataspace `attach()` call, the master creates a copy of the respective dataspace for every replica (a). For this purpose the master obtains additional empty dataspaces and copies the original dataspace's content. Thereafter, each replica gets its dedicated copy inserted into its region map (b).

L4Re uses an AVL tree[23] to store the region map. Such trees allow fast lookups, which in case of memory management are required for every page fault handling operation.

The intuitive solution to manage replicated memory in ROMAIN would be to let the master maintain one copy of the region map for every replica. This requires no modification to L4Re's region management code at all. However, it would multiply the number of lookups and modifications that need to be performed during every RM operation.

Looking closer at the problem we find that the replicas' address space layouts will never differ, because correctly functioning replicas will always attach identical dataspaces to identical memory regions. Incorrect replica operations will be detected by ROMAIN before any modification of the region map takes place. Hence, the region maps can be stored in a single AVL tree.

ROMAIN extends L4Re's region map implementation with a specialized leaf node type. ROMAIN's leaf nodes do not store a single mapping from region to dataspace, but instead store one dataspace capability for every replica. Thereby ROMAIN is able to find all dataspace copies for a memory region with a single lookup operation instead of performing N lookups when managing N replicas.

Page Fault Handling (Step 3) The ROMAIN master process sets itself up to be the page fault handler for all replica threads. This is necessary to prevent an external page fault handler from obtaining unvalidated replica state. Furthermore, it allows the master to remain in control of the replicas' address space layout.

When receiving a page fault message, the master process looks up the replicas' memory regions in the region map. It then translates the page

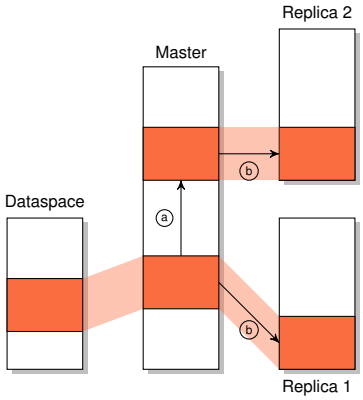

Figure 3.11: ROMAIN provides each replica with a dedicated copy of each memory region.

[23] Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998

fault address to an address within its own virtual address space to find the local mapping of the replica region. Finally, the master uses FIASCO.OC's memory mapping mechanism to map the respective memory regions into each replica address space.

3.5.3 What if Memory was ECC-Protected?

As I explained in Section 2.4.1 on page 28, many modern COTS memory controllers provide ECC protection of the stored data. Therefore, software fault tolerance mechanisms – such as SWIFT – often rely on such protection. They can thereby reduce their performance overhead by never validating data in memory.

ROMAIN maintains dedicated copies of all memory regions for every replica and therefore does not require this kind of memory protection. Any fault in a memory word that leads to incorrect outputs by a replica will be detected on the next externalization event. Furthermore, ROMAIN replicas do not suffer from cases where ECC protection does not suffice to detect errors, which were for instance reported by Hwang and colleagues.[24] However, ROMAIN's solution leads to increased memory overhead: N replicas require N times the amount of memory compared to a single application instance. If ECC memory protection allowed us to significantly decrease this memory consumption, it might therefore be a useful configuration option.

[24] Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 111–122, London, England, UK, 2012. ACM

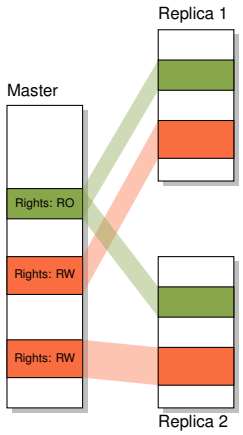

Figure 3.12: Combining ROMAIN and ECC-protected memory allows to reduce memory overhead by using a single copy of read-only memory for all replicas.

If we assume having functioning ECC-protected memory, ROMAIN can improve memory consumption for read-only regions: This kind of data never gets modified by the application, and the master can therefore use a single copy of a read-only region and map it to all replicas' address spaces. ECC will make sure that replicas always read correct data. Furthermore, the virtual memory's write protection can be used to intercept any attempt to overwrite this data by a faulty replica.

Unfortunately, this approach does not work for writable memory regions. As ROMAIN's replicas execute independently, they may read or write those regions at different points in time. If their memory accesses went to the same physical memory location, this might induce inconsistent states and application failures. In order to overcome this, replicas would have to synchronize upon every access to such a memory region, which in turn would largely increase ROMAIN's runtime overhead.

ROMAIN's memory manager supports an ECC mode as shown in Figure 3.12. Read-only memory regions are stored as a single copy and mapped to all replicas. Writable regions are copied as explained in the previous section. While this mode integrates ECC-protected memory into ROMAIN, it does not significantly reduce memory overhead. For example, the SPEC CPU 2006 benchmarks – which I use for evaluating ROMAIN's execution time overheads in Chapter 5 – may share their read-only executable code and data regions this way. However, these regions only occupy a few kilobytes of memory, whereas the benchmarks dynamically allocate hundreds of megabytes for their heap. Heap regions are however writable and hence need to be copied. As a result, ROMAIN's ECC optimization is disabled by default.

This optimization may however be useful in other scenarios where the text segment makes up a larger fraction of an application's address space. For

example, my work computer runs the `chromium` web browser. For this application, the text segment and the dynamically loaded libraries consume about 50% of the application's address space. In this setup, not physically replicating read-only memory segments would significantly reduce the memory overhead required for replication.

3.5.4 Increasing Memory Management Flexibility

Memory management requires a substantial amount of work on the side of the ROMAIN master process. It is therefore a source of runtime overhead. To quantify the memory-related runtime overhead I implemented a microbenchmark that stresses FIASCO.OC's memory subsystem. Listing 3.13 shows the benchmark as pseudocode.

```
1  void membench()
2  {
3     // Step 1: Dataspace allocation
4     Dataspace   ds = memory_allocator.alloc(1 GiB);
5
6     // Step 2: Attach to address space
7     Address  start = region_manager.attach(ds, size=1 GiB);
8
9     // Step 3: Touch memory
10    for (Address a : range(start, start + 1 GB)) {
11        *address   = 0;
12        address   += L4_PAGE_SIZE;
13    }
14 }
```

Listing 3.13: Memory management microbenchmark

An application first allocates a dataspace of 1 GiB size. This first phase constitutes simple object allocation that in the ROMAIN case is proxied by the master process. The application thereafter attaches this dataspace to its address space by calling its region manager. This second step triggers region management within the master process. We can use this second phase to compare ROMAIN's and L4Re's original region management. In the third phase, the benchmark touches every virtual memory page in this dataspace exactly once by writing a single memory word in each page. This last step is dominated by the page fault handling that every memory access will cause.

I first executed the benchmark natively on top of FIASCO.OC to get a baseline measurement. Thereafter I ran the benchmark as a single replica using ROMAIN. The second measurement therefore solely shows the overhead introduced by proxying system calls and memory management and does not contain replication cost. I will investigate replication cost for real-world applications in Chapter 5.

I executed the benchmark on an Intel Core i7 (Nehalem) clocked at 3.4 GHz with 4 GB of RAM. Figure 3.14 shows the benchmark results and breaks down total execution time into the dataspace allocation, region management, and page fault handling overhead. The results are arithmetic means of five benchmark runs each. The test machine was rebooted for each run to avoid cache effects. The standard deviation was below 0.1% for all measurements and is therefore not shown in the graph.

We see that page fault handling overhead dominates native execution. Allocating a dataspace and attaching it to a region each cost one IPC message

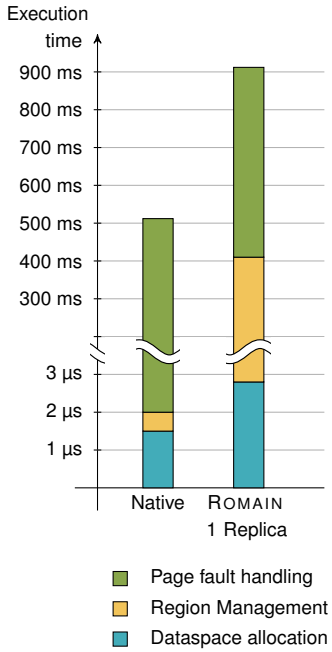

Figure 3.14: Microbenchmark: Memory Management in ROMAIN compared to native FIASCO.OC. (Note the changing scale of the Y axis.)

plus server-side request handling time. These steps are completed within 2 µs. The remainder of the time is spent resolving one page fault for every 4 KiB page. These faults result in sending $1GiB/4KiB = 262,144$ page fault messages, which the pager then translates to the same amount of dataspace mapping requests.

In the ROMAIN case we see that dataspace allocation takes slightly longer than in the native case. This overhead stems from the fact that the single allocation message is now intercepted and proxied by the master process. However, this overhead is negligible compared to the other two phases.

Region management costs around 400 ms in contrast to 0.5 µs in the native case. In contrast to native execution, the master has to perform additional work in this phase. Instead of only managing application regions, it also attaches all memory to its own address space, causing the respective page faults in this course. Given this fact, attaching a region with ROMAIN should therefore be as expensive as the total page fault handling time in the native case, because all 4 KiB page faults need to be resolved here as well. This is not the case because ROMAIN does not touch every single page, cause a page fault, and translate it into a dataspace mapping request. Instead, the master process uses the dataspace interface directly and thereby avoids additional page fault messages.

Once the region is attached in the master, the replica continues execution and touches all memory pages. This case is equivalent to the native case and causes the same amount of page faults to be handled.[25] Hence, the page fault handling phase takes as long in ROMAIN as in the native case.

[25] Remember, master and replica run in different address spaces. The page faults during region management made this memory only available within the master's address space.

Reducing Memory Management Cost I implemented two optimizations in ROMAIN that reduce the overhead for managing replica memory. First, ROMAIN tries to use memory pages with a larger granularity to manage replicas' address spaces and thereby reduce page fault overhead. As we will see this requires hardware support and is not always a viable option. As a second optimization, ROMAIN leverages a FIASCO.OC feature that allows to map more than a single memory page in the case of a page fault.

Using Larger Hardware Pages Page fault handling is expensive because x86/32 systems by default manage memory using page sizes of 4 KiB. While this allows for flexible memory allocation, it requires handling lots of page faults and has been observed to pollute the translation-lookaside buffer (TLB).[26] To address this issue, most modern processor architectures support larger page sizes for memory management. On x86/32, these larger pages of 4 MiB size are called *superpages*. In the example above, using superpages will reduce the number of page faults the master needs to service to $1GiB/4MiB = 256$. This decrease directly leads to a reduction of total page fault handling time.

[26] Narayanan Ganapathy and Curt Schimmel. General Purpose Operating System Support for Multiple Page Sizes. In *USENIX Annual Technical Conference*, ATC '98, Berkeley, CA, USA, 1998. USENIX Association

L4Re's dataspace manager for physical memory pages supports requesting superpage dataspaces, but clients do not use this feature by default for reasons explained in the next section. However, ROMAIN intercepts all replicas' dataspace allocation messages. It can therefore inspect these requests' allocation sizes and above a certain threshold (e.g., 4 MiB) modify the allocation to request a superpage dataspace in order to reduce page fault overhead.

Using Larger Memory Mappings Most modern processor architectures (x86, ARM, SPARC, PowerPC) support some form of superpage mappings. However, it is not useful to solely rely on this feature for two reasons: First, many applications allocate memory in chunks much smaller than 4 MiB. In these cases, using superpages wastes otherwise available physical memory.

Second, using multiple different page sizes makes memory management more complex: Allocating superpages requires contiguous physical memory regions of 4 MiB size to be available, which may not always be the case because of fragmentation arising from memory allocations with smaller page sizes. As an implementation artifact, L4Re's physical dataspace manager restricts allocation even more and requires superpage dataspaces to be completely physically contiguous, meaning that allocating a 1 GiB dataspace composed of superpages requires exactly 1 GiB of contiguous physical memory to be available, lest allocation fails.

For these reasons applications by default refrain from using superpages. However, the FIASCO.OC kernel provides an additional mechanism to reduce page fault processing cost that is independent of the actual hardware page size. When sending a memory mapping via IPC, the sender may chose to send more than a single page at once. The kernel reacts upon such requests by simply modifying multiple page table entries during the map operation. The FIASCO.OC Application Binary Interface (ABI) allows to send memory mappings sized with any power of two, e.g., 4 KiB, 8 KiB, 16 KiB and so on, but requires the start address of such mappings to be a multiple of the mapping size.[27]

Figure 3.15 illustrates the alignment problem. We see a master and a replica address space consisting of 8 pages each. A three-page region shall be mapped from the master (pages 3-5) to the replica (pages 4-6). The maximum possible mapping size for this region is two pages. However, the pages are improperly aligned, so that every page fault on an even page number in the replica will be resolved with a mapping from an odd page number in the master and vice versa. This does not suit FIASCO.OC's alignment requirements and the master therefore has to map three single pages.

The master can reduce paging cost by properly aligning memory regions according to the replica's needs as shown in Figure 3.16. Here, master pages 2-4 are mapped to replica pages 4-6. Any page fault in replica pages 4 or 5 allows the master to handle this page fault by mapping a complete two-page region (pages 2 and 3) to the replica at once. This reduces the number of replica page faults and therefore decreases page fault handling overhead.

ROMAIN leverages FIASCO.OC's support for multi-page mappings to reduce the number of page faults. Whenever a replica raises a page fault, the master process tries to map the largest possible power-of-two region that contains the page fault address to the replica. To facilitate this approach, during every region `attach()` call the master identifies the best possible alignment and largest possible mapping size for each replica and master region. This approach makes sure that later page faults can be resolved using the largest possible memory mapping.

[27] This requirement allows the kernel to fit memory mappings into the least possible amount of page table entries and simplifies the map operation.

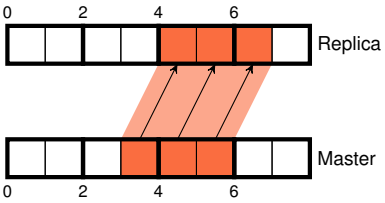

Figure 3.15: If source and destination regions are improperly aligned, the master cannot send large-page mappings to the replica.

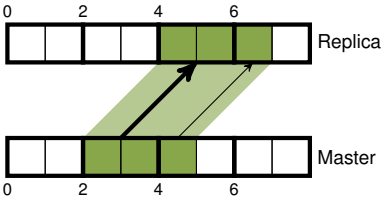

Figure 3.16: Adjusting alignment in replica and master memory reduces the number of page faults to handle.

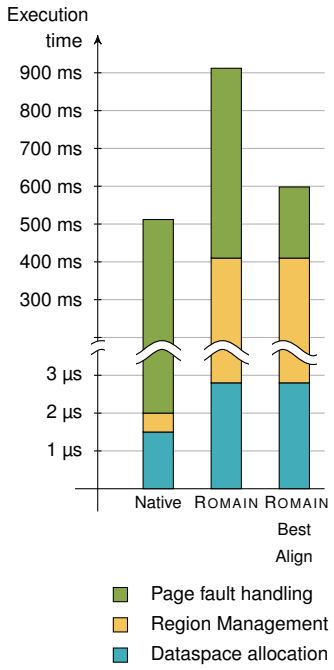

Figure 3.17: Microbenchmark: Memory Management minimizing number of page faults compared to previous microbenchmarks. (Note the changing scale of the Y axis.)

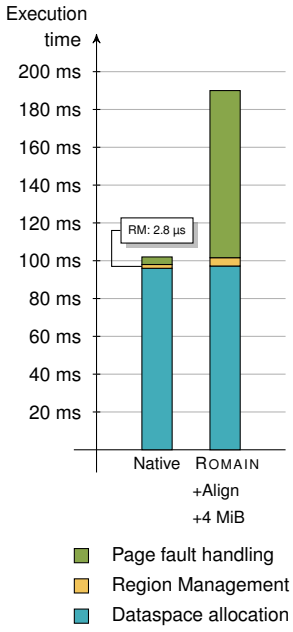

Figure 3.18: Microbenchmark: Memory Management combining superpages and the best alignment strategy

[28] The IEEE and The Open Group. The Open Group Base Specifications – Issue 7. `http://pubs.opengroup.org`, 2013

Effect of Larger Mappings I repeated the microbenchmark introduced in the beginning of this section and applied the previously described memory management optimizations. First I enabled the alignment optimization that reduces the number of page faults independent from hardware-supported superpages. I configured the ROMAIN master process to map up to 4 MiB of memory at once during page fault handling. Figure 3.17 compares the benchmark's outcome (Best Align) to the previously measured results.

We see that dataspace allocation and region management do not change in ROMAIN. This is expected, because we did not modify these parts of the system. Additionally, we see that the Best Align strategy reduces page fault handling cost for ROMAIN. Overall ROMAIN execution time is reduced by 30% for this benchmark. The overhead compared to native execution is reduced to 16%.

Effect of Using Superpages In the next step I enabled use of superpages for mapping the 1 GiB memory region. This modification reduces the number of page faults to be handled to $1GiB/4MiB = 256$. Comparing the result to the previous native benchmark would not be fair, because this would compare native execution with thousands of page faults to ROMAIN with much fewer faults. Instead, I also adjusted the native version of the benchmark to use superpages and compare the results in Figure 3.18.

We see that native execution of the microbenchmark with superpages enabled is already five times faster than the previous native benchmark (102 ms vs. 512 ms). Dataspace allocation actually gets much slower (97 ms vs. 1.5 µs), because the dataspace manager has to perform more work to reserve a physically contiguous memory region. The observation that this is external dataspace management overhead is underlined by the fact that dataspace allocation in ROMAIN is as fast as in native execution, because most of this time is spent executing outside ROMAIN's sphere of replication. Execution in ROMAIN also gets faster than in the previous benchmark (190 ms vs. 590 ms). However, the relative overhead compared to native execution increases to 90%.

Are These Optimizations FIASCO.OC*-Specific?* The optimizations I introduced in this section appear to leverage features specific to the FIASCO.OC microkernel. This raises the question whether they are micro-optimizations for a specific kernel or can be applied to other OS environments as well.

Superpages are a feature of the underlying hardware and are supported by many operating systems. Linux' `mmap()` system call supports allocation of anonymous superpage regions using the `HUGE_TLB` flag. The SystemV `shmget()` operation to create shared memory regions also supports shared memory segments to consist of superpages.[28] Hence, a ROMAIN implementation on Linux could apply optimizations similar to the ones I implemented on top of FIASCO.OC.

SUMMARY: ROMAIN manages replica memory by maintaining a dedicated copy of each memory region for each replica. To do that, ROMAIN interposes dataspace allocation, address space management, and page fault handling for the replicated application.

Memory overhead can be reduced by relying on ECC-protected memory. ROMAIN can then work with a single copy for each read-only memory region and does not need to copy these. The effect of this optimization is limited, because most memory regions are writable and must still be copied.

Replica memory management leads to a significant execution time overhead. I mitigate this overhead by reducing the number of page faults that need to be serviced by the master process. For this purpose ROMAIN uses hardware-provided superpages and maps the largest possible amount of memory when handling a single page fault.

3.6 Managing Memory Shared with External Applications

As we saw in the previous section, memory that is private to a replicated application can be efficiently replicated using ROMAIN. However, additional mechanisms are required to deal with shared memory regions. I consider all virtual memory regions that are accessible to more than one process as shared memory. Microkernel-based systems use such regions for instance to implement communication channels that allow streaming data between applications without requiring kernel interaction.[29]

Shared memory therefore constitutes both input and output from the perspective of a replicated application. Due to its nature, shared memory content is not under full control of the ROMAIN master process. This results in two problems, which I call *read inconsistency* and *write propagation*.

[29] Jork Löser, Lars Reuther, and Hermann Härtig. A Streaming Interface for Real-Time Interprocess Communication. Technical report, TU Dresden, August 2001. URL: `http://os.inf.tu-dresden.de/papers_ps/dsi_tech_report.pdf`

Read Inconsistencies Replicas in a redundant multithreading scenario execute independently and may access memory at different points in time. For example, assume we have a read-only shared memory region that is accessible to a replicated application. Further assume the master process has found a way to directly map this region to all replicas.

A *read inconsistency* between two replicas occurs if first a replica R_1 reads a value v from the shared region and starts to process this datum. Thereafter, an external application updates v to a value v' with $v \neq v'$. Last, a second replica R_2 reads the new value v' and processes it. This scenario violates the determinism principle: Replicas have to obtain identical inputs at all times. Otherwise they may execute different code paths, which leads to falsely detected errors and respective recovery overhead.

Write Propagation If shared memory channels are writable for the replicated application, a second problem needs to be solved: We saw in the previous section that independently running replicas must perform all their write operations to a privately owned memory region. However, in the shared memory scenario the ROMAIN master needs to make the replicas' modifications visible to outside users of the shared memory channel. This *write propagation* needs to be done at a point where all replicas' private copies contain identical and consistent content. It is impossible for the master process to know when this is the case without cooperation from or knowledge about the replicated application.

ROMAIN solves both the read inconsistency and the write propagation problem by translating all accesses to shared memory into externalization events. I implemented two strategies to achieve this: *Trap & Emulate* intercepts every memory access and emulates the trapping instruction. *Copy & Execute* removes the software complexity and execution overhead of this emulator from the ROMAIN master process.

3.6.1 Trap & Emulate

As accesses to shared memory may be input or output operations, the ROMAIN master needs to intercept all these accesses and validate them to ensure correct execution. If replicas got shared memory regions directly mapped, such interception of accesses would be impossible. Therefore, ROMAIN handles page faults in shared memory regions differently than for private regions.

When a replicated application accesses a shared region for the first time, the resulting page fault is delivered to the master process. Instead of establishing a memory mapping to the replica, the master now emulates the faulting instruction and adjusts the faulting vCPU as well as the master's view of the shared memory region as if the access was successfully performed. Thereafter, replica execution resumes at the next instruction.

This approach is conceptually identical to the *trap & emulate* concept that Popek and Goldberg developed for implementing virtual machines.[30] By trapping all read and write operations to shared memory, trap & emulate solves both the read inconsistency and the write propagation problem.

[30] Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, July 1974

An Instruction Emulator for x86 I added an instruction emulator to ROMAIN that is able to emulate the most common memory-related instructions of the x86 instruction set architecture. The emulator disassembles instructions using the UDIS86 disassembler.[31] Based on UDIS86' output the emulator is able to emulate the `call`, `mov`, `movs`, `push`, `pop`, and `stos` instructions.[32]

[31] `https://github.com/vmt/udis86`

[32] Intel Corp. Intel64 and IA-32 Architectures Software Developer's Manual. Technical Documentation at `http://www.intel.com`, 2013

These instructions only comprise a subset of all x86 instructions that access memory. However, these instructions suffice to start a "Hello World" application in ROMAIN and emulate all its memory accesses on the way. Implementing a full-featured instruction emulator is out of scope for this thesis. I will show in Section 3.6.2 that such an emulator is furthermore unnecessary for most shared memory use cases.

Overhead for Trap & Emulate Emulating instructions instead of allowing direct access to memory considerably slows down execution: Rather than reading a memory word within a few CPU cycles, a shared memory access causes a page fault that gets reflected to the ROMAIN master. This overhead stems from hardware and kernel-level fault handling. Additionally, the master adds overhead itself because the faulting instruction needs to be disassembled and emulated.

I implemented three microbenchmarks to quantify the overhead caused by trap & emulate. These benchmarks all work on a 1 GiB shared memory region and mimic typical memory access patterns:

1. *Memset:* The first benchmark fills the whole 1 GiB region with a constant value using the `memset()` function provided by the standard C library. A typical x86 compiler optimizes this function call into a set of `rep stos` (repeat store string) instructions.
2. *Memcopy:* The next benchmark uses the C library's `memcpy()` function to copy data from the first 512 MiB of the shared memory region into the second half. This function call usually gets optimized into `rep movs` (repeat move string) instructions.
3. *Random access:* The last benchmark writes a single data word to a random address within the shared memory region. As we will see, this benchmark is the most demanding in terms of emulation overhead, because random access due to their very nature cannot be optimized into anything else but single-word `mov` instructions.

I executed the benchmarks on the test machine described in Section 3.5.4 and once again compared native execution to the execution of a single replica in ROMAIN. Native execution measures the pure cost of memory operations – memory is directly mapped into the application, all page faults are resolved before the measurements begin. Within ROMAIN, I allocated and attached the shared memory region before starting the benchmark. The Memset and Memcopy benchmarks were repeated 50 times each. The random access benchmark performed 10,000,000 random memory accesses to the shared region.[33]

[33] The number of iterations were chosen so that all three benchmarks had comparable execution times.

Figure 3.19 shows the benchmark results. I compare my results to native execution and execution in ROMAIN with the memory marked as a private region. The latter two execution times are identical, because once memory is mapped to a replica, all memory accesses directly go to the hardware and there is no difference to native execution. In contrast, if the memory region is marked as shared memory in ROMAIN, there is a visible overhead for all benchmarks.

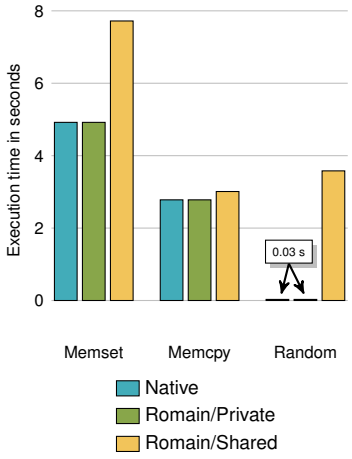

Figure 3.19: Microbenchmark: Overhead for trap & emulate memory handling

For the Memset and Memcopy benchmarks emulating memory accesses is comparatively cheap (57% overhead for Memset, 9% overhead for Memcopy). These overheads result from the optimizations I explained above: in both cases the compiler replaces calls to the C library's `memset()` and `memcpy()` functions by a single x86 instruction with a `rep` prefix. As a result, each benchmark run results in only a single emulation fault for the whole operation.

In contrast, randomly accessing memory leads to a slowdown by a factor of 120. In this benchmark every single memory access causes a separate emulation fault. This means that the ROMAIN instruction emulator is called 10,000,000 times. Random memory access therefore constitutes a worst case for instruction emulation.

A Closer Look at Random Access Cost To find out how the benchmarks relate to a real-world scenario, I inspected the source code of SHMC, an L4Re library that provides packet-based communication channels through a shared memory ring buffer. SHMC uses `memcpy()` operations for the potentially

large payload-related operations. However, SHMC additionally needs to do packet bookkeeping within the shared data region. For the latter purpose, SHMC requires random access operations.

I therefore had a closer look at where the overhead for emulating random shared memory accesses comes from. Figure 3.20 breaks down a single emulated memory access into four phases: First, an emulated access causes a page fault in the CPU, which gets delivered to the ROMAIN master process. This exception handling part takes roughly 1,900 CPU cycles. Another 1,400 CPU cycles are spent by the master for validating CPU state and dispatching the event to the PageFault observer. Inside the emulator, time is spent on disassembling the faulting instruction (6,400 cycles) and emulating the write effects (2,400 cycles).

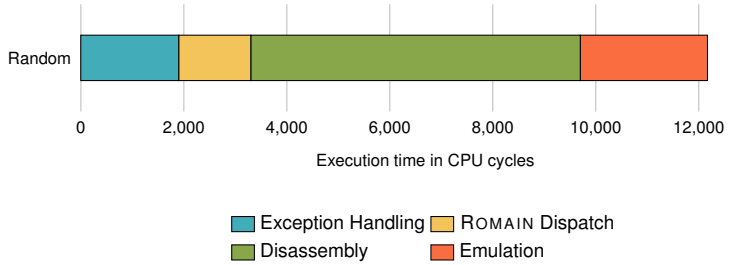

Figure 3.20: Breakdown of random access emulation cost (trap & emulate)

In total, ROMAIN spends around 12,000 CPU cycles emulating a shared memory access, whereas the benchmark indicates a cost of about 100 cycles for a native memory access. About half of this time is spent inside the UDIS86 disassembler for parsing the faulting instruction and filling a disassembler data structure. (While 100 cycles sound high for a memory access, Intel's Performance Analysis Guide roughly approximates an uncached DRAM access to cost about 60 ns, which would come down to about 150 cycles on my test computer.[34] As the Random Access microbenchmark uses 1 GiB of memory, randomly picking one word for the next access is highly likely to miss the cache.).

Additionally, the disassembly and emulation infrastructure add a large amount of source code complexity to the ROMAIN master. The UDIS86 library alone comprises about 8,000 lines of code. For these reasons I implemented an alternative shared memory interception technique that does not use the disassembler at all.

[34] David Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors. Technical report, Intep Corp., 2009. https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf

3.6.2 *Copy & Execute*

The ROMAIN instruction emulator has two jobs: First, replica-virtual addresses need to be translated into master-virtual ones, because instruction emulation takes place in the master and the master's address space layout may differ from the replicas. Second, the actual instruction needs to be emulated and the replica vCPUs need to be adjusted accordingly. If we can solve both problems without a disassembler, we can avoid both the implementation complexity and the performance drawbacks explained above.

Avoiding Address Translation Using Identity Mappings As explained previously, the master process keeps dedicated copies of every private memory region for every replica. For this reason, virtual addresses in the replicas and the master may differ. However, shared memory regions are never copied, but only exist as a single instance in the master. ROMAIN uses this fact to solve the address translation problem. The master intercepts IPC calls that are known to obtain capabilities to shared-memory dataspaces. When the replicated application then tries to attach such a dataspace to its address space, the master maps the respective region to the same virtual address in replica and master address spaces.

Due to this 1:1 mapping of shared memory regions, no address translation is necessary anymore. Unfortunately, this approach only works if the virtual memory region requested by the replica is still available in the master process. This may be a problem if replicas request mappings to a fixed address. However, most L4Re applications leave selection of an attached region's virtual address to their memory manager.[35] In case of the replicas this is the ROMAIN master, which can then select a suitable region.

[35] This behavior is similar to most Linux applications that obtain anonymous memory to an arbitrary address using the `mmap()` function.

A Fast and Complete Instruction Emulator Using identity mappings any faulting memory instruction will now use the faulting replicas' register state and virtual addresses that are identical to addresses in the master. Hence, there is no need for instruction emulation anymore. Instead, we can perform the faulting shared memory access directly in the master process using the fastest and most complete instruction "emulator" available: the physical CPU!

I added a shared memory emulation strategy to ROMAIN that I call *copy & execute*. This strategy performs shared memory accesses by executing the following four steps:

1. *Create a dynamic function*: Allocate a buffer in memory, fill this buffer with the faulting instruction followed by a return instruction (Byte `0xC3`).
2. *Adjust physical CPU state*: Store the current physical CPU state in memory and copy the faulting replica's CPU state into the physical CPU.
3. *Perform the shared memory access*: Call the dynamic function created in step 1. This will perform the memory access in the master's context and works because we use identity-mapped shared memory regions. After the memory access, the dynamic function will return to the previous call site.
4. *Restore CPU state*: Store the potentially modified physical CPU state into the replica's vCPU. Restore the previously stored master CPU state.

The copy & execute strategy does not require expensive instruction disassembly. However, as the x86 instruction set uses variable-length instructions, we need to identify the actual length of the faulting instruction to perform the copy operation in step 1. For this purpose I used MLDE32,[36] a tiny instruction length decoder that does this job faster than UDIS86.

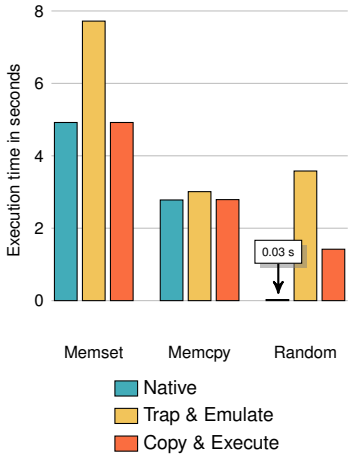

Figure 3.21: Microbenchmark: The Copy & Execute strategy improves the performance of memory access emulation.

Benefit of Copy & Execute I repeated the previously introduced memory microbenchmarks and show the results in Figure 3.21. The Memset and Memcopy benchmarks do not show any overhead anymore. The overhead for random accesses decreased from a factor of 120 down to about a factor of 47.

[36] `http://www.woodmann.com/collaborative/tools/index.php/Mlde32`

Furthermore, Figure 3.22 compares the cost for a single random memory access for copy & execute to the trap & emulate case I presented before. Exception handling and ROMAIN's internal event dispatch are not touched by the modifications at all and therefore remain the same. For the copy & execute case, disassembly contains the cost of determining the instruction length, which is significantly smaller than for trap & emulate. In total, a random access to shared memory using copy & execute takes about 4,800 CPU cycles.

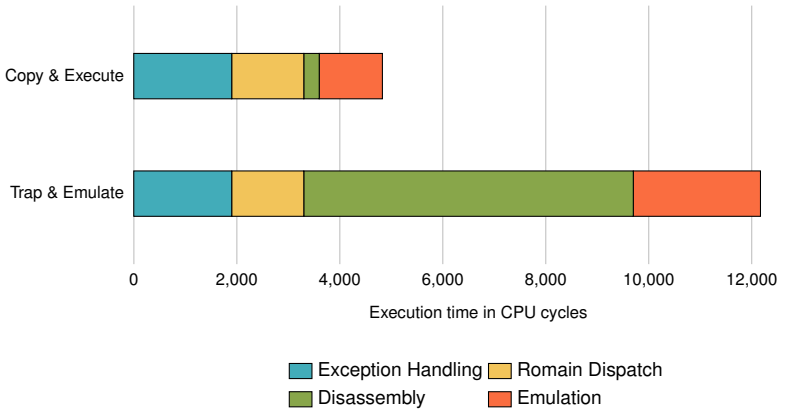

Figure 3.22: Breakdown of random access emulation cost (copy & execute)

Limitations of Copy & Execute While copy & execute significantly decreases shared memory access cost, this approach has two drawbacks: It does not support all types of memory accesses, and it relies on identity-mapped memory regions. I will explain these limitations below.

Emulating memory accesses using copy & execute only supports memory instructions that modify the state of shared memory and the general-purpose registers. It does not support memory-indirect modifications of the instruction pointer, such as a jump through a function pointer, because this would divert control flow within the master process. So far I did not encounter any application that uses function pointers stored in a shared memory region. I therefore argue that it is safe to assume that well-behaving applications never modify the instruction pointer based on a shared-memory access.

I already explained that the ROMAIN master process can establish identity mappings for the majority of shared memory regions. Most memory instructions on x86 only have a single memory operand, so that they will always read or write a dedicated 1:1-mapped shared memory location. There is one exception to this rule, though. The `rep movs` instruction, which is for instance used to implement `memcpy()`, uses two memory operands. In this case, one of the addresses may point to a replica-private virtual address that still requires translation into a master address.

For this special case, ROMAIN's implementation of copy & execute includes a pattern-matching check for the respective opcode bytes (`0xF3 0xA5`). If this opcode is detected, ROMAIN additionally inspects the instruction's operands for whether they point into a shared memory region. If one of the pointers is replica-private, the master rewrites this operand to the respective master address.

SUMMARY: From the perspective of a replicated application, shared memory content may constitute input as well as output. The ROMAIN master never gives replicas direct access to these regions, but instead handles every access as an externalization event and emulating the access.

I showed that the overhead of the naive trap & emulate approach to emulating memory operations can be significantly reduced by a copy & execute strategy. The latter strategy is however not universally applicable because it makes assumptions about the type of memory-accessing instructions and their memory operands.

3.7 Hardware-Induced Non-Determinism

The ROMAIN master process intercepts all replica system calls, administrates kernel objects on behalf of replicas, and manages the replicas' address space layouts. These three mechanisms cover close to all replica input and output paths. They ensure that replicas execute deterministically and that replica states are validated before data leaves the sphere of replication.

In addition to these sources of input, applications can also obtain input through hardware features, such as reading time information or using a hardware-implemented random number generator. Hence, accesses to such hardware resources needs to be intercepted and handled by the ROMAIN master in order to ensure deterministic replica execution.

3.7.1 Source #1: `gettimeofday()`

Software can read the current system clock through the `gettimeofday()` function provided by the C library. To speed up clock access, many operating systems implement this function without using an expensive system call. Linux for example provides fast access to clock information through the virtual dynamic shared object (vDSO).[37] The vDSO is a read-only memory region shared between kernel and user processes. The kernel's timer interrupt handler updates the vDSO's time entry on every timer interrupt. Based on this mechanism the `gettimeofday()` function can be implemented as a simple read operation from the vDSO.

[37] Matt Davis. Creating a vDSO: The Colonel's Other Chicken. Linux Journal, mirror: `http://tudos.org/~doebel/phd/vdso2012/`, February 2012

FIASCO.OC provides a mechanism similar to the vDSO, which is called the *Kernel Info Page (KIP)*. The KIP provides information about the running kernel's features as well as a clock field that is used to obtain time information. An intuitive solution to make time access through a KIP or vDSO mechanism deterministic would be to handle these regions as shared memory. With this approach the ROMAIN master would intercept and emulate all accesses to time information.

Avoiding KIP Access Emulation Virtual memory only allows us to configure the access rights for whole memory pages (i.e., 4 KiB regions). ROMAIN can therefore only mark the whole KIP as shared memory and emulate all accesses. Unfortunately, the KIP not only contains a clock field, but also a heavily used memory region specifying FIASCO.OC's kernel entry code. This means that the KIP is accessed multiple times for every system call. Emulating all KIP accesses to maintain control over time input is therefore prohibitive, because we would as a collateral damage slow down every system call by several orders of magnitude.

To avoid this slowdown, I implemented TimeObserver, an event observer that emulates the `gettimeofday()` function within the ROMAIN master. During application startup, the observer patches the replicated application's code and places a software breakpoint (byte `0xCC`) on the first instruction of the `gettimeofday()` function. When this function gets called by the replicated program at a later point in time, this will cause a breakpoint trap in the CPU. FIASCO.OC then notifies the ROMAIN master about this CPU exception. During event processing, the TimeObserver then reads the KIP's clock value once and adjusts the replicas' vCPU states as if a real call to the instrumented function had taken place.

Limitations of the TimeObserver To patch the function entry point, TimeObserver needs to know the start address of the `gettimeofday()` function. This symbol information is not available for every binary program that ROMAIN replicates. If the binary was for instance stripped from symbol information, TimeObserver could not perform its instrumentation duties. However, software vendors in practice often provide debug symbol information along with their binary-only software.[38] This information can be sourced by TimeObserver to determine the respective address. If such information is not available, ROMAIN could still fall back to emulating all KIP accesses as I explained above. However, I did not implement this mechanism yet.

[38] Microsoft Corp. Symbol Stores and Symbol Servers. Microsoft Developer Network, accessed on July 12th 2014, `http://msdn.microsoft.com/library/windows/hardware/ff558840(v=vs.85).aspx`

3.7.2 Source #2: Hardware Time Stamp Counter

Apart from timing information provided through a kernel interface, CPUs often have dedicated instructions that allow to determine time. On x86 the `rdtsc` instruction provides such a mechanism and allows an application to determine the number of clock cycles since the CPU was started.[39]

By default, `rdtsc` can be executed at any CPU privilege level. Replicas may use this instruction to once again obtain different inputs depending on their temporal order and the physical CPU they run on. ROMAIN therefore needs to intercept `rdtsc` calls to provide deterministic input.

This problem can in principle be solved using the TimeObserver approach. We would have to know all locations of `rdtsc` instructions in advance and would then convert these instructions into software breakpoints during application startup. However, as explained in Section 3.3, finding specific x86 instructions within an unstructured instruction stream is difficult. To avoid these difficulties, ROMAIN instead uses a feature provided by x86 hardware: Kernel code can set the TimeStampDisable (TSD) bit in the CPU's CR4 control register to disallow execution of the `rdtsc` instruction in user mode.[40]

[39] Since the Nehalem microarchitecture `rdtsc` is actually incremented using its own frequency to provide a constant clock regardless of CPU-internal frequency scaling. This detail is not important for my explanation.

[40] Intel Corp. Intel64 and IA-32 Architectures Software Developer's Manual. Technical Documentation at `http://www.intel.com`, 2013

ROMAIN asks the kernel to set the TSD bit for every replica. Executing the `rdtsc` instruction within the replica will then cause a General Protection Fault. This fault is reflected to the master process and then handled by the TimeObserver without having to patch any code beforehand.

3.7.3 Source #3: Random Number Generation

In addition to time-related hardware accesses, modern processors provide special instructions to access other non-deterministic hardware features. Intel's most recent CPUs for instance provide access to a hardware random number generator through the `rdrand` instruction.[41] Intel have furthermore announced the introduction of the Software Guard Extensions (SGX) instruction set extension in future processor generations. SGX allows to execute parts of an application within an isolated compartment, called an enclave.[42] Enclave memory is protected from outside access by encryption with a random encryption key.

[41] Intel Corp. Intel Digital Random Number Generator (DRNG) – Software Implementation Guide. Technical Documentation at `http://www.intel.com`, 2012

[42] Intel Corp. Software Guard Extensions – Programming Reference. Technical Documentation at `http://www.intel.com`, 2013

For replication purposes both `rdrand` and SGX instructions need to be intercepted by ROMAIN in order to ensure determinism across replicas. I did not implement this kind of interception yet, but there are two options to do so:

1. *Disallow Instructions*: Both kinds of instructions will only be available in a subset of Intel's CPUs and software is therefore required to check the availability of these extensions on the current CPU before using them. This is done using the `cpuid` instruction. ROMAIN can intercept `cpuid` and pretend to a replicated application that these instructions are unavailable on the current CPU. Software will then have to work around this problem and must not use the non-deterministic instructions.

2. *Virtualize Instructions*: Intel's VMX virtualization extensions allow the user to configure for which reasons a virtual machine will cause a VM exit that is then seen by the hypervisor. Random number generation, SGX, as well as `rdtsc` can thereby be configured to raise a visible externalization event. ROMAIN could be extended to run replicas not only as OS processes but as hardware-supported virtual machines and thereby intercept and emulate these instructions.

Florian Pester demonstrated that replication based on hardware-assisted virtual machines is feasible.[43] With such an extension we can implement both of the above options. However, while the first alternative will be easier to implement, some applications may simply refuse to work if their expected hardware functionality is unavailable. Therefore, I suggest to investigate option number two in future work.

[43] Florian Pester. ELK Herder: Replicating Linux Processes with Virtual Machines. Diploma thesis, TU Dresden, 2014

3.8 Error Detection and Recovery

With the mechanisms described above, ROMAIN is able to execute replicas of a single-threaded application. The replicas run as isolated processes and the management mechanisms I presented in the previous sections provide four isolation properties, which are fundamental for successful error detection and recovery:

1. *Replicas have an identical view of their kernel objects.* Kernel objects are created using system calls. ROMAIN intercepts all system calls and is therefore able to ensure that replicas always have an identical view of these objects. As a result, we can assume all system calls that target the same object to have the same semantics.
2. *Replicas have identical address space layouts.* The ROMAIN master process acts as the memory manager for all replicas. For this purpose it intercepts all system calls related to dataspace acquisition and address space management. ROMAIN thereby establishes identical address space layouts in all replicas. By servicing replica page faults, the master process furthermore ensures that the replicas' accessible memory regions exactly match.
3. *Replicas obtain identical inputs.* Replicas receive inputs through system calls, shared memory, and timing-specific mechanisms. ROMAIN intercepts all these sources of input and thereby provides all replicas with the same inputs. Replicas execute the same code and therefore will deterministically produce the same output unless they suffer from the effects of hardware faults.
4. *Data never leaves the sphere of replication without being validated.* Replicas output data using system calls or shared memory channels. ROMAIN intercepts both types of output. As a consequence of properties 1, 2, and 3 these outputs will be identical as long as the replicas do not experience hardware faults.

3.8.1 Comparing Replica States

While executing a replicated application, the FIASCO.OC kernel reflects all externalization events to the ROMAIN master process. Once all replicas reach their next externalization event, the master compares the replicas' register states and the content of their UTCBs. If all these data match, the intercepted externalization event is valid and can be further handled by the master.

The replica state comparison only validates that the replicas at this point in time still agree about their outputs. In order to decrease comparison overhead the master does not compare the replicas' memory contents. If a hardware error modifies a replica's memory state and the replica still reaches its next system call in the same state as all other replicas, this faulty state remains undetected.

Such an undetected error can lead to two scenarios: First, the erroneous memory value does not case the application to misbehave at all. This is a case of a benign fault and not detecting it does not harm the replicated application. Redundant multithreading mechanisms often avoid detecting benign errors in order to reduce their execution time overhead.

In the second scenario, the faulty memory location is used to compute subsequent output operations. The affected replica will then produce a future externalization event that differs from the other replicas and ROMAIN will then detect the error. Not detecting faulty memory state immediately in this scenario increases error detection latency, but does not impact the correctness of the replication mechanism.

If replica states mismatch, the master initiates a recovery procedure. First, ROMAIN tries to perform forward recovery using majority voting. The master checks if the state of a majority of replicas matches. If this is the case, the faulty replicas' states are overwritten with the majority's state. This includes overwriting registers, UTCBs, as well as all memory content. After successful forward recovery all replicas are in an identical state once again and may immediately continue operation.

If the number of replicas does not allow to make a majority decision, ROMAIN falls back to providing fail-stop behavior. The replicated application is terminated and an error message is returned.

3.8.2 Reducing Error Detection Latency

ROMAIN detects faulty replicas as soon as they cause an externalization event that differs from the other running replicas. The underlying assumption is that applications regularly cause externalization events in the form of page faults and system calls that trigger validation. Unfortunately, this assumption does not hold in all cases.

Replicas Stuck in an Infinite Loop The first problem results from errors that cause replicas to execute infinite loops that do not make any progress. This may for instance happen if a hardware fault modifies the state of a loop variable in a way that the loop's terminating condition is never reached.[44] In this case the ROMAIN master will never be able to compare all replicas' states because the faulty replica never reaches its next externalization event for comparison.

[44] Martin Unzner. Implementation of a Fault Injection Framework for L4Re. Belegarbeit, TU Dresden, 2013

ROMAIN solves this problem by starting a watchdog timer whenever the first replica raises an externalization event. If this watchdog expires before all replicas reach their next externalization event, error correction is triggered. In case a majority of replicas reached their externalization event, the remaining replicas are considered faulty. ROMAIN then halts these replicas and triggers recovery as described before. If fewer than half of the replicas reached their externalization event, these replicas may be the faulty ones. ROMAIN then continues to wait for externalization events from the remaining replicas before performing error detection.

Bounding Validation Latencies A second problem related to error detection was analyzed by Martin Kriegel in his Bachelor's Thesis[45] which I advised. Compute-bound applications may perform long stretches of computation in between system calls as shown in Figure 3.23 on the following page. This increases the period between state validations, t_v. A fault happening within this computation may have a potentially long error detection latency te_1.

[45] Martin Kriegel. Bounding Error Detection Latencies for Replicated Execution. Bachelor's thesis, TU Dresden, 2013

Kriegel identified this as a problem in three cases:

1. *Multiple errors:* If t_v becomes greater than the expected inter-arrival time of hardware faults, a replicated application may suffer from multiple independent faults before validation takes place. In this case, recovery through majority voting may no longer be possible.

2. *Checkpoint Overhead:* If ROMAIN operates in fail-stop mode, recovery may trigger checkpoint rollback and re-computation. The longer t_v is, the longer the required re-computation takes.
3. *Loss of Timing Guarantees:* The above two effects may eventually lead to loss of timing guarantees due to hardware errors. Note, that this thesis does not deal with providing real-time guarantees for replicated applications in the presence of hardware faults. Research on this topic is ongoing and has for instance been published by Philip Axer.[46]

[46] Philip Axer, Moritz Neukirchner, Sophie Quinton, Rolf Ernst, Björn Döbel, and Hermann Härtig. Response-Time Analysis of Parallel Fork-Join Workloads with Real-Time Constraints. In *Euromicro Conference on Real-Time Systems*, ECRTS'13, Jul 2013

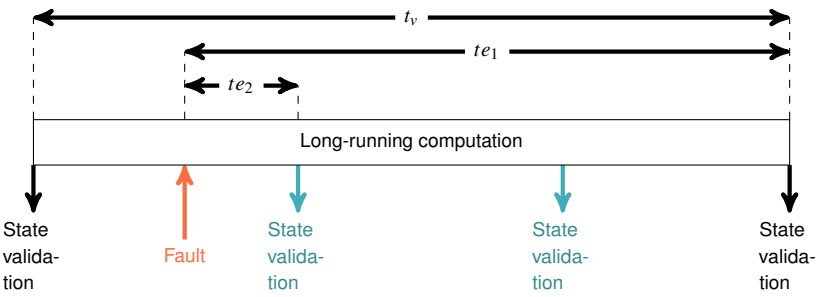

Figure 3.23: Long-running computations increase error detection latency if ROMAIN only relies on system calls for state comparison.

To address these issues, Kriegel proposed to insert artificial state validation operations at intervals smaller than t_v. These intervals are shown in cyan in Figure 3.23. With these additional state validations, detection latency is reduced to te_2 and ROMAIN may therefore trigger recovery operations faster.

Kriegel validated his proposal with an extension to ROMAIN and the FIASCO.OC kernel. This extension inserts artificial exceptions into running replicas by leveraging a hardware extension. Modern CPUs provide programmable performance counters to monitor CPU-level events.[47] These counters can be programmed to trigger an exception upon overflow. Kriegel used this feature to trigger exceptions for instance once a replica retired 100,000 instructions. As replicas execute deterministically, they will raise such an exception at the same point within their execution and the ROMAIN master can use these exceptions to validate their states.

[47] Intel Corp. Intel64 and IA-32 Architectures Software Developer's Manual. Technical Documentation at `http://www.intel.com`, 2013

During his work, Kriegel found that counting retired instructions leads to imprecise interrupts, because this performance counter depends on complex CPU features, such as speculative execution. Depending on the workload and other hardware effects, speculation may lead to the retirement of multiple instructions within the same cycle. Due to this effect, replicas may get interrupted by a performance counter overflow, but still their instruction pointers and states differ because some replicas may already have executed more instructions than others. Kriegel devised a complicated algorithm to let replicas catch up with each other in such situations. However, the fundamental problem is well-known and Intel's Performance Analysis Guide suggests to use other performance counters – such as *Branches Taken* – to obtain more precise events.[48]

[48] David Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors. Technical report, Intep Corp., 2009. `https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`

SUMMARY: ROMAIN detects erroneous replicas upon their next externalization event by comparing the states of all replicas. If possible, ROMAIN provides forward recovery by majority voting among the replicas. If this is impossible, ROMAIN falls back to fail-stop behavior.

ROMAIN relies on frequent externalization events, but some applications may not use system calls often enough. In this case, hardware performance counters can be used to insert artificial state validation operations.

In this chapter I described how ROMAIN instantiates application replicas, manages their resources, and validates their states. Redundant multithreading assumes that replicas always execute deterministically if presented with the same inputs. This is unfortunately not the case if the replicated application is multithreaded, because scheduling decisions made by the underlying kernel may introduce additional non-determinism. Replicating multithreaded applications therefore requires additional mechanisms, which I am going to discuss in the next chapter.

4
Can we Put the Concurrency Back Into Redundant Multithreading?

Many software-level fault tolerance methods — such as SWIFT introduced in Section 2.4.2 on page 29 — were developed or tested solely targeting single-threaded application benchmarks. And despite their names even redundant multithreading techniques — such as PLR and ROMAIN in the version I introduced until now — cannot replicate multithreaded applications. In this chapter I explain that scheduling non-determinism causes false positives in error detection for such programs. Multithreaded replication needs to correctly distinguish these false positives from real errors in order to be both correct and efficient.

Deterministic multithreading techniques solve the non-determinism problem. I review related work in this area and then present *enforced* and *cooperative* determinism – two mechanisms that allow ROMAIN to achieve deterministic multithreaded replication by making lock acquisition and release deterministic across replicas. I compare these two mechanisms with respect to their execution time overhead and their reliability implications.

The ideas discussed in this chapter were published at EMSOFT 2014.[1]

[1] Björn Döbel and Hermann Härtig. Can We Put Concurrency Back Into Redundant Multithreading? In *14th International Conference on Embedded Software*, EMSOFT'14, New Delhi, India, 2014

4.1 What is the Problem with Multithreaded Replication?

Developers make use of modern multicore CPUs by adapting their applications to leverage the available resources concurrently. *Multithreaded programming* frameworks — such as OpenMP[2], Cilk,[3] or Callisto[4] — support this adaptation. These frameworks usually build on a low-level threading implementation.

The POSIX thread library (`libpthread`[5]) is one of the most widely used low-level thread libraries. Extending ROMAIN to support `libpthread` applications therefore allows to replicate a wide range of multithreaded programs. Throughout this chapter I will therefore focus on applications using `libpthread`. In this section I first give a short overview of multithreading primitives. Thereafter I explain how non-determinism in multithreaded environments counteracts replicated execution.

[2] L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998

[3] Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Conference on Programming Language Design and Implementation*, PLDI'98, pages 212–223, Montreal, Quebec, Canada, June 1998

[4] Tim Harris, Martin Maas, and Virendra J. Marathe. Callisto: Co-Scheduling Parallel Runtime Systems. In *European Conference on Computer Systems*, EuroSys '14, Amsterdam, The Netherlands, 2014. ACM

[5] The IEEE and The Open Group. POSIX Thread Extensions 1003.1c-1995. `http://pubs.opengroup.org`, 2013

4.1.1 Multithreading: An Overview

A thread is the *fundamental software abstraction* of a physcial processor in a multithreaded application and represents a single activity within this program. The thread library manages thread properties, such as what code it executes and which stack it uses. The *execution order* of concurrently running threads is determined by the underlying OS scheduler. This separation has two advantages: first, applications do not need to be aware of the actual number of CPUs and can launch as many threads as they need. The OS will then take care of selecting which thread gets to run when and on which CPU. Second, in contrast to a single application, the OS can incorporate global system knowledge into its load balancing decisions.

To cooperatively compute results, threads use both *global and local resources*. While local resources are only accessed by a single thread, global resources are shared among all threads. The state of global resources and hence program results heavily depend on the order in which threads read and write this shared data. These situations are called *data races*. Races and the potential misbehavior they may induce are an important concern when developing and testing parallel applications.[6]

A code path where threads may race for access to a shared global resource is called a *critical section*. Thread libraries provide *synchronization mechanisms*, which developers can use to protect critical sections from data races. These mechanisms include a range of different interfaces, such as blocking and non-blocking locks, condition variables, monitors, and semaphores.[7]

[6] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Workshop on Binary Instrumentation and Applications*, WBIA'09, pages 62–71, New York, NY, USA, 2009. ACM

[7] Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008

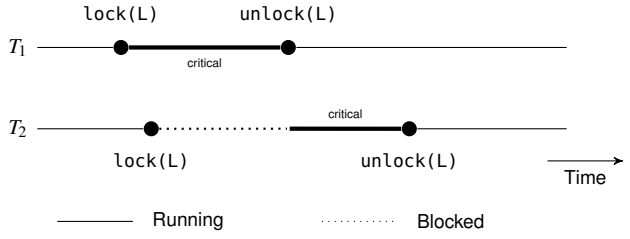

Figure 4.1: Blocking Synchronization

Figure 4.1 gives a general overview of how synchronization primitives work. Critical sections are protected by one or more synchronization variables, such as a lock L. Whenever a thread T_1 tries to execute a critical section, it issues a synchronization call, such as `lock(L)`, to mark the critical section as busy. When a thread T_2 tries to enter a critical section protected by the same lock while the lock is owned by T_1, T_2 gets blocked until T_1 leaves its critical section by calling `unlock(L)`. The synchronization mechanism thereby makes sure that only one thread at a time can execute the critical section.

4.1.2 Multithreading Meets Replication

As presented in the previous chapter, ROMAIN implements fault tolerance by replicating an application N times and validating the replicas' system call parameters. Intuitively, extending this approach to multithreaded applications is straightforward: ROMAIN should launch N replicas of every application thread and compare these threads' system calls independently. Unfortunately,

this approach fails for applications where threads behave differently depending on the state of a global resource and the time and order of accesses to this resource.

To illustrate the problem let us consider a multithreaded application that uses the ThreadPool design pattern[8] to distribute work across a set of worker threads as shown in Figure 4.2. An application consists of two worker threads W_1 and W_2. The workers operate on work packets (A-C) which they obtain one packet at a time from a globally shared work queue. The workers execute the code shown in Listing 4.3: A packet is first removed from the work queue (`get_packet()`). Let us assume this function is properly synchronized so that no data race exists and it always removes and returns the first entry from the shared work queue. In the second step, the worker processes this packet (`process()`). Finally, the worker makes this operation's results externally visible (`output()`).

[8] Doug Lea. *Concurrent Programming In Java. Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999

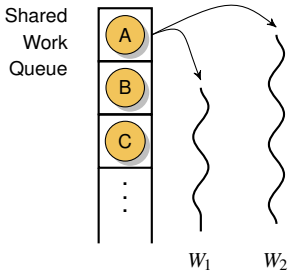

Figure 4.2: Example ThreadPool: Two worker threads obtain work items from a globally shared work queue.

If the two worker threads are scheduled concurrently by the OS scheduler, their behavior depends on who gets to access the work queue first. Figures 4.4 ⓐ and ⓑ show two possible schedules for processing work packets A and B. Both schedules are valid, but lead to different program behavior from an external observer's point of view. Schedule a) will produce the event sequence "`output(A); output(B)`", whereas schedule b) will produce "`output(B); output(A)`."

```
1  void worker()
2  {
3     while (true) {
4        p = get_packet();
5        process(p);
6        output(p);
7     }
8  }
```

Listing 4.3: Worker thread implementation

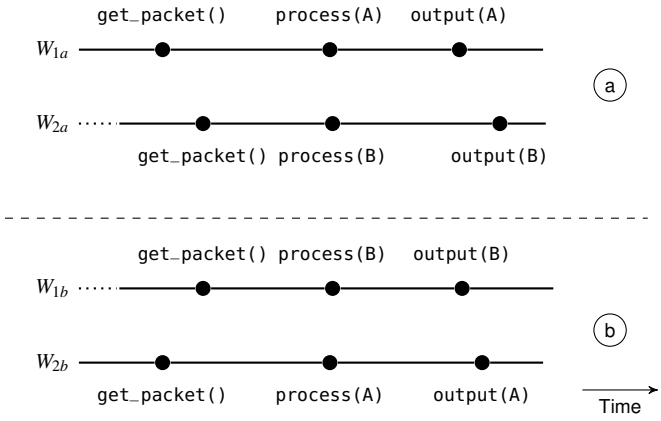

Figure 4.4: Example schedules for the ThreadPool example

We see in the example that different timing of events can impact program behavior and lead to non-deterministic execution even given the same inputs. Scheduling decisions made by the underlying OS are a main source of such non-determinism and remain out of control of the application or the thread library.[9]

[9] That is, unless the application uses OS functionality to micromanage all its threads and thereby forgoes any benefits from OS-level load balancing and scheduling optimizations.

Let us now assume, we use ROMAIN to replicate our application using the intuitive approach of replicating threads independently. Each application thread is instantiated twice: W_{1a} and W_{1b} are replicas of worker 1, W_{2a} and W_{2b} are replicas of worker 2. Replicas execute independently and ROMAIN intercepts their externalization events (`output()`) to validate their states. In this scenario schedules ⓐ and ⓑ from Figure 4.4 may constitute schedules executed by the two application replicas.

To detect errors ROMAIN will compare externalization events generated by replicas of the same application thread. The master will thereby find that replica W_{1a} executes `output(A)`, while replica W_{1b} calls `output(B)`.

ROMAIN will deem this a replica mismatch, report a detected hardware fault, and trigger error recovery. The same will happen for replicas W_{2a} and W_{2b}.

In the best case, this *false positive* error detection will induce additional execution time overhead, because ROMAIN performs unnecessary error recovery. In the worst case, non-deterministic execution will lead all replicas of an application to execute different schedules and produce different events. In this case, no majority of threads with identical states may be found. ROMAIN is then no longer able to perform any error recovery at all.

SUMMARY: Scheduling-induced non-determinism may yield multiple valid schedules of a multithreaded application that generate different outputs. This non-determinism may either seriously impact replication performance or hinder successful replication completely.

4.2 Can we make Multithreading Deterministic?

Non-determinism originating from data races and from OS-level scheduling decisions makes development of concurrent software complex and error-prone. These issues raise the question whether thread-parallel programming really is a useful paradigm and if we can replace traditional concurrency models with a more comprehensible approach.[10] *Deterministic multithreading (DMT)* is such an alternative.

[10] Edward A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006

The goal of DMT is to make every run of a parallel application exhibit identical behavior. Methods to do so have been proposed at the levels of programming languages, middleware and operating systems. Applying these mechanisms to multithreaded replicas in ROMAIN will solve the non-determinism problem introduced in the previous section — deterministic replicas yield deterministic externalization events unless affected by a hardware error. I will therefore review DMT to find techniques that are applicable in the context of ROMAIN.

Language-Level Determinism Programming-language extensions introduce new syntactic constructs or compiler-level analyses to allow developers to express concurrency while maintaining freedom of data races. As an example, Deterministic Parallel Java augments the Java programming language with annotations to specify which regions in memory are accessed by a piece of code. Developers specify these regions and then use parallel programming constructs to parallelize code segments. The compiler uses the annotations to verify that concurrent code segments are race-free.[11]

[11] Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'09, pages 97–116, Orlando, Florida, USA, 2009. ACM

If applications are implemented to be completely deterministic, they will never exhibit alternative schedules. Such deterministic programs can be replicated without intervention from the replication system. ROMAIN therefore benefits from these language extensions. However, my aim is to support a wide range of binary-only applications and it is hence impractical to rely on all applications being implemented using specific programming languages.

Determinism in Multithreaded Distributed Systems Distributed systems often use state machine replication to distribute work across compute nodes and tolerate node failures. Recent distributed replication frameworks address the fact that the software running on single nodes may exhibit behavioral variations due to multithreaded non-determinism.

Cui's Tern[12] analyzes parallel applications and their inputs with respect to the schedules they induce. The Tern runtime then classifies incoming data and tries to force scheduling and lock acquisition down a path that was previously learned from similar inputs. Tern batches inputs into larger chunks that are executed concurrently. The runtime can thereby reduce the number of input classifications. Tern has low execution time overhead unless data does not match the precomputed schedule. In this latter case, the runtime has to start a new learning run. As Tern requires applications to be analyzed before running them, it is not a practical alternative for ROMAIN because that would require adding a complex input classification and schedule prediction engine to the master process.

[12] Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable Deterministic Multithreading Through Schedule Memoization. In *Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–13, Vancouver, BC, Canada, 2010. USENIX Association

Similar to Tern, Storyboard enforces deterministic replication by relying on application-specific input knowledge to force applications to take precomputed schedules.[13] EVE batches inputs into groups that are likely to have no data conflicts. If this is the case, concurrent processing of these inputs is likely to have no non-deterministic effects.[14] Rex avoids an expensive training phase by using a leader/follower scheme. Leader replicas execute, log their non-deterministic decisions, and finally validate that they reached the same result. Follower replicas then consume the logged values and replay the logged decisions.[15]

As ROMAIN replicates operating system processes, batching their system calls and distributing them deterministically is unfortunately not an option. We will however see on page 78 that the idea of leader/follower determinism can be applied to multithreaded replication as well.

[13] Rüdiger Kapitza, Matthias Schunter, Christian Cachin, Klaus Stengel, and Tobias Distler. Storyboard: Optimistic Deterministic Multithreading. In *Workshop on Hot Topics in System Dependability*, HotDep'10, pages 1–8, Vancouver, BC, Canada, 2010. USENIX Association

[14] M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. EVE: Execute-Verify Replication for Multi-Core Servers. In *Symposium on Opearting Systems Design & Implementation*, OSDI'12, Oct 2012

[15] Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the Speed of Multi-core. In *European Conference on Computer Systems*, EuroSys '14, Amsterdam, The Netherlands, 2014. ACM

Deterministic Memory Consistency Models Memory models at both the hardware and the programming language level describe how modifications to data in memory become visible to the rest of the system. Based on these models we can reason about whether programs will expose deterministic behavior.

Lamport defined *sequential consistency* as a parallel execution that orders memory writes as if they were executed by one arbitrary interleaving of sequential threads on a single processor.[16] Most importantly, all threads observe the same interleaving. Note that sequential consistency does not provide freedom from data races — it simply provides a framework to reason about the existence of these problems.

[16] Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979

Other researchers recognized that sequential consistency is too strict as it forbids compiler-level or hardware-level optimizations, which would improve the performance of concurrent execution. Alternatives — such as *release consistency* — therefore weaken consistency rules to allow for optimizations.[17] Release consistency diverges from the need for a common shared view of memory and supports arbitrary reordering of memory accesses. However, the model requires all accesses to globally shared objects to be protected by a pair of `acquire` and `release` operations. In combination, reordering can be used

[17] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture*, ISCA '90, pages 15–26, Seattle, Washington, USA, 1990. ACM

to improve concurrent performance, while the rules about protecting global objects still allow to reason about the order of object updates and hence about the existence of data races.

Aviram and Ford argued that in order to reduce implementation complexity, we not only need a memory model to reason about races, but one that enforces determinism. They proposed a model that ensures completely deterministic execution and called it *workspace consistency*.[18] Instead of acquiring access to each global object, workspace consistency requires threads to work on dedicated copies of these objects. Threads obtain copies using a `fork` operation and merge this copy back into the global program state using a `join` call. This approach is similar to release consistency. However, instead of preventing concurrent modification, workspace consistency lets each thread modify its local object copy. The consistency model furthermore defines rules about the order in which updated copies are merged back into the global application view. Thereby any potential data races that exist between threads are resolved in a deterministic order. As a result, the whole program becomes deterministic.

[18] Amittai Aviram, Bryan Ford, and Yu Zhang. Workspace Consistency: A Programming Model for Shared Memory Parallelism. In *Workshop on Determinism and Correctness in Parallel Programming*, WoDet'11, Newport Beach, CA, 2011

Aviram first implemented workspace consistency in the Determinator operating system.[19] This approach requires all programs to be rewritten for a new system and is therefore impractical if we want to reuse the large quantities of existing applications on existing operating systems. The authors later demonstrated that the idea of workspace consistency can also be retrofitted into existing parallel programming frameworks. For that purpose they implemented a deterministic version of OpenMP and showed that most state-of-the-art parallel benchmarks can be adapted to use workspace consistency.[20] Merrifield later added workspace-consistent memory management to Linux and showed that many concurrent applications – including deterministic threading systems, shared-memory data structures, and garbage collectors – can benefit from such management mechanisms being present in the OS.[21]

[19] Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient System-enforced Deterministic Parallelism. pages 193–206, Vancouver, BC, Canada, 2010. USENIX Association

[20] Amittai Aviram and Bryan Ford. Deterministic OpenMP for Race-Free Parallelism. In *Conference on Hot Topics in Parallelism*, HotPar'11, Berkeley, CA, 2011. USENIX Association

[21] Timothy Merrifield and Jakob Eriksson. Conversion: Multi-Version Concurrency Control for Main Memory Segments. In *European Conference on Computer Systems*, EuroSys '13, pages 127–139, Prague, Czech Republic, 2013. ACM

Programs using workspace consistency are automatically deterministic. As with language-level determinism, replicating such applications does not require additional support from ROMAIN and works out of the box. Also similar to language-level approaches we can however not assume that all applications are implemented deterministically. Hence, deterministic memory consistency provides no silver bullet for replication.

Deterministic Runtimes In addition to developing new deterministic programming methods, researchers proposed ways to retrofit existing systems with determinism. Bergan's CoreDet splits multithreaded execution into parallel and serial phases and dynamically assigns each memory segment an owner.[22] In the parallel phase threads are only allowed to modify memory regions they own privately. Once a thread accesses a shared variable it is blocked until the serial phase. Serial execution is started after a preset amount of time. Here, threads perform their accesses to shared state in a deterministic order. CoreDet relies on compiler-generated hints to track memory ownership and provides a runtime that periodically switches between parallel and serial thread execution.

[22] Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 53–64, Pittsburgh, Pennsylvania, USA, 2010. ACM

A large quantity of multithreaded applications already exists and rewriting or recompiling them to become deterministic is often infeasible. As mentioned previously, while these applications may use one of many different parallel programming paradigms, these paradigms in the end map to a low-level thread library, such as `libpthread`.

Most of today's applications are linked dynamically.[23] These programs do not provide their own version of commonly used libraries – such as `libC`, `libpthread` and `libX11` – but use a library version globally provided by the underlying system. While this concept was originally introduced to reduce binary program sizes and save system memory, it also allows to transparently replace a system's implementation of a library. Deterministic versions of `libpthread` have been proposed as a drop-in replacement using this approach.

[23] John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999

Strongly deterministic libraries — such as DTHREADS[24] and Grace[25] — provide fully deterministic ordering of every memory access. Both approaches do so by emulating workspace consistency: each thread runs in a dedicated address space and works on dedicated copies of data. When threads reach predefined synchronization points — such as well-known `libpthread` functions — their changes are merged back into the main address space deterministically.

[24] Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient Deterministic Multithreading. In *Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, Cascais, Portugal, 2011. ACM

[25] Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe Multithreaded Programming for C/C++. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 81–96, Orlando, Florida, USA, 2009. ACM

Spawning per-thread address spaces and merging data back and forth does not come for free. DTHREADS' authors report a slowdown of up to a 4 times when comparing DTHREADS applications to their native `libpthread` versions. Parrot[26] reduces DTHREADS's overhead using developer hints. These hints allow the programmer to specify concurrent regions and performance-critical non-deterministic sections within their application. This approach is unfortunately no option for ROMAIN because it a) requires modifications to the application and b) forgoes determinism to improve performance, which is not a viable alternative for replicated execution.

[26] Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *ACM Symposium on Operating Systems Principles*, SOSP'13, pages 388–405, Farminton, Pennsylvania, 2013. ACM

Olszewski's Kendo[27] and Basile's LSA algorithm[28] observe that as long as a multithreaded application is race-free and protects all accesses to shared data with locks, we do not need to enforce deterministic ordering of every memory access. Instead, it suffices to ensure that all lock acquisition and release operations are performed in a deterministic order. Their *weakly deterministic* libraries implement such behavior by intercepting `libpthread`'s `mutex_lock` and `mutex_unlock` operations.

[27] Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 97–108, Washington, DC, USA, 2009. ACM

[28] Claudio Basile, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Active Replication of Multithreaded Applications. *Transactions on Parallel Distributed Systems*, 17(5):448–465, May 2006

Weak determinism provides lower execution time overheads than strongly deterministic methods. Kendo's authors report less than 20% execution time overhead compared to native `libpthread`. As a downside, their approach requires applications to be race-free and therefore limits its applicability.

Deterministic Multithreading for Replicating Multithreaded Applications
While many of the previously discussed solutions use replication as one example to motivate their work, only few researchers showed that their approach actually works for this purpose.

Bergan implemented dOS, an operating system modification in Linux that adds CoreDet deterministic management to a group of processes and thereby makes this subset of Linux applications deterministic.[29] Using this solution, the authors were able to replicate a multithreaded web server. However, their

[29] Tom Bergan, Nicholas Hunt, Luis Ceze, and Steve Gribble. Deterministic Process Groups in dOS. In *Symposium on Operating Systems Design & Implementation*, OSDI'10, pages 177–192, Vancouver, BC, Canada, 2010. USENIX Association

solution relies on a significant modification to the Linux kernel. Their patch adds and modifies more than 8,000 lines of code in the kernel and touches all major subsystems, such as memory management, scheduling, file systems, and networking.

Mushtaq implemented a deterministic replicated thread library on Linux that requires only minor kernel modifications.[30] His solution uses a modified `libpthread`. A leader process executes `libpthread` calls and logs their order and results into a shared memory area. A second follower process executes behind the leader and reads the leader's log data. The follower uses this log to detect errors by comparing his results to the logged ones. Furthermore, the follower uses the log to assign locks to his threads in the same order as the leader process.

[30] Hamid Mushtaq, Zaid Al-Ars, and Koen L. M. Bertels. Efficient Software Based Fault Tolerance Approach on Multicore Platforms. In *Design, Automation & Test in Europe Conference*, Grenoble, France, March 2013

Mushtaq's work is attractive because it works solely in user space[31] and he reports low overheads for replicated execution. Upon detecting an error, Mushtaq rolls back to a previous checkpoint. Similar to other Linux checkpointing solutions,[32] he creates a lightweight checkpoint using the `fork()` system call: `fork()` creates a copy-on-write copy of the calling process. Parent and child thereby share all memory until the parent starts modifying pages, which are then dynamically copied by the OS kernel. In the checkpointing scenario, the child process never runs. It is solely used to store the memory contents of its parent. If the system detects an error, rollback is achieved by killing the erroneous parent and continuing execution in its child.

[31] Their only addition to Linux is a non-POSIX-compliant multithreaded `fork` system call.

[32] Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 77–90, Pittsburgh, Pennsylvania, USA, 2010. ACM

Checkpointing using `fork()` works well for restarting after a software error, but is seriously flawed if we want to tolerate hardware errors: Due to the copy-on-write nature of a `fork`ed process, the original process and its checkpoint will share any data that is read-only. If this data is affected by a fault in the memory hardware, the data will be modified but no copy will be created. Hence, if this fault leads to a failure in the protected process, rolling back to the previous checkpoint will not fix the problem but instead re-execute from a corrupted checkpoint. Using ECC-protected memory would help to detect such corruption, but Mushtaq discusses neither the problem nor the solution in his paper.

SUMMARY: To replicate multithreaded applications we need to make their execution deterministic. Deterministic multithreading methods provide this feature and implementations exist at the level of programming languages, runtime environments, and operating systems.

Modifying the system's `libpthread` thread library seems to most generic option to make replication deterministic, because these mechanisms apply to a wide range of programs and do not rely on the program or the underlying OS to be modified in any way.

4.3 Replication Using Lock-Based Determinism

I will now describe how I extended ROMAIN to support multithreaded applications. Starting from the status presented in the previous chapter, I will first explain how concurrent threads generating externalization events are handled by the master process. Thereafter I will describe how multithreaded replicas are made deterministic in order to avoid the problems introduced in the previous section.

4.3.1 An Execution Model for Multithreaded Replication

I extended ROMAIN's replication model with another abstraction to facilitate multithreaded replication. Figure 4.5 illustrates the resulting terminology. As before, ROMAIN runs multiple *replicas* (Replica 1, Replica 2) of an application. These replicas constitute isolated address spaces and serve as resource containers. Similar to Kendo, which I introduced on page 77, each replica of a multithreaded application runs all its *replica threads* concurrently within the replica address space. In the figure we see three such threads in every replica. To distinguish threads across replicas I will from now on denote $T_{i,j}$ as the j-th thread in the i-th replica.

The master process intercepts and handles replica threads' externalization events. In Section 3.3 I described how the master waits for all replica threads to reach their next externalization event, compares their results, and then handles the respective event. This is insufficient for multithreaded replication, because we now no longer need to wait for *all* replica threads, but only for all replicas of *the same* replicated application thread. For instance, if all else is identical, then thread $T_{1,1}$ in Replica 1 and $T_{2,1}$ in Replica 2 will execute the same code and their externalization events need to be compared for error detection. I refer to the set of threads that have the same local thread ID and execute identical code within different replicas as a *thread group*.

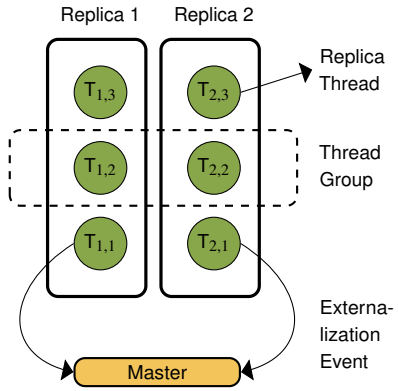

Figure 4.5: Terminology overview

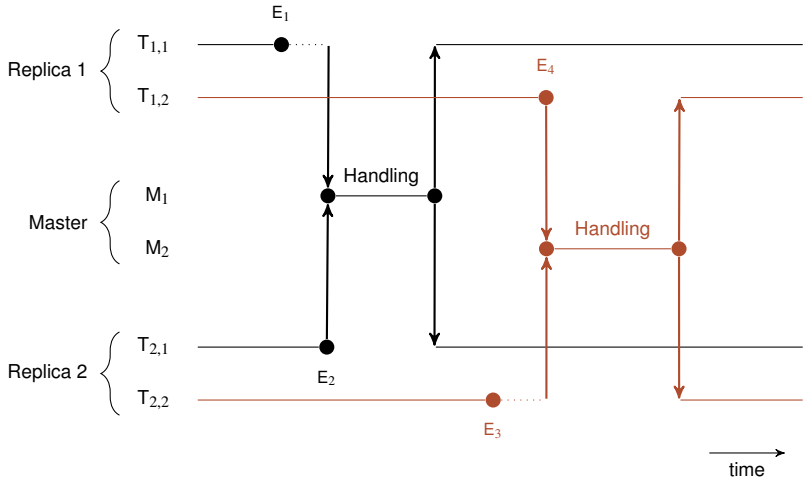

Figure 4.6: Multithreaded event handling in ROMAIN

Figure 4.6 shows how thread groups are handled by the master. We see two replicas running two threads each. The thread pairs ($T_{1,1}$, $T_{2,1}$) and ($T_{1,2}$, $T_{2,2}$) form two thread groups. Once $T_{1,1}$ raises an externalization event E_1 it gets blocked until the other thread in its thread group, $T_{2.1}$, also reaches its next externalization event (E_2). The events are reflected to the master for

event handling in thread M_1, which resumes execution of threads $T_{1,1}$ and $T_{2,1}$ once handling is finished. The second thread group executes independently from the first one. While $T_{1,1}$ and $T_{2,1}$ are handled by the master, $T_{1,2}$ and $T_{2,2}$ continue execution until they hit their next externalization events E_3 and E_4, which are then handled by the master in thread M_2.

Besides distinguishing events by their thread groups, ROMAIN handles system calls and manages resources in the same way as I described in Chapter 3. However, in order to ensure that thread groups always behave identically, we need to resolve non-deterministic behavior in our replicas as discussed in the previous section.

4.3.2 Options for Deterministic Multithreading in ROMAIN

As stated in Chapter 2, one of my design goals for ASTEROID is to protect unmodified binary applications against the effects of hardware faults. When considering multithreaded replication, this requirement rules out any solutions that demand applications to be rewritten using DMT mechanisms or recompiled using a DMT-aware compiler. Based on this requirement and the review of DMT techniques in the previous section, two options remain for protecting multithreaded applications: I can make applications deterministic by either modifying the whole system or by implementing a deterministic multithreading library.

Determinism at the System or Library Level? To provide *system-level determinism* I need to adapt the FIASCO.OC kernel, the L4 Runtime environment, as well as important libraries — such as libC — to guarantee deterministic execution for multithreaded applications. This approach mirrors CoreDet and dOS, which I introduced on page 76 and has been shown to induce low execution time overheads. However, this approach affects all applications running on FIASCO.OC regardless of whether they are protected by ROMAIN. As a consequence, applications that protect themselves against hardware faults suffer from overheads related to deterministic execution even if they are not affected by non-determinism at all.

Library-level determinism uses a modified version of the `libpthread` library to implement deterministic multithreading on a per-application basis. As explained previously, the dynamic nature of this library allows us to replace its implementation without modifying application code. Furthermore, a system can provide different implementations of this library: When ROMAIN loads a replicated application it can dynamically load a deterministic version of `libpthread`. In contrast, L4Re's system-wide application loader may still use an unmodified, non-deterministic `libpthread` for applications that should not suffer from determinism-related overheads.

SUMMARY: I decided to implement multithreaded replication using library-level determinism as this approach provides more flexibility to the system's users and also promises to be less complex to implement than modifying FIASCO.OC and L4Re.

Weak or Strong Determinism? In the previous section I distinguished between strongly and weakly deterministic thread libraries. Weak determinism — such as Kendo — requires the application to be race-free and protect all shared data accesses using synchronization operations. This approach promises low execution time overheads and low resource requirements: all threads run within the same address space and share global resources.

Strong determinism — as implemented by DTHREADS — provides determinism even in the presence of data races. This benefit is paid for with higher runtime overheads and resource requirements: by running all threads in individual address spaces and implementing workspace consistency, DTHREADS needs a copy of all globally shared memory for every thread. In the worst case this means that a deterministic application with N threads requires N times the amount of memory of the original application.[33]

[33] DTHREADS reduces this worst case by not copying thread-private memory.

As described in Section 3.5.2, ROMAIN maintains a copy of each memory region for every replica it runs. If we replicated a DTHREADS application with N threads using M replicas, this means we need $M \times N$ times the amount of memory of the unreplicated and non-deterministic version. This requirement leads to practical limitations: I developed ROMAIN for the x86/32 architecture where every application can address 3 GiB of memory in user space.[34] The ROMAIN master is the pager for all replicas and hence needs to service their page faults from his private 3 GiB of user memory.

[34] 1 GiB is reserved for the FIASCO.OC kernel. Linux and Windows do the same.

Let us now assume we replicate a multithreaded application in triple-modular redundant mode. Memory replication demands that this application can use at most 1 GiB per replica lest the ROMAIN master cannot serve all replica page faults. If we now assume our application to run 4 threads, this 1 GiB includes workspace-consistent copies of memory regions for each thread. In the worst case this means that our application can only use 256 MiB of distinct memory for computation purposes.

To reduce replication overhead in terms of execution time and resource usage I therefore decided to implement a weakly deterministic thread library for replicating multithreaded applications. I am going to describe this weakly deterministic library and its integration into ROMAIN in the upcoming sections. In Section 4.3.7 I will outline how a strongly deterministic solution would differ from the design presented in this section.

4.3.3 Enforced Determinism

Ensuring weakly deterministic multithreaded execution requires that ROMAIN replicas reach an agreement on the order in which threads acquire locks. In my first approach to implement this agreement, I let the master process decide on lock ordering. I call this approach *enforced determinism* because an external instance imposes ordering on the otherwise non-deterministic replicas.

Adapting the Thread Library I implemented enforced determinism with an adapted `libpthread` library that transforms synchronization operations into externalization events visible to the ROMAIN master. For this purpose I analyzed the synchronization operations in L4Re's `libpthread` library, which is derived from μClibC.[35]

[35] `http://www.uclibc.org/`

Four functions within μClibC need to be adapted in order to enforce deterministic ordering of synchronization events:

1. `pthread_mutex_lock()`,
2. `pthread_mutex_unlock()`,
3. `__mutex_lock()`, and
4. `__mutex_unlock()`.

The former two functions implement `libpthread`'s mutex synchronization primitive. The latter two functions are used for synchronizing concurrent accesses to data structures internal to `libpthread`. If we manage to ensure proper ordering for calls to these functions across all replicas, we will also achieve deterministic ordering of higher-level synchronization primitives – such as barriers, condition variables, and semaphores – because `libpthread` internally implements these primitives using the above four operations.

I adapted L4Re's `libpthread` implementation and replaced the entry points to the four synchronization functions with an `INT3` instruction as shown in Listing 4.7. This single-byte x86 instruction raises a debug trap when it is executed. Thereby, whenever a program calls into one of the synchronization functions, the program will raise a debug exception, which gets reflected to the ROMAIN master process.

```
1  int
2  attribute_hidden
3  __pthread_mutex_lock
4     (pthread_mutex_t * mutex)
5  {
6    asm volatile ("int3");
7
8    /* rest of code omitted
9     * [..]
10    */
11 }
```

Listing 4.7: Introducing debug exceptions into `libpthread` mutex operations

Lock Event Handling in the Master To enforce deterministic replica operation, the master process needs to implement ordering while handling the debug exceptions raised by lock operations. For this purpose I added a new event observer to ROMAIN's chain of event handlers, the *LockObserver*.

The LockObserver handles all exceptions related to lock operations by intercepting these events through ROMAIN's event handling mechanism. The observer mirrors each `libpthread` mutex M that exist in the replicas with a dedicated mutex M' within the context of the master process. The LockObserver handles synchronization events in three steps:

1. *Inspect parameters:* We inspect the faulting replica's stack to determine the current function calls' parameters.[36] Thereby we obtain the mutex ID M that is currently being used as well as the return address at which the replicas shall resume execution after the lock operation.

   The LockObserver uses a hash table to map mutex M to the corresponding master mutex M'. If no such entry exists in the hash table, a new mutex M' is allocated and initialized.

2. *Carry out lock operation:* The replica's synchronization operation is performed by the master using the master mutex M'. All thread groups that perform lock operations will go through the LockObserver's event handler. If these thread groups use the same replica mutex M, they will at this point carry out an operation on the same master mutex M'. Thereby, the LockObserver achieves synchronization between concurrently executing thread groups identical to the synchronization a native mutex operation achieves in the context of a single application instance.

3. *Adjust replica state:* Once the master synchronization operation returns, we know that we can also return in the replica context. To do so, the

[36] Remember, the master has full access to all replica memory as it also functions as the replicas' memory manager.

LockObserver adjusts the thread group's states to emulate a return from the lock function. This means setting the replica threads' instruction pointers to the previously determined return address and setting the return value (EAX on x86_32) and stack pointer registers appropriately.

The LockObserver mechanism provides deterministic lock acquisition across replicas, because all synchronization operations are serialized and ordered within the master's handler function. In contrast to other DMT techniques, multiple runs of the same application will not necessarily produce identical behavior. The LockObserver still works for replication purposes, because we are only interested in deterministic ordering between the replicas in a single application run.

4.3.4 *Understanding the Cost of Enforced Determinism: Worst Case Experiments*

I implemented a microbenchmark to evaluate the worst case overhead induced by using enforced determinism. Two concurrent threads execute the code shown in Listing 4.8, so that each thread increments a global counter variable 5,000,000 times. For each increment a global mutex `mtx` is acquired and released. The benchmark therefore spends most of its time within `libpthread`'s synchronization operations and will suffer most from any slowdown induced by a DMT mechanism.

```
1  int counter = 0;
2  const int increments = 5000000;
3  pthread_mutex_t mtx = PTHREAD_MUTEX_INITIALIZER;
4
5  void thread()
6  {
7     for (unsigned i = 0; i < increments; ++i) {
8        pthread_mutex_lock(&mtx);
9        counter++;
10       pthread_mutex_unlock(&mtx);
11    }
12 }
```

Listing 4.8: Thread microbenchmark

Similar to the microbenchmarks in Chapter 3, I executed this benchmark using ROMAIN with one, two, and three replicas and compared their execution times to native execution. For these experiments I used a system with 12 physical Intel Xeon 5650 CPU cores running at 2.67 GHz and distributed across two sockets. Each replica thread as well as each native thread were pinned to a dedicated physical CPU to maximize concurrency.

Figure 4.9 shows the measured execution times for the benchmark. The results represent the average over five benchmark runs in each setup. The runs' standard deviation was below 0.1% in all cases and is therefore not shown in the figure. We see that the pure overhead of intercepting all lock and unlock operations already slows down the benchmark by a factor of 121. Double and triple-modular redundancy increase this cost even more with a maximum slowdown of 309 for TMR. These high overheads demand a more thorough investigation to find their sources.

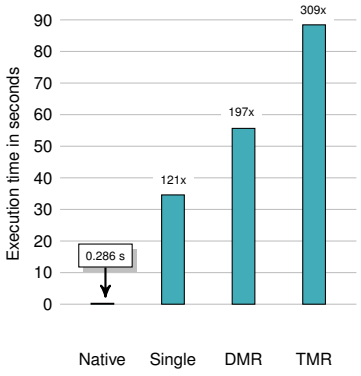

Figure 4.9: Execution times measured for the multithreading microbenchmark

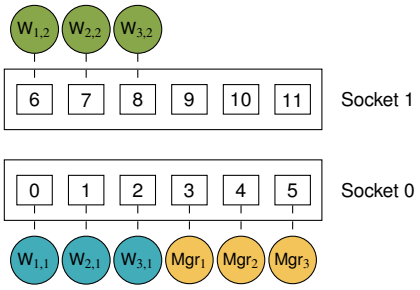

Figure 4.10: Sequential assignment of the benchmark threads to CPU cores on the test machine

Adjusting CPU Placement While the benchmark is designed to run two worker threads, a closer inspection of `libpthread` showed that in addition to these worker threads, a third manager thread is launched. `libpthread` uses this thread to internally distribute signals, launch new threads, and perform cleanups once threads are torn down. The manager thread is launched lazily once an application becomes multithreaded, e.g., when it first calls `pthread_create()`. As a result, the startup order of these threads is Worker 1 – Manager – Worker 2.

As mentioned before, ROMAIN assigns replica threads to dedicated physical CPUs. My naive implementation of this assignment was to distribute threads sequentially across all CPUs starting at CPU 0. As my test machine has two CPU sockets with six cores each, the replicas in a TMR setup will be distributed as shown in Figure 4.10.

Unfortunately, this setup assigns replicas of the two heavily synchronizing worker threads to different processor sockets. Every synchronization operation requires messages to be sent between the sockets. While L4Re implements such messaging using FIASCO.OC's IPC primitives, these will eventually require Inter Processor Interrupts (IPIs) to be sent between CPUs.

Using a microbenchmark I found that sending messages between cores on the same socket requires about 8,500 CPU cycles, whereas sending messages between cores on different sockets costs about 19,500 CPU cycles.[37] Additionally, the manager thread does not perform any real work in the benchmark, so distributing these replica threads across CPUs does not gain any performance and only wastes resources.

To optimize for low IPI latencies I manually adapted the CPU placement algorithm. The idle replicas of the manager thread are co-located with the first worker's thread group, making room to place the second worker thread group on CPUs 3–5. As Figure 4.11 illustrates, all replica threads thereby run on a single socket and therefore benefit from reduced IPI latencies for synchronization purposes.

[37] FIASCO.OC provides a `pingpong` benchmark suite to evaluate the cost of kernel operations and hardware features.

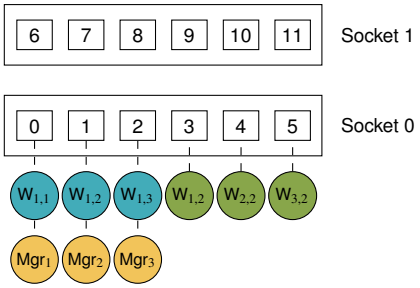

Figure 4.11: Assignment of the benchmark threads to CPU cores on the test machine, optimized to minimize synchronization cost.

Reducing Synchronization Cost In a next step I instrumented the ROMAIN master to determine where the remaining overhead comes from. I separated replica execution into four different phases, which are depicted in Figure 4.12. These phases distinguish between *active* and *passive* replicas: One active replica enters the master, performs state validation and potential event handling. The remaining replicas are passive. They wait for the active replica to finish its processing and then resume execution based on its results.

1. **User time** (■) measures the time spent executing outside the master process. This includes both actual application time as well as the time spent in the FIASCO.OC kernel for delivering vCPU exceptions. Note that this time does not include time spent in actual system calls, because those are handled within the master process.

2. **Pre-Synchronization** (■) measures the time between entering the master process and executing event handling. For passive replicas this is equivalent to the time waiting for all other replicas to enter the master. For active replicas, this includes state validation time as well as the management overhead for processing the list of event observers.

3. **Observer time (■)** measures the time an active replica spends in one of ROMAIN's event observers for handling replica events. This time also includes time the master spends in system calls on behalf of the replica as described in Section 3.4 on page 45.
4. **Post-Synchronization time (■)** tracks the time spent between event handling and resuming replica execution. For the active replica this includes time for storing the event handling result and waking up all passive replicas. For passive replicas this is the time to obtain the leader's state and resume execution.

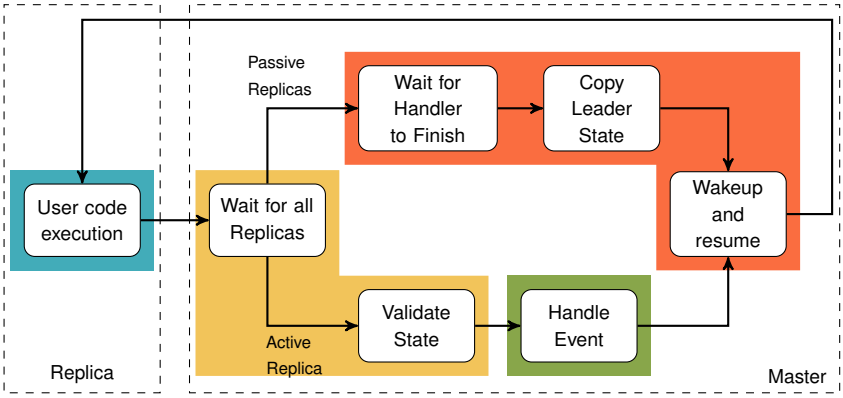

Figure 4.12: Execution phases of a single replica

I re-executed the previous microbenchmark with the CPU placement optimization turned on and measured the fraction of time replicas spend in each of the four phases and show the results in Figure 4.13. The `libpthread` manager thread that runs in every application spends 100% of its time in event handling, because it is indefinitely waiting for an incoming management request. It is not shown in the figure. The bars show the average time each worker thread replica spends in one of the four phases. I always show the distribution for the first replica of the first worker thread. All other replicas have similar phase distributions with standard deviations below 1%.

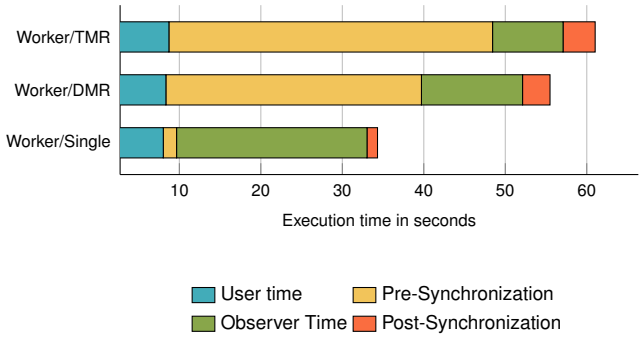

Figure 4.13: Time the microbenchmark threads spend in the execution phases shown in Figure 4.12 when running with optimized CPU placement.

First of all we see that the workers spend about 8 seconds of the benchmark in the actual user code and that this value does not change when increasing the number of replicas. Given the test machine's clock speed of 2.67 GHz and the fact that we execute 10 million lock and unlock operations in each thread, this number maps to an average user time of around 2,000 CPU cycles per lock operation. I confirmed with a microbenchmark that this is roughly equivalent

to the cost of delivering a FIASCO.OC vCPU exception and resuming the vCPU afterwards. This shows that user time is dominated by vCPU exception delivery.

Second, the results show that in the single-replica case we spend most of the master execution time (87.4%) in the master's event handler. This time decreases for double and triple modular redundancy: Here the replicas spend most of their time (73.3% for DMR, 82.8% for TMR) in the pre- and post-synchronization phases.

I had a closer look at where synchronization overhead comes from and found two sources of overhead. First, replicas within a thread group need to wait for each other whenever they enter the master. This is a problem for exception-heavy benchmarks as the multithreaded one: When an exception is handled, the active replica wakes up the passive ones and performs cleanup work. In the meantime the passive replicas resume to user space and immediately raise their next exception. Here they have to wait for the active replica to catch up. This overhead reason is inherent to replicated execution and fortunately only a problem for such microbenchmarks.

As a second reason I found that in my initial implementation, the master used a `libpthread` condition variable at the beginning of event handling to wait for incoming replicas and at the end of the event handling phase to wait for all replicas to leave the master. Each of these synchronization operations requires expensive message-based notifications.

I implemented an optimization for the synchronization phase that replaces the condition variables with globally shared synchronization variables and let the replicas poll on this variable instead of using message-based synchronization. This optimization avoids synchronization IPIs between replicas and I call it *fast synchronization*.

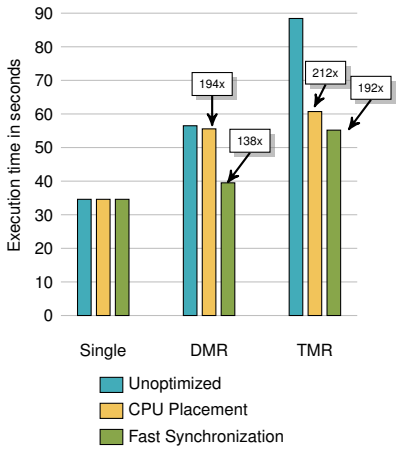

Figure 4.14: Execution times measured for the multithreading microbenchmark when run with optimizations turned on. Unoptimized data from Figure 4.9 plotted for reference.

Improved Synchronization Overhead I compare the effects of the two optimizations to the previously presented microbenchmark results in Figure 4.14. We see that TMR execution benefits most from the CPU placement optimization, whereas DMR shows nearly no effect because the DMR replicas were already placed on a single socket in the first place. In turn, DMR shows a better improvement from the fast synchronization optimization than TMR. This is due to the fact that in the TMR case more overhead is spent waiting for other replicas to catch up whereas in the DMR case more time is spent sending synchronization messages.

Figure 4.15 breaks down the fully optimized version (CPU placement and fast synchronization) of the benchmark into user, synchronization and event handling times. Compared to the previous results shown in Figure 4.13 we see a decrease in both the pre and post synchronization phases for DMR and TMR execution.

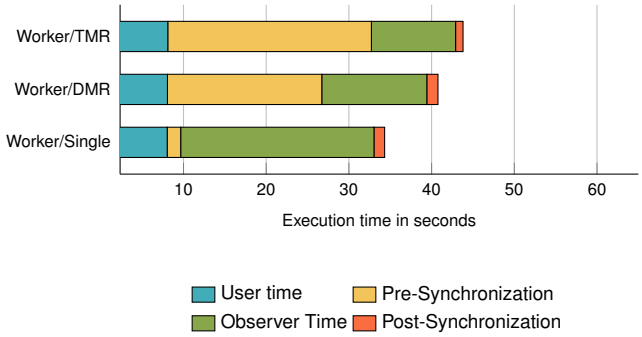

Figure 4.15: Breakdown of benchmark overhead sources with CPU placement and fast synchronization optimizations turned on.

SUMMARY: I presented enforced determinism, a mechanism that transforms lock operations into exceptions visible to the ROMAIN master process. The master handles these exceptions and thereby establishes lock ordering. In turn, multithreaded replicas behave deterministically.

I used a microbenchmark to analyze replication overhead and found that the placement of replicas on different CPUs influences overhead. I furthermore pinpointed inter-replica synchronization during event handling as a second source of overhead. I devised optimizations to address both of these issues in order to reduce replication overhead.

4.3.5 Cooperative Determinism

We saw in the previous section that enforced determinism has a high worst-case overhead. Even after optimizing, TMR execution is slowed down by two orders of magnitude. We also saw that there are three main contributors to overhead:

1. **Mirroring of data structures and work in the master:** All lock operations in the replicas are mirrored with additional data structures and operations in the ROMAIN master process. Furthermore, every lock operation has to go through the master's event handling mechanism.

2. **CPU traps per lock operation:** Every lock operation leads to a CPU trap. This adds a constant cost of about 2,000 CPU cycles to each operation, whereas a normal lock function call would require less than 100 CPU cycles if the lock is uncontended.

3. **Replica synchronization:** Waiting for all replicas to reach their next lock operation is the biggest contributor to replication overhead. This synchronization is often unnecessary: If a replica thread wants to acquire a lock and the other threads in the same thread group simply lag behind this first thread, there is actually no need to wait for them. The first thread can optimistically continue and trust the others to make the same decision afterwards.

   This optimistic locking solution does not impact reliability, because ROMAIN will still compare replica states at externalization events. This

approach only reduces the number of such events in order to save execution time overhead. Even optimistic locking however needs to properly synchronize lock operations whenever more than one thread group tries to acquire a lock.

To address these issues, I designed a replication aware `libpthread` library, which I call `libpthread_rep`. Replicas no longer reflect their lock operations to the master process but instead cooperate internally to achieve deterministic lock ordering. This solution – which I call *cooperative determinism* – avoids mirroring lock operations in the master (problem #1), eliminates the need for CPU traps in lock operations (problem #2), and reduces the amount of inter-replica synchronization to those cases where it is really necessary (problem #3).

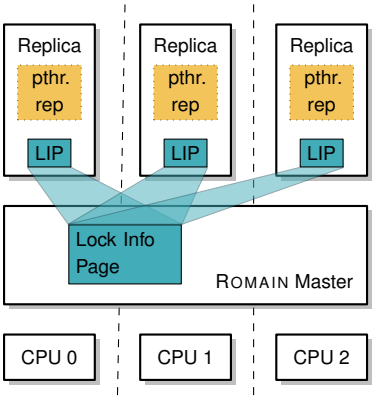

Figure 4.16: Cooperative Determinism: Applications are linked with a replication-aware thread library. This library uses a lock info page shared among all replicas to establish lock ordering.

Architecture for Cooperative Determinism ROMAIN provides an infrastructure for cooperative determinism using two building blocks shown in Figure 4.16. First, replicated applications use `libpthread_rep` as a drop-in replacement for `libpthread`. Second, `libpthread_rep` establishes ordering of lock operations using a *lock info page (LIP)* that is shared among all replicas. The master process is only involved in this architecture in the setup phase: it loads `libpthread_rep` into the replicas, creates the LIP and makes sure that this LIP is mapped into each replica's address space at a pre-defined address.

The Lock Info Page `libpthread_rep` uses the LIP to share information about the state of lock acquisitions between replicas without requiring expensive synchronization messages. The LIP contains information about the number of replicas as well as information about all locks that are used by the replicated application. Listing 4.17 shows the LIP data structure. In my current prototype, the LIP is dimensioned to support 2,048 unique locks. This number suffices for the benchmarks I present in this thesis and may be adapted at compile time if necessary.

```
1 struct LIP {
2    unsigned num_replicas;
3    struct {
4       unsigned spinlock;
5       Address  owner;
6       unsigned acq_count;
7       unsigned epoch;
8    } locks[MAX_LOCKS];
9 };
```

Listing 4.17: Lock Info Page data structure

For each lock in the application, the LIP stores a `spinlock` field that protects access to the LIP's lock information from concurrent access by the replicas. The `owner` field keeps track of the ID of the thread that currently possesses the lock. `acq_count` is used internally by `libpthread_rep` to count the number of replica threads that already entered a critical section. Finally, the `epoch` field serves as a logical progress indicator for threads and is incremented whenever a thread calls a lock or unlock function.

Using the LIP to Enforce Lock Ordering As with enforced determinism (explained in Section 4.3.3) `libpthread_rep` adjusts four functions from `libpthread`: `pthread_mutex_lock()`, `pthread_mutex_unlock()`, `__mutex_lock()`, and `__mutex_unlock()`. I modified both of the lock functions to call `lock_rep()`, which I show as a control flow graph in Figure 4.18 on the facing page.

When a thread tries to acquire a lock, it consults the shared LIP to inspect the lock's global state. If the lock is currently marked as free, the thread stores its own thread ID and epoch counter in the global state and continues. At this

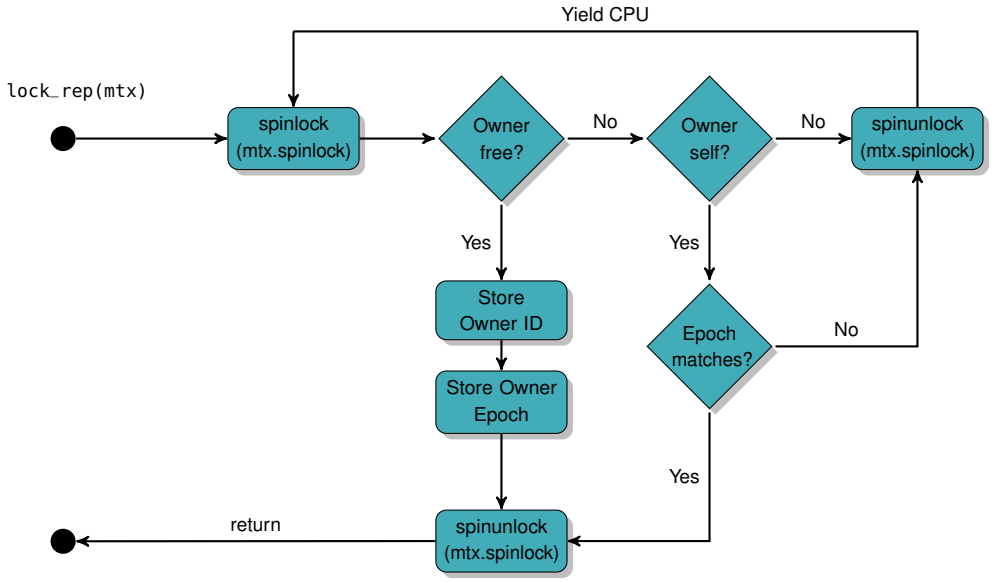

Figure 4.18: `lock_rep()`: a replication-aware lock function

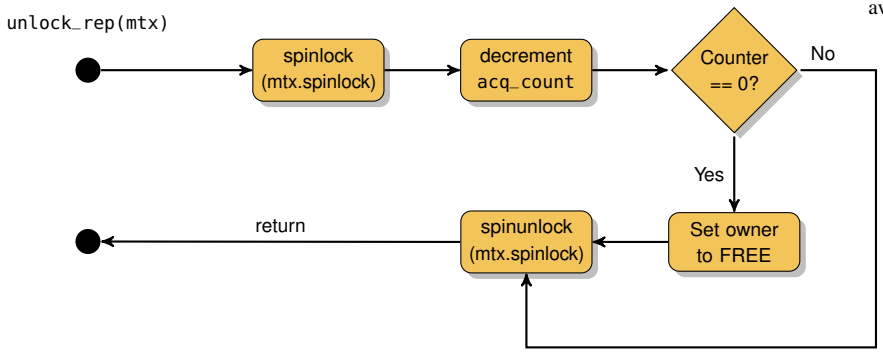

Figure 4.19: `unlock_rep()`: a replication-aware unlock function

point the thread has acquired the lock and can continue its operation without having to wait for the other threads in its thread group.

If the acquiring thread finds the lock to be taken, it checks the owner's thread ID. If the owner ID matches the calling thread's ID, this means that another thread from the same thread group already acquired the lock and the calling thread can continue operation. However, if the owner ID does not match the calling thread, we found a situation where a different thread group owns the lock. This means, the calling thread has to wait until the lock either becomes free or changes ownership to the caller's ID.

Inconsistencies may arise if a thread releases a lock and then tries to reacquire it before all other threads of the thread group released the lock. The epoch counter is used to detect such a situation and prevent this thread from overtaking the rest of its thread group.

Figure 4.19 shows a flow chart for the unlock operation. The `acq_count` counter tracks how many threads of a thread group need to release the lock before a new owner can be established. Each thread releasing a lock decrements this counter. Only the last thread to leave a critical section has to additionally reset the `owner` field.

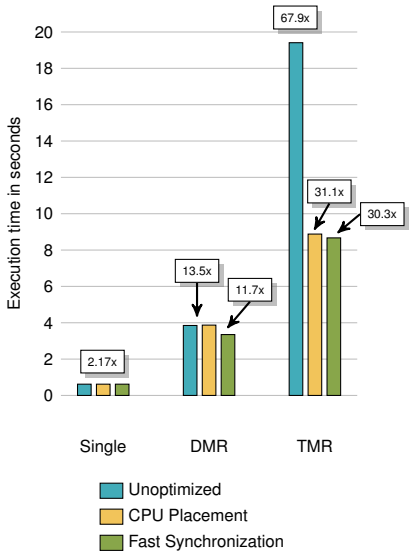

Figure 4.20: Execution times measured for the multithreading microbenchmark running with different optimization levels and cooperative determinism

Cooperative Determinism Runtime Overhead To evaluate the efficiency of cooperative determinism, I repeated the worst-case microbenchmark that I used to evaluate enforced determinism in Section 4.3.4 on page 83. Again, I ran this microbenchmark (native execution time: 0.286 s) with no optimizations, with optimized CPU placement, as well as with fast synchronization and show the results in Figure 4.20. At first glance we see that the worst case overhead for the optimized version of cooperative determinism is about six times lower than for the similarly optimized version of enforced determinism (TMR: 30.3x vs. 192x).

Once again, triple-modular redundant execution benefits significantly from optimizing replica placement. Even though cooperatively deterministic replicas do not use explicit synchronization upon every lock operation, the replicas still use the shared-memory LIP, which needs to be kept consistent by the CPU's cache coherency implementation. Hardware cache coherency again requires cross-CPU messaging, which is less expensive on a single CPU socket. Hence, we see better performance if all replicas run on a single socket.

As a last point, the cooperatively deterministic benchmark causes only 136 externalization events compared to more than 20,000,000 events the master needs to handle in the enforced deterministic case. The master process is therefore seldom involved and as a consequence, the fast synchronization optimization that speeds up master-level event handling has nearly no effect on this benchmark.

SUMMARY: I designed a replication-aware thread library that uses a lock info page shared among all replicas to establish deterministic ordering of lock acquisitions. This solution avoids expensive externalization events and inter-replica synchronization wherever possible and thereby achieves six times lower execution time overheads compared to enforced determinism.

4.3.6 Limitations of Lock-Based Determinism

The solution for deterministic replication I presented in this chapter assumes the application to solely use the synchronization operations provided by the `libpthread` library. Otherwise, the application must be free of data races. This requirement limits ROMAIN's applicability for those programs that use ad-hoc synchronization (spinlocks) or lock-free data structures.[38] This problem can be solved by adapting the respective libraries to be replication-aware similar to my adaptation of `libpthread`. As an alternative solution, fully deterministic execution may resolve or at least detect data races in non-locked accesses while merging thread-local data back into the globally consistent view. I explain how my solution would be extended to full determinism in the next section.

[38] Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993

4.3.7 Fully Deterministic Execution

In Section 4.3.2 I based my decision to only support weakly deterministic multithreading in ROMAIN on the resource overheads that would be required to implement strongly deterministic multithreading. Especially, I pointed out that having to maintain per-thread copies of all shared memory regions limits the practical applicability of DTHREADS-like strong determinism on 32-bit processor architectures.

While I developed ROMAIN focussing on the x86/32 architecture, modern 64-bit systems allow processes to address much larger amounts of physical memory. Hence, these resource limitations will become less of a problem in the future.[39] Therefore, I will now discuss the required changes to support strong determinism in ROMAIN similar to DTHREADS.

[39] Porting ROMAIN to x86/64 is work in progress at the time of this writing.

Adjusting the Execution Model The execution model I described in Section 4.3.1 also applies to strong determinism. The ROMAIN master process still monitors replica execution and each thread group's externalization events need to be handled independently. In contrast to my implementation, each replica thread would execute within a dedicated address space. These address spaces would be allocated by the master process whenever an application executes `pthread_create()` and then remain fixed for the whole lifetime of each replica thread.

Memory Management As strongly deterministic multithreading requires more than one address space per replica, ROMAIN's memory management needs to be aligned with these new requirements. To implement workspace consistency, ROMAIN needs to maintain one *reference copy* of all memory regions plus one additional copy for every replica thread.

Whenever a thread group raises a page fault, the ROMAIN master needs to serve it by mapping the respective duplicate memory regions into each replica thread's address space. For this purpose, the master needs to maintain a mapping between these copies and the respective replica threads.

The page fault handler can use a lazy memory allocation strategy, so that per-replica memory copies are only created for those replicas that actually access a region. This will help to reduce the amount of memory required for strong determinism.

Workspace Consistency At synchronization points replica threads need to merge their local updates into the reference view and vice versa. Similar to DTHREADS we can do so by instrumenting well-known `libpthread` functions and reflecting them to the ROMAIN master for performing deterministic memory updates.

To reduce merge effort, we can use the same optimization that was proposed by DTHREADS's authors: The ROMAIN master can map reference memory regions read-only to the replica threads, so that they share read-only data to reduce resource overhead. As discussed in Section 3.5 this still requires one copied region for every distinct replica if we want to avoid relying on ECC-protected memory.

SUMMARY: It is possible to extend ROMAIN to support strongly deterministic multithreading. This approach allows replication of multithreaded applications with data races, but its feasibility is limited due to resource constraints on 32-bit architectures. I propose to provide strong determinism using workspace consistency, which requires adapting ROMAIN's memory manager and the replication-aware thread library.

4.4 Reliability Implications of Multithreaded Replication

So far I focussed this chapter on implementing deterministic multithreaded replication with a low execution time overhead. However, the aim of the ROMAIN operating system service is to detect and correct errors. While error detection is equivalent to the single-threaded case described in Chapter 3, recovery requires additional care in a multithreaded environment.

4.4.1 Error Detection and Recovery

The architecture I presented in this chapter allows to deterministically replicate multithreaded applications on top of ROMAIN. ROMAIN's architecture makes sure that replicas perceive identical inputs. The deterministic multithreading extensions described in the previous section make sure that threads process these inputs in the same order and hence deterministically generate identical outputs.

Hardware-induced errors may modify a replica's state and may therefore cause a replica thread to behave differently. Similar to the single-threaded case, the ROMAIN master process will detect such a deviation by comparing the thread's state with the other threads within its thread group while processing externalization events. Once the ROMAIN master process detects a state mismatch, it starts an error recovery routine. In line with related work I assume a single-fault model here, which means hardware faults are rare enough so that we can assume only one fault to be active at a given point in time.

In contrast to single-threaded recovery we face an additional layer of complexity, though: before a faulty replica thread triggers error detection, it may have written arbitrary data to memory. As ROMAIN executes all threads of a replica in the same address space, all other threads in this replica may have read these faulty values. Hence, *all threads of a faulting replica must be assumed faulty*, even if they did not yet trigger their next externalization event. Given the single-fault assumption we can however assume that *all other replicas are still correct*.

To return the replicated application into a correct state, the ROMAIN master first halts all replica threads.[40] As threads execute independently, they will be stopped at an arbitrary point within their execution. Even threads in the same thread group will most certainly be interrupted at different points in their execution because they execute independently.

ROMAIN selects one of the correct replicas R_c to be the *recovery template*. Then, the master brings all other replicas into the state of the template:

[40] On Fiasco.OC, we can halt threads by setting their priority to 0.

1. The recovery template's address space and memory layout is copied to all other replicas.

2. Each replica thread $T_{i,j}$ (where i is the replica number and j is the respective thread group number) has its architectural state set to the state of the corresponding recovery template thread $T_{c,j}$ from its thread group.

Returning all replicas – even the correct ones – to the exact state of the recovery template allows us to handle the fact that we potentially halted other replica threads from the same thread group at different points within their computation. Eventually, all replica threads will be returned to an identical and correct state. The replicas can then resume execution.

4.4.2 Recovery Limitations

The recovery approach I presented in this section assumes that we can always return all replicas into a consistent and correct state. This assumption is true as long as replicas work on isolated resources, because then we can simply copy the state of a correct resource over the state of a faulty one. However, this assumption no longer holds if we have resources shared among all replicas for two reasons:

1. Faulty data may propagate through the shared resource to other replicas and thereby forgo fault isolation.

2. If no replicated copies of a shared resource exist, we cannot return the resource to a correct state.

This problem applies to the lock info page that ROMAIN uses to implement cooperative determinism: the respective memory area is shared among all replicas. Corrupting the LIP may affect other replicas' behavior (problem #1), for instance by blocking threads that would otherwise be able to acquire a lock. Furthermore, as the LIP only exists as a single copy, we cannot return it to a consistent state during recovery (problem #2).

To work around these issues, we need to have a closer look at what errors the LIP may suffer from. As we deal with hardware faults, we do not need to protect ourselves against targeted attacks of a single replica on the LIP. Instead, the LIP may be affected by hardware errors that I roughly classified into two categories:

1. **Corruption of the LIP:** LIP data is part of an application's address space. As the result of a fault, this data may become corrupted:

   - A fault affecting the target pointer of a write operation may change this pointer to point into the LIP. The write operation will then overwrite arbitrary data within the LIP.
   - A fault affecting the size of a write operation (e.g., the `size` parameter of a `memcpy()` operation) may cause a valid write operation to overflow its target buffer. If this buffer is located before the LIP in the replica's address space, the overflow may affect LIP content.
   - Last, an SEU in memory or during a computation may affect the LIP directly.

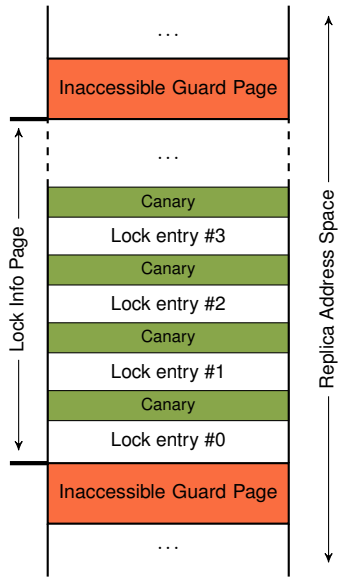

Figure 4.21: Protecting the LIP within the replica's address space: Guard pages prevent overflow writes from different sources, canaries aim to detect faulty writes within the LIP.

[41] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Exposition*, volume 2, pages 119–129 vol.2, 2000

[42] The guard page *behind* the LIP is necessary as some `memcpy` implementations copy backwards.

[43] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generative Software-Based Memory Error Detection and Correction for Operating System Data Structures. In *International Conference on Dependable Systems and Networks*, DSN'13. IEEE Computer Society Press, June 2013

2. **Inconsistent lock acquisition:** Even if the LIP is correctly modified with respect to the cooperative determinism protocol, its content may be inconsistent for recovery purposes:

   - A replica may decide to acquire a lock based on a previous error. In this case, the replica will correctly acquire the lock, but none of the correct threads in the same thread group will ever do so. Given that we only reset lock ownership once all threads of a thread group have called `rep_unlock()`, this lock will remain locked infinitely and other threads trying to correctly acquire the lock will block.
   - I explained above that ROMAIN halts all threads during recovery and that we may stop threads at different points in their execution. As a result, we may encounter situations, where one correct thread already acquired a lock, while a second correct thread did not reach the point of acquisition yet. During recovery, we need to bring all threads into the same state. Consequentially, we need to either release or acquire the lock for everyone, depending on which replica we select as the recovery template.

Detecting Corruption using Guards and Canaries In order to reduce the chance of LIP corruption, I modified the way ROMAIN attaches the LIP to each replica's address space as shown in Figure 4.21. This modification is inspired by Cowan and colleagues' work on protecting memory against buffer overflow attacks.[41]

To prevent buffer overflows in application data from corrupting the LIP, ROMAIN places an inaccessible guard page before and behind the LIP. Thereby, any overflowing write sequence will cause a page fault exception before modifying LIP state.[42]

To increase the likelihood of detecting corruptions within the LIP, all lock entries are separated by a canary value. The canary is a fixed bit pattern that will never be modified by normal operation. Whenever a thread tries to acquire a lock, it first validates the correctness of the respective lock entry's canary value. If the canary is found to be corrupt, the thread notifies the master of an unrecoverable error. As the LIP is shared across all replicas, the only suitable reaction to such an event is for the master to terminate the replicated application. However, this approach at least makes sure that replicas generate no incorrect output.

An alternative solution to increase the chance of successful recovery would be to replicate the LIP itself similar to Borchert's replication of kernel data structures.[43] However, I did not implement this alternative.

Consistent LIP Recovery using Acquisition Bitmaps I addressed the problem of consistent LIP recovery by adding a mechanism that tracks which replica thread already acquired a lock. For this purpose I added an `acq_bitmap` field to each LIP lock entry. When a thread $T_{i,j}$ from thread group j acquires a lock, it sets the i-th bit in this field to 1. Upon lock release, the thread resets this bit to 0.

Using this mechanism, we can address the two inconsistent lock acquisition subproblems as follows:

1. If a thread erroneously acquired a lock while the correct threads did not, the `acq_bitmap` will have this thread's bit set to 1, while all other bits are 0. During recovery, ROMAIN can set this bit to 0 and thereby return the respective lock entry to a consistent and correct state.

2. If correct threads are stopped during recovery and some threads halt before acquiring a lock and others halt afterwards, the respective lock entry will have both 0 and 1 entries in its `acq_bitmap`. To correct this consistently, ROMAIN sets all bits in the bitmap to the value of the recovery template's bit. Hence, only if the recovery template thread already acquired the lock, all other threads will acquire it during recovery.

While the above enhancements increase LIP reliability, they also increase resource and execution time overhead. I chose to use 16-bit canaries, so that the LIP size grows by $2,048 * 2 = 4,096$ bytes. Furthermore, I repeated the microbenchmark that I used in the previous sections with cooperative determinism and the enhancements described above. DMR runtime increases about 10%, TMR runtime increases about 15%.

SUMMARY: Detecting errors in multithreaded replicas works similar to error detection for single-threaded ones. Error recovery needs additional care, because all replicas and their threads need to be returned to the same consistent state.

Cooperative determinism suffers from the problem that the LIP is shared among all replicas. Corrupt LIP entries may therefore constitute unrecoverable errors. I combined guard pages, lock entry canaries, and fine-grained ownership tracking to reduce the chance that such a situation arises.

The above analysis shows that there is another dimension when evaluating different reliability mechanisms: While cooperative determinism provides a low execution time overhead, it is harder to protect against corruption from a faulty replica. In contrast, enforced determinism does not need to care for shared data, but suffers from higher execution time overheads.

In Chapter 6 I will show that this difference between enforced and cooperative determinism is only one particular instance of a more general problem: every software-implemented fault tolerance mechanism relies on specific (but different) software and hardware features to function correctly. We need to identify and specially protect these components to achieve full-system reliability.

5
Evaluation

In the previous two chapters I described the design of ROMAIN and used microbenchmarks to motivate my design decisions. In this chapter I present a larger-scale evaluation of how well ROMAIN achieves its goal of providing fault-tolerant execution to binary-only user applications on top of FIASCO.OC. For this purpose I analyze ROMAIN's error detection capabilities (error coverage and detection latency), the accompanying resource and execution time overheads, as well as ROMAIN's implementation complexity.

I show that ROMAIN detects 100% of all errors within a replicated application and recovers from more than 99.6% of them. ROMAIN's best-case execution time overhead for replicating single-threaded applications is about 13% for triple-modular redundant execution. Multithreaded replication is more costly and implies up to 65% overhead when running three replicas of an application with four worker threads. By providing majority voting, ROMAIN allows for fast error recovery.

Parts of the experiments presented in this chapter were published in SOBRES 2013[1] and EMSOFT 2014.[2]

5.1 Methodology

Wilken identified five properties that can be analyzed to assess a fault-tolerant system.[3] I apply his taxonomy to analyze ROMAIN and evaluate these properties:

1. *Error Coverage* measures the fraction of errors that a fault tolerance mechanism is able to detect and recover from.
2. *Error Detection Latency* determines how long a fault resides in the system before it is detected.
3. Replicated execution incurs *execution time overhead* because the ROMAIN master process spends additional time waiting for replicas, inspecting their states, and performing replicated resource management.
4. Replicated execution furthermore induces a *resource overhead* by maintaining resource copies to facilitate independent execution of replicas.
5. ROMAIN adds *code complexity* to the L4 Runtime Environment by adding the master process component.

[1] Björn Döbel and Hermann Härtig. Where Have all the Cycles Gone? – Investigating Runtime Overheads of OS-Assisted Replication. In *Workshop on Software-Based Methods for Robust Embedded Systems*, SOBRES'13, Koblenz, Germany, 2013

[2] Björn Döbel and Hermann Härtig. Can We Put Concurrency Back Into Redundant Multithreading? In *14th International Conference on Embedded Software*, EMSOFT'14, New Delhi, India, 2014

[3] Kent D. Wilken and John Paul Shen. Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 9(6):629–641, 1990

I evaluate error coverage and detection latency using fault injection experiments in Section 5.2. Thereafter, I evaluate ROMAIN's execution time and resource overhead using application benchmarks in Section 5.3. I analyze the master process' code complexity in Section 5.4. Finally, I compare the achieved results to related work in Section 5.5.

In line with related work[4,5] I assume my fault model to be single-event upsets in memory and registers. In this fault model, we can detect errors by running two replicas (double-modular redundancy – DMR) and we can correct detected errors by majority voting when running three replicas (triple-modular redundancy – TMR). I therefore only investigate DMR and TMR setups and leave the analysis of multi-error fault models and ROMAIN running more than three replicas for future work.

[4] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *International Conference on Computer Safety, Reliability and Security*, Safecomp'10, Vienna, Austria, 2010

[5] V. B. Kleeberger, C. Gimmler-Dumont, C. Weis, A. Herkersdorf, D. Mueller-Gritschneder, S. R. Nassif, U. Schlichtmann, and N. Wehn. A Cross-Layer Technology-Based Study of how Memory Errors Impact System Resilience. *IEEE Micro*, 33(4):46–55, 2013

5.2 Error Coverage and Detection Latency

The main goal of every fault tolerance mechanism is to detect and recover from errors before they cause system failure. Given a fault model and a workload, the mechanism's error coverage measures the ratio of faults of this specific type that are detected and corrected:

$$Coverage = \frac{N_{detected}}{N_{total}}$$

In addition to error coverage, we can measure the error detection latency as the time between the activation of an error and its detection by the respective fault tolerance mechanism.

As I explained in Chapter 2, redundant multithreading (RMT) techniques – such as ROMAIN – aim to maintain best possible error coverage while reducing the mechanism's execution time overheads as much as possible. For this purpose, RMT often trades higher detection latencies for lower overhead by reducing state validation. This trade-off does not affect error coverage as long as two properties hold:

1. A fault must never lead to a visible application failure.
2. The error detection latency must not exceed the minimum expected inter-arrival time of hardware faults. Otherwise, the number of replicas in the system might no longer suffice to detect and recover from these faults.

Similar to other RMT techniques, ROMAIN ensures the first property by performing state validation whenever application state is about to become visible to an external observer. In Chapter 3 I showed the techniques the ROMAIN master uses to intercept all possible externalization events for this purpose.

As RMT techniques only validate state at externalization points, ROMAIN may suffer from problems related to the second property. This will be the case if a replicated application performs long stretches of computation without performing any interceptable externalization event in between. However, ROMAIN implements two mechanisms that allow to cope with such situations:

1. The user can *tune the number of replicas* to the number of expected concurrent faults using Schneider's $2N+1$ rule[6] and thereby create a setup

[6] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990

that can handle multiple concurrent faults. This number is a startup option passed to the ROMAIN master.

2. As I explained in Section 3.8.2, Martin Kriegel extended ROMAIN with a mechanism to *bound error detection latencies* by enforcing checks after a fixed number of retired instructions.[7] The application can thereby be forced into state validation in periods shorter than the expected inter-arrival time of faults.

[7] Martin Kriegel. Bounding Error Detection Latencies for Replicated Execution. Bachelor's thesis, TU Dresden, 2013

5.2.1 Vulnerability Analysis Using Fault Injection

Fault injection (FI)[8] is a standard technique to evaluate the behavior of a system in the presence of faults. In contrast to waiting for errors to manifest in physical hardware, these experiments allow us to insert these errors in a controlled environment. Given a fault model, such a controlled environment also allows us to inject every possible error and thereby assess the error coverage of a given fault tolerance method.

[8] Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault Injection Techniques and Tools. *IEEE Computer*, 30(4):75–82, Apr 1997

Fault-injection methods can be implemented at the hardware and software levels. Hardware-level injectors work on implementations of the hardware in question and often augment this hardware with specific points to inject faults. These approaches can perform fine-grained instrumentation of the hardware and inject faults down to the transistor level. Sterpone used an extended FPGA implementation of a microprocessor for this purpose.[9] In contrast, Heinig and colleagues performed fault injection on an embedded ARM processor and used an attached hardware debugger to inject faults.[10] The downside of these approaches is that they require dedicated hardware and labor-intensive manual setup.

[9] Luca Sterpone and Massimo Violante. An Analysis of SEU Effects in Embedded Operating Systems for Real-Time Applications. In *International Symposium on Industrial Electronics*, pages 3345–3349, June 2007

[10] Andreas Heinig, Ingo Korb, Florian Schmoll, Peter Marwedel, and Michael Engel. Fast and Low-Cost Instruction-Aware Fault Injection. In *GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*, 2013

Software-level fault injection tools try to inject faults without requiring direct access to the underlying hardware. For example, Gu performed fault injection experiments for Linux on x86 hardware using a debugging kernel module and hardware breakpoints, but without access to an expensive x86 hardware debugger.[11] Other researchers – such as Yalcin[12] – proposed to use hardware simulators instead of real hardware for fault injection. Simulator-based fault injection restricts reliability assessment to the level of detail provided by the underlying simulator. Cho pointed out that these inaccuracies make it hard to use simulator-based FI to draw any conclusions about the vulnerability properties of physical hardware.[13]

[11] Weining Gu, Z. Kalbarczyk, and R.K. Iyer. Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors. In *Conference on Dependable Systems and Networks*, DSN'04, pages 887–896, June 2004

[12] G. Yalcin, O.S. Unsal, A. Cristal, and M. Valero. FIMSIM: A Fault Injection Infrastructure for Microarchitectural Simulators. In *International Conference on Computer Design*, ICCD'11, 2011

[13] Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A. Abraham, and Subhasish Mitra. Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–10, 2013

In this section I am not interested in hardware properties, but in the reaction of ROMAIN to misbehaving hardware. For this purpose, software-level FI still works: we can select a representative fault model (such as SEUs in memory), apply this fault model in a simulator and observe if ROMAIN detects and corrects these errors. While the resulting error distributions may not always be representative of physical hardware behavior, we can still make an apples-to-apples comparison between the same experiment running without replication and protected by ROMAIN.

Under these circumstances we can benefit from another property of simulation-based fault injection: we can now run many such simulators in parallel and thereby drastically reduce the time needed to perform large-scale fault injection experiments.

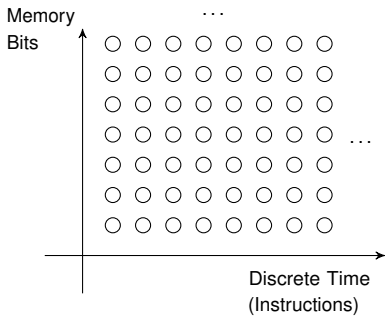

Figure 5.1: Fault Space for Single-Event Upsets in Memory

In order to determine the coverage of a fault tolerance mechanism using fault injection, we need to inject all faults that might affect the given system and compute the ratio between detected and injected faults. This *fault space* often grows very large. As an example, Figure 5.1 shows the required set of experiments if we assume single-event upsets in memory as the fault model. In this example we need to perform one experiment for every discrete time instant and every bit in memory.

For most applications, this fault space is too large to fully enumerate even with fast FI mechanisms. Therefore, two techniques are applied to obtain practical results in a timely manner:

1. *Fault Space Sampling:* Instead of performing all necessary experiments, sampling approaches select a subset of these experiments and try to approximate the global result by performing only the experiments in this sample. The accuracy of the obtained results varies depending on the sample size and the method with which the sample was chosen.

   Random sampling selects a random subset of experiments. This approach works if the selected sample is large enough and we are only interested in high-level questions, such as "What fraction of faults is going to crash the application?" Gu's Linux study used this approach.[11] Random sampling fails if we are interested in the vulnerability of specific data structures or functions, because the random sample may misrepresent them.

   An alternative – which I call *workload reduction* – focuses fault injection on interesting subsections of a larger workload and fully enumerates these. This approach was for instance applied by Arlat and colleagues to analyze the reliability of kernel subsystems in LynxOS.[14]

2. *Fault Space Pruning:* If we take a closer look at the fault space we find groups of experiments that will produce identical results. As an example, consider a scenario where a memory word W is read at time instants 0 and 10. Any fault in instants 1 through 10 will have the same consequence: the wrong value will be read at instant 10 and impact program behavior thereafter. Hence, we can speed up experimentation by only performing a single one of these experiments and apply its result to all others.

   Modern fault injection frameworks, such as Relyzer[15] and FAIL*[16] analyze the fault space to identify such groups. They thereby reduce the number of required experiments without reducing fault injection accuracy.

I use the FAIL* fault injection framework to analyze ROMAIN's error coverage and detection latencies. FAIL* uses the Bochs[17] emulator to inject faults and monitor program behavior. Bochs performs instruction-level emulation of the underlying platform and can therefore only model faults that are visible at this granularity. I specifically look at two fault models that FAIL* supports out of the box: SEUs in (1) memory and (2) general-purpose CPU registers.

[14] Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frédéric Salles. Dependability of COTS Microkernel-Based Systems. *IEEE Transactions on Computing*, 51(2):138–163, February 2002

[15] Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 123–134, New York, NY, USA, 2012. ACM

[16] Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. FAIL*: Towards a Versatile Fault-Injection Experiment Framework. In Gero Mühl, Jan Richling, and Andreas Herkersdorf, editors, *International Conference on Architecture of Computing Systems*, volume 200 of *ARCS'12*, pages 201–210. German Society of Informatics, March 2012

[17] `http://bochs.sourceforge.net`

The current version of FAIL* does not distinguish between different address spaces on top of a modern operating system. As I want to inject faults into a specific replica address space, I therefore use an extension to FAIL* that was originally developed by Martin Unzner in a thesis I advised.[18] With this extension we can distinguish between code executing within one dedicated address space (i.e., the faulty replica) and code executing outside this address space.

[18] Martin Unzner. Implementation of a Fault Injection Framework for L4Re. Belegarbeit, TU Dresden, 2013

As FAIL* executes all experiments in the Bochs emulator, it has no notion of wall-clock time. Instead, FAIL* discretizes time in terms of instructions retired by the observed platform. Hence, time (i.e., error detection latency) in this subsection is measured in retired instructions.

FAIL* provides a campaign server that sends experiments to concurrent instances of the FAIL* fault injection client and collects the results. This allows to parallelize fault injection runs and I had the opportunity to do so on the Taurus HPC Cluster at the Center for Information Services and High Performance Computing (ZIH) at TU Dresden. The FI experiments I describe below consumed a total of 66,000 CPU hours on this cluster.

5.2.2 Benchmark Setup

I selected four applications as benchmarks to evaluate ROMAIN's fault tolerance capabilities. To keep fault injection manageable, I focus on small benchmarks. I will use longer-running examples to validate ROMAIN's computational overhead in the next section.

1. *Bitcount* is a simple, compute-bound benchmark from the MiBench benchmark suite.[19] It compares different implementations of bit counter algorithms. This property makes its results susceptible to bit flip effects.
2. *Dijkstra* is another benchmark from the MiBench suite. It represents an implementation of Dijkstra's path finding algorithm[20] that is used for instance in network routing.
3. *IPC* is an example for FIASCO.OC's inter-process communication mechanism. Two threads run inside the same address space and exchange a message. This benchmark therefore focuses on faults that happen directly before or after invoking a system call on FIASCO.OC.
4. *CRC32* uses the Boost[21] C++ implementation of the CRC32 checksum to compute a checksum over a chunk of data in memory. CRC32 is a commonly used checksum algorithm in network applications.[22]

[19] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *International Workshop on Workload Characterization*, pages 3–14, Austin, TX, USA, 2001. IEEE Computer Society

[20] Edsger. W. Dijkstra. A Note on Two Problems in Connexion With Graphs. *Numerische Mathematik*, 1:269–271, 1959

[21] `http://www.boost.org`

[22] Philip Koopman. 32bit Cyclic Redundancy Codes for Internet Applications. In *Conference on Dependable Systems and Networks*, DSN '02, pages 459–472, Washington, DC, USA, 2002. IEEE Computer Society

I prepared the experiments by creating boot images of the application setups that can be run by FAIL*. For every setup I created one image that runs the benchmark natively on L4Re and a second image that runs the benchmark in ROMAIN with TMR. Table 5.1 shows the dimension of my fault injection campaigns. While previous works[23,24] used random sampling with a sample size of up to 10,000 experiments, my study covers the whole fault space of these four applications and thereby represents several million experiments for every benchmark and fault model.

[23] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *International Conference on Computer Safety, Reliability and Security*, Safecomp'10, Vienna, Austria, 2010

[24] Weining Gu, Z. Kalbarczyk, and R.K. Iyer. Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors. In *Conference on Dependable Systems and Networks*, DSN'04, pages 887–896, June 2004

To reduce fault injection time, I reduced the workloads to their interesting parts – i.e. the main work loop of the benchmarks – leaving out application startup and teardown. The table shows the number of instructions executed in the injection phase of each benchmark.

I furthermore benefit from FAIL*'s fault space pruning to reduce the number of required experiments while still covering the whole fault space. The table therefore shows the total number of experiments covered by each fault injection campaign as well as the pruned number of experiments, i.e., the runs that I actually had to perform.

Benchmark	Fault Model	# of Instructions	Total Experiments	Experiments after Pruning
Bitcount	Register SEU	54,866	9,100,824	1,635,808
	Memory SEU		405,710,740	374,865
Dijkstra	Register SEU	108,171	16,990,912	3,694,177
	Memory SEU		2,399,115,368	1,221,705
IPC	Register SEU	10,800	1,567,088	341,049
	Memory SEU		46,656,894	116,881
CRC32	Register SEU	108,089	23,976,128	2,408,329
	Memory SEU		510,278,440	288,881

Table 5.1: Overview of Fault Injection Experiments

For each FI experiment I collected the following information for later processing of the results:

- *Experiment Information:* For every experiment I keep track of what kind of fault (e.g., a bit flip in which bit of which register) was injected at which discrete point in time.
- *Experiment Outcome and Output:* A fault injection experiment is executed until the program reaches a predefined terminating instruction (successful completion) or a timeout expires. I set this timeout to 4,000,000 instructions, which is 40 times larger than the longest of the benchmarks in question. This long timeout is necessary to give ROMAIN sufficient time for error detection and correction in the failure case.

  Depending on the result type I classify the experiment outcome:

  1. *No Effect:* The experiment reached the terminating instruction and the experiments' output matches the output of an initial fault-free run. In case of an injected fault this means that the fault did not alter visible program behavior.
  2. *Silent Data Corruption (SDC):* The program successfully terminated, but the output differs from the initial fault-free run. This happens if a hardware fault modifies the program's output but does not lead to a visible crash.
  3. *Crash:* The experiment terminated, but the terminating instruction pointer or the logged output indicate that the program crashed, for instance by accessing an invalid memory address.
  4. *Timeout:* The program did not reach its final instruction and the output does not indicate a user-visible crash. This happens for instance if a hardware fault makes the program get stuck in an infinite loop.

5. *Corrected Error:* When injecting faults while running in ROMAIN, the replication service detected and corrected an error. In the discussion below I furthermore distinguish between two sources of error detection:
   (a) *Detection By State Comparison:* All replicas reached their next externalization event and ROMAIN detected a state mismatch.
   (b) *Detection By Timeout:* At least one of the replicas did not reach the next externalization event before the first replica's event watchdog timeout expired.

- *Faulty Execution Time:* I log the number of instructions the faulty replica executes before an error is detected. As we will see in Section 5.2.4, this data is the closest information we can get about error detection latency in a FAIL* setup.

5.2.3 Coverage Results

I executed one fault injection campaign for every benchmark and every fault model. In every campaign I first injected faults into the benchmark without protection to get a base distribution of fault outcomes. Thereafter I injected faults into the same benchmark running with ROMAIN in TMR mode. Figure 5.2 shows the distribution of fault types when injecting SEUs into general-purpose registers during the runtime of the benchmarks. Figure 5.3 shows the same distribution when injecting bit flips into application memory. (Single benchmark names represent the native runs. Replicated benchmark results are suffixed with TMR.)

We see crash, timeout, and SDC failures in native execution across all benchmarks. Register SEUs are more likely to lead to crashes, because registers are often used for dereferencing pointers. A bit flip in such a pointer may modify the target address to point into an invalid memory area, leading to an unhandled page fault. A closer look at the native benchmark outcomes confirms that in all campaigns more than 80% of the crashes can be attributed to page faults.

When looking at native memory errors, the CRC32 benchmark stands out as it shows an SDC rate of 97%. This can be explained by the fact that most memory accesses performed by this benchmark target the memory region that a CRC checksum is computed from, because the program does not access any other memory. Hence, bit flips in this area will lead to a diverging checksum and show up as SDC errors.

Comparing the native experiments to the TMR experiments we see that ROMAIN detects and corrects all memory errors and close to all register SEUs. This confirms that ROMAIN works as intended and provides fault tolerant execution to the applications it protects.

I had a closer look at the CRASH errors that appear when injecting faults into the Bitcount, IPC, and Dijkstra benchmarks. In all of these cases, an error is detected by ROMAIN but recovery does not succeed within the bounds of the fault injection experiment. While these experiments are a small fraction of all injections, they still number in the thousands. I manually repeated several of those experiments and in all cases recovery succeeded during my repeated experiments. I conclude that these crashes were only identified as

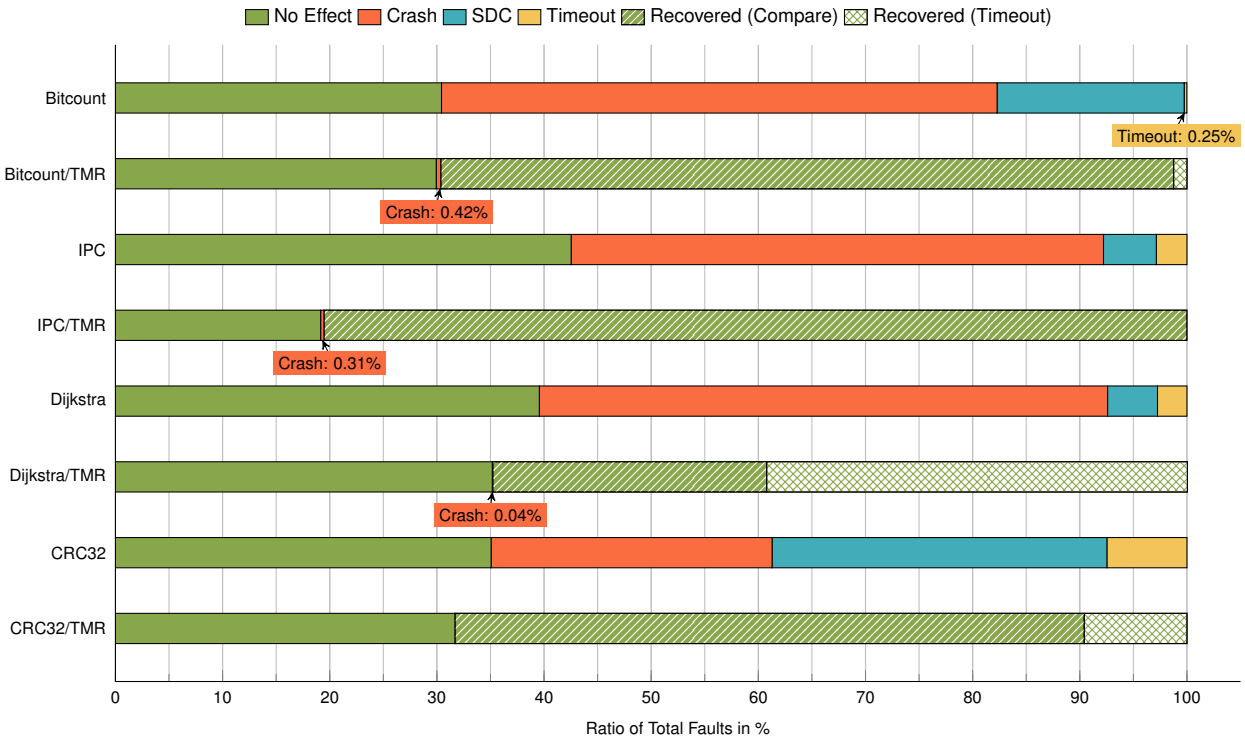

Figure 5.2: ROMAIN Error Coverage, Fault Model: SEUs in General-Purpose Registers

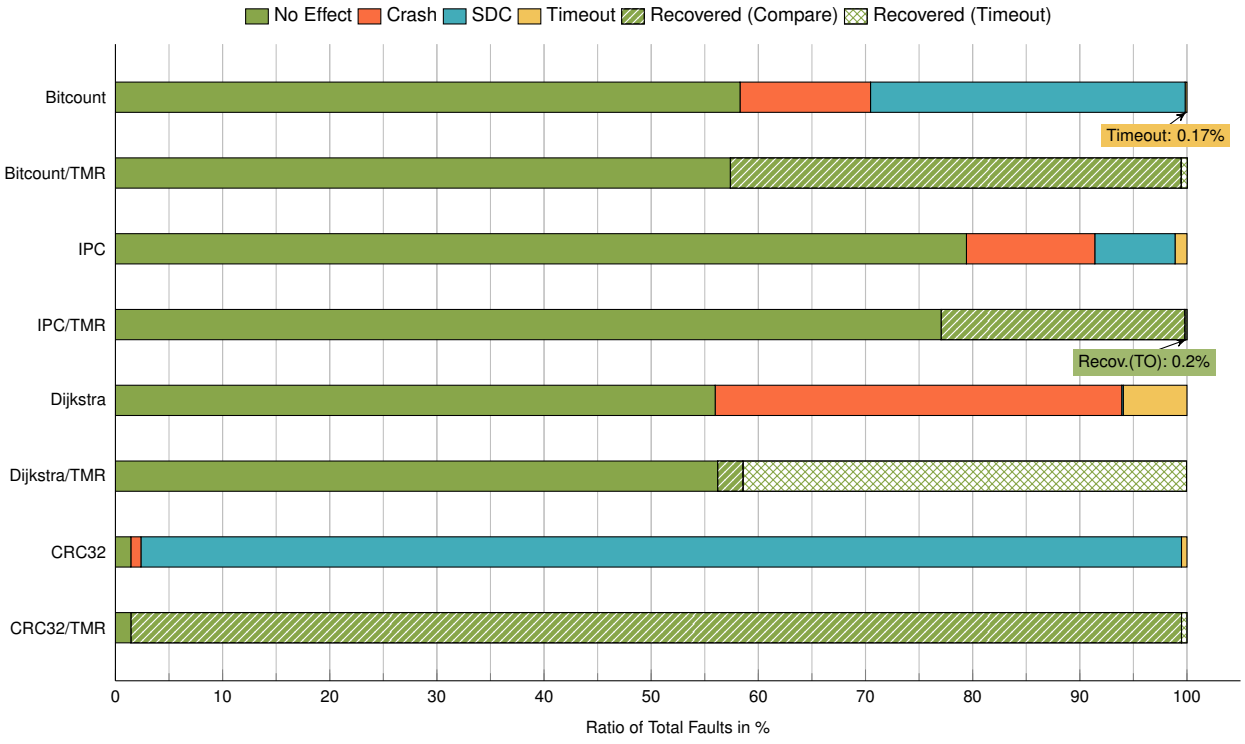

Figure 5.3: ROMAIN Error Coverage, Fault Model: SEUs in memory accessed by the application

such because the experiments did not run to completion before the timeout of my fault injection experiments triggered.

Nevertheless I show these experiments as crashes in my experiment results, because I did not revalidate all of the experiments. From the results we see that ROMAIN's error detection rate is 100%. Recovery succeeds in at least 99.6% of the injected register errors and in 100% of the injected memory errors.

The largest fraction of errors are detected by state comparison. Only few errors (less than 10% of the faults in Bitcount, IPC, and CRC32) are found as the result of ROMAIN's event watchdog expiring. This is contrasted by the Dijkstra benchmark, where most errors are found as the result of a timeout.

In contrast to the other benchmarks, the distance between externalization events in Dijkstra is high, because the program executes the whole path finding algorithm before performing another system call. Hence, the correct replicas execute for a long time before reaching this call. If we now inject a fault and the affected replica fails fast, e.g., by raising a page fault, the replica will then enter the ROMAIN master and wait for the remaining replicas to raise an event. As these replicas still execute for a long time, the faulty replica's watchdog expires before the correct ones arrive for state comparison. Recovery then finds that no majority of replicas is available, waits for all other replicas and then succeeds in returning the program to a correct state. Hence, these errors are actually corrected by state comparison, but my automated outcome classification tool marked them as "detected by timeout" due to their output.

I repeated the Dijkstra memory experiment with the event watchdog programmed to a timeout three times as high as before. In this case, the correct replicas reach their next externalization event before the faulty replica's timeout expires. In turn, 93% of the detected errors are then classified as "detected by state comparison."

All in all 100% error detection and close to 100% correction rates show that ROMAIN achieves its goal of protecting applications against the effects of hardware faults. Note however that these coverages are computed only for code running inside the protected application. While this is in line with all other SWIFT mechanisms I am aware of, it ignores the fact that there are other software components that remain unprotected by ROMAIN. These components include the operating system kernel as well as the ROMAIN master process and I will return to this problem in Chapter 6.

5.2.4 Error Detection Latency

As I explained previously, redundant multithreading trades higher error detection latencies for reduced execution time overhead. We saw in the previous experiments that ROMAIN achieves fault tolerance and we will see in Section 5.3 that the respective execution time overheads are indeed lower than for other non-RMT methods. In this section I now try to estimate ROMAIN's error detection latency.

One main assumption I make in this thesis is that modern hardware provides a sufficient number of physical processors and that ROMAIN can therefore execute every replica on a dedicated CPU as shown in Figure 5.4.

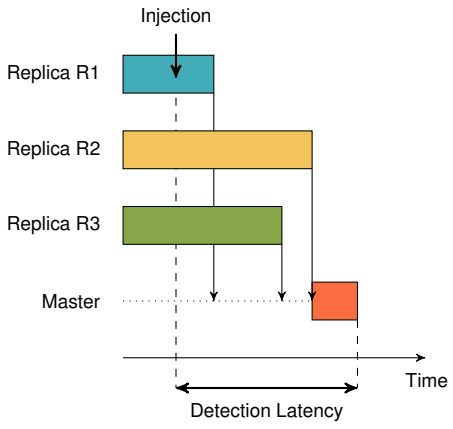

Figure 5.4: Error Detection Latency in a ROMAIN TMR setup on real hardware

Three replicas execute concurrently and the ROMAIN master validates their states whenever they execute an externalization event.

Let us now assume that we inject a fault into Replica 1 as indicated in the figure. The replicas will run until their next externalization event, the ROMAIN master will compare their states and flag an error. On a physical computer we can therefore measure the error detection latency as the wall clock time difference between the injection and detection times.

In contrast to physical hardware, FAIL* executes all replicas on a single emulated CPU. Therefore, replicas share this CPU and their execution order will be determined by FIASCO.OC's scheduler. Depending on the actual situation, replicas may be scheduled in arbitrary order, such as one of the orderings shown in Figure 5.5. As we see in the figure, the measured wall clock time may vary depending on the chosen schedule and it is therefore difficult to draw any conclusions about error detection latency.

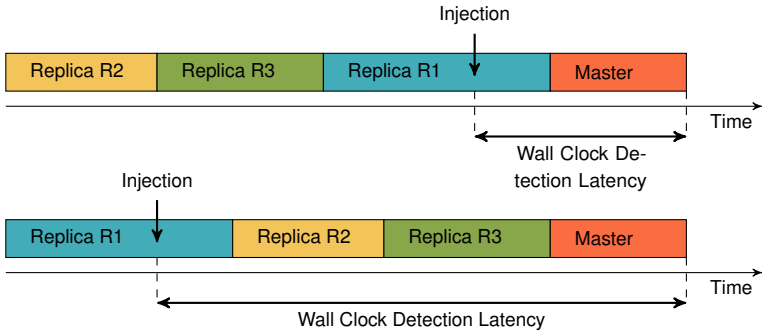

Figure 5.5: Possible replica schedules in a single-CPU ROMAIN TMR setup.

However, my instrumentation in FAIL* allows me to measure the number of instructions the faulty replica executes before ROMAIN detects an error and corrects it. Figure 5.6 plots the cumulative distribution function for this faulty execution time for each of my fault injection campaigns. Each plot distinguishes between the results for memory and register SEUs. I furthermore show separate plots for errors that were detected by comparing replica states (=) and errors that were detected due to ROMAIN's event watchdog (=).

The distributions differ across applications. Every application has a specific offset at which all errors that will be detected by state comparison are found. Bitcount, which performs several system calls during the fault injection experiment, has this offset at around 50,000 instructions. The fairly short IPC benchmark reaches it at 10,000 instructions. In contrast, Dijkstra and CRC32 perform long stretches of computation and therefore only reach this point later (2 million instructions for Dijkstra, 150,000 instructions for CRC32).

The results furthermore show ROMAIN's event watchdog in action. In the Bitcount, IPC, and CRC32 benchmarks, the rate of errors detected by watchdog timeout jumps from close to zero to 100% at around 500,000 instructions after the injection. This corresponds to the timeout value that I configured for these experiments. We also see that this is not the case for the Dijkstra benchmark. Again, this is caused by the fact that Dijkstra computes for several million cycles before reaching its next externalization event and the timeouts we see in Dijkstra are caused by the faulty replica whose watchdog expires before the correct replicas reach an externalization event.

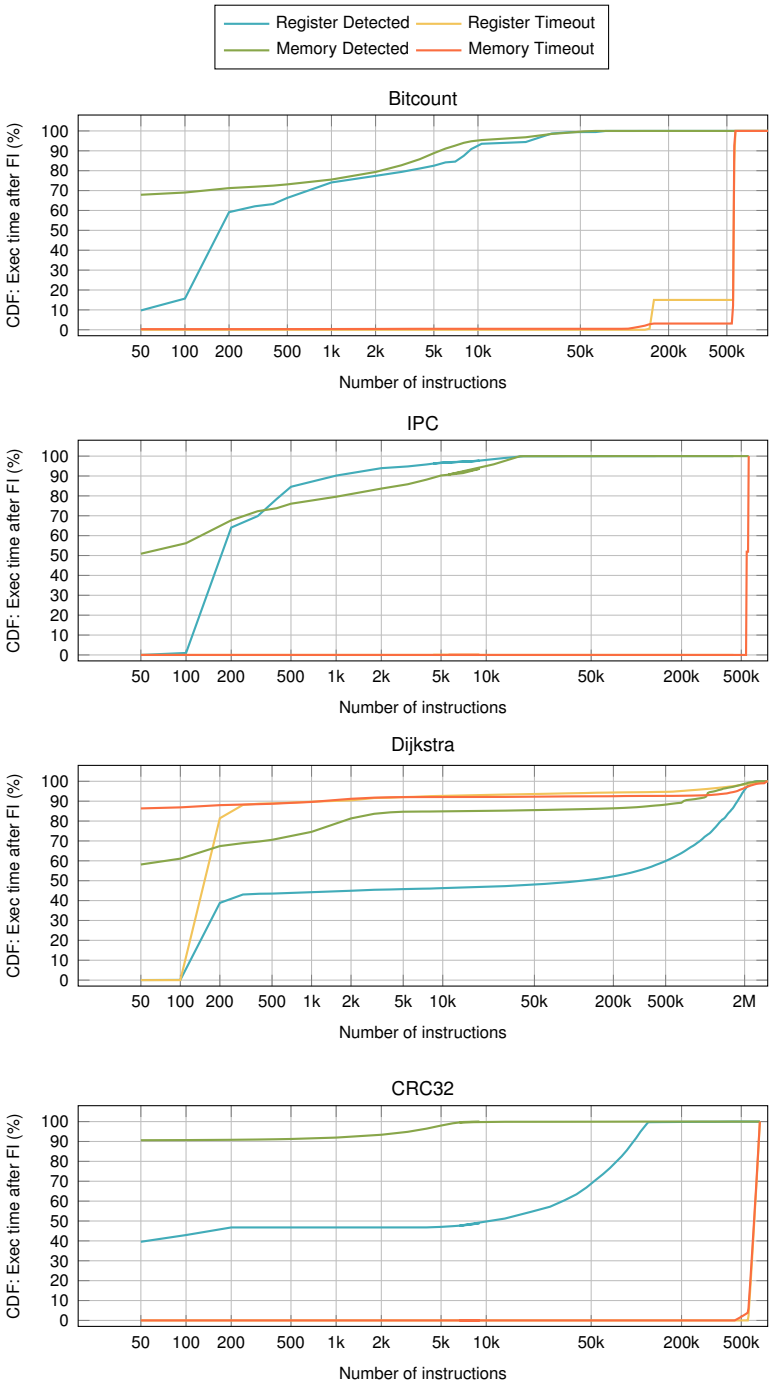

Figure 5.6: Number of instructions the faulty replica executed before ROMAIN detected an error (CDF over all FI experiments)

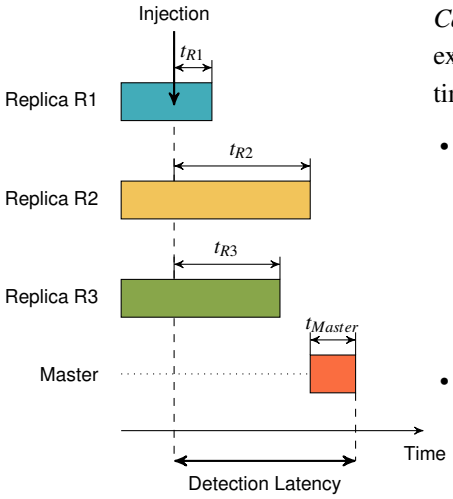

Figure 5.7: Computing error detection latency

Can we Compute Error Detection Latency? If we reconsider concurrently executing replicas as in Figure 5.7, we see that the faulty replica's execution time is not identical to the error detection latency:

- ROMAIN performs state comparison once all replicas reach their next externalization event. As replicas execute in parallel and do not access shared resources in this time, their wall clock execution time is the maximum of the single replica execution times:

$$t_{r_{max}} = max(t_{R1}, t_{R2}, t_{R3})$$

- The error detection latency needs to additionally incorporate the time t_{Master} that the master process requires to perform replica state comparison as well as the time t_{Kernel} that is spent inside FIASCO.OC for processing scheduling interrupts as well as delivering replica events to the master process.

  The error detection latency can therefore be expressed as

$$t_{detect} = t_{r_{max}} + t_{Master} + t_{Kernel}$$

Calculating this latency requires proper measurements or analysis of the respective components. Such an analysis is out of scope of my thesis, but appears to be an interesting direction for future research. I would like to provide two starting points for this research here:

1. In order to determine $t_{r_{max}}$ we need to measure the maximum time it may take a correct replica to execute before reaching its next externalization event. This time provides a lower bound for when the next replica state comparison will set in. As replicas execute independently on different cores and do not access any shared state between these events, such an analysis will be similar to traditional worst case execution time (WCET) analysis.[25]
2. In addition to $t_{r_{max}}$ we furthermore need to incorporate t_{Master} and t_{Kernel}. These components can first be determined using separate WCET analyses of the respective components. The replicas, the master, and the operating system kernel can then be modeled as a sequence of fork-join parallel tasks. Axer showed that it is possible to perform a response time analysis for this class of tasks.[26]

[25] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008

[26] Philip Axer, Moritz Neukirchner, Sophie Quinton, Rolf Ernst, Björn Döbel, and Hermann Härtig. Response-Time Analysis of Parallel Fork-Join Workloads with Real-Time Constraints. In *Euromicro Conference on Real-Time Systems*, ECRTS'13, Jul 2013

SUMMARY: I performed fault injection experiments to analyze ROMAIN's error coverage. Injecting SEUs into memory and general-purpose registers I found that ROMAIN detects 100% of the errors in a replicated application and is able to recover from more than 99.6% of these errors.

I furthermore explored opportunities to analyze error detection latency. My fault injection experiments show that replicas, depending

on the actual application, execute several thousands up to several millions of instructions before an error is detected. Actual determination of error detection latencies bears similarities with worst-case execution time analysis and is left as an open issue for future work.

5.3 Runtime and Resource Overhead

Having shown that ROMAIN achieves its goal of detecting and correcting hardware errors before they lead to application failure, I now investigate the runtime characteristics of the replication service. I use the SPEC CPU 2006 and SPLASH2 benchmark suites to evaluate replication overhead. Thereafter I measure the slowdown ROMAIN introduces when replicating a shared-memory application and show that the time needed for error recovery is dominated by the replica's memory footprint.

5.3.1 Test Machine and Setup

All experiments in this section are executed on a machine with 12 physcial Intel Xeon X5650 CPU cores running at 2.67 GHz. The cores are distributed across two sockets with six cores each. Each socket has a 12 MiB L3 cache shared among all cores. Each core has 256 KiB local L2 cache. I turned off Hyperthreading, TurboBoost, and dynamic frequency scaling in order to obtain reproducible results.

On the test machine I run 32-bit versions of the FIASCO.OC microkernel, L4Re, ROMAIN, and the respective benchmarks. Therefore, the available memory for my experiments is limited to 3 GiB. All software was compiled with a recent version of GCC (Debian 4.8.3).

I assume a single-fault model, where at most one erroneous replica exists at a single point in time. As I explained before, we need two replicas to detect an error, and three replicas to also provide error correction in such a scenario.[27] I therefore measure execution times for ROMAIN running two and three replicas of an application. I compare these results to native execution of the same application on top of L4Re. In addition to that, I also measure the execution time of ROMAIN running a single replica. While this does not give any benefit in terms of fault tolerance, this benchmark allows us to estimate the overhead of ROMAIN's mechanism to intercept externalization events.

I configured all runs (native and replicated) so that every thread executes on a dedicated physical CPU. No background workload interfered with the experiment runs. The results therefore represent the best possible execution time we can achieve on top of ROMAIN.

[27] Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990

5.3.2 Single-Threaded Replication: SPEC CPU 2006

I analyze ROMAIN's execution time overhead for replicating single-threaded applications using the SPEC CPU 2006 benchmark suite. SPEC CPU is a computation-heavy suite that does not intensively communicate with the operating system or other applications. The programs are nevertheless representative for common use cases, such as video decoding, spam filtering, image processing, and computer gaming.

Benchmark Coverage and Methodology SPEC CPU 2006 consists of 29 benchmark programs. ROMAIN is able to replicate 19 of them. The remaining benchmarks either do not work on L4Re at all (2 benchmarks) or acquire too many resources so that replicating them is infeasible on a 32-bit system (8 benchmarks):

- The `453.povray` benchmark requires a working UNIX system providing the `fork()` and `system()` functions. These are not available on L4Re.
- The `483.xalancbmk` uses a deprecated feature of the C++ Standard Template Library (`std::strstream`), which is not provided by L4Re's version of this library.
- As explained in the previous section, the setup I used for my benchmarks is only able to address 3 GiB of memory. This memory needs to suffice for the microkernel, the L4Re resource managers, the ROMAIN master, and the respective benchmarks. While this is enough to run a single instance of each benchmark, some SPEC CPU programs allocate so much memory that there is not enough left to run a second or third replica of this application.

  Under these circumstances, the `410.bwaves`, `434.zeusmp`, and `450.soplex` benchmarks were only able to run as a single replica on top of ROMAIN. The `403.gcc`, `436.cactusADM`, `447.dealII`, `459.GemsFDTD`, and `481.wrf` benchmarks were only able to run up to two replicas and failed to allocate sufficient memory for a third instance.

I executed each benchmark in four modes: natively, and with ROMAIN running one, two, and three replicas. In each mode I executed five iterations of each benchmark and computed the average benchmark execution time over these runs. The standard deviation across all modes was below 1%, except for `433.milc` (8.24%), `454.calculix` (5.65%), `465.tonto` (1.8%), `481.wrf` (6.64%), and `482.sphinx3` (1.1%).

Benchmark Results Figure 5.8 shows the execution time overheads for those benchmarks that ROMAIN was completely able to replicate. The results are normalized to native execution of the benchmark on L4Re and constitute averages over five benchmark runs each. For these 19 benchmarks the geometric mean overhead for running a single replica in ROMAIN is 0.3%. The overhead for double-modular redundancy is 6.4%, and the overhead for triple-modular redundancy is 13.4%.

Remember that these results represent the best-case overhead: all replica threads run on dedicated CPUs with no background load. While the results therefore might appear overly optimistic for real-world deployments, I argue that future hardware platforms are likely to come with an abundant amount of CPUs and therefore running replicas on their own CPUs is feasible. However, computer architects have suggested that such platforms might not be able to power all CPUs at the same time.[28] Therefore, consolidating replicas onto fewer processors while maintaining acceptable overheads is an interesting area for future research.

Figure 5.9 on the next page shows the benchmark results for those benchmarks that were only able to run one or two replicas. If we incorporate these additional results into the overhead calculation, running a single replica in ROMAIN increases execution time by 1.8%. Running two replicas adds 7.3%

[28] Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Annual International Symposium on Computer Architecture*, ISCA'11, pages 365–376, San Jose, California, USA, 2011. ACM

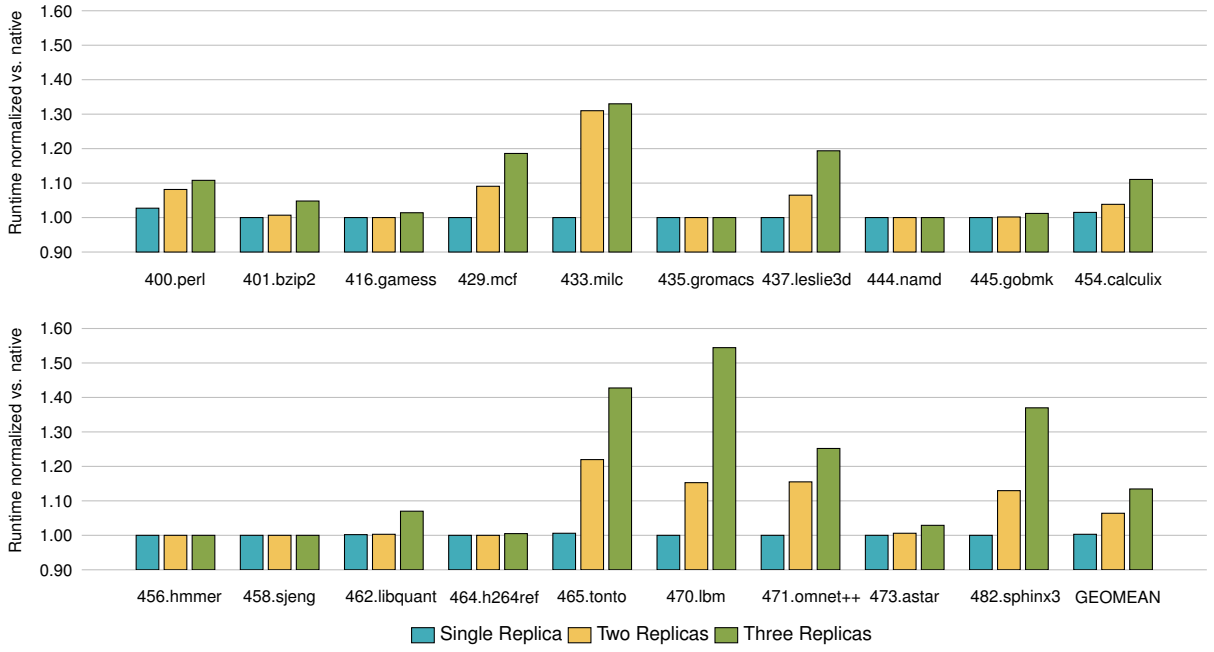

Figure 5.8: SPEC CPU 2006: Normalized execution time overhead for replication

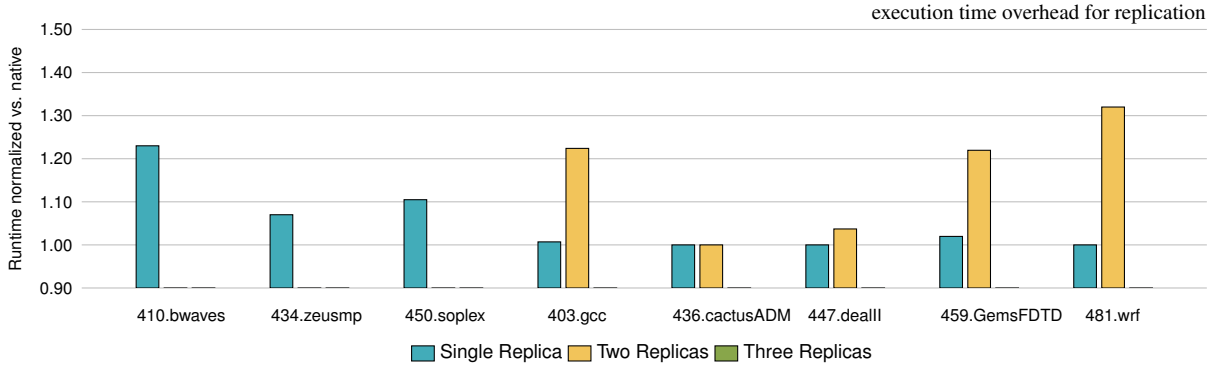

Figure 5.9: SPEC CPU: Overhead for incompletely replicated benchmarks

execution time, whereas triple-modular redundancy remains at a normalized overhead of 13.4%.

Is There a Connection Between Overhead and Master Invocations? In the experiment results we see that some SPEC CPU benchmarks have close to no overhead even when we replicate them, whereas other benchmarks show high overheads. My first intuition to explain the cause of overheads was to look at ROMAIN's internals: the master process is invoked whenever a replica raises an externalization event. Hence, benchmarks with more externalization events should have higher overheads.

To validate this intuition I counted the externalization events (system calls and page faults) for each benchmark and normalized these counts to the benchmarks' native execution time. Figure 5.10 on the following page plots replication overhead as a function of this normalized event rate. I highlight the eight benchmarks with the largest replication-induced execution time overheads. There appears to be a general trend that higher event rates also lead to higher execution time overheads. However, there are also exceptions, which require further investigation.

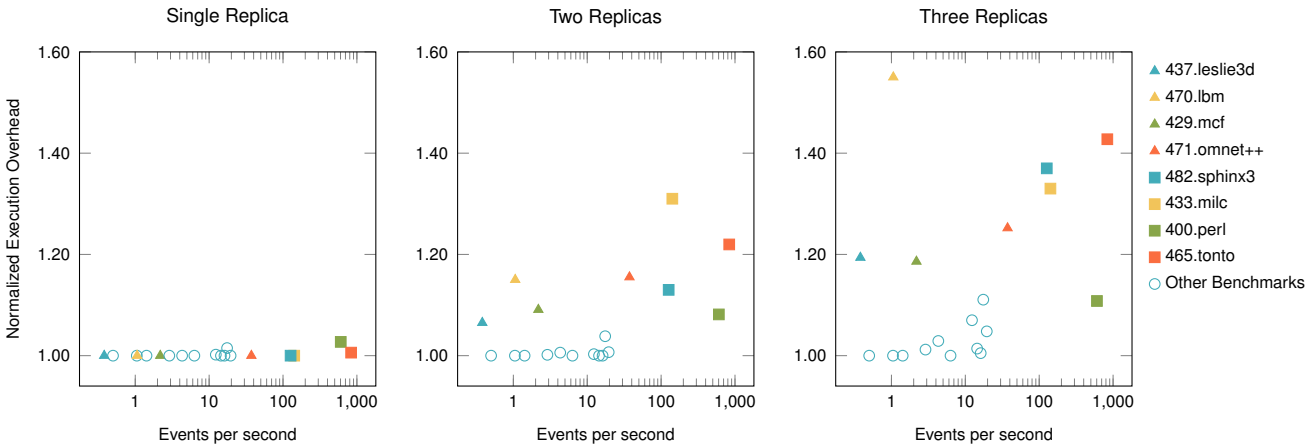

Figure 5.10: Relation between externalization event rate and replication overhead (logarithmic x axis)

`465.tonto` and `400.perl` are the benchmarks with the highest externalization event rates. Nevertheless, `400.perl` shows significantly lower execution time overhead due to replication. Closer investigation into these two benchmarks reveals that they differ in the types of externalization events they raise. `400.perl` performs a large amount of IPC calls to an external log server as it writes data to the standard output. In contrast, `465.tonto` dominantly allocates and deallocates memory regions. As I explained in Section 3.4.1, IPC messages are simply proxied by the ROMAIN master. Unlike IPC messages, memory management requires more work at the master's side, because the master needs to maintain per-replica memory copies as I described in Section 3.5.2. Therefore, replicating `465.tonto` is more expensive than replicating `400.perl`.

The Effect of Caches on Replication Overhead The second interesting result from Figure 5.10 is that the `429.mcf`, `437.leslie3d` and `470.lbm` benchmarks have relatively high replication overheads even though their rate of externalization events is low. My first suspicion was that these benchmarks perform special system calls that require costly handling in the master process. To substantiate this suspicion I measured the time these applications spent in user and master code during replicated execution.

Figure 5.11 on the facing page compares the normalized execution times of the eight SPEC benchmarks with the highest execution time overheads. The figure also shows the ratio of user code execution ( ) versus master execution ( ). We see that different classes of benchmarks exist:

1. The `400.perl` and `465.tonto` benchmarks have nearly constant user execution time while their master execution time increases with increasing number of replicas. Their overhead is therefore explained by additional execution within the master process and these benchmarks confirm the initial intuition.

2. `429.mcf`, `437.leslie3d`, `470.lbm`, `473.omnet++`, and `482.sphinx3` spend nearly all their time executing application code. Their execution

time increases with increasing number of replicas, but this increase cannot be attributed to master execution. My initial intuition is not true for these benchmarks.

3. `433.milc` shows signs of both effects. Its user execution time increases with increasing replicas, but most of the replication overhead still comes from increased time spent executing ROMAIN master code.

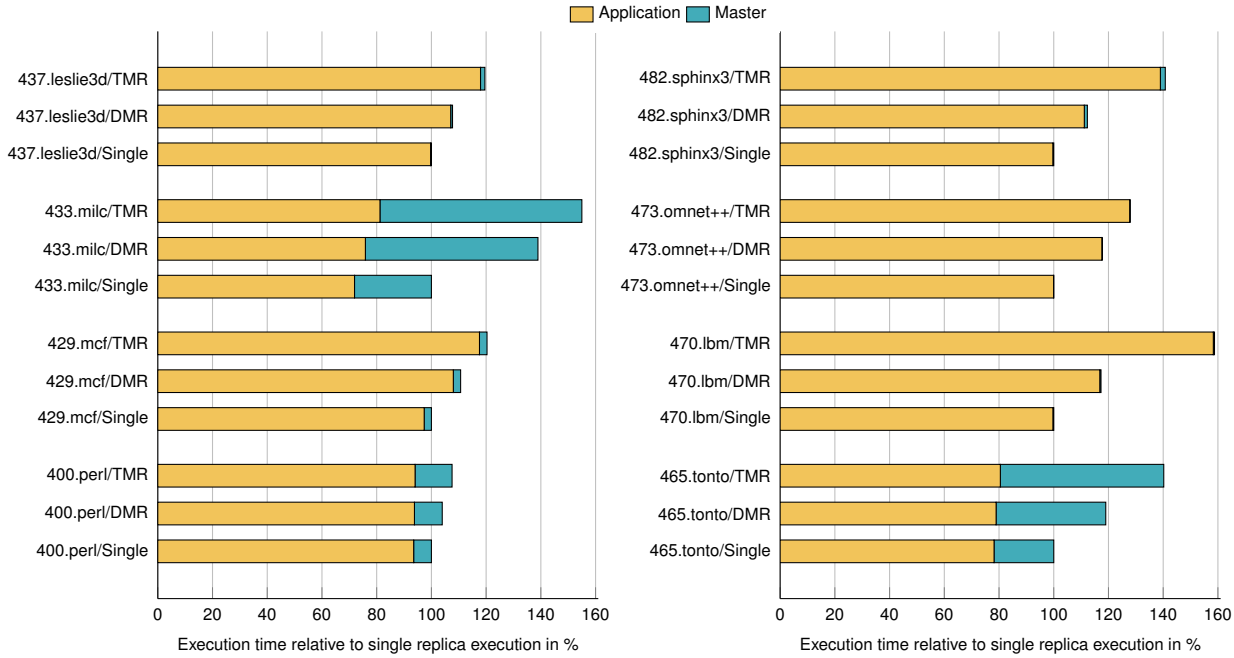

Figure 5.11: SPEC CPU: Breakdown of user vs. master ratio of overhead

The fact that replication-induced execution time overheads mainly appear in user code indicates that these overheads may have hardware-level causes. I used hardware performance counters available in the test machine to analyze the last-level cache miss rate of each benchmark. Table 5.2 shows the total number of last-level cache misses when running one, two, and three replicas as well as the relative increase of cache misses in comparison to single-replica execution. We see that the cache miss rates increase manifold when running multiple replicas.

For the `433.milc` and `465.tonto` benchmarks this increase in cache misses is a result of master interaction. Whenever the applications raise an externalization event, we switch to the ROMAIN master process. As the master's address space is disjoint from the replica address space, caches need to be flushed during this context switch. This leads to increased last-level cache misses, because after switching back to the benchmark previously cached data needs to be read from main memory again.

The increased cache miss rates also explain increased overheads for the other SPEC benchmarks. Here, the miss rates rise because multiple instances of the same application compete for the L3 cache, which can only fit 12 MiB of data. Hence, where a single replica can still use all of this cache, three replicas need to share the cache. In the best case this leaves only 4 MiB of L3 cache for each replica.

Benchmark	Single Replica	Two Replicas (vs. single replica)	Three Replicas (vs. single replica)
`400.perl`	0.5×10^6	2.9×10^6 (x 5.3)	4.6×10^6 (x 8.44)
`429.mcf`	9.87×10^6	15.6×10^6 (x 1.6)	19.44×10^6 (x 1.99)
`433.milc`	14.14×10^6	14.77×10^6 (x 1.04)	19.1×10^6 (x 1.35)
`437.leslie3d`	5.49×10^6	7.34×10^6 (x 1.34)	8.46×10^6 (x 1.54)
`465.tonto`	0.07×10^6	0.58×10^6 (x 7.99)	0.96×10^6 (x 13.28)
`470.lbm`	3.11×10^6	6.9×10^6 (x 2.22)	11.48×10^6 (x 3.69)
`471.omnet++`	5.2×10^6	7.97×10^6 (x 1.53)	8.59×10^6 (x 1.65)
`482.sphinx3`	0.19×10^6	4.93×10^6 (x 26.51)	9.42×10^6 (x 50.65)

Table 5.2: Last-Level (L3) Cache Miss Rates (Misses per second of execution time) for eight SPEC CPU benchmarks. Miss rates are normalized to single-replica execution time.

Given my observations I hypothesize that the SPEC benchmarks in question are cache-bound. Replicating them reduces the available cache per replica and thereby impacts execution time even for benchmarks that do not heavily interact with the ROMAIN master. Apart from my measurements, this hypothesis is supported by a similar analysis by Harris and colleagues.[29]

[29] Tim Harris, Martin Maas, and Virendra J. Marathe. Callisto: Co-Scheduling Parallel Runtime Systems. In *European Conference on Computer Systems*, EuroSys '14, Amsterdam, The Netherlands, 2014. ACM

Better Performance Using Reduced Cache Miss Rates I demonstrated in Section 4.3.4 that replication overhead can be reduced by placing replica threads on the same CPU socket. This happened because this way of placing replicas reduced the cost of synchronization messages that are sent between replicas for every externalization event. It turns out that this optimization only benefits communication-intensive applications, such as the microbenchmark I used to evaluate multithreaded replication. In contrast, my analysis in this section indicates that this strategy might not be ideal for cache-bound applications, such as at least some of the SPEC CPU benchmarks.

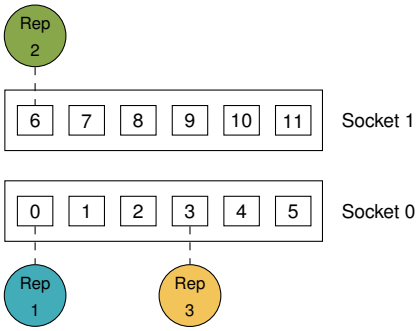

Figure 5.12: Assigning replicas to available CPUs in order to optimize L3 cache utilization

Given the test machine described in Section 5.3.1, forcing all replicas to run on the first socket and share an L3 cache leaves the second socket with an additional 12 MiB of cache completely unused. If my hypothesis is true, distributing replicas across all sockets should reduce replication overhead by doubling the amount of available L3 cache. To confirm this theory, I adapted ROMAIN's replica placement algorithm as shown in Figure 5.12: I now place the first and third replica of an application on socket 0, whereas the second replica always runs on socket 1.

Using this adjusted setup, I repeated my experiments for the eight SPEC benchmarks in question and show the improved execution time overheads in Figure 5.13 on the next page. The numbers in the figure show the relative improvement for the given setup compared to the execution times shown in Figure 5.8 on page 111.

We see that cache-aware CPU assignment reduces replication overhead for five of the eight benchmarks. Running two replicas is nearly as cheap as running a single one, because the second replica uses a dedicated L3 cache and does not interfere with the first replica. The exceptions to this observation are once again `433.milc` and `465.tonto`, whose overheads do not differ from the previous runs. This confirms the assumption that these benchmarks

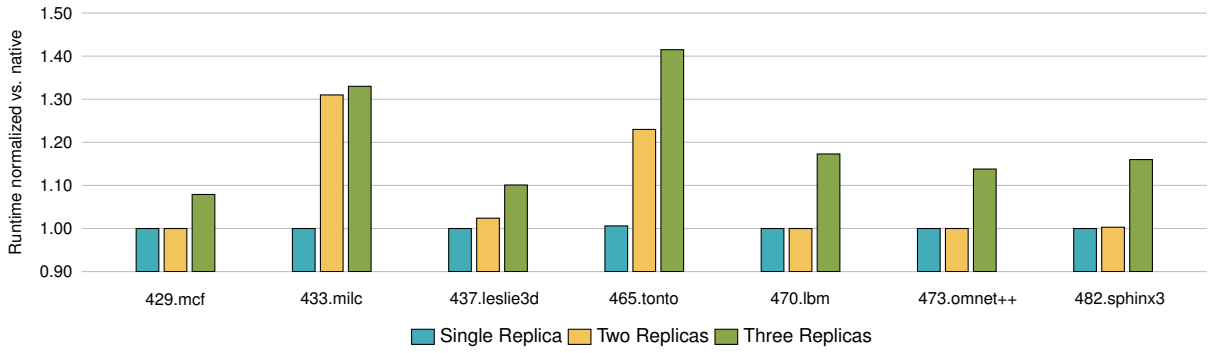

Figure 5.13: SPEC CPU: Execution time overhead with improved replica placement

are dominated by interactions with the ROMAIN master instead of suffering from cache thrashing.

SUMMARY: ROMAIN provides efficient replication for single-threaded applications. Based on measurements using the SPEC CPU 2006 benchmark suite, the geometric mean overhead for running two replicas is 7.3%. Three replica instances lead to an overhead of 13.4%.

In addition to interaction with the ROMAIN master process, cache utilization has a major impact on replication performance. While a single application instance may fit its data into the local cache, running multiple instances may exceed the available caches. This problem may be mitigated by cache-aware placement of replicas on CPUs.

5.3.3 Multithreaded Replication: SPLASH2

The SPEC CPU benchmarks I used in the previous section are all single-threaded and hence do not leverage ROMAIN's support for replicating multi-threaded applications that I introduced in Chapter 4. To cover such programs with my evaluation, I evaluate ROMAIN's overhead using the SPLASH2 benchmark suite.[30] As previous research found, these benchmarks contain data races[31] and thereby violate my requirement that multithreaded applications need to be race-free in order to replicate them. I therefore analyzed these benchmarks using Valgrind's data race detector[32] and removed data races from the `Barnes`, `FMM`, `Ocean` and `Radiosity` benchmarks.[33]

I compiled the benchmarks using cooperative determinism provided by the replication-aware `libpthread_rep` library I introduced in Section 4.3.5. Figure 5.14 shows the execution time overheads for these benchmarks running with two application threads normalized to native execution without ROMAIN. The geometric mean overheads are 13% for double-modular redundant execution and 24% for triple-modular redundant execution.

These overheads become higher if we run four application threads in each benchmark. The overheads, shown in Figure 5.15, are 22% for DMR execution and 65% for TMR execution.

Investigating the sources of overhead I found similar causes as for single-threaded replication. `FMM`, `Ocean`, `FFT`, and `Radix` allocate a significant

[30] Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *SIGARCH Comput. Archit. News*, 23(2):24–36, May 1995

[31] Adrian Nistor, Darko Marinov, and Josep Torrellas. Light64: lightweight hardware support for data race detection during systematic testing of parallel programs. In *International Symposium on Microarchitecture*, MICRO 42, pages 541–552, New York, NY, USA, 2009. ACM

[32] Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Workshop on Binary Instrumentation and Applications*, WBIA'09, pages 62–71, New York, NY, USA, 2009. ACM

[33] I make the respective patches available at `http://tudos.org/~doebel/emsoft14`.

amount of memory and the measurements show that most of their overhead comes from the initialization phase where these memory resources are allocated. These benchmarks also measure their compute times without initialization. For these measurements the data shows TMR overheads of less than 5% with two worker threads and less than 10% with four worker threads.

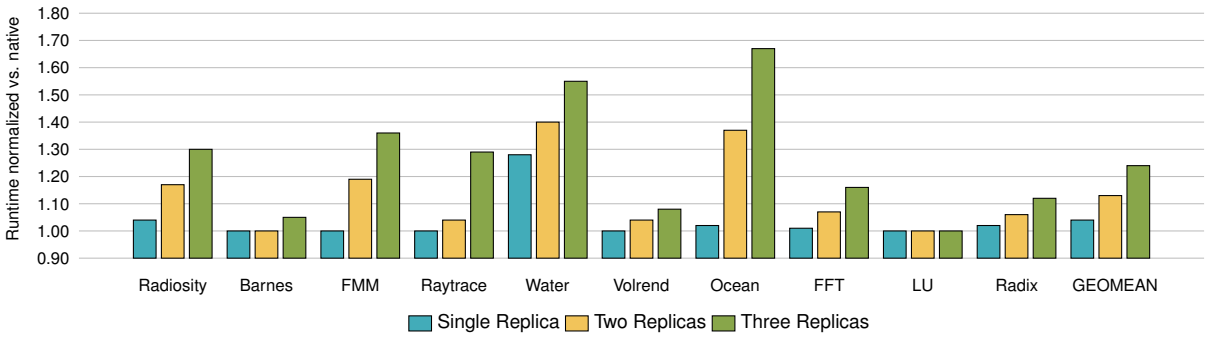

Figure 5.14: SPLASH2: Replication overhead for two application threads

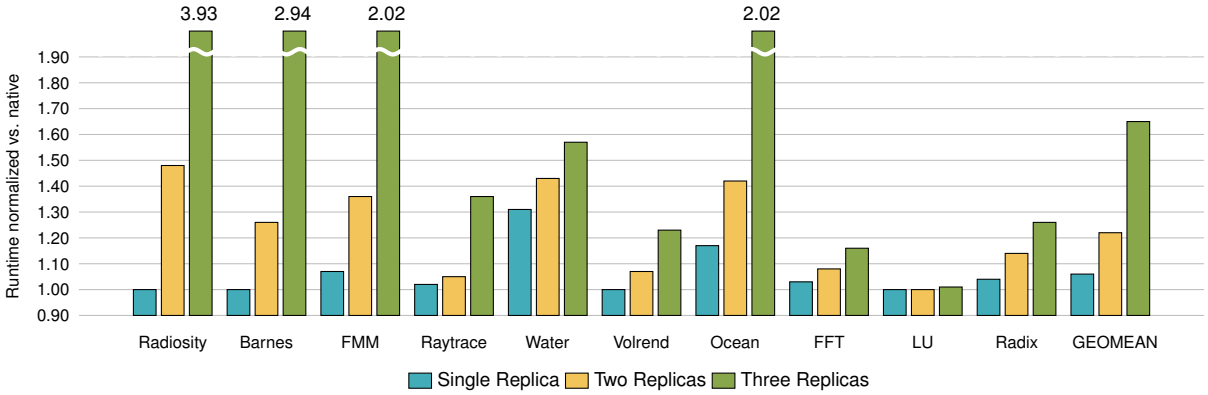

Figure 5.15: SPLASH2: Replication overhead for four application threads

Benchmark	Lock Operations per second
Radiosity	13,800,000
Barnes	1,455,000
FMM	1,040,000
Water	850,000
Raytrace	451,000
Volrend	169,000
Ocean	7,100
FFT	141
LU	75
Radix	7

Table 5.3: Rate of lock operations for the SPLASH2 benchmarks executing natively with two worker threads

The `Barnes` benchmark is interesting because it has fairly low overhead when replicating the version using two worker threads, but shows drastic increases in overhead when running four workers. Throughout its execution, the benchmark touches nearly 900 MiB of memory and therefore three replicas use most of the available RAM in the test computer. I measured the L3 cache miss rates of each benchmark run and found that the total number of L3 misses across all cores and replica threads is identical when running a single replica of the two-worker and the four-worker versions. However, when running three replicas, the two-worker version doubles its L3 miss rate whereas the four-worker version's L3 miss rate multiplies by five. This effect explains part of the huge overhead of the `Barnes` benchmark with four worker threads.

The same findings can however not be applied to the `Radiosity`, `Raytrace`, and `Water` benchmarks. They show high replication-induced overheads even though their memory footprint is low – they use only around 40 MiB of memory. I suspected concurrency to be the replication bottleneck here and therefore measured the rate of lock/unlock operations all benchmarks perform when executing natively with two threads. I show these rates in Table 5.3.

We see that the benchmarks in question are among the ones with the highest rates of lock operations. I therefore attribute the overheads of these benchmarks to their lock ratio. Note that the `Barnes` and `FMM` benchmarks are also in the group with high lock rates, but in their case locking-induced overheads and memory-related overheads overlap.

Figure 5.16 shows the reason lock operations imply overhead. We see three replicas with two threads each that compete for access to a lock L. In the example $R_{1,1}$ executes the lock acquistion first and therefore becomes the lock owner as explained in Section 4.3.5. This lock ownership ensures that replicas $R_{2,1}$ and $R_{3,1}$ from the same thread group will also acquire the lock even though from the perspective of timing replicas $R_{2,2}$ and $R_{3,2}$ reach the respective lock operation first.

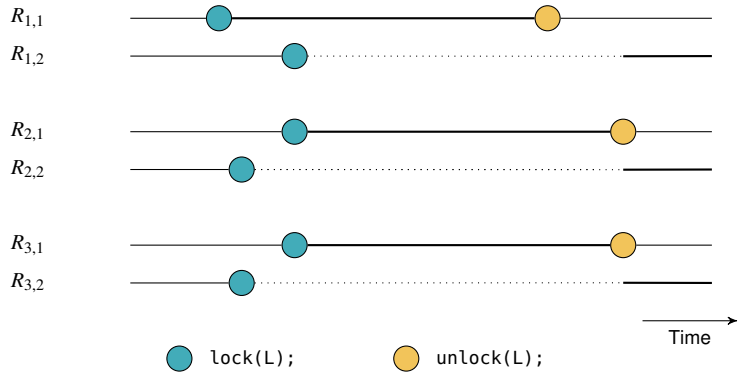

Figure 5.16: Multithreaded replication: The first replica to obtain a lock slows down all other replicas if their threads have different timing behavior.

Using this approach, ROMAIN ensures deterministic execution of multithreaded replicas. However, the example shows that this mechanism also reduces concurrency, because replicas $R_{2,2}$ and $R_{3,2}$ have to wait even though they could enter their critical section in the native case. This situation may occur at any replicated lock operation and is more likely to happen if the replica performs more lock operations. Hence, lock-intensive applications show higher replication-induced overheads and these overheads increase with more replicated threads.

For completeness, I also show the execution time overheads when running the benchmarks using enforced determinism, which I introduced in Section 4.3.3 and which reflects every lock operation as an externalization event to the master. Figure 5.17 shows the execution time overheads when running SPLASH2 with two worker threads. We see that this approach enlarges the overheads we observed for cooperative determinism.

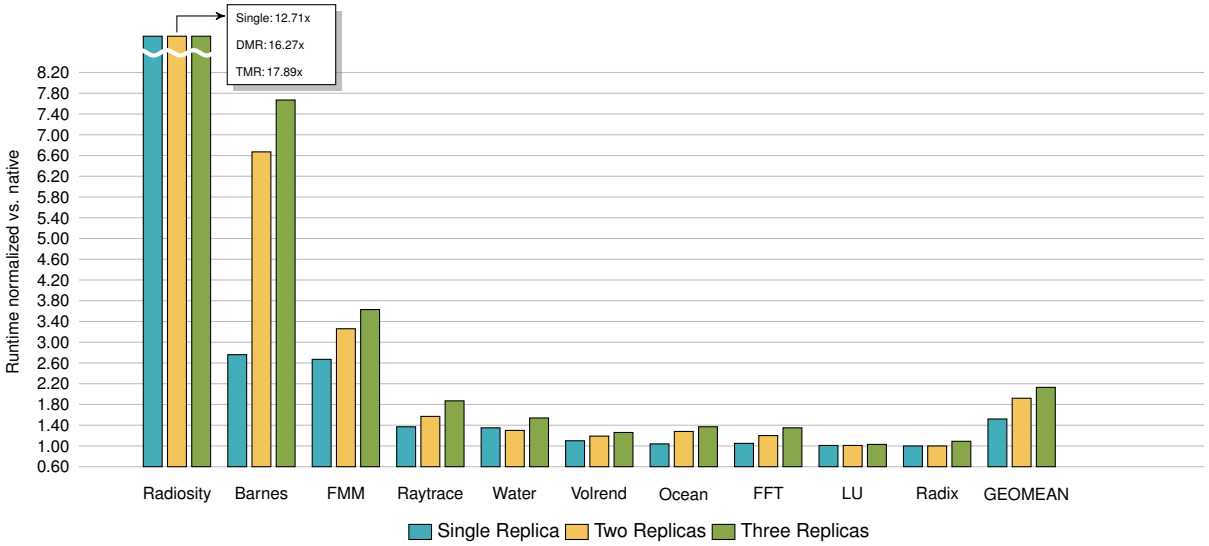

Figure 5.17: SPLASH2: Replication overhead for two application threads using enforced determinism

SUMMARY: The execution time overhead of ROMAIN for replicating multithreaded applications is higher than for single-threaded ones. Using the SPLASH2 benchmark suite as a case study I showed that in addition to the previously measured overhead sources (system calls and memory effects), multithreaded applications also suffer from replication overhead due to their lock density. High lock densities are known to be a scalability bottleneck in concurrent applications and replication magnifies this effect as replicas have to wait for each other in order to deterministically acquire locks.

5.3.4 *Application Benchmark: SHMC*

In Section 3.6 I showed that ROMAIN also supports replicating applications that require access to memory regions shared with other processes. In contrast to replica-private memory, these accesses must be intercepted by the master process as these operations constitute externalization events as well as input to the replicas.

I evaluate shared-memory performance overhead using a worst-case scenario depicted in Figure 5.18. Two applications, a sender and a receiver, share a 512 KiB shared memory channel. The sender uses L4Re's packet-based shared memory ring buffer protocol, SHMC, to send 400 MiB of payload data to the receiver. The receiver only reads the data and does no processing on it. The scenario therefore evaluates the best possible throughput we can achieve in such a scenario.

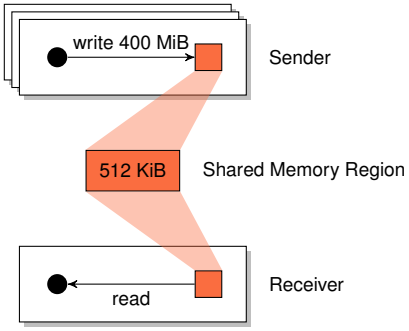

Figure 5.18: SHMC Application Benchmark

I executed the benchmark natively on L4Re and varied the packet size used by the SHMC protocol from 32 bytes up to 2,048 bytes. Every packet going through the channel requires additional work by the protocol implementation. Hence, increasing the packet size will also increase SHMC channel throughput because the implementation needs to perform less management work.

I then replicated the sender application using ROMAIN and performed the same variation in packet size as in the native case. Figure 5.19 shows the obtained throughputs, distinguishes between the two shared memory interception solutions (trap & emulate and copy & execute) I presented in Section 3.6 and relates them to the results for native execution.

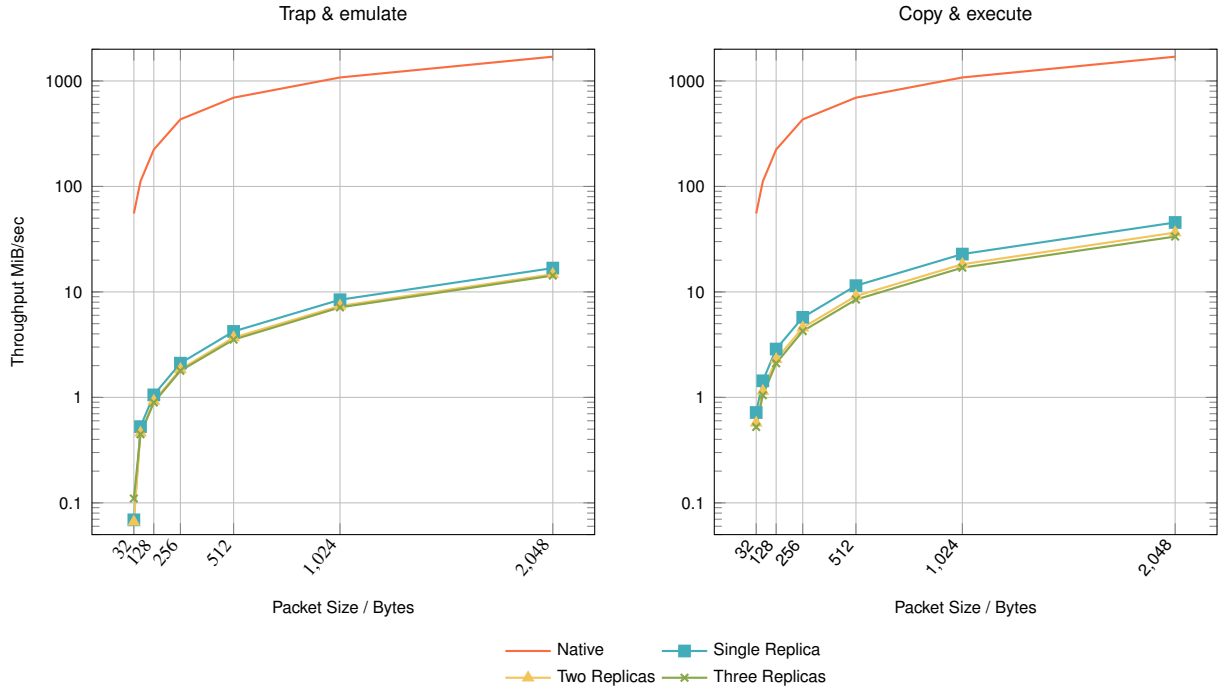

Figure 5.19: Throughput achieved when replicating an application transmitting data through a shared-memory channel. (Note the logarithmic y axis.)

The results show that replicating shared-memory accesses is costly. While native execution achieves throughputs between 55 MiB/s (32 byte packets) and 1.7 GiB/s (2,048 byte packets), replicated shared memory accesses are 2 orders of magnitude slower. Trap & emulate replication achieves throughputs between 110 KiB/s (32 byte packets) and 14,7 MiB/s (2,048 byte packets) for three replicas. Copy & execute – which avoids using an instruction emulator to perform memory accesses – performs significantly better and achieves 520 KiB/s for 32 byte packets and 33,5 MiB/s for 2,048 byte packets.

In both native and replicated execution increasing the packet size also increases throughput. I inspected the protocol closer and found that for every packet, SHMC performs 23 shared-memory accesses. One of these accesses is a `rep movs` instruction for which we saw in Section 3.6 that its overhead is negligible. The remaining 22 memory accesses are however random accesses to SHMC's management data structures. Handling such instructions is expensive and explains why the replication-induced overhead for small packets is higher (factor 100 for copy & execute TMR) than for larger packets (factor 50 for copy & execute TMR).

SUMMARY: While ROMAIN is capable of replicating applications that use shared memory, replication will significantly slow down these applications.

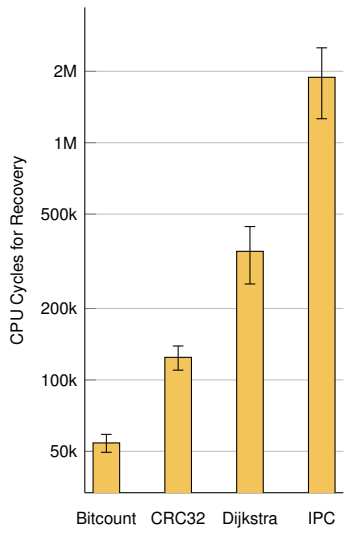

Figure 5.20: Average time (CPU cycles) to recover from a detected error in the fault injection benchmarks (Logarithmic Y axis, error bars represent standard deviation across 10 experiments.)

5.3.5 *The Cost of Recovery*

As a next experiment I investigated the cost of recovering from a detected error. For this purpose I executed those benchmarks on my test machine that I used for fault injection experiments in Section 5.2. I randomly selected ten injections into general purpose registers that lead to a detected error in my previous experiments and injected those errors manually during native execution using a special ROMAIN fault injection observer. I injected one such error into every benchmark run and repeated each injection ten times to compute an average of the time it took ROMAIN to perform recovery after the error had been detected. Figure 5.20 shows the averages I observed.

Recovering from an error in ROMAIN consists of bringing the register states and memory content of all replicas into an identical state. This process is dominated by the cost of memory recovery. Returning architectural registers into the same state cost around 700 CPU cycles in every experiment and I do not show this in the plot. The remaining thousands to millions of cycles are spent correcting memory content. As I described in Section 3.5, the ROMAIN master process has access to all memory of the replicas. Hence, this part of the recovery process maps to a set of `memcpy` operations from a correct into a faulty replica.

I suspected that the difference in recovery times we see across the benchmarks is related to the amount of memory these benchmarks consume. Two facts support this hypothesis:

1. In my setup, Bitcount consumes 192 KiB of memory, CRC32 consumes 434 KiB, Dijkstra uses 1.3 MiB and the IPC benchmark allocates 3.4 MiB. This is reflected by the respective recovery times.

2. The Dijkstra and IPC benchmarks show a large standard deviation. This is due to the fact that in each of those two experiments, one out of the ten injections constantly showed much lower recovery times (60,000 cycles vs. 350,000 cycles (Dijkstra) and 1.8 million cycles (IPC)) than the remaining 9 runs. In both cases, the fast recovery runs stem from faults that were injected early within the benchmark's `main()` functions, that is before the benchmarks allocate all their memory. Hence, recovery does not need to copy as much data as in the remaining runs.

To further validate that recovery time is dominated by copying memory content, I created another microbenchmark: An application allocates a memory buffer and touches all data in this buffer once, so that the ROMAIN master has to map the memory pages into all replicas of the program. Thereafter, I flip a bit in the first replica, which is detected and corrected by the ROMAIN master.

I varied the buffer size from 1 MiB to 500 MiB, executed ten fault injection runs for each size, and plot the average recovery times and the resulting recovery throughput in Figure 5.21. The standard deviations in this experiment were below 0.1% and are therefore not plotted. We see that recovery time grows linearly with the replicated application's memory footprint and even recovering a replicated application with 500 MiB of memory is done within 130 ms. This forward recovery mechanism is extremely fast compared to traditional checkpoint/rollback mechanisms. For instance, DMTCP reports

a restart overhead between one and four seconds for a set of application benchmarks.[34] Note that this advantage in recovery speed is not a special feature of ROMAIN, but is inherent to all replication-based approaches that provide forward recovery.

Recovery throughput – that is the amount of memory recovered per second – is high for buffer sizes below 4 MiB. Remember, my test machine has 12 MiB of last-level cache per socket. When running three replicas with a memory footprint of up to 4 MiB, this data fits into the L3 cache and in fact it is prefetched into this cache because the benchmark touches all memory before injecting a fault. For larger footprints, recovery becomes memory-bound. The larger the footprint becomes, the closer recovery throughput gets to around 3.8 GiB/second. I measured the theoretical optimum for `memcpy()` on my test machine to be 4.5 GiB/second. The difference to recovery throughput comes from the fact that ROMAIN's recovery does not copy a single continuous space of memory, but several smaller ones.

[34] Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009

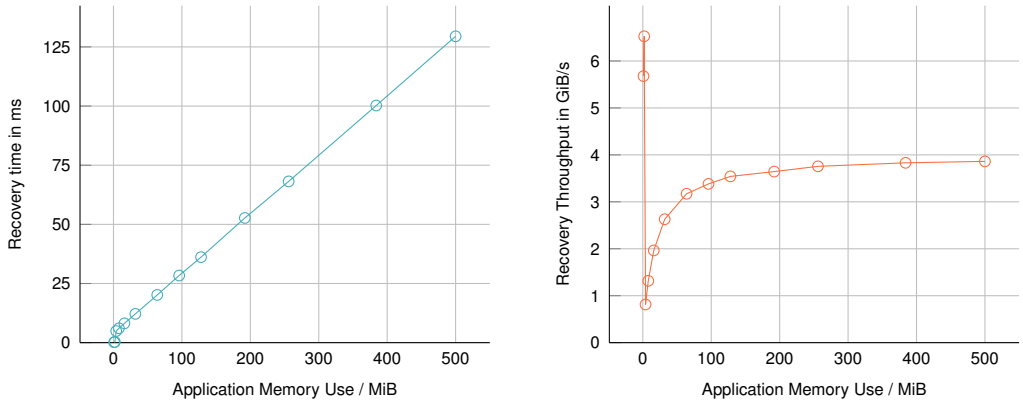

Figure 5.21: Recovery time depending on application memory footprint

SUMMARY: ROMAIN provides forward recovery by majority voting. Recovery times relate linearly to the replicated application's memory footprint. Forward recovery takes place in few milliseconds whereas state-of-the-art backward recovery may take an order of magnitude more time.

5.4 Implementation Complexity

ROMAIN adds complexity in terms of source code to FIASCO.OC's L4 Runtime Environment. I measure this complexity by giving the source lines of code (SLOC) required to implement the features described in this thesis. I used David A. Wheeler's `sloccount`[35] to evaluate these numbers for ROMAIN and show the results in Table 5.4.

The two components required to implement replication as an OS service are the ROMAIN master process (7,124 SLOC) and the replication-aware `libpthread` library (756 SLOC). I furthermore categorized the master process' code into five categories to gain more insight into where implementation complexity originates from:

[35] `http://www.dwheeler.com/sloccount`

1. *Replica Infrastructure* subsumes code for loading an application binary during startup, creating the respective number of replicas and maintaining each replica's memory layout.
2. *Fault Observers* lists the implementation complexity of the specific event observers I introduced in Section 3.3. While most observers are fairly small and comprehensible, system call handling and the implementation of deterministic locking are substantially more complex. Observers for debugging replicas are available as well but can be disabled at compile time to reduce complexity.
3. *Event Handling* includes all code to intercept replicas' externalization events and perform error detection and recovery using majority voting.
4. *Shared Memory Interception* comprises the implementations of replicated shared memory access I described in Section 3.6. The table shows that the source code complexity for implementing each interception method (trap & emulate and copy & execute) in ROMAIN is about the same. Note however, that the trap & emulate approach requires an additional x86 disassembler library. For this purpose I used the `libudis86` disassembler, which adds another 2,187 lines of code to the implementation.
5. *Runtime Support* includes all remaining code, such as master startup and logging. This runtime support furthermore includes Martin Kriegel's implementation of a hardware watchdog to bound error detection latencies as described in Section 3.8.2.

ROMAIN Component	SLOC
Master	**7,124**
Replica Infrastructure	2,652
Binary Loading	385
Replica Management	1,484
Memory Management	783
Fault Observers	1,962
Observer Infrastructure	190
Time Input	112
Trap Handling	82
Debugging	438
Page Fault Handling	186
Locking	430
System Calls	524

ROMAIN Component	SLOC
Master (ctd.)	
Event Handling	511
Shared Memory Int.	950
Common	378
Trap & emulate	262
Copy & execute	310
Runtime Support	1,049
Startup, Logging	562
Hardware Watchdog	487
`libpthread_rep`	**756**

Table 5.4: ROMAIN: Source Lines of Code

5.5 Comparison with Related Work

With the experiments in this chapter I demonstrated that ROMAIN provides an operating system service that efficiently detects and recovers from hardware-induced errors in binary-only user applications. I now compare ROMAIN to other software-implemented fault tolerance techniques. I do not compare against hardware-level techniques as a major point in software-based fault tolerance methods is to avoid custom hardware features and solely work

using software primitives. For the comparison I selected ROMAIN and five additional mechanisms that provide this property:

1. *Software-Implemented Fault Tolerance (SWIFT)* is a compiler technique that duplicates operations using different hardware resources (such as registers) and compares the results of these operations to detect errors.[36]
2. *Encoding Compiler – ANB (EC-ANB)* is a compiler that arithmetically encodes operands and control flow of an instrumented program.[37] This approach provides higher error coverage than SWIFT.
3. *Process-Level Redundancy (PLR)* pioneered the idea of using operating system processes as the sphere of replication, which ROMAIN builds upon.[38]
4. *Efficient Software-Based Fault Tolerance on Multicore Platforms (EFTMP)* provides applications with a set of system call wrappers and a replication-aware deterministic thread library to support replication of multithreaded applications.[39]
5. *Runtime Asynchronous Fault Tolerance (RAFT)* improves the speed of PLR by speculatively executing system calls instead of waiting for all replicas to reach their next system call for state comparison.[40]

Table 5.5 summarizes the comparison between ROMAIN and these other mechanisms. The numbers and properties shown in the table are taken from the scientific papers and this thesis respectively. I explain the different points in more detail below.

[36] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization*, CGO '05, pages 243–254, 2005

[37] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *International Conference on Computer Safety, Reliability and Security*, Safecomp'10, Vienna, Austria, 2010

[38] 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, 2009

[39] Hamid Mushtaq, Zaid Al-Ars, and Koen L. M. Bertels. Efficient Software Based Fault Tolerance Approach on Multicore Platforms. In *Design, Automation & Test in Europe Conference*, Grenoble, France, March 2013

[40] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. Runtime Asynchronous Fault Tolerance via Speculation. In *International Symposium on Code Generation and Optimization*, CGO '12, pages 145–154, 2012

	Compiler-Based Approaches		Replication-Based Approaches			
	SWIFT	**EC-ANB**	**PLR**	**EFTMP**	**RAFT**	**ROMAIN**
Covers Memory Errors	No	Yes	Yes	No	Yes	Yes
Covers Processor SEUs	Yes	Yes	Yes	Yes	Yes	Yes
Error Coverage	>92%	>99%	100%	Unknown	100%	100%
Recovery Built In	No	No	Yes	No	No	Yes
Memory Overhead	1x	2x	Nx	2x	2x	Nx
Exec. Overhead, Single	41%	2x - 75x	17% (detection) 41% (recovery)	Unknown	5 - 10%	7% (detection) 13% (recovery)
Support for Multithreading	Unknown	Unknown	No	Yes	No	Yes
Exec. Overhead, Multithreading	Unknown	Unknown	Not supported	< 18%	Not supported	22% (detection) 65% (recovery)
Source Code Required	Yes	Yes	No	No	No	No

Table 5.5: Comparison of ROMAIN and other software fault tolerance methods

Note that a direct comparison of overhead numbers and coverage rates is not always possible: SWIFT was for instance implemented on the Itanium architecture whereas all other tools were implemented for x86. Error coverage rates were computed from different kinds of experiments: SWIFT, PLR, and RAFT used a similar random sampling approach and injected several thousands of bit flips into application memory, general purpose and control registers. EC-ANB used a custom fault injector that modified computations, operators, and memory operations at runtime. ROMAIN's was tested by injecting errors into memory and general purpose registers using the FAIL* framework.

Error Coverage and Recovery All mechanisms in this comparison support detection of SEUs in the processor, such as incorrect computations, register SEUs, and control flow errors. With the exception of SWIFT and EFTMP, all mechanisms furthermore support detection of memory errors. SWIFT explicitly rules out memory errors and requires ECC-protected RAM. EFTMP simply ignores the fact that it is not safe against memory errors as I explained in Section 4.2.

In contrast to all other mechanisms, PLR and ROMAIN support recovery from errors using majority voting as a built-in feature. This feature is paid for with a higher memory overhead, because at least three instances of a replica need to be maintained to allow voting. All other mechanisms rely on orthogonal methods for recovery, such as a working checkpointing mechanism, to be available. Using such recovery methods will lead to additional overheads (e.g., for taking checkpoints and re-executing from the last checkpoint during recovery) that are not included in the overhead numbers provided by the respective authors.

Overheads SWIFT is the only mechanism in this analysis that does not require additional memory. As explained before, PLR and ROMAIN multiply memory consumption by the number of replicas they are running. EC-ANB doubles the amount of memory because it transforms all 32-bit data words into 64-bit encoded data. EFTMP and RAFT run two replicas of an application and hence require double the amount of memory.

All mechanisms except EC-ANB have execution time overheads between 5% and 41% when running single-threaded benchmarks. ROMAIN's 13% overhead for running three replicas is competitive and has the advantage of providing forward recovery.

ROMAIN was evaluated on top of FIASCO.OC whereas the other mechanisms were analyzed on Linux. These systems differ in the type of externalization events that need to be handled, which may impact execution time overheads. However, Florian Pester's Linux version of ROMAIN, which I mentioned in Section 3.3, shows similar overheads (16.3% for TMR execution) for the same benchmarks (SPEC CPU 2006) on the same test machine that I used for my evaluation.[41]

[41] Florian Pester. ELK Herder: Replicating Linux Processes with Virtual Machines. Diploma thesis, TU Dresden, 2014

Supported Applications Only EFTMP and ROMAIN support protection of multithreaded applications. SWIFT and EC-ANB do not discuss this problem and have not been evaluated using multithreaded benchmarks. PLR and RAFT

explicitly rule out replication of multithreaded programs for their prototypes. EFTMP provides lower runtime overheads than ROMAIN. However, as I pointed out above, ROMAIN covers a wider range of hardware errors and provides forward recovery.

As a last point of comparison, compiler-based solutions, such as SWIFT and EC-ANB, require the whole source code of the protected application to be recompiled. External binary code, such as the standard C library, remains unprotected by these mechanisms. In contrast, PLR and RAFT use binary recompilation to protect complete application binaries. EFTMP protects applications by requiring them to use a custom-designed `libpthread` library. ROMAIN avoids expensive recompilation by running at the operating system level and leveraging FIASCO.OC's virtualization support to intercept externalization events.

SUMMARY: ROMAIN combines the advantages of other software-implemented fault tolerance mechanisms and addresses their deficiencies:

- ROMAIN protects binary-only applications and does not require source code availability in contrast to compiler-based solutions.
- ROMAIN protects both single-threaded and multithreaded applications against CPU and memory errors, whereas most other mechanisms were only tested using single-threaded workloads.
- ROMAIN provides built-in forward recovery using majority voting, while most other techniques only allow for error detection. As a matter of flexibility, ROMAIN can however also be executed in error detection mode, which reduces its overhead, but in turn requires the combination with an external error recovery mechanism, such as application-level checkpointing.

While ROMAIN provides efficient and flexible error detection and recovery for unmodified user-level applications, there is a remaining drawback that my solution shares with most other software fault tolerance mechanisms: They only protect application code and do not detect and recover from errors in the fault-tolerant runtime or the underlying operating system kernel. This is a serious problem, because those components are crucial for the correct functioning of the whole system. I will therefore continue my thesis with an investigation of this Reliable Computing Base.

6
Who Watches the Watchmen?

At several points in the previous chapters we realized that ROMAIN relies on specific hardware and software features to function correctly. Protecting this *Reliable Computing Base (RCB)* remains an open issue. In this chapter I argue that every software-implemented fault tolerance mechanism has a specific RCB and investigate what constitutes ROMAIN's RCB. After having a closer look at what constitutes the RCB, I present three case studies which future work may extend in order to achieve full system protection.

1. As the OS kernel is the largest part of ROMAIN's RCB, I first study the vulnerability of the FIASCO.OC microkernel to understand how RCB code reacts to hardware faults and give ideas about how the kernel can be better protected.
2. Thereafter, I investigate how current hardware trends can lead to an architecture providing a mixture of reliable and less reliable CPU cores and how the ASTEROID OS architecture can benefit from such a platform.
3. Lastly, I point out that ASTEROID's software-level RCB is purely open-source software and therefore allows the application of compiler-based fault tolerance mechanisms. Lacking a suitable compiler, I use simulation experiments to estimate the performance impact applying such mechanisms could have on ASTEROID as a whole.

I developed the concept of the Reliable Computing base together with Michael Engel.[1] Two of the case studies I present in this chapter were previously published in HotDep 2012[2] (mixed-reliability hardware) and SOBRES 2013[3] (estimating the effect of compiler-based RCB protection).

[1] Michael Engel and Björn Döbel. The Reliable Computing Base: A Paradigm for Software-Based Reliability. In *Workshop on Software-Based Methods for Robust Embedded Systems*, 2012

[2] Björn Döbel and Hermann Härtig. Who Watches the Watchmen? – Protecting Operating System Reliability Mechanisms. In *Workshop on Hot Topics in System Dependability*, HotDep'12, Hollywood, CA, 2012

[3] Björn Döbel and Hermann Härtig. Where Have all the Cycles Gone? – Investigating Runtime Overheads of OS-Assisted Replication. In *Workshop on Software-Based Methods for Robust Embedded Systems*, SOBRES'13, Koblenz, Germany, 2013

[4] Department of Defense. *Trusted Computer System Evaluation Criteria*, December 1985. DOD 5200.28-STD (supersedes CSC-STD-001-83)

6.1 The Reliable Computing Base

Computer systems security is often evaluated with respect to the *Trusted Computing Base (TCB)*. The US Department of Defense's "Trusted Computer Systems Evaluation Criteria"[4] define the TCB as

> "[...] all of the elements of the system responsible for supporting the security policy and supporting the isolation of objects (code and data) on which the protection is based. [...] the TCB includes hardware, firmware, and software critical to protection and must be designed and implemented such that system elements excluded from it need not be trusted to maintain protection."

Hence, the TCB comprises those hardware and software components that need to be trusted in order to obtain a secure service from a computer system.

Industry experts estimate that the main source of security vulnerabilities in modern system are programming or configuration errors at the software level.[5] As the number of software errors correlates with code size,[6] security research focuses on minimizing the amount of code within the TCB in order to improve system security.[7]

When implementing software-level fault tolerance mechanisms – such as ROMAIN – we face a similar problem: we rely on the correct functioning of hardware and software components in order to implement fault tolerance. In the case of ROMAIN these components are:

- *Memory Management Hardware*: ROMAIN provides fault isolation between replicas by running them in different address spaces. This isolation relies on the hardware responsible for enforcing address space isolation (i.e., the MMU) to work properly.
- *Hardware Exception Delivery*: ROMAIN intercepts the externalization events generated by a replicated application. For this purpose it configures the replicas in a way that all these exceptions get reflected to the master process for state comparison and event processing. Using the watchdog mechanism I described in Section 3.8.2, ROMAIN can already cope with missing hardware exceptions. However, the master still relies on the fact that exception state is written to the right memory location within the master and does not accidentally overwrite important master state.
- *Operating System Kernel*: The FIASCO.OC kernel is crucial for ROMAIN because it configures the hardware properties I mentioned above and ROMAIN needs to rely on this configuration to work properly. Furthermore, the kernel schedules replicas and delivers hardware exceptions to the master process.
- ROMAIN *Master Process*: The master process manages replicas and their resources and furthermore handles externalization events. While running in user mode on top of FIASCO.OC, the master is still unreplicated and therefore unable to detect and recover from the effects of a hardware fault.

Inspired by the term TCB, in our paper we opted to call those components that are unprotected by a software fault tolerance mechanism and still need to work in order for the whole system to tolerate hardware faults, the *Reliable Computing Base (RCB)*:[1]

> "The Reliable Computing Base (RCB) is a subset of software and hardware components that ensures the operation of software-based fault-tolerance methods and that we distinguish from a much larger amount of components that can be affected by faults without affecting the program's desired results."

6.1.1 Minimizing the RCB

As the RCB is unprotected by our software fault tolerance mechanisms, any code that runs within the RCB remains vulnerable to hardware faults. In order to reduce the probability of an error striking during unprotected execution, we argue that similar to the TCB, the RCB should be minimized. This raises the question, how we can accomplish this minimization.

[5] Cristian Florian. Report: Most Vulnerable Operating Systems and Applications in 2013. GFI Blog, accessed on July 29th 2014, `http://www.gfi.com/blog/report-most-vulnerable-operating-systems-and-applications-in-2013/`

[6] Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, Redmond, WA, 2 edition, 2004

[7] Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies. In *European Conference on Computer Systems*, EuroSys'06, pages 161–174, 2006

Minimizing Code Size As mentioned above, minimizing the TCB is achieved by reducing the amount of code inside the TCB because there is a direct relation between code size and security vulnerabilities. This is not the case for dependability: A program can – but does not need to – actually become more dependable using more code, for instance when this code is used to implement fault-tolerant algorithms or data validation.

Minimizing RCB Execution Time As hardware faults are often modeled as being uniformly distributed over time,[8] reducing the time spent executing RCB code will reduce the probability of an error occurring within the RCB. Two design decisions I presented in this thesis provide a reduction of RCB execution time:

1. By running on top of a microkernel, I limit the amount of time spent executing unprotected kernel code. Code that would traditionally run inside a monolithic OS kernel – such as a file system – runs as a user-level application and can therefore be replicated and protected using ROMAIN.
2. I presented several performance optimizations that reduce the time spent executing in the ROMAIN master:
   - In Section 3.5 I described how ROMAIN's memory manager maps multiple pages at once during page fault handling.
   - The fast synchronization mechanism I introduced in Section 4.3.4 reduces the time replicas spend in synchronization operations.

   While these optimizations were mainly implemented to decrease ROMAIN's performance overhead, a useful side effect is that they also reduce the time spent executing unprotected kernel and master code.

[8] Shubhendu Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008

Minimizing Software/Hardware Vulnerability The previous examples seem to indicate that anything that speeds up kernel execution will also reduce the RCB's vulnerability. As an example, when comparing replica states we could chose to perform a hardware-assisted SSE4 `memcmp()` instead of using a pure C implementation.[9]

However, computer architecture researchers pointed out that different functional units of a CPU[10] or different types of instructions of an instruction set architecture[11] have different levels of vulnerability against hardware errors. Hence, using a more complex functional unit to reduce RCB execution time may lead to an increase in total vulnerability as more functional units participate in execution of this RCB code.

[9] Richard T. Saunders. A Study in Memcmp. *Python Developer List*, 2011

[10] Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *International Symposium on Microarchitecture*, MICRO 36, Washington, DC, USA, 2003. IEEE Computer Society

[11] Semeen Rehman, Muhammad Shafique, Florian Kriebel, and Jörg Henkel. Reliable Software for Unreliable Hardware: Embedded Code Generation Aiming at Reliability. In *International Conference on Hardware/-Software Codesign and System Synthesis*, CODES+ISSS '11, pages 237–246, Taipei, Taiwan, 2011. ACM

Future Research The most promising approach to reducing the vulnerability of the RCB against hardware faults is to consider both software-level and hardware-level vulnerability metrics. I argue that Sridharan's Program Vulnerability Factor (PVF) may be a good starting point for such analysis.[12]

As a side project of this thesis, I implemented a PVF analysis tool for x86 binary code[13] and showed that

- PVF analysis is a fast alternative to traditional fault injection analysis and can predict the impact of a hardware error on sequences of instructions, and

[12] Vilas Sridharan and David R. Kaeli. Quantifying Software Vulnerability. In *Workshop on Radiation effects and fault tolerance in nanometer technologies*, WREFT '08, pages 323–328, Ischia, Italy, 2008. ACM

[13] Björn Döbel, Horst Schirmeier, and Michael Engel. Investigating the Limitations of PVF for Realistic Program Vulnerability Assessment. In *Workshop on Design For Reliability (DFR)*, 2013

- In its current state, PVF is limited to predicting whether an error will be benign or lead to a program failure. PVF analysis does not incorporate quality information that could indicate whether a wrong result is still "good enough" given a specific application scenario.

By incorporating PVF analysis tools into the software development process, modifications to RCB components can be evaluated for their impact on the component's vulnerability in addition to traditional correctness and performance analysis.

6.1.2 The RCB in Software Fault Tolerance Mechanisms

ROMAIN is not the only software-implemented fault tolerance mechanism that relies on RCB components. I will now have a look at other mechanisms that implement fault tolerance using compiler-level or infrastructure-level techniques and show that most of these solutions have an RCB specific to the respective mechanism.

The RCB and Compiler-Based Fault Tolerance Compiler-based fault tolerance mechanisms should be able to compile the complete software stack including all RCB components if their source code is available. However, in some cases additionally inserted code – such as SWIFT's result validation[14] – remains unprotected from hardware faults and hence forms the RCB of these mechanisms.

Furthermore, to the best of my knowledge none of the compiler-based mechanisms available has been applied to an operating system kernel so far. As none of these tools are openly available for download, it is impossible to try this with FIASCO.OC. I argue that fault tolerant compilation of an OS kernel will have to solve three problems:

1. *Asynchronous Execution* arises from the necessity to run interrupt handling code whenever a hardware interrupt needs to be serviced. This may result in random jumps that confuse signature-based control-flow checking[15] or arithmetic protection of the instruction pointer.[16]
2. *Interaction with Hardware* requires accessing specific I/O memory regions or using I/O-specific instructions. These accesses work on concrete values dictated by the hardware specification. I/O values cannot be arithmetically encoded or otherwise replicated. Hence, additional code to convert between encoded and concrete values is required, which may in turn prove to be a single point of failure for the respective solution.
3. Thread-based replication[17] relies on a *runtime* that implements threading and mechanisms to communicate results among these threads. Such mechanisms are usually implemented by the OS kernel. Therefore, the threading implementation would require additional protection.

These problems can be solved. For instance, Borchert presented an aspect-oriented compiler that is able to protect important kernel data structures in the long term[18] using checksums and data replication. However, his work does not protect those data structures during modification and it does not apply to short-term storage like the stack, so that parts of the kernel still remain in the

[14] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization*, CGO '05, pages 243–254, 2005

[15] Namsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control-Flow Checking by Software Signatures. *IEEE Transactions on Reliability*, 51(1):111 –122, March 2002

[16] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *International Conference on Computer Safety, Reliability and Security*, Safecomp'10, Vienna, Austria, 2010

[17] Yun Zhang, Jae W. Lee, Nick P. Johnson, and David I. August. DAFT: Decoupled Acyclic Fault Tolerance. In *International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 87–98, Vienna, Austria, 2010. ACM

[18] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generative Software-Based Memory Error Detection and Correction for Operating System Data Structures. In *International Conference on Dependable Systems and Networks*, DSN'13. IEEE Computer Society Press, June 2013

RCB. For now I therefore assume that the OS kernel remains a less-protected part of the RCB for compiler-based fault tolerance mechanisms.

The RCB and Fault Tolerant Infrastructure In contrast to methods that generate fault-tolerant code, other approaches integrate a fault-tolerant infrastructure into a system. Replication-based methods add such infrastructure in the form of a replica manager. I already explained that in the case of ROMAIN this manager as well as the underlying OS kernel remain unprotected. This is in line with other such work: The authors of PLR[19] and RAFT[20] explicitly state that their mechanisms do not support protecting the underlying OS.

Other infrastructure approaches rework the operating system to be inherently more fault tolerant. For example, the Tandem NonStop system aimed to structure all kernel and application code to use transactions and thereby guarantee that a consistent state can be reached in the case of any error.[21] Lenharth and colleagues implemented a similar idea in Linux: their Recovery Domains restructure kernel code paths to allow rollback.[22]

However, Gray acknowledges that Tandem's transactions rely on a transaction manager to correctly implement rollback recovery. Lenharth's solution does not cover important kernel parts, such as scheduling, interrupt handling, and memory allocation. We therefore see that even those mechanisms have an RCB that remains vulnerable to the effects of hardware faults.

As a notable exception, Lovelette's software fault tolerance mechanism for the ARGOS space project actually aimed at protecting the whole RCB. ARGOS sent commercial-off-the-shelf processors into space and tried to protect them using software-only methods. As a result of the high rate of memory errors they saw in flight, they implemented a software-level ECC that scrubbed all important code and data – including their OS kernel – periodically to detect and recover from these errors before they led to system malfunction.[23] Furthermore, they duplicated the ECC scrubber in order to avoid malfunctions in this area.

SUMMARY: Although software-implemented fault tolerance mechanisms protect user applications, they still rely on a set of hardware and software components – the Reliable Computing Base – to function correctly. In most cases the RCB includes the OS kernel as well as additional infrastructure leveraged by the respective mechanisms. These RCB components are the Achilles' Heel of any such mechanism and require additional effort in order to achieve full-system fault tolerance.

6.2 Case Study #1: How Vulnerable is the Operating System?

The examples in the previous section showed that the operating system kernel is part of the Reliable Computing Base for all software-implemented fault tolerance mechanisms. For the case of ROMAIN, this kernel is FIASCO.OC. I therefore conducted a series of fault injection (FI) campaigns to understand how FIASCO.OC behaves in the presence of hardware-induced errors and gain ideas about what mechanisms are suitable to protect ROMAIN's RCB.

[19] 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, 2009

[20] Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. Runtime Asynchronous Fault Tolerance via Speculation. In *International Symposium on Code Generation and Optimization*, CGO '12, pages 145–154, 2012

[21] Jim Gray. Why Do Computers Stop and What Can Be Done About It? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986

[22] Andrew Lenharth, Vikram S. Adve, and Samuel T. King. Recovery Domains: An Organizing Principle for Recoverable Operating Systems. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 49–60, New York, NY, USA, 2009. ACM

[23] M.N. Lovellette, K.S. Wood, D. L. Wood, J.H. Beall, P.P. Shirvani, N. Oh, and E.J. McCluskey. Strategies for Fault-Tolerant, Space-Based Computing: Lessons Learned from the ARGOS Testbed. In *Aerospace Conference Proceedings, 2002. IEEE*, volume 5, pages 5–2109–5–2119 vol.5, 2002

My analysis is similar to an older study performed by Arlat and colleagues that investigated the Chorus and LynxOS microkernels.[24] Their study used microbenchmarks to drive execution towards interesting kernel components. They then injected transient memory faults into regions used by the kernel and observed the outcome. Gu[25] and Sterpone[26] also used benchmarks to drive fault injection experiments on the Linux kernel.

In contrast to these previous studies, which resorted to random sampling the fault space, I perform fault injection for *all potential faults in the fault space*. As in Section 5.2 I use the FAIL* fault injection framework to run these experiments in parallel and use fault space pruning to eliminate experiments whose outcome is already known.

[24] Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frédéric Salles. Dependability of COTS Microkernel-Based Systems. *IEEE Transactions on Computing*, 51(2):138–163, February 2002

[25] Weining Gu, Z. Kalbarczyk, and R.K. Iyer. Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors. In *Conference on Dependable Systems and Networks*, DSN'04, pages 887–896, June 2004

[26] Luca Sterpone and Massimo Violante. An Analysis of SEU Effects in Embedded Operating Systems for Real-Time Applications. In *International Symposium on Industrial Electronics*, pages 3345–3349, June 2007

6.2.1 Benchmarks and Setup

I selected four microbenchmarks to trigger commonly used mechanisms within FIASCO.OC for fault injection purposes. In the selection process I focussed on triggering important kernel mechanisms. My experiments may therefore over-represent these paths in the kernel while they do not cover rarely used paths, such as error handling code.

1. *Inter-Process Communication (IPC4, IPC252)*: Microkernel-based systems are built from small software components that run isolated for the purpose of safety and security. These components exchange data and delegate access rights through IPC channels implemented by the kernel. IPC is often considered the most important mechanism in a microkernel-based system.[27]

   This microbenchmark consists of two threads running in the same address space. The first thread sends a message to the second one, which sends a reply in return. I inject faults into both phases of the IPC operation and consider the experiment correct if the first thread successfully prints the reply message.

   I ran this benchmark in two versions: IPC4 sends a minimum IPC message with a payload of 4 bytes. IPC252 extends this size to the maximum possible payload size of 252 bytes.
2. *ThreadCreate:* As I explained previously, FIASCO.OC manages kernel objects, such as address spaces, threads, and communication channels as basic building blocks for user applications. The correct functioning of the system therefore depends on the proper creation of such objects.

   The second microbenchmark creates a user thread and lets this thread print a message. This involves kernel activity to allocate a new kernel data structure, register the thread with the kernel's scheduler, and run the creator as well as the newly created thread appropriately. I inject faults into each of the system calls required to start up a thread.
3. *MapPage:* Microkernels implement resource management in user-level applications. The kernel facilitates this management by providing a mechanism to delegate resources from one program to another. In FIASCO.OC terms this mechanism is called *resource mapping*.

   This third microbenchmark exemplifies resource mappings using a virtual memory page. I run two threads in different address spaces. The first thread requests a memory resource from the second one, which then

[27] Jochen Liedtke. Improving IPC by Kernel Design. In *ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, Asheville, North Carolina, USA, 1993. ACM

selects a virtual memory page from its address space and delegates it to the requestor. I inject faults into the map operation and validate benchmark success by inspecting the content of the mapped page on the receiver side.

4. *vCPU:* In Section 3.3 I explained that ROMAIN leverages FIASCO.OC's virtual CPU (vCPU) mechanism to monitor all externalization events generated by replicas. This event delivery mechanism is therefore crucial for the correctness of ROMAIN.

   In this benchmark I launch a new vCPU, which executes a couple of `mov` instructions to bring its registers into a dedicated state. Then the vCPU raises an event that is intercepted by the vCPU master. I inject faults into the kernel's mechanism for exception delivery.

I compiled 32bit versions of FIASCO.OC, the L4 Runtime Environment, and the benchmarks with GCC (version Debian 4.8.1-10). I then used FAIL* to select and perform FI experiments. I inject bit flips into memory and general-purpose registers. My experiments focus on execution in privileged kernel mode, while I assume that user-level code is protected by ROMAIN for which I already showed that it detects all such errors that manifest during user-level execution in Section 5.2.

Table 6.1 shows the number of instructions each benchmark executed inside the kernel, the number of register and memory bit flips that make up the whole fault space, and the number of experiments I actually had to carry out after FAIL* successfully pruned known-outcome experiments.

Benchmark	Fault Model	# of Instructions	Total Experiments	Experiments after Pruning
IPC4	Register SEU	2,193	326,632	70,857
	Memory SEU		9,553,400	21,865
IPC252	Register SEU	2,313	341,992	74,697
	Memory SEU		13,542,904	25,705
ThreadCreate	Register SEU	26,893	5,095,840	891,481
	Memory SEU		175,464,208	57,905
MapPage	Register SEU	6,956	1,048,040	223,074
	Memory SEU		40,377,232	67,074
vCPU	Register SEU	535	73,456	16,330
	Memory SEU		591,384	4,530

Table 6.1: Overview of FIASCO.OC Fault Injection Experiments

6.2.2 *Experiment Results*

I executed the fault injection campaigns for every microbenchmark and as a first step classified the experiment outcomes similar to my classification in Section 5.2:

1. *OK* means the experiment terminated and produced exactly the same output as an unmodified run.
2. *CRASH* indicates that the experiment terminated with a visible error message either in the kernel or user space.
3. *TIMEOUT* indicates that the experiment did not terminate within a predefined period of time. I set this timeout to 200,000 instructions, which –

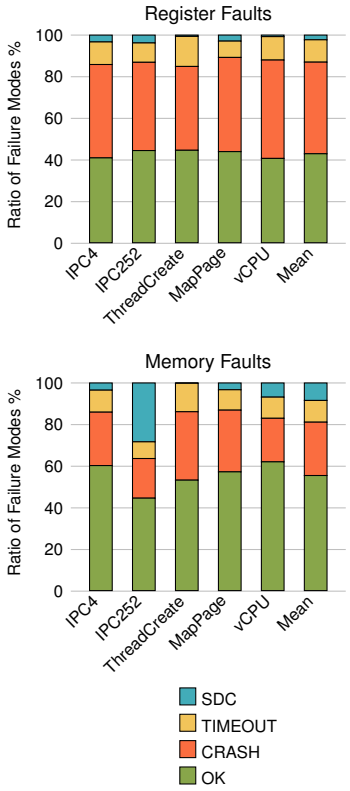

Figure 6.1: Distribution of FI Results targetting the FIASCO.OC kernel

depending on the workload – means the benchmark executed 10 to 100 times as long as the initial run at this point.

4. *Silent Data Corruption (SDC)* means that an experiment terminated and did not raise an error. However, the experiment's output differed from the initial, fault-free run.

Figure 6.1 shows the distributions of experiment results for all benchmarks and distinguishes between register and memory SEUs. On average, 43% of the register faults and 56% of the memory faults did not lead to a change in system behavior. This observation confirms other studies that also found that many SEUs have no influence on the outcome of an application run, such as Arlat's study[24] and my user-level fault injection experiments discussed in Chapter 5.

Note, that my experiments focus on the execution of a single microbenchmark. They therefore only cover the short-term effects of an injected fault. Experiments that are classified OK after my experiments may still corrupt kernel state in a way that affects the execution of a different system call at a later point in time. Future work needs to investigate for which experiments this would be the case and which kernel data structures are prone to such long-term corruption issues. These data structures will then be likely candidates for Borchert's aspect-oriented data protection.[28]

Those injection runs leading to a visible deviation of application behavior are dominated by CRASH errors: an average of 44% of all register faults and 26% of all memory faults led to crashes, whereas for both fault models about 10% of the experiments timed out. In contrast, only 2% of the register faults and 8% of the memory faults led to SDC errors. The SDC numbers are significantly smaller than Hari's previously reported error rates for user-level applications.[29]

Understanding Silent Data Corruption I attribute the difference in SDC behavior to the fact that my experiments focus on kernel-level code. Hari's study pointed out that SDC errors are often the result of long-running computations on large chunks of data. Such computations rarely occur in kernel mode. Instead, the kernel touches vital data structures, such as the scheduler's run queue or hardware page tables. These structures have a direct impact on the behavior of the system and corrupting them is therefore more likely to lead to a crash.

There is an outlier in my measurements that confirms this hypothesis: When injecting memory errors into the IPC252 benchmark we see an SDC rate of 28%. Closer inspection of this result reveals that nearly all SDC errors here happen within two distinct memory regions: the IPC sender's and receiver's user-level thread control blocks (UTCBs). As I explained in Section 3.4, the UTCB contains a program's system call parameters when entering the kernel. In the case of IPC this is the message payload to be transmitted. As the kernel only copies UTCB content without performing any processing, errors within the UTCB show up as silent data corruption.

As FIASCO.OC is a microkernel, the amount of memory accessed during these microbenchmarks benchmark is fairly low. For the IPC4 and IPC252 benchmarks, which only differ in the amount of bytes transferred through the

[28] Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generative Software-Based Memory Error Detection and Correction for Operating System Data Structures. In *International Conference on Dependable Systems and Networks*, DSN'13. IEEE Computer Society Press, June 2013

[29] Siva K. S. Hari, Sarita V. Adve, and Helia Naeimi. Low-Cost Program-Level Detectors for Reducing Silent Data Corruptions. In *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2012

IPC mechanism, the kernel touches about 1 KiB (IPC4) and 1.5 KiB (IPC252) of memory respectively. The increase in IPC252's memory footprint is directly explained by the increased message payload: 248 more bytes in the sender UTCB plus 248 more bytes in receiver UTCB lead to an increase of 496 bytes. Consequentially, the increase in SDC errors we observe is caused by these additional bytes and underlines the fact that user-level data passing through the UTCB is more prone to undetected data corruption than errors in internal kernel data structures.

Investigating CRASH Errors As CRASH errors are the most dominant misbehavior seen in the previous experiments, I had a closer look on what kind of crashes we are seeing as a result of injecting faults into the kernel. I inspected the output and termination information of those experiments labeled as CRASH in the previous result distribution and further distinguished between three types of crashes:

1. *MEMORY* failures are those where the error caused the kernel to access an invalid memory address, i.e., an address that is not part of any valid virtual memory region. These crashes trigger kernel-level page fault or double fault handler functions.
2. *PROTECTION* failures indicate those runs where the injected fault caused the kernel to raise hardware protection faults (e.g., by writing to a read-only memory page) or access invalid kernel objects (e.g., because a capability index was corrupted).
3. *USER* failures classify experiments where control returned an error condition to a user-level application. This happens in one of two ways: first, the kernel may return from the current system call with an error that is then detected by the user application. Second, an injected fault may lead to an exception that FIASCO.OC deems to originate from the user (e.g, because of a page fault in user-addressable memory). This exception is then delivered to a user-level exception handler.

Figure 6.2 breaks down the CRASH errors from the previous fault injection campaigns into these three categories. We see that slightly more than half of all crashes are reflected to user space (register faults: 57%, memory faults: 54%). An average of 37% of the register faults and 27% of the memory faults led to memory-related exceptions within the kernel. Protection failures make up the smallest fraction of results with 18% of the injected memory faults and 6% of the injected register faults.

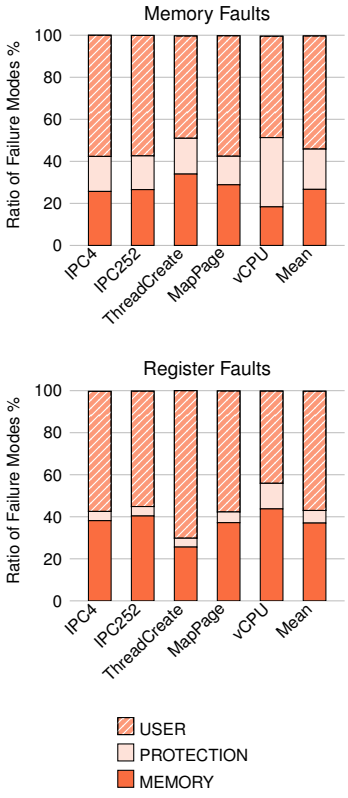

Figure 6.2: Distribution of CRASH failure types

6.2.3 Directions for Future Research

The distributions I showed above allow us to understand what impact SEUs have on kernel execution. Based on these results we can draw conclusions about what kinds of kernel errors we can detect and recover from.

Handling Silent Data Corruption Detecting and correcting SDC failures depends on the actual workload. If these errors happen within the payload of an IPC message, FIASCO.OC's execution is not affected at all. User-level programs can instead detect those errors using message checksums. To

demonstrate this, I modified the IPC252 benchmark so that the sender adds a CRC32 checksum of the message payload and the receiver validates this checksum. With this approach I was able to detect 94.6% of all SDCs in the IPC252 benchmark.

Incorporating checksums and making IPC protocols retry failed message transfers requires a substantial amount of work, because every application needs to be adapted for this purpose. Future research should therefore investigate whether and how this step can be automated.

SDCs in other parts of the kernel may not be as easy to detect, as they for instance they rely on the correctness of hardware mechanisms. As an example, the vCPU benchmark relies on correct delivery of the exception state to the vCPU master. This involves the CPU's exception mechanism to dump the proper register state onto the kernel stack.[30] No software mechanism will be able to detect a data corruption happening in this step, because software will only get called after the exception state is copied to the stack. The respective window of vulnerability could be reduced by a hardware extension that adds a checksum to the exception state on the stack. However, this modification would require customized hardware instead of relying on COTS hardware features. As SDCs only constitute a tiny fraction of the kernel failures we are seeing, we should rather focus on detecting more prominent failure types.

[30] Intel Corp. Intel64 and IA-32 Architectures Software Developer's Manual. Technical Documentation at `http://www.intel.com`, 2013

Kernel Crashes are Detected Errors Given my breakdown of crash errors above, we saw that hardware faults may trigger unhandled page faults or protection faults inside the FIASCO.OC kernel. FIASCO.OC itself is designed in a way that these events will never happen during normal kernel execution: kernel memory is always mapped and accesses never lead to page faults. Protection faults will also only happen in the case of a programming error or a hardware fault.

If we assume FIASCO.OC to be thoroughly tested before going to production, a software error leading to a kernel crash will be extremely rare. Therefore, if we encounter a page fault or protection fault hardware exception at runtime, we can assume this to be the result of a hardware fault and start error recovery. Hence, all MEMORY and PROTECTION failures we saw, actually constitute detected errors.

Crashes Reflected to User Space are Detected Errors In addition to kernel crashes we saw that more than half of all CRASH errors are reflected to user space. If we assume that programs at the user level are replicated using ROMAIN, then these exceptions will get sent to the ROMAIN master, which will then detect a deviation from other non-faulty replicas and initiate error recovery. Unfortunately, this only covers the rare case if a kernel error occurs while a replica is executing, for instance because the kernel's timer interrupt handler was triggered for scheduling reasons.

In contrast, replication does not protect us from kernel errors that arise during system call handling. As I explained in Chapter 3, the ROMAIN master executes all system calls on behalf of the replicas. As the master is not replicated, a failing system call cannot be handled using replication.

Nevertheless, these errors are noticed by user-level code:

- If the error gets reflected to user space in the form of an exception message, the ROMAIN master's exception handler will detect this issue. Again, by design we can expect no exceptions to occur during normal execution of a system call. Therefore, these exceptions constitute detected hardware errors.
- If the error becomes visible as an error code returned by the system call, ROMAIN will deliver it to the replicated application just as in the case of any other system call return error. The program is then responsible to handle the error code properly.

Can We Recover from Those Crashes? While the previous analysis showed that CRASH failures can be detected, they require recovery procedures at three different levels: kernel failures need to be handled inside FIASCO.OC, visible exceptions require handling by the ROMAIN master, and system call errors need to be recovered from by the application.

As an experiment I implemented a mechanism to bridge the gap between those layers by turning all CRASH failures into system call errors. I modified FIASCO.OC and ROMAIN so that in the case of an unhandled exception during kernel execution they turn this exception into a system call error visible to the application:

- FIASCO.OC's double fault, page fault, and protection fault handler functions return an error value to the currently active thread instead of stopping execution as they do by default.
- I modified ROMAIN's exception handler so that in the case of an exception during a system call, the replica currently executing this system call is resumed with an error return value.

With this mechanism, no recovery needs to be implemented in the lower levels and only application code has to deal with these issues. Unfortunately, this places the burden of recovery on each application developer. To relieve developers, I implemented a generic recovery mechanism, which is part of FIASCO.OC's system call bindings, so that application code is completely oblivious of error handling and recovery:

- Before issuing a system call, the user-level thread pushes its current register state and system call parameters from the UTCB to the stack.
- The thread then issues its system call.
- If the call returns an error value, the original state is retrieved from the stack and the system call is issued again.

This approach allowed me to recover from a set of manually crafted kernel crashes. However, when I evaluated this approach with fault injection experiments on a larger scale, I discovered that there are many cases where generic recovery fails either by crashing the kernel again or by getting stuck inside one of the next system calls. The experiment therefore was a failure and requires future work to investigate the following three problems:

1. *Non-idempotent system calls*: Assuming a user application receives a system call error indicating a kernel crash, the application does not know at what point during the system call the crash happened. The kernel may at this point already have modified kernel state by creating new kernel objects or deleting old ones.

   Generic recovery only works if system calls are idempotent – such as FIASCO.OC's IPC send operation) – and requires additional care if kernel state might have been modified.

2. *Thread Dependencies:* Inter-Process Communication – FIASCO.OC's most heavily used system call – involves the synchronization between a sender and a receiver thread. The IPC path is therefore a series of updates to the state of two complex state machines. If the sender of a message crashes, my modifications were successful in returning an error to the sender. However, depending on which part of the IPC protocol is affected by the crash, we may also have to return an error on the receiver side of the IPC. Implementing this feature was not completed in time for this thesis.

3. *Recovery for multithreaded programs on multicore platforms:* My experiment only worked for recovering single-threaded programs running on a single-core CPU. I did not yet consider potential race conditions and other side effects that may arise when trying to recover on a system that runs other threads concurrently.

Timeouts are Difficult In addition to SDC and CRASH errors, my fault injection campaigns show that a non-negligible amount of hardware-induced errors lead to TIMEOUTs. The reasons for these errors are manifold. In the simplest case the kernel skips delivering an IPC message because of a bit flip, and immediately returns to the sender. This scenario can be handled by a fault-aware user application that retries requests if no proper answer is received within a certain time interval.

Other TIMEOUT errors are harder to detect: I noticed that most timeouts occur due to corruption of the kernel's scheduling data structures. In these cases the kernel may lose track of a thread and never schedule it again even though it would be ready to run. Unfortunately, detecting these errors is often impossible because we cannot distinguish between a system that simply has no work to do and a system that lost a thread due to a hardware error. I conclude from this observation that scheduling data structures need to be additionally protected using redundancy in order to avoid timeouts.

SUMMARY: I conducted a series of fault injection campaigns to analyse the FIASCO.OC kernel's reaction to SEUs in memory and general-purpose registers. I found that detectable crash failures constitute the largest source of hardware-induced misbehavior for the kernel. However, implementing recovery in such situations remains an open issue.

Silent data corruption rarely occurs during kernel execution. Most of the SDC errors happened within the message payload of an IPC message. Error detection and recovery may be achieved using message checksums and failure-aware communication protocols.

Corruption of scheduling-related data may furthermore lead to TIMEOUT errors where execution of an active thread does not resume properly. These errors are hard to detect and the affected data structures therefore require additional protection.

6.3 Case Study #2: Mixed-Reliability Hardware Platforms

In Chapter 2 I argued that ROMAIN should run on commercial-off-the-shelf (COTS) hardware because COTS components make up a majority of modern embedded, workspace, and high-performance computers. In contrast, hardware that is custom-tailored for reliability is often very expensive. Reliability features are therefore seldom added to COTS hardware. On the other hand, we are currently seeing an increase in hardware diversity due to the advent of heterogeneous manycore platforms provided by major CPU vendors. It is likely that these heterogeneous CPUs are also heterogeneous with respect to their vulnerability against hardware faults.

While heterogeneous compute platforms are currently optimized for their compute throughput and their energy consumption, I argue that we are likely to see platforms combining specially reliable CPUs with cheap, but vulnerable non-reliable cores in the future. The ASTEROID OS architecture I developed in this thesis suits such an architecture, because it allows protecting RCB components by running them on reliable processors while executing replicas on less reliable CPUs.

6.3.1 Heterogeneous Hardware Platforms

Some of the most widely available heterogeneous platforms today come as I/O board extensions to desktop and data center computers. These platforms, such as Intel's Xeon Phi[31] and Nvidia's Kepler platform[32] are derived from previous generations of graphics accelerators. The main features provided by these *general-purpose graphics processing units (GPGPUs)* are additional compute cores and specialized vector processing units. With these properties they extend their predecessors' focus on graphics processing to general-purpose compute-intensive and parallel applications. As a side effect, these systems also often perform the same computation at a lower energy consumption,[32] which makes them additionally attractive for large-scale use.

ARM pioneered a slightly different architecture with its *big.LITTLE* platform.[33] This architecture is motivated by the observation that while a computer might need a fast processor for some applications, a lot of the remaining work can be done on a slower, more energy-efficient CPU. big.LITTLE therefore combines four energy-efficient, in-order ARM Cortex A7 CPUs with four fast, super-scalar, and more power-hungry Cortex A15 processors. The operating system can dynamically assign applications to the CPU that suits their needs and switch off the remaining cores to save energy in the meantime.

[31] James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013

[32] Nvidia Corp. Kepler: The World's Fastest, Most Efficient HPC Architecture. `http://www.nvidia.com/object/nvidia-kepler.html`, accessed August 1st, 2014, 2014

[33] ARM Ltd. Big.LITTLE processing with ARM Cortex-A15. Whitepaper, 2011

When looking at hardware reliability, we see that big.LITTLE already combines two types of processors with different reliability characteristics. The Cortex A7 has fewer features and requires less chip area. This makes the processor less vulnerable to the effects of cosmic radiation than the larger Cortex A15.

While the previously mentioned platforms are used in production today, other research platforms investigated the idea of using hardware heterogeneity explicitly to separate between CPU cores that are heavily protected against hardware faults and smaller, less protected ones. Leem proposed the Error-Resilient System Architecture (ERSA) that consists of a small set of *Super-Reliable Cores (SRCs)* whereas the majority of the chip area is spent for smaller, faster, and less reliable *Relaxed Reliability Cores (RRCs)*.[34] Leem intended these RRCs to execute stochastic compute workloads. These programs have the specific property that few errors during computation will not have a dramatic influence on the correctness of a program run. In contrast, he proposed to use SRCs to execute software that must never fail due to the effects of hardware errors, such as the operating system.

[34] L. Leem, Hyungmin Cho, J. Bau, Q.A. Jacobson, and S Mitra. ERSA: Error Resilient System Architecture for Probabilistic Applications. In *Design, Automation Test in Europe Conference Exhibition*, DATE'10, pages 1560–1565, 2010

In a different work, Motruk and colleagues presented *IDAMC*, a reconfigurable manycore platform that allows to safely isolate programs in space (by enforcing hardware resource partitioning between groups of CPUs) and time (by partitioning access to the network-on-chip (NoC)).[35] IDAMC thereby allows to run mixed-criticality workloads concurrently on the same chip while still fulfilling automotive safety regulations.[36] In this architecture we also find worker cores that are solely intended to perform computations at the application level combined with monitor cores that implement special monitoring and configuration tasks. Only monitor cores are allowed to reconfigure the system-wide isolation setup and assign resources to workers cores. Therefore, monitors need to be designed to be less vulnerable to hardware faults than the workers.

[35] Boris Motruk, Jonas Diemer, Rainer Buchty, Rolf Ernst, and Mladen Berekovic. IDAMC: A Many-Core Platform with Run-Time Monitoring for Mixed-Criticality. In *International Symposium on High-Assurance Systems Engineering*, HASE'12, pages 24–31, Oct 2012

[36] International Organization for Standardization. ISO 26262: Road Vehicles – Functional Safety, 2011

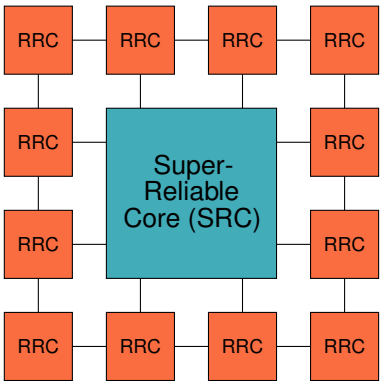

Figure 6.3: Mixed-Reliability Manycore Architecture (RRC = Relaxed Reliability Core)

The observations discussed above indicate that we are likely to see heterogeneous multicore systems integrating cores of different levels of hardware vulnerability in the future. These systems will consist of at least two kinds of processors as shown in Figure 6.3. Borrowing Leem's naming I call these processor types Super-Reliable Cores and Relaxed Reliability Cores.

- *Super-Reliable Cores (SRCs)* are processors that are specially protected against hardware faults. This protection may be implemented by replicating hardware units. Alternatively, SRCs might be produced with a larger structure size to be less vulnerable against production- as well as temperature- and radation-induced errors. To reduce vulnerability against variation in signal runtime, they may furthermore be clocked at a lower rate than other CPUs.

- *Relaxed Reliability Cores (RRCs)* will be produced at the smallest possible structure size to integrate more cores and accelerators onto the chip. These can then run at the highest possible clock rate to achieve best performance. As a consequence, these cores are more likely to suffer from the error effects I explained in Chapter 2.

If such platforms become COTS hardware, their inherent properties may allow to protect ASTEROID's Reliable Computing Base by running the

FIASCO.OC kernel and the ROMAIN master process on SRCs, while replicas may be executed on RRCs respectively. Such a system will protect RCB components in hardware and concurrently use ROMAIN to protect application software using replication. In contrast to statically replicated setups, this architecture benefits from ASTEROID's additional flexibility: the system designer can decide, whether he wants to replicate an important application or rather run it on an SRC. Furthermore, less important applications may run unreplicated on RRCs.

There is one open problem that we need to solve in order to run ROMAIN on top of a mixed-reliability manycore architecture. The mechanisms I presented in this thesis so far execute replicas concurrently until they raise an externalization event. At this point, the replicas wait for each other and then one of the replicas switches to master execution and performs the actual event handling. This handling is executed on any CPU the replica is currently running on and does not switch to another more reliable core for this purpose.

To fit a mixed-reliability platform, ROMAIN has to move execution of master code to an SRC. This requires interaction between RRC and SRC code. I investigate three ways to do so in the following section. I compare those alternatives to ROMAIN's original event handling mechanism, which I will refer to as *local event handling*.

6.3.2 Communication Across Resilience Levels

Based on the previous discussion of a mixed-reliability architecture I now make the following assumptions:

- The platform consists of SRCs and RRCs.
- SRCs and RRCs can communicate over the network-on-chip (NoC). Rambo and colleagues pointed out that the NoC also requires protection against the effects of hardware faults.[37]
- Hardware enforces resource isolation between RRCs. This isolation is configured by software running on an SRC as it was demonstrated by Motruk's IDAMC.
- The operating system and the ROMAIN master run on an SRC and schedule replica execution on RRCs.

[37] Eberle A. Rambo, Alexander Tschiene, Jonas Diemer, Leonie Ahrendts, and Rolf Ernst. Failure Analysis of a Network-on-chip for Real-Time Mixed-Critical Systems. In *Design, Automation Test in Europe Conference Exhibition*, DATE'14, 2014

Given these assumptions, we need efficient and reliable communication between RCB and non-RCB components. For this purpose I implemented and evaluated three inter-core communication mechanisms, which I describe below: (1) thread migration, (2) synchronous messaging, and (3) shared-memory polling.

Thread Migration The first option to reliably handle replica events is to migrate a replica thread from an RRC to the master SRC in the case of an event. I show this in Figure 6.4. After this migration, event handling proceeds as in the local fault handling scenario, but benefits from hardware protection mechanisms provided by the SRC. Once event handling is complete, the replica threads are migrated back to their RRCs.

This mechanism requires that the underlying operating system supports migration of threads between cores. This feature is provided by FIASCO.OC, but it remains to be investigated whether migration will still be able on a hardware platform consisting of isolated SRC and RRC processors.

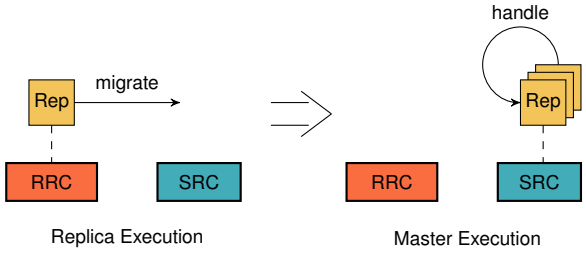

Figure 6.4: Switching to an SRC by thread migration

Synchronous Messaging (Sync IPC) I implemented a second technique that avoids migrating all threads to an SRC. Instead, I start a helper thread HT on the SRC, which waits for event notifications. Replicas send these notifications through a dedicated messaging mechanism, such as FIASCO.OC's IPC channels. Alternatively, this communication could also be implemented using specially protected hardware mechanisms similar to the message passing extensions Intel proposed in their Single Chip Cloud Computer (SCC).[38]

[38] Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight Communications on Intel's Single-Chip Cloud Computer Processor. *SIGOPS Operating Systems Review*, 45(1):73–83, February 2011

As I show in Figure 6.5, once all replicas have sent their state to the helper thread on the master side, the helper validates their correctness and performs event handling. In the meantime, the replicas block waiting for a reply. After the helper completes event processing, it replies to the replicas with an update to their states. The replicas then resume concurrent execution on their RRCs.

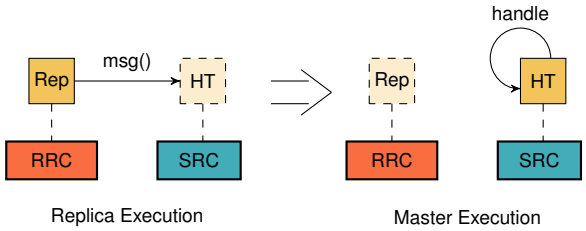

Figure 6.5: Triggering SRC execution using synchronous IPC

Shared-Memory Polling (SHM Poll) Finally, my third mechanism avoids relying on a dedicated messaging mechanism and instead uses shared memory between RRCs and SRCs to transfer notifications and replica states. This mechanism was motivated by FlexSC, which observed that asynchronous messaging primitives may lead to better system call throughput and latencies.[39] I therefore implemented a variant of the previous mechanism where the SRC helper thread and the RRC replicas poll on a shared-memory region for updates as depicted in Figure 6.6.

[39] Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, 2010

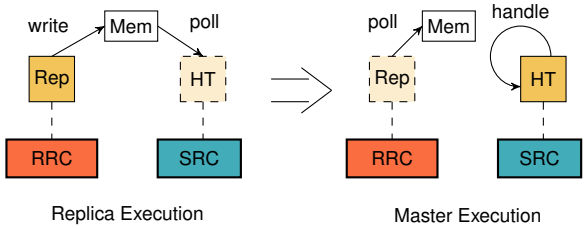

Figure 6.6: Notifying the SRC using shared-memory polling

6.3.3 Evaluation

As I implemented ROMAIN for the 32bit x86 architecture, there is no current platform available that provides mixed-reliability hardware. I therefore evaluate my communication mechanisms on the test machine described in Section 5.3.1. I simulate SRCs and RRCs the following way:

- I assume one of the twelve available cores to be an SRC, while all other cores are RRCs.
- I modified the ROMAIN master process to perform all event handling on the SRC. For this purpose I integrated the three communication mechanisms explained above into ROMAIN and adjusted the master's event handling accordingly.
- As explained before, the SRC might be protected from signal fluctuations by running at a lower clock speed than the remaining cores. I simulate this fact by artificially slowing down master event handling: whenever the ROMAIN master handles a replica event, I measure the time required to handle this event. Thereafter, I introduce a wait phase, which is a multiple of the event handling time to simulate the SRC being slower than an RRC.

  In the subsequent experiments I show three different speed ratios between SRCs and RRCs: I measure overhead for both processors running at the same speed (ratio 1:1), as well as for the SRC being five times (ratio 1:5) and ten times (ratio 1:10) slower than an RRC.

I selected four benchmarks from the MiBench benchmark suite for this evaluation:[40]

1. *Bitcount* is a purely computation-bound benchmark that does not perform expensive system calls. The benchmark spends a large fraction of its time executing user code and is therefore likely to not suffer from RCB slowdown.
2. *Susan* and *Lame* both memory-map an input file and then perform image processing and audio decoding respectively. They therefore represent a mix of kernel interaction and intensive compute work.
3. *CRC32* memory-maps 30 files to its address space and then computes a checksum of their content. Checksum computation is relatively cheap so that the benchmark is dominated by the cost of loading the data files. CRC32 therefore heavily interacts with the kernel and the ROMAIN master and I expect it to suffer most from RCB-induced slowdown.

[40] M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *International Workshop on Workload Characterization*, pages 3–14, Austin, TX, USA, 2001. IEEE Computer Society

As in previous experiments, I execute these benchmarks natively on L4Re and then in ROMAIN running one, two, and three replicas. In Figures 6.7 – 6.10 I plot the experiment results for each of the four benchmarks. I group the results by the replica slowdown ratios (1:1 1:5 1:10). Within each group I order the results by the communication mechanism used. I furthermore show the overhead for ROMAIN's original local fault handling for reference.

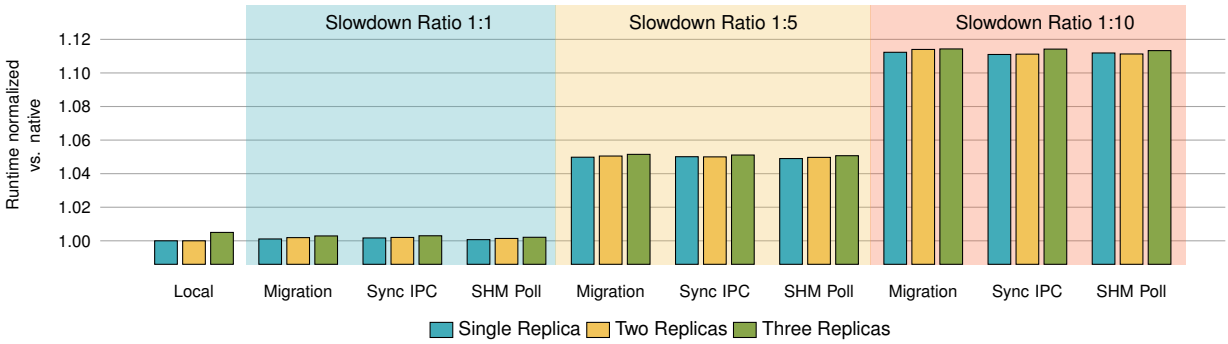

Figure 6.7: Bitcount: Replication Overhead when running RCB code on a Super-Reliable Core with different cross-core communication mechanisms and SRC slowdowns

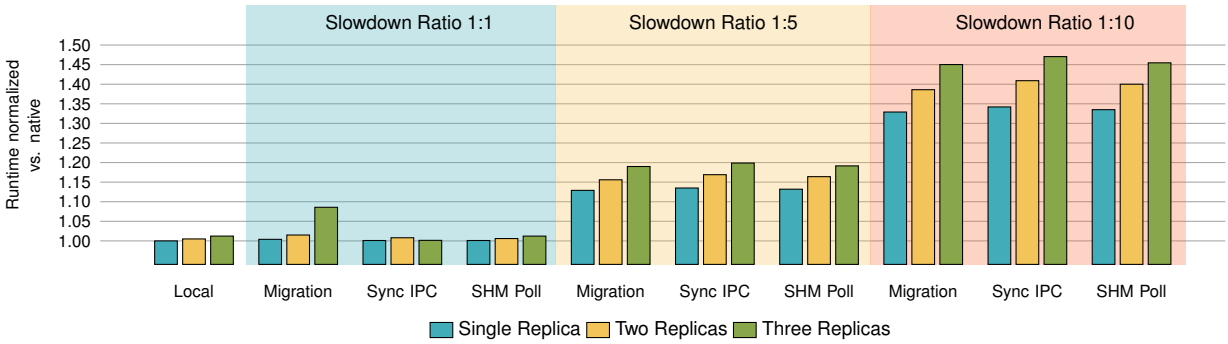

Figure 6.8: Susan: Replication Overhead when running RCB code on a Super-Reliable Core with different cross-core communication mechanisms and SRC slowdowns

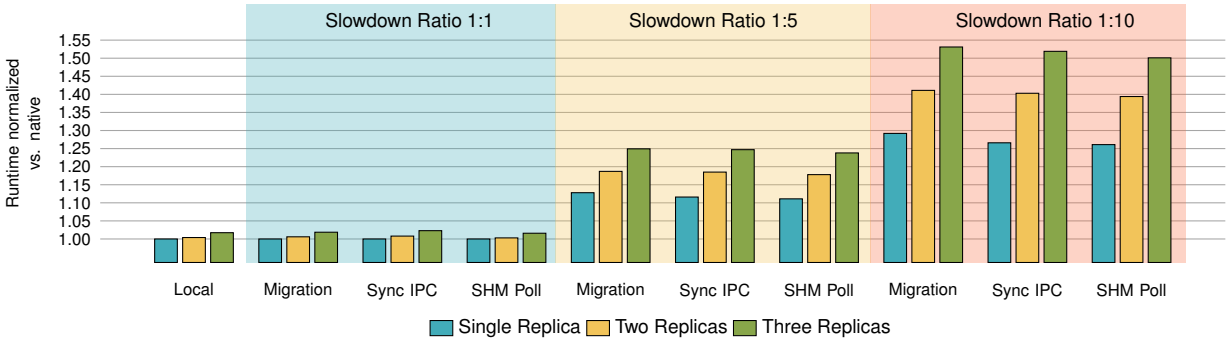

Figure 6.9: Lame: Replication Overhead when running RCB code on a Super-Reliable Core with different cross-core communication mechanisms and SRC slowdowns

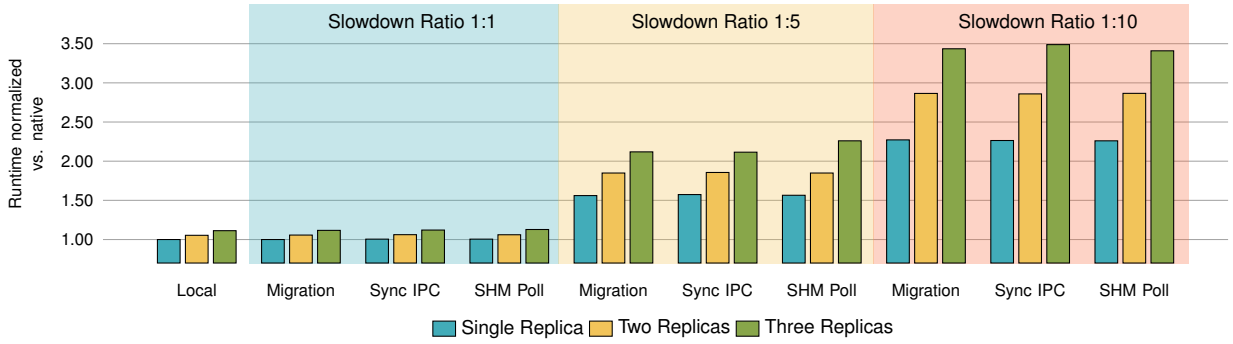

Figure 6.10: CRC32: Replication Overhead when running RCB code on a Super-Reliable Core with different cross-core communication mechanisms and SRC slowdowns

As expected, we see that overheads rise when we slow down execution of RCB code on the SRC. We also see that slowing down RCB execution has a higher impact on system-call heavy workloads. Replicating CRC32 with three replicas multiplies exectuion time by a factor of 3.5 when the RCB is executed ten times slower. In contrast, the overhead for Bitcount in the same setup is merely 11%.

While there are small fluctuations across the different communication mechanisms, the main trend is that their impact on replication appears to be nearly identical. This is not surprising: I measured the cost for handling externalization events and handling a page fault for example costs an average of 10,000 CPU cycles in the ROMAIN master. Redirecting IPC messages and other system calls is even more costly: The Bitcount benchmark spends an average of 1.3 million cycles on every system call it performs. Most of this time is spent for processing these calls at the other end, such as writing data to a serial terminal or allocating memory dataspaces. In contrast, delivering events to the ROMAIN master contributes between a few 100 and 1,000 CPU cycles to these handling times depending on the selected delivery method. Hence, the impact of the messaging mechanism on overall processing is low. This impact becomes even lower when we slow down the RCB as this increases processing times even more.

As the choice of cross-core communication mechanism does not influence replication performance, we should focus on the reliability properties of a mechanism in order to best protect the Reliable Computing Base. A thorough analysis of this issue is left for future work because it requires a functioning mixed-reliability hardware platform. However, for the following reasons I anticipate that synchronous messaging may be a useful communication mechanism from an RCB perspective:

- Thread migration requires kernel and hardware support. Migrating threads between mixed-reliable cores could be disallowed by the hardware platform for the purpose of isolating the resources of SRCs and RRCs.

- For a similar reason, shared-memory polling might not work on such platforms. Relaxed Reliability Cores might encounter hardware errors that cause them to overwrite arbitrary memory regions. A simple hardware solution to prevent those errors to affect SRCs is to disallow sharing of memory regions across reliability zones. If this is implemented in hardware, sharing memory is impossible

- In contrast, synchronous IPC can be implemented using a specially protected hardware message channel. The feasibility of such hardware mechanisms in multicore platforms was already demonstrated by Motruk's IDAMC and the Intel SCC.

SUMMARY: Trends towards heterogeneous manycore platforms indicate that future manycore systems will comprise processors with different levels of reliability. Researchers already proposed hardware that is built from many cheap and fast Relaxed Reliability Cores (RRCs) and few, specially protected Super-Reliable Cores (SRCs).

The ASTEROID operating system architecture I developed in this thesis fits well onto such an architecture. Components of the Reliable Computing Base – such as the FIASCO.OC kernel and the ROMAIN master process – can run on an SRC while user applications can be protected by running them on RRCs in a replicated fashion.

6.4 Case Study #3: Compiler-Assisted RCB Protection

The motivation for implementing ROMAIN as an OS service was the need to support arbitrary binary-only applications without requiring a recompilation from source code. When we think of protecting the Reliable Computing Base against hardware faults, we may reconsider this argument. All components that are within ROMAIN's RCB – the FIASCO.OC kernel, services of the L4 Runtime Environment, as well as the ROMAIN master process – are available as open source software. In this case it is certainly possible to protect those components using compiler-assisted reliable code transformations, such as the ones I discussed in Section 2.4.2.

Unfortunately, to the best of my knowledge none of the commonly referenced fault-tolerant compilers is freely available for download. Furthermore, as I explained in Section 6.1.2, none of these solutions were previously applied to operating system kernels. Hence, implementing such a compiler and applying it to RCB components is out of the scope of this thesis.

The Cost of Compiler-Assisted RCB Protection Nevertheless, we can try to estimate what impact such a compiler-based solution would have on ROMAIN's performance, resource usage and reliability. In ASTEROID, user-level applications are protected against hardware faults using replicated execution provided by ROMAIN. We can improve the reliability of the whole system by compiling its RCB components using a fault-tolerant compiler. Using state-of-the-art approaches, such as encoded processing,[41] this will allow us to detect close to 100% of commonly visible hardware errors that affect the RCB.

Applying compiler-assisted fault tolerance to the RCB will also lead to resource and execution time overheads. Encoded processing for instance adds redundancy to data by transforming 32bit data words into 64bit encoded values. This means, the RCB's memory footprint is likely to double from applying such techniques. Neither the FIASCO.OC kernel nor the ROMAIN

[41] Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *International Conference on Computer Safety, Reliability and Security*, Safecomp'10, Vienna, Austria, 2010

master process require more then a few megabytes of data for their management needs. The largest fraction of data in a system usually belongs to user-level applications. These programs are protected by replication and as I explained in Chapter 3, ROMAIN maintains copies of their memory for every replica. Therefore, while compiler-assisted fault tolerance will increase the memory footprint of the RCB, its impact will be negligible in comparison to the impact of replication-induced memory overhead for user applications.

Execution Time Overhead I performed a simulation experiment to estimate the performance impact of compiler-based fault tolerance methods on the RCB. For this purpose I executed the integer subset of the SPEC CPU 2006 benchmarks on top of ROMAIN using triple-modular redundancy. Similar to the experiment in the previous section I executed the benchmarks on the test machine described in Section 5.3.1 and slowed down all master execution by a given factor. In contrast to the previous experiment, I did not distinguish between reliable and non-reliable CPUs, because in this setup the RCB is protected using pure software methods. For the slowdown I selected three factors that represent widely cited compiler-assisted fault tolerance methods:

1. *SWIFT* represents Reis and colleagues' Software-Implemented Fault Tolerance,[42] a low-overhead mechanism that duplicates instructions and compares their results. SWIFT has a reported mean overhead of 9.5%.
2. *EC-ANBD* represents Schiffel and colleagues's AN-encoding compiler.[41] ANBD improves on SWIFT's error coverage, but reports a significantly larger execution time overhead of about 289%.
3. *SRMT* refers to Wang and colleagues' implementation of redundant multithreading in software.[43] The authors report an overhead of 900% when running their replicated threads on different CPU sockets. While other RMT approaches have reported lower overheads, I selected this mechanism explicitly to show the impact of high slowdowns on RCB execution.

[42] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization*, CGO '05, pages 243–254, 2005

[43] Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed Software-based Redundant Multithreading for Transient Fault Detection. In *International Symposium on Code Generation and Optimization*, CGO '07, pages 244–258, 2007

Figure 6.11 shows the resulting overheads, which include slowing down all inter-replica synchronization as well as handling of page faults in the ROMAIN master and any system calls the master performs on behalf of the replicas. As was to be expected, we see that SWIFT's 9.5% overhead does not matter at all and execution overheads are identical to execution with an unprotected RCB.

`445.gobmk`, `458.sjeng` and `473.astar` are purely compute-bound benchmarks and slowing down the RCB does not increase their execution time. Similarly, the overhead of `429.mcf` and `471.omnet++` results mainly from cache-related effects that I explained in Section 5.3.2. These benchmarks also do not spend a lot of time executing RCB code and therefore do not suffer from slowing down the RCB.

In contrast, the remaining benchmarks (`400.perl`, `401.bzip2`, `456.hmmer`, `462.libquant`, `464.h264ref`) perform system calls and interact with the memory manager. Here we see that for a system-call intensive workload, such as `400.perl`, slowing down the kernel and the replication master significantly increases replication overhead.

Nevertheless, the total overheads for combining replication with an RCB protected by fault-tolerant code transformations are significantly lower than

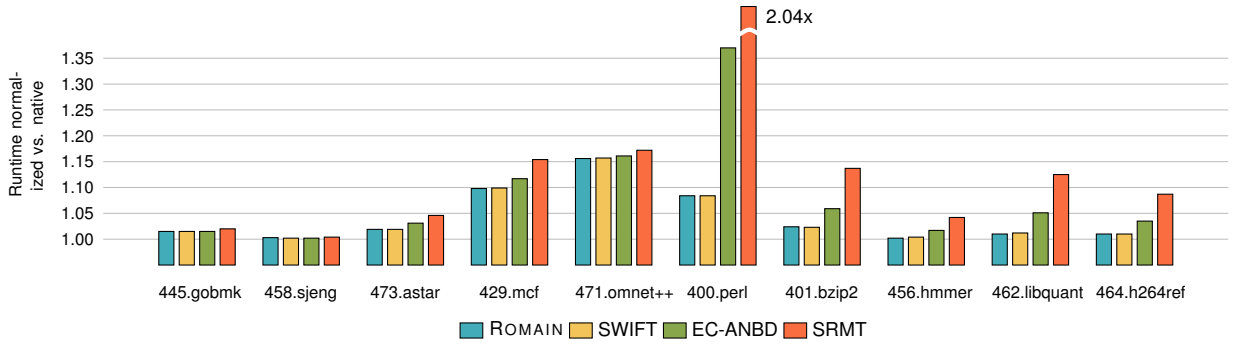

Figure 6.11: Estimating the cost of compiler-assisted RCB protected for the SPEC INT 2006 benchmarks running in triple-modular redundancy within ROMAIN

those slowdowns the authors reported for protecting user applications solely using compiler-assisted fault tolerance. This is because my proposed combination of user-level replication and compiler-protected RCB code retains the advantages of ROMAIN (low execution time overheads) for user-level code and only applies expensive compiler-level safeguards to those parts of the system that ROMAIN is unable to protect.

I conclude from this experiment that a combination of replication and compiler techniques to protect RCB code appears to be a promising path towards achieving full-system protection against hardware errors on commercial-off-the-shelf platforms. To further investigate this idea, future work will first have to solve the remaining problems for applying fault-tolerant compiler transformations to kernel code as I explained in Section 6.1.

SUMMARY: As all software parts of ROMAIN's RCB are available as open source software, we can protect them against hardware errors using compiler-based fault tolerance methods. Based on a simulation of potential RCB slowdowns, the approach of combining replication at the user-level and potentially expensive compiler-level protection at the RCB level appears to be a promising solution for fully protecting the software stack while achieving low execution time overheads.

7
Conclusions and Future Work

In this thesis I developed the ASTEROID operating system architecture to protect user applications against the effects of hardware errors. In this chapter I summarize the contributions of my thesis. Thereafter I outline ideas for future work, which mainly focus on reducing ASTEROID's resource footprint.

7.1 Operating-System Assisted Replication of Multithreaded Binary-Only Applications

The ASTEROID operating system architecture protects user-level applications against the effects of hardware errors arising in commercial-off-the-shelf (COTS) hardware. ASTEROID's main component is ROMAIN, an operating system service that replicates unmodified binary-only multi-threaded applications. ASTEROID meets the design goals that I identified in Section 2.5:

1. *COTS Hardware Support and Hardware-Level Concurrency:* I designed ASTEROID to work on COTS hardware. ROMAIN replicates applications for error detection and correction. To make replication efficient, I leverage the availability of parallel processors.

2. *Use of a Componentized System:* I implemented ASTEROID on top of the FIASCO.OC microkernel. Using a microkernel design allows ASTEROID to benefit from a system that is split into small, isolated components that can independently be recovered in the case of a failure. Furthermore, as microkernels run most of the traditional OS services – such as file systems and network stacks – in user space, these applications can be transparently protected against hardware errors using replication.

3. *Binary Application Support:* ROMAIN does not make any assumptions about applications with respect to the development model, programming language, libraries, or tools used for their implementation. My solution therefore allows to replicate any binary application that is able to run on top of FIASCO.OC's L4 Runtime Environment.

   There are still two exceptions that limit ROMAIN's applicability:

   (a) As explained in Chapter 4, ROMAIN requires multithreaded applications to be race-free and use a standard lock implementation in order to be replicated. In Section 4.3.7 I showed how this limitation can be removed using strongly deterministic multithreading.

(b) Device drivers make accesses to input/output resources that may have side effects, such as sending a network packet or writing data to a hard disk. Due to this fact, such accesses cannot easily be replicated and ROMAIN therefore is unable to replicate device drivers yet. This limitation needs to be addressed in future work.

4. *Efficient Error Detection, Correction, and Replication of Multi-threaded Programs:* My evaluation in Chapter 5 showed that ROMAIN is able to efficiently replicate single- and multithreaded application software. In my fault injection experiments I demonstrated that ROMAIN detects 100% of all injected single-event upsets in memory and general-purpose registers. ROMAIN furthermore provides error recovery using majority voting, which succeeded in at least 99.6% of my experiments.

   Throughout this thesis I discussed how ROMAIN manages replicated applications as well as their resources. I evaluated design alternatives to select those mechanisms that make ROMAIN efficient:

   - By replicating applications via redundant multithreading, ROMAIN achieves low replication overheads because it limits the number of state validation operations to those locations where application state becomes visible outside the sphere of replication. While this strategy reduces validation and recovery overhead, we saw in Section 5.2 that it may in turn lead to replicas executing several thousand instructions before an error is detected. For this reason ROMAIN also includes a mechanism to artificially force long-running applications to trigger state validation from time to time. This mechanism was developed by Martin Kriegel[1] and I explained it in Section 3.8.2.
   - In Section 3.5.2 I advocated to use hardware-supported large page mappings and proactive handling of page faults to reduce the memory overhead that replicated execution implies.
   - In Section 4.3 I compared two strategies to achieve deterministic replication of multithreaded, race-free applications by enforcing deterministic lock ordering across replicas. I showed that we can reduce the overhead of intercepting lock acquisition and release operations by leveraging a replication-aware `libpthread` library.
   - In Section 3.6 I showed that replicated access to shared memory has a large impact on performance and designed a copy & execute strategy for emulating shared memory accesses that is faster than traditional trap & emulate mechanisms.

5. *Protection of the Reliable Computing Base:* In Chapter 6 I explained that software-level fault tolerance mechanisms always require a correctly functioning set of software and hardware components, the Reliable Computing Base (RCB). I explained what comprises ASTEROID's RCB and discussed ideas towards protecting the RCB. While my thesis shows that these ideas – such as making kernel failures visible to user applications, leveraging mixed-criticality hardware, and protecting the RCB using compiler-assisted protection mechanisms – are feasible, I left their implementation and thorough evaluation for future work.

[1] Martin Kriegel. Bounding Error Detection Latencies for Replicated Execution. Bachelor's thesis, TU Dresden, 2013

7.2 Directions for Future Research

ROMAIN replicates applications with low execution time overhead. I now outline ideas for future research to reduce this execution time overhead and improve ROMAIN's error coverage for multi-error scenarios. For this purpose I distinguish between ideas to reduce replica resource consumption and other promising optimizations.

7.2.1 Reducing Replication-Induced Resource Consumption

Resource overhead is a major problem for any mechanism that uses replication to provide fault tolerance. Running N replicas will usually require N times the amount of resources of a single application instance. This leads to problems in systems, such as embedded computers, that need to constrain resource availability to reduce energy consumption and production cost. We furthermore saw in Section 5.3.2 that resource replication may lead to secondary problems, such as performance reduction due to an increase of last-level cache misses.

I believe that in order to reduce the resource consumption of replicated systems we have to investigate how we can reduce the number of running replicas while maintaining the fault tolerance properties ROMAIN provides.

Dynamically Adapting the Number of Replicas Replication systems, such as ROMAIN, usually fix the number of active replicas to cope with given static assumptions about the expected rate of faults. Real-world systems may however experience dynamically changing fault rates depending on environmental and software-level conditions:

- Systems become more vulnerable to soft errors when they experience higher temperatures or are located at a higher altitude above the sea level.[2] In such situations it may be beneficial to increase the number of running replicas when environmental conditions – reported by external sensors – change.
- Different parts of a program may have different vulnerabilities to soft errors. Program Vulnerability Factor (PVF) analysis allows to detect these variations.[3] Some software functionality may therefore require increased protection while other sequences of code may run less protected.

Both observations open up the potential for reducing the number of replicas and hence the amount of replicated resources for periods of lower vulnerability. Robert Muschner extended ROMAIN to support the dynamic adjustment of replicas in his Diploma Thesis, which I co-advised with Michael Roitzsch.[4]

The general idea of his thesis was to increase or decrease replica count when triggered by an external sensor. To increase the number of replicas, new replica vCPUs are started and brought into the same state as existing vCPUs by copying the state from a previously validated replica. This process is similar to how error recovery works in ROMAIN. The difference here is that new memory needs to be allocated for the newly spawned replica. In order to decrease the number of replicas, we simply halt an existing replica at its next externalization event and release the accompanying resources.

Muschner's thesis shows the feasibility of this approach. He also pointed out that dynamic replicas do not come for free: the adjustment requires

[2] Ziegler, James F. and Curtis, Huntington W. et al. IBM Experiments in Soft Fails in Computer Electronics (1978–1994). *IBM Journal of Research and Development*, 40(1):3–18, 1996

[3] Vilas Sridharan and David R. Kaeli. Quantifying Software Vulnerability. In *Workshop on Radiation effects and fault tolerance in nanometer technologies*, WREFT '08, pages 323–328, Ischia, Italy, 2008. ACM

[4] Robert Muschner. Resource Optimization for Replicated Applications. Diploma thesis, TU Dresden, 2013

additional execution time for acquiring and releasing resources. This overhead limits the frequency at which we can adjust the number of running replicas. To hide these latencies, Muschner proposed to perform resource releases in the background while the active replicas commence operation. Furthermore, he devised a copy-on-write scheme for allocating resources to a newly added replica. This latter scheme limits error coverage as replicas sharing data copy-on-write will suffer from undetected errors affecting these regions. The approach therefore requires combination with a hardware-level error detection mechanism, such as Error-Correcting Codes (ECC).[5]

[5] Shubhendu Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008

Reducing Resource Consumption by Leveraging ECC Memory In Section 3.5.3 I already pointed out that ROMAIN can benefit from a combination with ECC hardware, because then we can avoid creating copies of read-only memory regions and share them across all replicas. Future work can extend this approach by leveraging application-specific memory access patterns: memory is often written once and later only accessed in a read-only fashion. To save replica resources, ROMAIN could duplicate such memory regions during the write phases and later remove all copies except one, which would then be mapped read-only to all replicas. This idea is closely related to the field of memory deduplication for virtual machines, where such regions are searched for in order to reduce memory consumption in data centers.[6]

[6] Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More Effective Memory Deduplication Scanners Through Cross-Layer Hints. In *USENIX Annual Technical Conference*, USENIX ATC'13, San Jose, CA, USA, 2013

ECC-protected read-only memory furthermore relates to Walfield's idea of discardable memory:[7] here applications can allocate and use memory to cache data as long as there is no memory pressure. In the case of memory scarcity, the OS drops discardable data and notifies the application, which in turn may re-obtain the data once it is needed at a later point in time. ROMAIN could maintain read-only copies of such cached objects. In the case of a detected ECC failure, it would then drop all discardable memory regions and let the application take care of recovering this cached data from its previous source.

[7] Neal H. Walfield. Viengoos: A Framework For Stakeholder-Directed Resource Allocation. Technical report, 2009

Heuristics for Error Recovery I described in this thesis that ROMAIN uses majority voting to determine which replicas are faulty and need to be corrected. I argue that the existence of a majority is not always required to perform recovery. If a fault causes a replica to crash, for instance by accessing an invalid memory region, two replicas suffice for correction: the faulting replica will raise a page fault in a region that is either unknown to the ROMAIN memory manager or a region where ROMAIN knows that a valid mapping was previously established. In this case, a second replica, raising a valid externalization event, can be assumed to be correct and serve as the origin for recovery.

The situation becomes more difficult for the correction of silent data corruption (SDC) errors. If we only have two replicas running in this case, we will see two valid system calls that differ in their system call arguments. I believe that many applications have a specific set of valid system call arguments and that we can use machine learning techniques[8] to train a classifier to distinguish between valid and invalid arguments.

[8] Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997

Once trained, the classifier will be able to decide which of two replicas is the faulty one in the case of SDC. Future work will have to evaluate for what kinds of applications such heuristics work and what kind of probabilistic recovery guarantees this enables.

7.2.2 Optimizations Using Application-Level Knowledge and Hardware Extensions

Besides the previously described experiments towards reducing the number of running replicas, future work should also investigate performance and error coverage optimizations that may be enabled by incorporating more application-level knowledge into the process of replication or by leveraging hardware extensions.

Application-Defined State Validation As I explained in Section 3.8, ROMAIN currently checks the replicas' architectural register states and user-level thread control blocks in order to detect errors. For performance reasons I do not perform a complete comparison of all replica memory, arguing that as long as erroneous data remains internal to the replica, it does not hurt system correctness. However, such erroneous data may remain stored for a long time and if this time frame exceeds the expected inter-arrival rate of hardware errors, an independent second error may affect another replica and finally lead to a situation where the number of running replicas does no longer suffice for error correction by majority voting.

One way to address this problem would be to make replication-aware applications that report important in-memory state to the ROMAIN master. The master can then incorporate this state into validation in order to improve error coverage.

ROMAIN furthermore assumes that any system call is equally important for the outcome of a program and constitutes a point where data leaves the sphere of replication. There may be situations, where this is not necessarily the case:

- Applications might occasionally write debug or reporting output. Errors in such messages may affect program analysis or debugging. However, depending on the actual application they may not necessarily constitute bugs from the perspective of a user of the affected service.
- Applications might store temporary data in files on a file system. Writes to these files will therefore be considered data leaving the sphere or replication. However, if this file is never read by an external observer, it will once again not affect the service obtained by an outside observer.

If an application was replication-aware, it could mark such system calls as *less important*. ROMAIN could then decide to forgo state validation and the accompanying replica synchronization overhead and just execute a system call unchecked. Alternatively, ROMAIN would only check a fraction of these system calls to maintain error coverage, but avoid checking all of them.

Leveraging Application Knowledge to Improve Shared Memory Replication My analysis in Section 5.3.4 showed that while ROMAIN supports the replication of shared-memory applications, intercepting those shared-memory accesses incurs a significant execution time cost. The main problem here is that ROMAIN conservatively needs to assume every such access to suffer from potential inconsistencies. Depending on the specific applications involved in a shared-memory scenario, ROMAIN may improve replication performance by leveraging application knowledge.

Consider for example a shared-memory scenario where a producer application uses shared memory to send a large amount of data to a consumer as it is often the case in zero-copy data transmission scenarios.[9] In this case the producer knows exactly when data is ready to be sent to the consumer and the consumer will never read or modify data before this point in time. If a replication-aware consumer is able to convey information about data being ready to ROMAIN, we can implement the following optimization for shared-memory replication:

1. The ROMAIN master maps a private copy of the shared-memory region to each replica of a replicated producer. The replicas then treat this memory as private memory and can directly read and write it.

2. Once data is ready, the producer notifies ROMAIN about data being updated. ROMAIN can also try to infer this information by inspecting the producer's system calls as such notifications are often sent through dedicated software interrupt system calls in FIASCO.OC.

3. If the notification is seen by the ROMAIN master, it first compares the replicas' private memory regions. Upon success, the content of one such region is merged back into the original shared memory region that is seen by the consumer.

4. Finally, ROMAIN delivers the data update notification to the consumer, which then reads the data.

Hardware-Level Execution Fingerprinting to Improve Error Coverage My previous optimization suggestions focused on reducing the number of state comparisons and the amount of resources required for replicated execution. If we are willing to accept specialized hardware instead of running ROMAIN on COTS components, replication can furthermore benefit from hardware extensions aiming at improving the fault tolerance of software.

Smolens proposed an extension to the CPU pipeline that computes hash sums of the instructions and data accessed by the different pipeline stages.[10] Philip Axer implemented a similar extension in hardware for the SPARC LEON3 processor.[11] Both works showed that implementing such fingerprinting is cheap in terms of chip area and energy consumption. I had the opportunity to supervise Christian Menard's implementation of such fingerprinting in a simulated, in-order x86 processor in the GEM5 hardware simulator.[12] Menard also showed the feasibility of integrating this mechanism into ROMAIN for fast state comparison.

The benefit of the fingerprinting approach is twofold: First, instead of comparing registers and memory areas, the ROMAIN master only needs

[9] Julian Stecklina. Shrinking the Hypervisor one Subsystem at a Time: A Userspace Packet Switch for Virtual Machines. In *Conference on Virtual Execution Environments*, VEE'14, pages 189–200, 2014

[10] Jared C. Smolens, Brian T. Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Fingerprinting: Bounding Soft-Error Detection Latency and Bandwidth. In *Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 224–234, Boston, MA, USA, 2004

[11] Philip Axer, Rolf Ernst, Björn Döbel, and Hermann Härtig. Designing an Analyzable and Resilient Embedded Operating System. In *Workshop on Software-Based Methods for Robust Embedded Systems*, SOBRES'12, Braunschweig, Germany, 2012

[12] Christian Menard. Improving replication performance and error coverage using instruction and data signatures. Study thesis, TU Dresden, 2014

to compare two registers per replica (the instruction and memory footprint registers). This speeds up comparison. Second, fingerprint comparison covers all data that influenced application behavior and this approach is therefore also able to detect erroneous data in memory that ROMAIN otherwise would not find or only find at a later point in time.

The remaining problem with pipeline fingerprinting is that all existing implementations (including Menard's x86 one) were only done for in-order processors. Modern CPUs however use sophisticated speculation and prefetching techniques that make it hard to exactly determine when an instruction or a datum from memory should be incorporated into the checksum and which data should be discarded for hash computation. This problem needs to be overcome in order to make a fingerprinting extension viable for real-world high-performance processors.

Acknowledgements

The process of research and writing a dissertation is a long and winding road. By intent we move off the beaten track to discover new and interesting things. But it is easy to get lost in this jungle and more often than we would like to admit, we need a helping hand that leads us back to the path. I had the pleasure to work with bright people that lended me this hand whenever I needed it.

First of all, I would like to thank my advisor Professor Hermann Härtig for giving me the opportunity to join his group and for giving me the freedom to explore the topics that interested me most. My colleagues in the TU Dresden Operating Systems Group supported my research with curiosity, encouragement, criticism – whichever was necessary at a given point in time. Carsten Weinhold and Michael Roitzsch had an open ear for the problems that were haunting me and often pointed out details or shortcuts that I was too blind to see. Adam Lackorzynski and Alexander Warg developed the L4 Runtime Environment, which this work is based on and patiently explained its intricacies to me over and over again. Martin Pohlack and Ronald Aigner were the first persons to introduce me to research and scientific writing and I hope that this thesis meets their expectations. Benjamin Engel, Bernhard Kauer, Julian Stecklina, and Tobias Stumpf gave valuable feedback and ideas for the development of ROMAIN. Thomas Knauth and Stephan Diestelhorst burdened themselves with commenting on many early versions of research papers I wrote. Angela Spehr was always there to help me deal with the bureaucratic tribulations of a German university.

At TU Dresden I also was in the lucky position to advise some extraordinarily bright students during their Master's Thesises. This dissertation benefited from the accompanying discussions, their questions, and their results. I would therefore like to thank Dirk Vogt, Martin Unzner, Martin Kriegel, Robert Muschner, Florian Pester, and Christian Menard.

ROMAIN and the ASTEROID OS architecture were designed within the DFG-funded project ASTEROID. Philip Axer was my partner in this project and it was a pleasure to work and collaborate with him. Horst Schirmeier developed the FAIL* fault injection framework and went out of his way to help me with my fault injection encounters. Additionally, I enjoyed my work with Michael Engel on the Reliable Computing Base concept and any other discussions we had on our research interests.

I furthermore had the opportunity to get to know industrial research and kernel development during two internships at Microsoft Research, Cambridge (UK), and at VMWare in Palo Alto, CA. During these months I learned

a lot about research and problem solving from Eno Thereska, Daniel Arai, Bernhard Poess, and Bharath Chandramohan.

Attending conferences and visiting other universities allowed me to get early feedback on my work from colleagues. I enjoyed fruitful discussions with Gernot Heiser, Olaf Spinczyk, Rüdiger Kapitza, Frank Müller, Frank Bellosa, Jan Stoess, and Marius Hillenbrand.

Last but not least I would like to thank my family. My parents supported me in the endeavor of becoming a researcher. My wife Christiane shared the highs and lows of life with me and I would not want to miss any minute of it.

To summarize: Thank you! You guys rock!

List of Figures

1.1 ASTEROID Resilient OS Architecture 10
1.2 Replicated Application 11

2.1 MOSFET Transistor 15
2.2 Switching MOSFET 15
2.3 Chain of errors 18
2.4 Temporal Reliability Metrics 19
2.5 Saggese's Fault Manifestation Study 23
2.6 Triple modular redundancy (TMR) 26
2.7 DIVA Architecture 27

3.1 ASTEROID System Architecture 40
3.2 ROMAIN Architecture 41
3.3 FIASCO.OC Exception Handling 43
3.4 Handling Externalization Events 44
3.5 Event Handling Loop 45
3.6 FIASCO.OC Object Capabilities 45
3.7 Replicated and Unreplicated Interaction 47
3.8 Translating Replica Capability Selectors 47
3.9 Partitioned Capability Tables 48
3.10 FIASCO.OC Region Management 50
3.11 Per-Replica Memory Regions 51
3.12 Replication Meets ECC 52
3.13 Memory management microbenchmark 53
3.14 Memory Management Overhead 53
3.15 The Mapping-Alignment Problem 55
3.16 Adjusting Alignment for Large Page Mappings 55
3.17 Reduced Page Fault Handling Overhead 56
3.18 Optimized Memory Management Results 56
3.19 Trap & emulate Runtime Overhead 59
3.20 Trap & emulate Emulation Cost 60
3.21 Copy & Execute Overhead 61
3.22 Copy & Execute Emulation Cost 62
3.23 Error Detection Latencies 68

4.1 Blocking Synchronization 72
4.2 Thread Pool Example 73
4.3 Worker thread implementation 73
4.4 Example Schedules 73

4.5 Terminology overview 79
4.6 Multithreaded event handling in ROMAIN 79
4.7 Externalizing Lock Operations 82
4.8 Thread microbenchmark 83
4.9 Worst-Case Multithreading Overhead 83
4.10 Sequential CPU Assignment 84
4.11 Optimized CPU Assignment 84
4.12 Execution phases of a single replica 85
4.13 Multithreaded Execution Breakdown 85
4.14 Optimized Multithreading Benchmark 86
4.15 Optimized Multithreading Breakdown 87
4.16 Lock Info Page Architecture 88
4.17 Lock Info Page Structure 88
4.18 Replication-Aware Lock Function 89
4.19 Replication-Aware Unlock Function 89
4.20 Cooperative Determinism Overhead 90
4.21 LIP Protection 94

5.1 Fault Space for Single-Event Upsets in Memory 100
5.2 Error Coverage: Register SEUs 104
5.3 Error Coverage: Memory SEUs 104
5.4 Error Detection Latency 105
5.5 Single-CPU Replica Schedules 106
5.6 Replica Execution after Fault Injection 107
5.7 Computing error detection latency 108
5.8 SPEC CPU Overhead 111
5.9 SPEC CPU Overhead (Incomplete) 111
5.10 Overhead by externalization event ratio 112
5.11 SPEC CPU Breakdown 113
5.12 Cache-Aware CPU Assignment 114
5.13 SPEC CPU Overhead (Improved) 115
5.14 SPLASH2: Replication overhead for two application threads 116
5.15 SPLASH2: Replication overhead for four application threads 116
5.16 Multithreaded Replication Problems 117
5.17 SPLASH2 Overhead with Enforced Determinism 118
5.18 SHMC Application Benchmark 118
5.19 Shared Memory Throughput 119
5.20 Microbenchmarks: Recovery Time 120
5.21 Recovery Time and Memory Footprint 121

6.1 FIASCO.OC Fault Injection Results 134
6.2 Distribution of CRASH failure types 135
6.3 Mixed Reliability Hardware Platform 140
6.4 Switching to an SRC by thread migration 142
6.5 Triggering SRC execution using synchronous IPC 142
6.6 Notifying the SRC using shared-memory polling 143
6.7 Bitcount overhead on SRC 144
6.8 Susan overhead on SRC 144
6.9 Lame overhead on SRC 144

6.10 CRC32 overhead on SRC 145
6.11 Estimation of Compiler-Based RCB Protection Overhead 148

8
Bibliography

Mike Accetta, Robert Baron, William Bolosky, David Golub, Richard Rashid, Avadis Tevanian, and Michael Young. Mach: A New Kernel Foundation for UNIX Development. In *USENIX Technical Conference*, pages 93–112, 1986.

Ronald Aigner. *Communication in Microkernel-Based Systems*. Dissertation, TU Dresden, 2011.

Muhammad Ashraful Alam, Haldun Kufluoglu, D. Varghese, and S. Mahapatra. A Comprehensive Model for PMOS NBTI Degradation: Recent Progress. *Microelectronics Reliability*, 47(6):853–862, 2007.

Jason Ansel, Kapil Arya, and Gene Cooperman. DMTCP: Transparent Checkpointing for Cluster Computations and the Desktop. In *23rd IEEE International Parallel and Distributed Processing Symposium*, Rome, Italy, May 2009.

Jean Arlat, Jean-Charles Fabre, Manuel Rodríguez, and Frédéric Salles. Dependability of COTS Microkernel-Based Systems. *IEEE Transactions on Computing*, 51(2):138–163, February 2002.

ARM Ltd. Big.LITTLE processing with ARM Cortex-A15. Whitepaper, 2011.

Mohit Aron, Luke Deller, Kevin Elphinstone, Trent Jaeger, Jochen Liedtke, and Yoonho Park. The SawMill Framework for Virtual Memory Diversity. In *Asia-Pacific Computer Systems Architecture Conference*, Bond University, Gold Coast, QLD, Australia, January 29–February 2 2001.

Todd M. Austin. DIVA: A Reliable Substrate for Deep Submicron Microarchitecture Design. In *International Symposium on Microarchitecture*, MICRO'32, pages 196–207, Haifa, Israel, 1999. IEEE Computer Society.

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 65nm SRAM. In *European Conference on Radiation and Its Effects on Components and Systems*, RADECS'08, pages 519–524, 2008.

Amittai Aviram and Bryan Ford. Deterministic OpenMP for Race-Free Parallelism. In *Conference on Hot Topics in Parallelism*, HotPar'11, Berkeley, CA, 2011. USENIX Association.

Amittai Aviram, Bryan Ford, and Yu Zhang. Workspace Consistency: A Programming Model for Shared Memory Parallelism. In *Workshop on Determinism and Correctness in Parallel Programming*, WoDet'11, Newport Beach, CA, 2011.

Amittai Aviram, Shu-Chun Weng, Sen Hu, and Bryan Ford. Efficient System-enforced Deterministic Parallelism. pages 193–206, Vancouver, BC, Canada, 2010. USENIX Association.

Algirdas Avizienis, Jean-Claude Laprie, Brian Randell, and Carl Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

Philip Axer, Rolf Ernst, Björn Döbel, and Hermann Härtig. Designing an Analyzable and Resilient Embedded Operating System. In *Workshop on Software-Based Methods for Robust Embedded Systems*, SOBRES'12, Braunschweig, Germany, 2012.

Philip Axer, Moritz Neukirchner, Sophie Quinton, Rolf Ernst, Björn Döbel, and Hermann Härtig. Response-Time Analysis of Parallel Fork-Join Workloads with Real-Time Constraints. In *Euromicro Conference on Real-Time Systems*, ECRTS'13, Jul 2013.

Claudio Basile, Zbigniew Kalbarczyk, and Ravishankar K. Iyer. Active Replication of Multithreaded Applications. *Transactions on Parallel Distributed Systems*, 17(5):448–465, May 2006.

Robert Baumann. Soft Errors in Advanced Computer Systems. *IEEE Design Test of Computers*, 22(3):258–266, 2005.

Tom Bergan, Owen Anderson, Joseph Devietti, Luis Ceze, and Dan Grossman. CoreDet: A Compiler and Runtime System for Deterministic Multithreaded Execution. In *Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 53–64, Pittsburgh, Pennsylvania, USA, 2010. ACM.

Tom Bergan, Nicholas Hunt, Luis Ceze, and Steve Gribble. Deterministic Process Groups in dOS. In *Symposium on Operating Systems Design & Implementation*, OSDI'10, pages 177–192, Vancouver, BC, Canada, 2010. USENIX Association.

Emery D. Berger, Ting Yang, Tongping Liu, and Gene Novark. Grace: Safe Multithreaded Programming for C/C++. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '09, pages 81–96, Orlando, Florida, USA, 2009. ACM.

David Bernick, Bill Bruckert, Paul del Vigna, David Garcia, Robert Jardine, Jim Klecka, and Jim Smullen. NonStop: Advanced Architecture. In *International Conference on Dependable Systems and Networks*, pages 12–21, June 2005.

James R. Black. Electromigration – A Brief Survey and Some Recent Results. *IEEE Transactions on Electron Devices*, 16(4):338–347, 1969.

Robert L. Bocchino, Jr., Vikram S. Adve, Danny Dig, Sarita V. Adve, Stephen Heumann, Rakesh Komuravelli, Jeffrey Overbey, Patrick Simmons, Hyojin Sung, and Mohsen Vakilian. A Type and Effect System for Deterministic Parallel Java. In *Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA'09, pages 97–116, Orlando, Florida, USA, 2009. ACM.

Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Protecting the dynamic dispatch in C++ by dependability aspects. In *GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '12)*, Lecture Notes in Informatics, pages 521–535. German Society of Informatics, September 2012.

Christoph Borchert, Horst Schirmeier, and Olaf Spinczyk. Generative Software-Based Memory Error Detection and Correction for Operating System Data Structures. In *International Conference on Dependable Systems and Networks*, DSN'13. IEEE Computer Society Press, June 2013.

Sherkar Borkar. Designing Reliable Systems From Unreliable Components: The Challenges of Transistor Variability and Degradation. *IEEE Micro*, 25(6):10 – 16, 2005.

Francisco V. Brasileiro, Paul D. Ezhilchelvan, Santosh K. Shrivastava, Neil A. Speirs, and S. Tao. Implementing Fail-Silent Nodes for Distributed Systems. *Computers, IEEE Transactions on*, 45(11):1226–1238, 1996.

Thomas C. Bressoud and Fred B. Schneider. Hypervisor-Based Fault Tolerance. *ACM Transactions on Computing Systems*, 14:80–107, February 1996.

W. G. Brown, J. Tierney, and R. Wasserman. Improvement of Electronic-Computer Reliability Through the Use of Redundancy. *IRE Transactions on Electronic Computers*, EC-10(3):407–416, 1961.

Derek Bruening and Qin Zhao. Practical Memory Checking with Dr. Memory. In *Symposium on Code Generation and Optimization*, CGO '11, pages 213–223, 2011.

George Candea, Shinichi Kawamoto, Yuichi Fujiki, Greg Friedman, and Armando Fox. Microreboot: A Technique For Cheap Recovery. In *Symposium on Operating Systems Design & Implementation*, OSDI'04, Berkeley, CA, USA, 2004. USENIX Association.

Hyungmin Cho, Shahrzad Mirkhani, Chen-Yong Cher, Jacob A. Abraham, and Subhasish Mitra. Quantitative Evaluation of Soft Error Injection Techniques for Robust System Design. In *Design Automation Conference (DAC), 2013 50th ACM / EDAC / IEEE*, pages 1–10, 2013.

Intel Corp. Intel Digital Random Number Generator (DRNG) – Software Implementation Guide. Technical Documentation at `http://www.intel.com`, 2012.

Intel Corp. Intel64 and IA-32 Architectures Software Developer's Manual. Technical Documentation at `http://www.intel.com`, 2013.

Intel Corp. Software Guard Extensions – Programming Reference. Technical Documentation at `http://www.intel.com`, 2013.

C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole. Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade. In *DARPA Information Survivability Conference and Exposition*, volume 2, pages 119–129 vol.2, 2000.

Heming Cui, Jiri Simsa, Yi-Hong Lin, Hao Li, Ben Blum, Xinan Xu, Junfeng Yang, Garth A. Gibson, and Randal E. Bryant. Parrot: A Practical Runtime for Deterministic, Stable, and Reliable Threads. In *ACM Symposium on Operating Systems Principles*, SOSP'13, pages 388–405, Farminton, Pennsylvania, 2013. ACM.

Heming Cui, Jingyue Wu, Chia-Che Tsai, and Junfeng Yang. Stable Deterministic Multithreading Through Schedule Memoization. In *Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–13, Vancouver, BC, Canada, 2010. USENIX Association.

L. Dagum and R. Menon. OpenMP: An Industry Standard API for Shared-Memory Programming. *Computational Science Engineering, IEEE*, 5(1):46–55, Jan 1998.

Matt Davis. Creating a vDSO: The Colonel's Other Chicken. Linux Journal, mirror: `http://tudos.org/~doebel/phd/vdso2012/`, February 2012.

Julian Delange and Laurent Lec. POK, an ARINC653-compliant operating system released under the BSD license. In *Realtime Linux Workshop*, RTLWS'11, 2011.

Timothy J. Dell. A White Paper on the Benefits of Chipkill-Correct ECC for PC Server Main Memory. IBM Whitepaper, 1997.

Department of Defense. *Trusted Computer System Evaluation Criteria*, December 1985. DOD 5200.28-STD (supersedes CSC-STD-001-83).

Alex Depoutovitch and Michael Stumm. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *European Conference on Computer Systems*, EuroSys '10, pages 181–194, Paris, France, 2010. ACM.

Edsger. W. Dijkstra. A Note on Two Problems in Connexion With Graphs. *Numerische Mathematik*, 1:269–271, 1959.

Artem Dinaburg. Bitsquatting: DNS Hijacking Without Exploitation. *BlackHat Conference*, 2011.

A. Dixit and Alan Wood. The Impact of new Technology on Soft Error Rates. In *IEEE Reliability Physics Symposium*, IRPS'11, pages 5B.4.1–5B.4.7, 2011.

Björn Döbel and Hermann Härtig. Who Watches the Watchmen? – Protecting Operating System Reliability Mechanisms. In *Workshop on Hot Topics in System Dependability*, HotDep'12, Hollywood, CA, 2012.

Björn Döbel and Hermann Härtig. Where Have all the Cycles Gone? – Investigating Runtime Overheads of OS-Assisted Replication. In *Workshop on Software-Based Methods for Robust Embedded Systems*, SOBRES'13, Koblenz, Germany, 2013.

Björn Döbel and Hermann Härtig. Can We Put Concurrency Back Into Redundant Multithreading? In *14th International Conference on Embedded Software*, EMSOFT'14, New Delhi, India, 2014.

Björn Döbel, Hermann Härtig, and Michael Engel. Operating System Support for Redundant Multithreading. In *12th International Conference on Embedded Software*, EMSOFT'12, Tampere, Finland, 2012.

Björn Döbel, Horst Schirmeier, and Michael Engel. Investigating the Limitations of PVF for Realistic Program Vulnerability Assessment. In *Workshop on Design For Reliability (DFR)*, 2013.

Nelson Elhage. Attack of the Cosmic Rays! KSPlice Blog, 2010, `https://blogs.oracle.com/ksplice/entry/attack_of_the_cosmic_rays1`, accessed on April 22nd 2013.

Kevin Elphinstone and Gernot Heiser. From L3 to seL4: What Have We Learnt in 20 Years of L4 Microkernels? In *Symposium on Operating Systems Principles*, SOSP'13, pages 133–150, Farminton, Pennsylvania, 2013. ACM.

Michael Engel and Björn Döbel. The Reliable Computing Base: A Paradigm for Software-Based Reliability. In *Workshop on Software-Based Methods for Robust Embedded Systems*, 2012.

Dawson Engler and David Yu et al. Chen. Bugs as Deviant Behavior: A General Approach to Inferring Errors in Systems Code. In *Symposium on Operating Systems Principles*, SOSP'01, pages 57–72, Banff, Alberta, Canada, 2001. ACM.

Ernst, Dan and Nam Sung Kim et al. Razor: A Low-Power Pipeline Based on Circuit-Level Timing Speculation. In *International Symposium on Microarchitecture*, MICRO'36, pages 7–18, 2003.

Hadi Esmaeilzadeh, Emily Blem, Renee St. Amant, Karthikeyan Sankaralingam, and Doug Burger. Dark Silicon and the End of Multicore Scaling. In *Annual International Symposium on Computer Architecture*, ISCA'11, pages 365–376, San Jose, California, USA, 2011. ACM.

Dae Hyun Kim et al. 3D-MAPS: 3D Massively Parallel Processor With Stacked Memory. In *Solid-State Circuits Conference Digest of Technical Papers (ISSCC), 2012 IEEE International*, pages 188–190, 2012.

David Fiala, Frank Mueller, Christian Engelmann, Rolf Riesen, Kurt Ferreira, and Ron Brightwell. Detection and Correction of Silent Data Corruption for Large-Scale High-Performance Computing. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 78:1–78:12, Salt Lake City, Utah, 2012. IEEE Computer Society Press.

Cristian Florian. Report: Most Vulnerable Operating Systems and Applications in 2013. GFI Blog, accessed on July 29th 2014, `http://www.gfi.com/blog/report-most-vulnerable-operating-systems-and-applications-in-2013/`.

International Organization for Standardization. ISO 26262: Road Vehicles – Functional Safety, 2011.

Matteo Frigo, Charles E. Leiserson, and Keith H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *Conference on Programming Language Design and Implementation*, PLDI'98, pages 212–223, Montreal, Quebec, Canada, June 1998.

Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1995.

Narayanan Ganapathy and Curt Schimmel. General Purpose Operating System Support for Multiple Page Sizes. In *USENIX Annual Technical Conference*, ATC '98, Berkeley, CA, USA, 1998. USENIX Association.

Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Phillip Gibbons, Anoop Gupta, and John Hennessy. Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors. In *International Symposium on Computer Architecture*, ISCA '90, pages 15–26, Seattle, Washington, USA, 1990. ACM.

Cristiano Giuffrida, Lorenzo Cavallaro, and Andrew S. Tanenbaum. We Crashed, Now What? In *Workshop on Hot Topics in System Dependability*, HotDep'10, Vancouver, BC, Canada, 2010. USENIX Association.

James Glanz. Power, Pollution and the Internet. The New York Times, accessed on July 1st 2013, mirror: `http://os.inf.tu-dresden.de/~doebel/phd/nyt2012util/article.html`, September 2012.

Jim Gray. Why Do Computers Stop and What Can Be Done About It? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, 1986.

Weining Gu, Z. Kalbarczyk, and R.K. Iyer. Error Sensitivity of the Linux Kernel Executing on PowerPC G4 and Pentium 4 Processors. In *Conference on Dependable Systems and Networks*, DSN'04, pages 887–896, June 2004.

Zhenyu Guo, Chuntao Hong, Mao Yang, Dong Zhou, Lidong Zhou, and Li Zhuang. Rex: Replication at the Speed of Multi-core. In *European Conference on Computer Systems*, EuroSys '14, Amsterdam, The Netherlands, 2014. ACM.

M. R. Guthaus, J. S. Ringenberg, D. Ernst, T. M. Austin, T. Mudge, and R. B. Brown. MiBench: A Free, Commercially Representative Embedded Benchmark Suite. In *International Workshop on Workload Characterization*, pages 3–14, Austin, TX, USA, 2001. IEEE Computer Society.

Tom R. Halfhill. Processor Watch: DRAM+CPU Hybrid Breaks Barriers. Linley Group, 2011, accessed on July 26th 2013, mirror: `http://tudos.org/~doebel/phd/linley11core/`.

Richard W. Hamming. Error Detecting And Error Correcting Codes. *Bell System Technical Journal*, 29:147–160, 1950.

Siva K. S. Hari, Sarita V. Adve, and Helia Naeimi. Low-Cost Program-Level Detectors for Reducing Silent Data Corruptions. In *42nd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 1–12, 2012.

Siva Kumar Sastry Hari, Sarita V. Adve, Helia Naeimi, and Pradeep Ramachandran. Relyzer: Exploiting Application-Level Fault Equivalence to Analyze Application Resiliency to Transient Faults. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 123–134, New York, NY, USA, 2012. ACM.

Tim Harris, Martin Maas, and Virendra J. Marathe. Callisto: Co-Scheduling Parallel Runtime Systems. In *European Conference on Computer Systems*, EuroSys '14, Amsterdam, The Netherlands, 2014. ACM.

Andreas Heinig, Ingo Korb, Florian Schmoll, Peter Marwedel, and Michael Engel. Fast and Low-Cost Instruction-Aware Fault Injection. In *GI Workshop on Software-Based Methods for Robust Embedded Systems (SOBRES '13)*, 2013.

Daniel Henderson and Jim Mitchell. POWER7 System RAS – Key Aspects of Power Systems Reliability, Availability, and Servicability. IBM Whitepaper, 2012.

Jörg Henkel, Lars Bauer, Nikil Dutt, Puneet Gupta, Sani Nassif, Muhammad Shafique, Mehdi Tahoori, and Norbert Wehn. Reliable On-chip Systems in the Nano-Era: Lessons Learnt and Future Trends. In *Annual Design Automation Conference*, DAC '13, pages 99:1–99:10, Austin, Texas, 2013. ACM.

John L. Hennessy and David A. Patterson. *Computer Architecture: A Quantitative Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 3rd edition, 2003.

Jorrit N. Herder. *Building a Dependable Operating System: Fault Tolerance in MINIX3*. Dissertation, Vrije Universiteit Amsterdam, 2010.

Maurice Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.

Maurice Herlihy and Nir Shavit. *The Art of Multiprocessor Programming*. Morgan Kaufmann Publishers, 2008.

Tomas Hruby, Dirk Vogt, Herbert Bos, and Andrew S. Tanenbaum. Keep Net Working - On a Dependable and Fast Networking Stack. In *Conference on Dependable Systems and Networks*, Boston, MA, June 2012.

Mei-Chen Hsueh, Timothy K. Tsai, and Ravishankar K. Iyer. Fault Injection Techniques and Tools. *IEEE Computer*, 30(4):75–82, Apr 1997.

Kuang-Hua Huang and Jacob A. Abraham. Algorithm-Based Fault Tolerance for Matrix Operations. *IEEE Transactions on Computers*, C-33(6):518–528, 1984.

Andy A. Hwang, Ioan A. Stefanovici, and Bianca Schroeder. Cosmic Rays Don't Strike Twice: Understanding the Nature of DRAM Errors and the Implications for System Design. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XVII, pages 111–122, London, England, UK, 2012. ACM.

IBM. PowerPC 750GX lockstep facility. IBM Application Note, 2008.

James Jeffers and James Reinders. *Intel Xeon Phi Coprocessor High Performance Programming*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2013.

Dawon Kahng. Electrified Field-Controlled Semiconductor Device. US Patent No. 3,102,230, `http://www.freepatentsonline.com/3102230.html`, 1963.

Rüdiger Kapitza, Matthias Schunter, Christian Cachin, Klaus Stengel, and Tobias Distler. Storyboard: Optimistic Deterministic Multithreading. In *Workshop on Hot Topics in System Dependability*, HotDep'10, pages 1–8, Vancouver, BC, Canada, 2010. USENIX Association.

M. Kapritsos, Y. Wang, V. Quema, A. Clement, L. Alvisi, and M. Dahlin. EVE: Execute-Verify Replication for Multi-Core Servers. In *Symposium on Opearting Systems Design & Implementation*, OSDI'12, Oct 2012.

John Keane and Chris H. Kim. An Odomoeter for CPUs. *IEEE Spectrum*, 48(5):28–33, 2011.

Piyus Kedia and Sorav Bansal. Fast Dynamic Binary Translation for the Kernel. In *Symposium on Operating Systems Principles*, SOSP '13, pages 101–115, Farminton, Pennsylvania, 2013. ACM.

Gabriele Keller, Toby Murray, Sidney Amani, Liam O'Connor, Zilin Chen, Leonid Ryzhyk, Gerwin Klein, and Gernot Heiser. File Systems Deserve Verification Too! In *Workshop on Programming Languages and Operating Systems*, PLOS '13, pages 1:1–1:7, Farmington, Pennsylvania, 2013. ACM.

Avi Kivity. KVM: The Linux Virtual Machine Monitor. In *The Ottawa Linux Symposium*, pages 225–230, July 2007.

V. B. Kleeberger, C. Gimmler-Dumont, C. Weis, A. Herkersdorf, D. Mueller-Gritschneder, S. R. Nassif, U. Schlichtmann, and N. Wehn. A Cross-Layer Technology-Based Study of how Memory Errors Impact System Resilience. *IEEE Micro*, 33(4):46–55, 2013.

Gerwin Klein, Kevin Elphinstone, Gernot Heiser, June Andronick, David Cock, Philip Derrin, Dhammika Elkaduwe, Kai Engelhardt, Rafal Kolanski, Michael Norrish, Thomas Sewell, Harvey Tuch, and Simon Winwood. seL4: Formal Verification of an OS Kernel. In *Symposium on Operating Systems Principles*, SOSP'09, pages 207–220, Big Sky, MT, USA, October 2009. ACM.

Donald E. Knuth. *The Art of Computer Programming, Volume 3: Sorting and Searching*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1998.

Philip Koopman. 32bit Cyclic Redundancy Codes for Internet Applications. In *Conference on Dependable Systems and Networks*, DSN '02, pages 459–472, Washington, DC, USA, 2002. IEEE Computer Society.

Martin Kriegel. Bounding Error Detection Latencies for Replicated Execution. Bachelor's thesis, TU Dresden, 2013.

Adam Lackorzynski. L^{4}Linux Porting Optimizations. Diploma thesis, TU Dresden, 2004.

Adam Lackorzynski and Alexander Warg. Taming Subsystems: Capabilities as Universal Resource Access Control in L4. In *Workshop on Isolation and Integration in Embedded Systems*, IIES'09, pages 25–30, Nuremburg, Germany, 2009. ACM.

Adam Lackorzynski, Alexander Warg, and Michael Peter. Generic Virtualization with Virtual Processors. In *Proceedings of Twelfth Real-Time Linux Workshop*, Nairobi, Kenya, October 2010.

Leslie Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers*, 28(9):690–691, September 1979.

Doug Lea. *Concurrent Programming In Java. Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

Dongyoon Lee, Benjamin Wester, Kaushik Veeraraghavan, Satish Narayanasamy, Peter M. Chen, and Jason Flinn. Respec: Efficient Online Multiprocessor Replay via Speculation and External Determinism. In *Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XV, pages 77–90, Pittsburgh, Pennsylvania, USA, 2010. ACM.

Edward A. Lee. The Problem with Threads. *Computer*, 39(5):33–42, May 2006.

L. Leem, Hyungmin Cho, J. Bau, Q.A. Jacobson, and S Mitra. ERSA: Error Resilient System Architecture for Probabilistic Applications. In *Design, Automation Test in Europe Conference Exhibition*, DATE'10, pages 1560–1565, 2010.

Andrew Lenharth, Vikram S. Adve, and Samuel T. King. Recovery Domains: An Organizing Principle for Recoverable Operating Systems. In *14th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 49–60, New York, NY, USA, 2009. ACM.

Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines. In *Symposium on Operating Systems Design and Implementation*, SOSP'04, San Francisco, CA, December 2004.

John R. Levine. *Linkers and Loaders*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 1999.

David Levinthal. Performance Analysis Guide for Intel Core i7 Processor and Intel Xeon 5500 Processors. Technical report, Intep Corp., 2009. `https://software.intel.com/sites/products/collateral/hpc/vtune/performance_analysis_guide.pdf`.

Dong Li, Zizhong Chen, Panruo Wu, and Jeffrey S. Vetter. Rethinking Algorithm-Based Fault Tolerance with a Cooperative Software-Hardware Approach. In *International Conference for High Performance Computing, Networking, Storage and Analysis*, SC'13, pages 44:1–44:12, Denver, Colorado, 2013. ACM.

Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIII, pages 265–276, Seattle, WA, USA, 2008. ACM.

Xin Li, Kai Shen, Michael C. Huang, and Lingkun Chu. A Memory Soft Error Measurement on Production Systems. In *USENIX Annual Technical Conference*, ATC'07, pages 275–280, June 2007.

Jochen Liedtke. Improving IPC by Kernel Design. In *ACM Symposium on Operating Systems Principles*, SOSP '93, pages 175–188, Asheville, North Carolina, USA, 1993. ACM.

Tongping Liu, Charlie Curtsinger, and Emery D. Berger. Dthreads: Efficient Deterministic Multithreading. In *Symposium on Operating Systems Principles*, SOSP '11, pages 327–336, Cascais, Portugal, 2011. ACM.

Jork Löser, Lars Reuther, and Hermann Härtig. A Streaming Interface for Real-Time Interprocess Communication. Technical report, TU Dresden, August 2001. URL: `http://os.inf.tu-dresden.de/papers_ps/dsi_tech_report.pdf`.

M.N. Lovellette, K.S. Wood, D. L. Wood, J.H. Beall, P.P. Shirvani, N. Oh, and E.J. McCluskey. Strategies for Fault-Tolerant, Space-Based Computing: Lessons Learned from the ARGOS Testbed. In *Aerospace Conference Proceedings, 2002. IEEE*, volume 5, pages 5–2109–5–2119 vol.5, 2002.

Chi-Keung Luk, Robert Cohn, Robert Muth, Harish Patil, Artur Klauser, Geoff Lowney, Steven Wallace, Vijay Janapa Reddi, and Kim Hazelwood. Pin: Building Customized Program Analysis Tools With Dynamic Instrumentation. In *ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '05, pages 190–200, New York, NY, USA, 2005. ACM.

Daniel Lyons. Sun Screen. Forbes Magazine, November 2000, accessed on April 22nd 2013, mirror: `http://tudos.org/~doebel/phd/forbes2000sun`.

Henrique Madeira, Raphael R. Some, Francisco Moreira, Diamantino Costa, and David Rennels. Experimental Evaluation of a COTS System for Space Applications. In *International Conference on Dependable Systems and Networks*, DSN 2002, pages 325–330, 2002.

Aamer Mahmood, Dorothy M. Andrews, and Edward J. McClusky. *Executable Assertions and Flight Software*. Center for Reliable Computing, Computer Systems Laboratory, Dept. of Electrical Engineering and Computer Science, Stanford University, 1984.

Jose Maiz, Scott Hareland, Kevin Zhang, and Patrick Armstrong. Characterization of Multi-Bit Soft Error Events in Advanced SRAMs. In *IEEE International Electron Devices Meeting*, pages 21.4.1–21.4.4, 2003.

Steve McConnell. *Code Complete: A Practical Handbook of Software Construction*. Microsoft Press, Redmond, WA, 2 edition, 2004.

Albert Meixner and Daniel J. Sorin. Detouring: Translating Software to Circumvent Hard Faults in Simple Cores. In *International Conference on Dependable Systems and Networks (DSN)*, pages 80–89, 2008.

Christian Menard. Improving replication performance and error coverage using instruction and data signatures. Study thesis, TU Dresden, 2014.

Timothy Merrifield and Jakob Eriksson. Conversion: Multi-Version Concurrency Control for Main Memory Segments. In *European Conference on Computer Systems*, EuroSys '13, pages 127–139, Prague, Czech Republic, 2013. ACM.

Microsoft Corp. Symbol Stores and Symbol Servers. Microsoft Developer Network, accessed on July 12th 2014, `http://msdn.microsoft.com/library/windows/hardware/ff558840(v=vs.85).aspx`.

Konrad Miller, Fabian Franz, Marc Rittinghaus, Marius Hillenbrand, and Frank Bellosa. XLH: More Effective Memory Deduplication Scanners Through Cross-Layer Hints. In *USENIX Annual Technical Conference*, USENIX ATC'13, San Jose, CA, USA, 2013.

Mark Miller, Ka-Ping Yee, Jonathan Shapiro, and Combex Inc. Capability Myths Demolished. Technical report, Johns Hopkins University, 2003.

Miguel Miranda. When Every Atom Counts. *IEEE Spectrum*, 49(7):32–32, 2012.

Thomas M. Mitchell. *Machine Learning*. McGraw-Hill, Inc., New York, NY, USA, 1 edition, 1997.

Gordon E. Moore. Cramming More Components Onto Integrated Circuits. *Electronics*, 38(8), 1965.

Boris Motruk, Jonas Diemer, Rainer Buchty, Rolf Ernst, and Mladen Berekovic. IDAMC: A Many-Core Platform with Run-Time Monitoring for Mixed-Criticality. In *International Symposium on High-Assurance Systems Engineering*, HASE'12, pages 24–31, Oct 2012.

Shubhendu Mukherjee. *Architecture Design for Soft Errors*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2008.

Shubhendu S. Mukherjee, Joel Emer, Tryggve Fossum, and Steven K. Reinhardt. Cache Scrubbing in Microprocessors: Myth or Necessity? In *Pacific Rim International Symposium on Dependable Computing*, PRDC '04, pages 37–42, Washington, DC, USA, 2004. IEEE Computer Society.

Shubhendu S. Mukherjee, Christopher Weaver, Joel Emer, Steven K. Reinhardt, and Todd Austin. A Systematic Methodology to Compute the Architectural Vulnerability Factors for a High-Performance Microprocessor. In *International Symposium on Microarchitecture*, MICRO 36, Washington, DC, USA, 2003. IEEE Computer Society.

Robert Muschner. Resource Optimization for Replicated Applications. Diploma thesis, TU Dresden, 2013.

Hamid Mushtaq, Zaid Al-Ars, and Koen L. M. Bertels. Efficient Software Based Fault Tolerance Approach on Multicore Platforms. In *Design, Automation & Test in Europe Conference*, Grenoble, France, March 2013.

Nicholas Nethercote and Julian Seward. Valgrind: A Framework for Heavyweight Dynamic Binary Instrumentation. In *ACM SIGPLAN conference on Programming Language Design and Implementation*, PLDI '07, pages 89–100, New York, NY, USA, 2007. ACM.

Adrian Nistor, Darko Marinov, and Josep Torrellas. Light64: lightweight hardware support for data race detection during systematic testing of parallel programs. In *International Symposium on Microarchitecture*, MICRO 42, pages 541–552, New York, NY, USA, 2009. ACM.

Nvidia Corp. Kepler: The World's Fastest, Most Efficient HPC Architecture. `http://www.nvidia.com/object/nvidia-kepler.html`, accessed August 1st, 2014, 2014.

Namsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Control-Flow Checking by Software Signatures. *IEEE Transactions on Reliability*, 51(1):111 –122, March 2002.

Namsuk Oh, Philip P. Shirvani, and Edward J. McCluskey. Error Detection by Duplicated Instructions in Super-Scalar Processors. *IEEE Transactions on Reliability*, 51(1):63–75, 2002.

Marek Olszewski, Jason Ansel, and Saman Amarasinghe. Kendo: Efficient Deterministic Multithreading in Software. In *Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XIV, pages 97–108, Washington, DC, USA, 2009. ACM.

Krishna V. Palem, Lakshmi N.B. Chakrapani, Zvi M. Kedem, Avinash Lingamneni, and Kirthi Krishna Muntimadugu. Sustaining Moore's Law in Embedded Computing Through Probabilistic and Approximate Design: Retrospects and Prospects. In *International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, CASES '09, pages 1–10, Grenoble, France, 2009. ACM.

Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. Faults in Linux: Ten Years Later. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '11, pages 305–318, Newport Beach, California, USA, 2011. ACM.

Florian Pester. ELK Herder: Replicating Linux Processes with Virtual Machines. Diploma thesis, TU Dresden, 2014.

Gerald J. Popek and Robert P. Goldberg. Formal Requirements for Virtualizable Third Generation Architectures. *Communications of the ACM*, 17(7):412–421, July 1974.

J. Postel. Transmission Control Protocol. RFC 793 (Standard), September 1981. Updated by RFCs 1122, 3168, 6093.

Eberle A. Rambo, Alexander Tschiene, Jonas Diemer, Leonie Ahrendts, and Rolf Ernst. Failure Analysis of a Network-on-chip for Real-Time Mixed-Critical Systems. In *Design, Automation Test in Europe Conference Exhibition*, DATE'14, 2014.

Brian Randell, Peter A. Lee, and Philip C. Treleaven. Reliability Issues in Computing System Design. *ACM Computing Surveys*, 10(2):123–165, June 1978.

Layali Rashid, Karthik Pattabiraman, and Sathish Gopalakrishnan. Towards Understanding The Effects Of Intermittent Hardware Faults on Programs. In *Workshops on Dependable Systems and Networks*, pages 101–106, June 2010.

David Ratter. FPGAs on Mars. *Xilinx Xcell Journal*, 2004.

Semeen Rehman, Muhammad Shafique, and Jörg Henkel. Instruction Scheduling for Reliability-Aware Compilation. In *Annual Design Automation Conference*, DAC '12, pages 1292–1300, San Francisco, California, 2012. ACM.

Semeen Rehman, Muhammad Shafique, Florian Kriebel, and Jörg Henkel. Reliable Software for Unreliable Hardware: Embedded Code Generation Aiming at Reliability. In *International Conference on Hardware/Software Codesign and System Synthesis*, CODES+ISSS '11, pages 237–246, Taipei, Taiwan, 2011. ACM.

Dave Reid, Campbell Millar, Gareth Roy, Scott Roy, and Asen Asenov. Analysis of Threshold Voltage Distribution Due to Random Dopants: A 100,000-Sample 3-D Simulation Study. *IEEE Transactions on Electron Devices*, 56(10):2255–2263, 2009.

Steven K. Reinhardt and Shubhendu S. Mukherjee. Transient Fault Detection via Simultaneous Multithreading. *SIGARCH Comput. Archit. News*, 28:25–36, May 2000.

George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. SWIFT: Software Implemented Fault Tolerance. In *International Symposium on Code Generation and Optimization*, CGO '05, pages 243–254, 2005.

John Rhea. BAE Systems Moves Into Third Generation RAD-hard Processors. Military & Aerospace Electronics, 2002, accessed on April 22nd 2013, mirror: `http://tudos.org/~doebel/phd/bae2002/`.

Kaushik Roy, Saibal Mukhopadhyay, and Hamid Mahmoodi-Meimand. Leakage Current Mechanisms and Leakage Reduction Techniques in Deep-Submicrometer CMOS Circuits. *Proceedings of the IEEE*, 91(2):305–327, 2003.

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, and Gernot Heiser. Dingo: Taming Device Drivers. In *ACM European Conference on Computer Systems*, EuroSys '09, pages 275–288, Nuremberg, Germany, 2009. ACM.

Leonid Ryzhyk, Peter Chubb, Ihor Kuz, Etienne Le Sueur, and Gernot Heiser. Automatic Device Driver Synthesis with Termite. In *Symposium on Operating Systems Principles*, SOSP '09, pages 73–86, Big Sky, Montana, USA, 2009. ACM.

Giacinto P. Saggese, Nicholas J. Wang, Zbigniew T. Kalbarczyk, Sanjay J. Patel, and Ravishankar K. Iyer. An Experimental Study of Soft Errors in Microprocessors. *IEEE Micro*, 25:30–39, November 2005.

Richard T. Saunders. A Study in Memcmp. *Python Developer List*, 2011.

Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. ANB- and ANBDmem-Encoding: Detecting Hardware Errors in Software. In *International Conference on Computer Safety, Reliability and Security*, Safecomp'10, Vienna, Austria, 2010.

Ute Schiffel, André Schmitt, Martin Süßkraut, and Christof Fetzer. Software-Implemented Hardware Error Detection: Costs and Gains. In *Third International Conference on Dependability*, DEPEND'10, pages 51–57, 2010.

Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. FAIL*: Towards a Versatile Fault-Injection Experiment Framework. In Gero Mühl, Jan Richling, and Andreas Herkersdorf, editors, *International Conference on Architecture of Computing Systems*, volume 200 of *ARCS'12*, pages 201–210. German Society of Informatics, March 2012.

Richard D. Schlichting and Fred B. Schneider. Fail-Stop Processors: An Approach to Designing Fault-tolerant Computing Systems. *ACM Transactions on Computer Systems*, 1:222–238, 1983.

Fred B. Schneider. Implementing Fault-Tolerant Services Using the State Machine Approach: A Tutorial. *ACM Computing Surveys*, 22(4):299–319, December 1990.

Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. DRAM Errors in the Wild: A Large-Scale Field Study. In *International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS'09, 2009.

Konstantin Serebryany and Timur Iskhodzhanov. ThreadSanitizer: Data Race Detection in Practice. In *Workshop on Binary Instrumentation and Applications*, WBIA'09, pages 62–71, New York, NY, USA, 2009. ACM.

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, 2009.

Lenin Singaravelu, Calton Pu, Hermann Härtig, and Christian Helmuth. Reducing TCB Complexity for Security-Sensitive Applications: Three Case Studies. In *European Conference on Computer Systems*, EuroSys'06, pages 161–174, 2006.

Timothy J. Slegel, Robert M. Averill III, Mark A. Check, Bruce C. Giamei, Barry W. Krumm, Christopher A. Krygowski, Wen H. Li, John S. Liptay, John D. MacDougall, Thomas J. McPherson, Jennifer A. Navarro, Eric M. Schwarz, Kevin Shum, and Charles F. Webb. IBM's S/390 G5 Microprocessor Design. *IEEE Micro*, 19(2):12–23, 1999.

Jared C. Smolens, Brian T. Gold, Jangwoo Kim, Babak Falsafi, James C. Hoe, and Andreas G. Nowatzyk. Fingerprinting: Bounding Soft-Error Detection Latency and Bandwidth. In *Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS XI, pages 224–234, Boston, MA, USA, 2004.

Livio Soares and Michael Stumm. FlexSC: Flexible System Call Scheduling with Exception-Less System Calls. In *Conference on Operating Systems Design and Implementation*, OSDI'10, pages 1–8, 2010.

Vilas Sridharan and David R. Kaeli. Quantifying Software Vulnerability. In *Workshop on Radiation effects and fault tolerance in nanometer technologies*, WREFT '08, pages 323–328, Ischia, Italy, 2008. ACM.

Vilas Sridharan and Dean Liberty. A Study of DRAM Failures in the Field. In *International Conference on High Performance Computing, Networking, Storage and Analysis*, SC '12, pages 76:1–76:11, Salt Lake City, Utah, 2012. IEEE Computer Society Press.

Julian Stecklina. Shrinking the Hypervisor one Subsystem at a Time: A Userspace Packet Switch for Virtual Machines. In *Conference on Virtual Execution Environments*, VEE'14, pages 189–200, 2014.

Luca Sterpone and Massimo Violante. An Analysis of SEU Effects in Embedded Operating Systems for Real-Time Applications. In *International Symposium on Industrial Electronics*, pages 3345–3349, June 2007.

Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. Recovering Device Drivers. *ACM Transactions on Computing Systems*, 24(4):333–360, November 2006.

Yuan Taur. The Incredible Shrinking Transistor. *IEEE Spectrum*, 36(7):25–29, 1999.

The IEEE and The Open Group. POSIX Thread Extensions 1003.1c-1995. `http://pubs.opengroup.org`, 2013.

The IEEE and The Open Group. The Open Group Base Specifications – Issue 7. `http://pubs.opengroup.org`, 2013.

Dean M. Tullsen, Susan J. Eggers, and Henry M. Levy. Simultaneous Multithreading: Maximizing on-chip Parallelism. In *International Symposium on Computer Architecture*, pages 392–403, 1995.

Andrew M. Tyrrell. Recovery Blocks and Algorithm-Based Fault Tolerance. In *EUROMICRO 96. Beyond 2000: Hardware and Software Design Strategies*, pages 292–299, 1996.

Martin Unzner. Implementation of a Fault Injection Framework for L4Re. Belegarbeit, TU Dresden, 2013.

Rob F. van der Wijngaart, Timothy G. Mattson, and Werner Haas. Light-weight Communications on Intel's Single-Chip Cloud Computer Processor. *SIGOPS Operating Systems Review*, 45(1):73–83, February 2011.

Dirk Vogt, Björn Döbel, and Adam Lackorzynski. Stay Strong, Stay Safe: Enhancing Reliability of a Secure Operating System. In *Workshop on Isolation and Integration for Dependable Systems*, IIDS'10, Paris, France, 2010. ACM.

Neal H. Walfield. Viengoos: A Framework For Stakeholder-Directed Resource Allocation. Technical report, 2009.

Cheng Wang, Ho-seop Kim, Youfeng Wu, and Victor Ying. Compiler-managed Software-based Redundant Multithreading for Transient Fault Detection. In *International Symposium on Code Generation and Optimization*, CGO '07, pages 244–258, 2007.

Nicholas Wang, Michael Fertig, and Sanjay Patel. Y-Branches: When You Come to a Fork in the Road, Take it. In *International Conference on Parallel Architectures and Compilation Techniques*, PACT '03, pages 56–, Washington, DC, USA, 2003. IEEE Computer Society.

Lucas Wanner, Charwak Apte, Rahul Balani, Puneet Gupta, and Mani Srivastava. Hardware Variability-Aware Duty Cycling for Embedded Sensors. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 21(6):1000–1012, 2013.

Carsten Weinhold and Hermann Härtig. jVPFS: Adding Robustness to a Secure Stacked File System with Untrusted Local Storage Components. In *USENIX Annual Technical Conference*, ATC'11, pages 32–32, Portland, OR, 2011. USENIX Association.

Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter Puschner, Jan Staschulat, and Per Stenström. The Worst-Case Execution-Time Problem – Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, 7(3):36:1–36:53, May 2008.

Kent D. Wilken and John Paul Shen. Continuous Signature Monitoring: Low-Cost Concurrent Detection of Processor Control Errors. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 9(6):629–641, 1990.

Waisum Wong, Ali Icel, and J.J. Liou. A Model for MOS Failure Prediction due to Hot-Carriers Injection. In *Electron Devices Meeting, 1996., IEEE Hong Kong*, pages 72–76, 1996.

Steven Cameron Woo, Moriyoshi Ohara, Evan Torrie, Jaswinder Pal Singh, and Anoop Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. *SIGARCH Comput. Archit. News*, 23(2):24–36, May 1995.

G. Yalcin, O.S. Unsal, A. Cristal, and M. Valero. FIMSIM: A Fault Injection Infrastructure for Microarchitectural Simulators. In *International Conference on Computer Design*, ICCD'11, 2011.

Ying-Chin Yeh. Triple-Triple Redundant 777 Primary Flight Computer. In *Aerospace Applications Conference*, volume 1, pages 293–307, 1996.

Takeshi Yoshimura, Hiroshi Yamada, and Kenji Kono. Is Linux Kernel Oops Useful or Not? In *Workshop on Hot Topics in System Dependability*, HotDep'12, pages 2–2, Hollywood, CA, 2012. USENIX Association.

Yun Zhang, Soumyadeep Ghosh, Jialu Huang, Jae W. Lee, Scott A. Mahlke, and David I. August. Runtime Asynchronous Fault Tolerance via Speculation. In *International Symposium on Code Generation and Optimization*, CGO '12, pages 145–154, 2012.

Yun Zhang, Jae W. Lee, Nick P. Johnson, and David I. August. DAFT: Decoupled Acyclic Fault Tolerance. In *International Conference on Parallel Architectures and Compilation Techniques*, PACT '10, pages 87–98, Vienna, Austria, 2010. ACM.

James F. Ziegler and William A. Lanford. Effect of Cosmic Rays on Computer Memories. *Science*, 206(4420):776–788, 1979.

Ziegler, James F. and Curtis, Huntington W. et al. IBM Experiments in Soft Fails in Computer Electronics (1978–1994). *IBM Journal of Research and Development*, 40(1):3–18, 1996.