Großer Beleg

Replicating Device Drivers on L4Re

Maurice Bailleu

14. September 2015

Technische Universität Dresden
Fakultät Informatik
Institut für Systemarchitektur
Professur Betriebssysteme

Betreuender Hochschullehrer: Prof. Dr. rer. nat. Hermann Härtig

Betreuender Mitarbeiter: M.Sc. Tobias Stumpf

TECHNISCHE UNIVERSITÄT DRESDEN

Faculty of Computer Science Institute of Systems Architecture

Großer Beleg – Student Thesis

Titel: Replicating Device Drivers on L4Re
Student: Maurice Bailleu
Supervisor: Tobias Stumpf & Florian Pester

Task:

The ASTEROID reliable operating system architecture uses replication to protect user-level applications against the effects of hardware errors. Being based on the Fiasco.OC microkernel, this architecture also allows to replicate traditional operating system services, such as file systems and network stacks [1].

The goal of this thesis is to enable replicated execution for device drivers as the last missing component to be protected at user space. Specifically, the assignment is to deal with the fact that replicated drivers must access I/O resources in a non-replicated fashion in order to avoid side effects on the hardware level.

The thesis shall be developed along the following milestones:

1) Replicating a serial device driver using the Romain replication service on top of L4Re using trap+emulate approach for device access
2) Replication of a more complex device driver (for instance Intel E1000 Network Driver) using Romain on L4Re using the above trap+emulate approach
3) Implementation of a device driver in a split fashion similar to the idea of microdrivers [2]: non-I/O specific device management code runs separated from I/O-specific code. Both driver components communicate via an IPC interface. Replication is applied to the non-I/O part only
4) Analysis how to protect the I/O-specific part of the split driver architecture (i.e., protection of the driver-specific Reliable Computing Base)

[1] Björn Döbel, Hermann Härtig, Michael Engel: Operating System Support for Redundant Multithreading, EMSOFT 2012
http://os.inf.tu-dresden.de/~doebel/papers/emsoft45-doebel.pdf
[2] V. Ganapathy, M. Renzelmann, A. Balakrishnan, M. Swift, S. Jha: The design and implementation of microdrivers, ASPLOS 2008
http://dl.acm.org/citation.cfm?id=1346303

Postadresse (Briefe)
TU Dresden, 01062 Dresden
Postadresse (Pakete u.ä.)
TU Dresden
Helmholtzstraße 10
01069 Dresden

Besucheradresse
Sekretariat:
Musterstr. 1
Zimmer 1

Steuernummer
(Inland)
203/149/02549
Umsatzsteuer-Id-Nr.
(Ausland)
DE 188 369 991

Bankverbindung
Commerzbank AG,
Filiale Dresden
IBAN
DE52 8504 0000 0800 4004 00
BIC COBADEFF850

Zufahrt
Rampe Seiteneingang, gekennzeichn. Parkfläche im Innenhof

Internet
http://tu-dresden.de

Wettbewerbsrechtlicher Hinweis

Die bloße Nennung von Namen, Produkten, Herstellern und Firmennamen dient lediglich als Information und stellt keine Verwendung des Warenzeichens sowie keine Empfehlung des Produktes oder der Firma dar.

Erklärung

Hiermit erkläre ich, dass ich diese Arbeit selbstständig erstellt und keine anderen als die angegebenen Hilfsmittel benutzt habe.

Dresden, den 14. September 2015

Maurice Bailleu

Abstract

As the physical components of logic circuits are getting smaller, error rates are getting higher due to various effects like radiation and electromagnetic coupling. Different strategies, such as costume hardware or replication of instructions through compiler features are proposed to mitigate these. Another approach is process level replication, which replicates processes and checks if the output of each process instance is identical. ROMAIN provides such a replicated application, for most unchanged processes.

ROMAIN is a major part of the ASTEROID Operating System (OS) architecture, which provides an Reliable Computing Base (RCB) for processes. All applications running outside the RCB are replicated and thereby protected by ROMAIN.

As drivers are a large part of the OS and frequently changed, it is desirable to exclude them from the RCB. Thus the goal of this work is to develop the required changes of ROMAIN and the ASTEROID OS architecture for a replication of device drivers.

This work introduces solutions for different communication mechanisms between hardware and software. These solutions have a small impact on the RCB and a feasible performance for most devices, while significantly increasing the fault tolerance of device drivers against unwanted bit-flips.

Contents

List of Figures

List of Tables

List of Acronyms

COTS Commercial off the Shelf

DMA Direct Memory Access

ICMP Internet Control Message Protocol

IPC Inter-process Communication

IRQ Interrupt Request

MEM System Memory

MMIO Memory-mapped I/O

PMIO Port-mapped I/O

RCB Reliable Computing Base

SDC Silent Data Corruption

SEU Single Event Upset

TSS Task State Segment

UART Universal Asynchronous Receiver/Transmitter

UTCB User-level Control Block

vCPU virtual CPU

1 Introduction

With the development of ever smaller transistors in modern computers, a different kind of error, besides of implementations errors, gets more prominent[11]. Single Event Upsets (SEUs) are bit-flips in hardware caused by cosmic rays[44], radioactive decay[2], aging effects[25], or electromagnetic coupling[26]. Most SEUs do not change the state of a bit permanently and can be corrected by resetting the respective transistors. These errors are named *soft-errors* or alternatively *transient faults*. Permanent faults through physical defects, which cannot be rectified in this way, are called *hard-errors*.

Since the probability of SEUs heightens with each new chip generation, even systems not used in a highly radioactive environment experience and have to mange these bit-flips. As these failures may be acceptable in consumer computers, they certainly can have catastrophic consequences on other systems, like medical equipment and high-availability servers. Furthermore, as these reliable and high-availability systems get more common solutions against SEUs have to be found and implemented.

As SEUs and therefore soft-errors are a hardware problems, it is very appealing to solve these failures in hardware. However, this requires greater development efforts. With the required features to secure the hardware against SEUs, the complexity of the components would increase, too. This is not a feasible option for device manufactures concentrating on the consumer market, as it would increase price and possibly reduce performance of the produced components. A producer of such hardware would probably loose market share. Equally, custom hardened hardware (not Commercial off the Shelf (COTS)) is expensive and therefore will be only used in systems which have a higher radiation background, for example airplanes and spacecraft[33, 43]. This makes solving the problem with hardware unfeasible in many situations. Software solutions running on COTS hardware are more sensible, as these can run on already existing hardware and do not have to be implemented anew for each hardware architecture.

Such a system has to detect, and if possible correct SEUs. For soft-errors the correction is simple, because setting the transistor anew for example through re-execution will eradicate the bit-flip. A hard-error in contrast presents a permanent fault and can only be repaired by replacing the faulty component. Most SEUs are soft-errors[39], thus correcting by rewriting respectively recalculating of the erroneous value should be preferred to exchanging the component, as this can be automated and reduce the maintenance costs greatly.

The ASTEROID OS architecture[13] provides a Reliable Computing Base (RCB) on which processes can run soft-error resilient on COTS hardware. This is achieved via the replication framework ROMAIN[12], which runs multiple instances of the task,

thereby taking care of copying the input into each instance and comparing their the output, checking for differences. If the outputs differ between replicas, ROMAIN will detect this, and if a majority voting is possible, correct the faulty instance.

ROMAIN can replicate nearly all processes, but the special requirements of communication with devices are not yet considered in ROMAIN. In this work necessary changes are introduced to successfully replicate I/O device drivers with ROMAIN, thereby removing them from the RCB of the ASTEROID OS architecture. This would greatly reduce the necessary effort to harden the RCB of ASTEROID OS.

Device drivers talk to devices over different hardware access methods, for each of them another approach is required. This work presents solutions for the following four major techniques: Port-mapped I/O (PMIO), Memory-mapped I/O (MMIO), Direct Memory Access (DMA), and Interrupt Request (IRQ). The goal of each approach is to minimize the increase of the RCB, while being as portable as possible regarding the underlying hardware as well as being completely device independent. Therefore the proposed designs are:

- For PMIO accesses to trap and emulate the instructions by ROMAIN similar to the already existing solution for shared memory.
- A MMIO access layer, thus all MMIO accesses have to be transformed into Interprocess Communication (IPC) messages, which get checked by ROMAIN.
- Reusing the shared memory feature of ROMAIN for DMA areas, by introducing a flag marking a memory area as used by DMA.
- For IRQ I will argue that ROMAIN already supports it.

This work is structured in the way that it will first introduce the technical background, related work, ASTEROID, and ROMAIN (2). Then the taken design decisions will be discussed for any of the different presented communication methods(3). Chapter 4 presents the performance as well as the reached fault resilience for the device drivers. At the end, this work is summarized and a prospect how the design can be improved in the future is given(5).

2 Technical Background

This chapter gives an overview of the fundamental problem why hardware suffers from unpredictable faults. Thereafter, it introduces the assumed fault model. Then some related work for fault tolerance is described. At the end, this chapter gives a description of ASTEROID and the replication framework ROMAIN on which this work is based upon.

2.1 Physical Background

As different studies show[39, 11] not all failures in program executions are attributable to software errors. Some of these failures originate in the technology used by integrated circuits of modern hardware, MOSFETs as shown in figure 2.1.

By applying a voltage to the gate, the gate acts with the bulk substrate separated by the isolating oxide layer as capacitor. This creates an electric field inside the MOSFET.

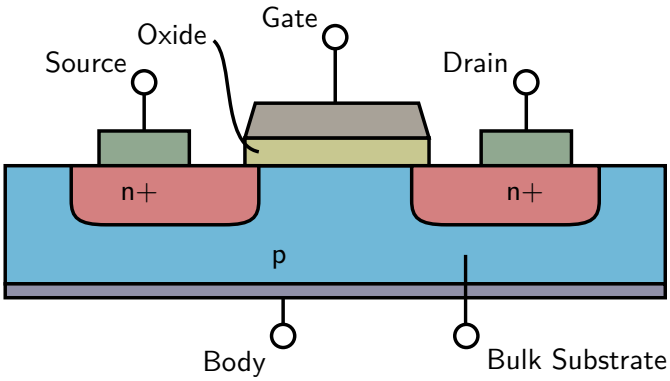

Figure 2.1: n-MOSFET

Based on: `https://upload.wikimedia.org/wikipedia/commons/7/79/Lateral_mosfet.svg?download`

An unwanted introduced charge(may be positive or negative) in the gate or bulk can raise/lower the current charge between gate and bulk. Going over or under the state switching charge, this leads to an unexpected state change of the transistor. For example cosmic rays[44] and radioactive decay[2] can alter the charge. Also aging effects like *hot-carrier injection*, *negative-bias temperature instability*, and *oxide breakdown* can change the properties of the transistor[25]. Furthermore, undesired interaction like electromagnetic coupling or conductive channels between unrelated components can climax in bit-flips as Kim et al. in [26] demonstrated.

The main effects of cosmic rays are *ionization effects*, *escape of long-range alpha-particles*, and *Coulomb-induced recoils*. These effects introduce different particle. At sea level these are largely neutrons, protons, electrons, and muons, and the energy of cosmic rays. Each particle type introduced by cosmic rays interacts with the transistor differently.[44]

The effects of spontaneous radioactive decay inside the substrate are similar to the effects of cosmic rays. But unlike them radioactive decay can be greatly reduced by using depleted material, especially depleted doping material.[2]

Aging effects are another source of hardware failure. Similar to radiation they are a consequence of the physics interacting with the hardware. The three main effects of aging transistors are *hot-carrier injection*, *negative-bias temperature instability*, and *oxide breakdown*.[25]

Hot-carriers are electrons or holes with a high kinetic energy. These carriers can be created by different sources. Through quantum mechanics hot-carriers can introduce current into the gate and the bulk substrate. This changes temporary the electric field. Moreover, hot-carriers change the physical properties of the transistor by altering the chemical structure.[41]

Another aging effect is negative-bias temperature instability, which is not fully understood yet. Like hot-carrier injection the chemical structure is changed. In contrast to hot-carrier injection this can happen at lower energy levels. Also a temporary effect occurs.[38, 21]

The third major reason for aging transistors is oxide breakdown. Here a short circuit through the isolating oxide layer between gate- and bulk-substrate is created.[10]

2.2 Relevance in Future Systems

The in section 2.1 discussed effects induce SEUs into electric logic circuits[39]. Dixit and Wood showed in [11] on which circuit's properties the soft-error rate is dependent. They found two main factors: the charge of the electric field and the area density of integrated circuits. Both relations are pretty straightforward and directly linked to size of the components.

The radiation by cosmic rays is distributed nearly isotropic at sea level[44]. Both radiation effects, cosmic rays as well as radioactive decay, can only induce a finite energy into the circuit at a given time frame. Therefore, the changes to the electric field by radiation is limited, making SEUs more probable for smaller transistors, as they also have a smaller electric field.

Higher circuit density leads to a increased probability of existing coupled circuits, since the increased density means reduced clearance between components. This makes electric field changes by fluctuating voltages more prominent in the transistors. Therefore, even previously benign electric coupling can have harmful impacts.

As these factors are directly dependent on the size of the circuit components, the rate of SEUs is expected to increase as consequence of the ongoing evolution of sophisticated semiconductor circuits[6, 11]. Moreover, efforts to lower the energy consumption of the components lead to a smaller gap between the used electric field changes and the necessary field changes for a state switch of a transistor. This also increases the risk of unwanted side effects having a error producing consequence[11].

2.3 Fault Model

As many of the discussed sources for faults are uniformly distributed over the set of integrated logic circuits, a prediction of the time and locality of a fault is not possible. This requires to secure every circuit. The mean time between failures for a MBit in DRAM is reported to be in the dimension of tens of thousands of hours[39]. This means even on a normal consumer computer these failures are fairly common. Nevertheless, through the relative scarcity of these events, it is very unlikely that more than one bit will be switched, thus a single bit-flip model is a good abstraction of the problem.

A data corruption can lead to different consequences depending on where and when they happen. Best case, the error occurs at a point where it does not change anything because the erroneous bits are not read or overwritten by a later write. This is called a *benign fault* and can be safely ignored. Yet other data corruptions can cause crashes (e.g. corrupted pointers), infinite loops (e.g. corrupted terminating conditions), or simply wrong results (e.g. changed interim result). *Silent Data Corruptions (SDCs)* are wrong results which cannot be detected as such and hence can tamper with a lot of subsequent data, causing cascading failures in the system.

2.4 Related Work

As argued in chapter 1 a software solution is preferable for most systems, since specially hardened hardware is expensive. Therefore, the discussion of related work is concentrated solely on software based solutions.

Shadow Drivers[42] increase the reliability of a system by monitoring device drivers and recovering them from failures by restarting and restoring the driver. This is achieved by recording all relevant information which gets passed between the kernel and the driver in the non-fault operation. If a driver fails the shadow driver switches modes and simulates the driver for the kernel, while restoring the driver and acting as kernel for the device driver. In doing so, shadow drivers can transparently mask detectable failures, but the authors found that they cannot detect 35% of the injected faults in their experiments. Knowledge over the driver class is also required to log all necessary information and recover the driver successfully, while concealing the failure to the kernel. This means that each driver class needs a special implementation of a shadow

driver. Drivers, which do not use the standard driver class interface, for example by using additional feature, can neither be monitored nor recovered, since the respectively shadow driver does not have any knowledge about it. Similarly shadow drivers are not able to track communication through an ad hoc-interface like shared memory, because they observe the communication between kernel and driver by replicating procedure calls. Due to the same reason they also cannot restore drivers with internal states which are not expressed by the kernel interface. Moreover, shadow drivers cannot guarantee, that after a crash, a request was handled. Duplication of request is also possible and has to be managed by either the device or "higher levels of software"[42, p.344]. In contrast to shadow drivers the solution presented in this work does not suffer of any of theses problems.

Herder et al. presented the idea of a *reincarnation server*[20]. Similar to shadow drivers the reincarnation server monitors the device drivers and restarts them in a failure case. But in contrast to shadow drivers, the reincarnation server does not mask the crash of the driver to the other processes. It is also not limited to device drivers but rather can be used for all sorts of services. The device driver as well as the processes depending on the device driver have to be aware of the reincarnation server, to be able to load and store state relevant information and get an update that a component has been restarted. Because the reincarnation server has to restart the driver, this approach cannot guarantee that no data gets lost, since it cannot determine when the driver misbehaved. This solution is also not suitable to recognize SDCs. Additionally the authors had some problems to save the state of the device driver in the reincarnation server, thus their implementation currently only works for stateless driver.

Another approach is to use compilers to ensure fault tolerance. One example for such a solution is *SWIFT*[36], which is not limited to drivers. SWIFT replicates each instruction, except stores, while each replica uses different registers. The process, compiled with SWIFT, compares before each store instruction the value of the replicas to be stored as well as the taken control flow up to this store. Because load operations have to be replicated to assure a error free computation and compiler-based reliability systems have no control over other processes, SWIFT cannot secure situations in which the value to be read could change between the replicated loads. Common cases are shared memory, exception, and interrupt handling. This is also a problem for multithreaded tasks, since scheduling and thread progress is non-deterministic. Therefore, SWIFT only supports single-threaded execution. As with every solution using compilers, the source code has to be available and every program has to be custom-compiled for a resilient execution. Because many drivers are dependent on interrupts and utilize shared memory for higher throughput, SWIFT is not suitable to secure most drivers against soft-errors. Also not all device manufactures release the source code of their drivers, making compiler-based solutions unsuitable.

A different way for a software based solution is *Process-Level Redundancy(PLR)*[40]. The idea of PLR is not to secure single hardware components against soft-errors, but to assure that the output of a process is correct. PLR replicates a process, guaranteeing

that all replicas get the same input and comparing the output of the copies. The authors divide theses redundant processes into a master process, which handles input and output events, and slave processes, which get copies of the input data and provide output data to be compared against. The comparison and copying takes place in a memory region shared between these. One input and output source are system calls. System calls, which do not change any external state, get executed by each replica. Other system calls have to be emulated for slave processes. This is done by the system call emulation layer, which is set in between the replicas and the OS. Soft-errors can also change the timing of an execution of replicas. To detect this a watchdog timer is used. Due to the replication of processes, PLR cannot run tasks which use sources of non-determinism, like shared memory or multithreaded executions.

2.5 ASTEROID OS Architecture

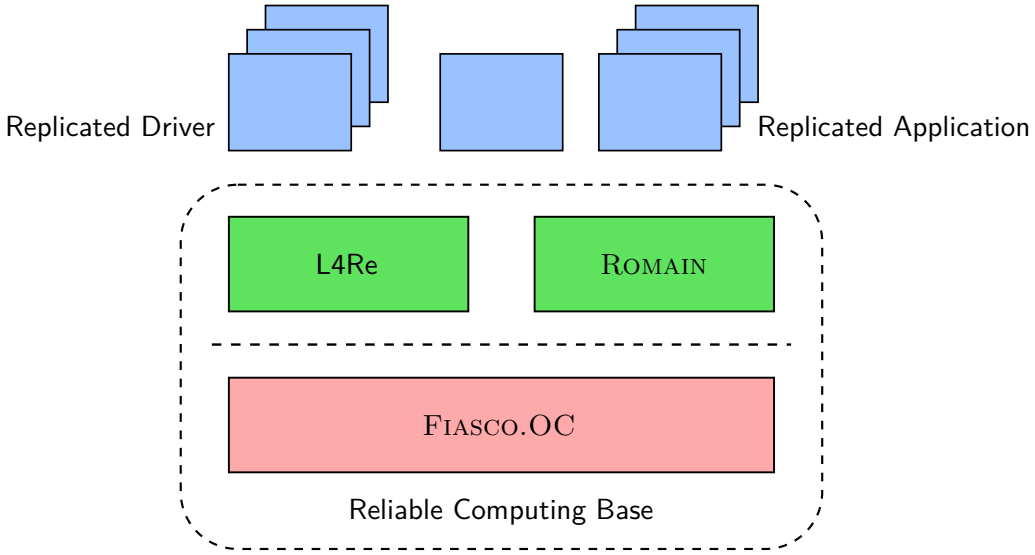

Figure 2.2: ASTEROID OS Architecture
Based on: [12, Figure 3.1]

The design goal of ASTEROID Operating System Architecture[13] is to have an OS with a defined RCB. The *Reliable Computing Base (RCB)* was introduced by Engel et al. in [15] and is described as the set of components, software and hardware components, which are needed and have to be trusted in order for processes to be resilient against faults. The RCB of ASTEROID allows to run fault unaware processes in a replicated fashion, therefore fault resilient, on COTS hardware. As figure 2.2 depicts, the ASTEROID's RCB is composed of the Fiasco.OC[17] microkernel, some fundamental system services provided by L4Re, and Romain[12], a replication framework. Section 2.6 will describe Romain further. All executions outside of the RCB of ASTEROID is protected due to replication, consequently the RCB has to be protected with other means.

2.6 Romain

For the replicated execution of unmodified binaries Döbel[12] has proposed *Robust Multithreaded Application Infrastructure*(Romain). Romain is a framework, which allows to run soft-error unconscious tasks transparently in an n-replicated way. By comparing the output of each replica, it can detect and correct hardware induced errors. It is part of the L4 Runtime Environment(L4Re)[18] and the ASTEROID OS architecture (Section 2.5). Romain can replicate most processes running on the Fiasco.OC microkernel.

Fiasco.OC is a third-generation microkernel, which, like every microkernel, seeks to minimize the size of the kernel. This means that Fiasco.OC does not provide any complex services. These have to be implemented in usermode. However, it offers all of its services in form of kernel objects, which can be accessed by task through *capabilities*.

A task consists of an address space, an object space, which holds the capabilities of the task, and an IO-port address space. It can have one or more threads which execute the code. Over IPC-Gates tasks are able to create a synchronized communication channel to other tasks.

Every task can only access capabilities which itself owns over a system call. This call is translated into an *Inter-process Communication (IPC)*. Each IPC consists of a memory region which is specific to a thread and is shared between task and kernel, the *User-level Control Block (UTCB)*. Fiasco.OC can then copy the necessary information from the UTCB of one thread to the UTCB of another thread. This not only allows to pass messages between tasks, but can also be used to give other tasks additional capabilities, for example memory pages.

This feature is used by Romain to guarantee that the replica only owns capabilities, over which Romain has full control. Derived from this, Romain has to act as program loader for the replicas to set the initial capabilities.

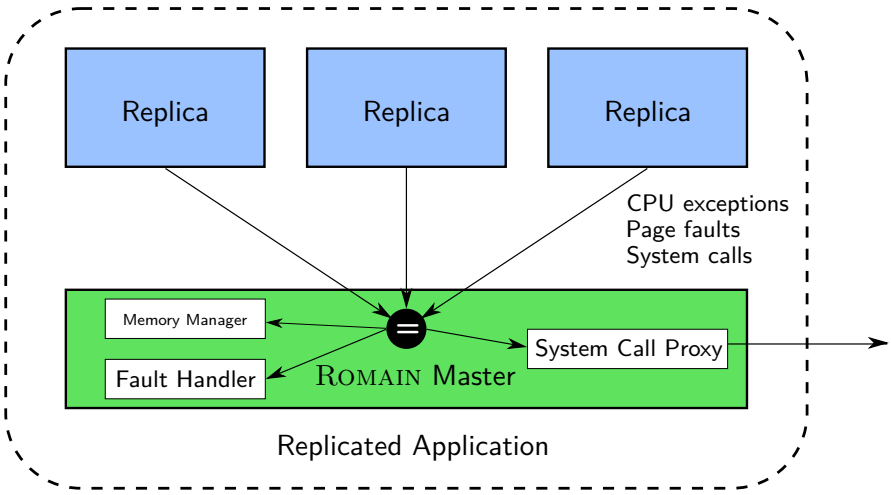

Figure 2.3: Romain Architecture
Based on: [12, Figure 3.2]

In figure 2.3 the general approach of ROMAIN is presented. A replicated application is composed of a ROMAIN master process and n-replicas running in their own address spaces. The master process compares each output of all replicas and therefore can detect mismatches between them. Because ROMAIN acts as process loader and resource manager, it can set up all replicas identical, this means that each replica should compute the same. Hence, if the output mismatches a soft-error has occurred. Similar to PLR, this only holds true if all inputs are the same, so ROMAIN has to replicate all inputs, too.

To be able to compare all outputs, ROMAIN uses a feature from FIASCO.OC called *virtual CPU (vCPU)*[27]. vCPUs are a thread substitute introduced for OS rehosting by Lackorzynski et al. vCPUs run user-level code like threads in an address space, but instead of the kernel handling all CPU traps, the vCPU gets migrated to a guest kernel, which then can act upon this event.

Through the use of vCPUs for replicas, ROMAIN can handle all externalization events that are visible for the kernel, for example system calls and page faults of the replicas. Because ROMAIN, from the viewpoint of the vCPUs, acts as guest kernel, all vCPUs of the replicas get migrated to it. ROMAIN can then inspect the vCPUs and status of the replicas. Each externalization event is also a synchronization point for the replicas, as depicted in figure 2.4. When a replica enters an externalization event, the replica will be halted by ROMAIN until all replicas have triggered the event. This enables ROMAIN to compare the output and the register state at each CPU trap. Then it acts accordingly as:

System call proxy
: doing the system call only once for all replicas and coping the result back into the replicas

Memory manager
: taking care that all replicas have the same memory layout

Fault-handler
: reacting to faults identical for all replicas

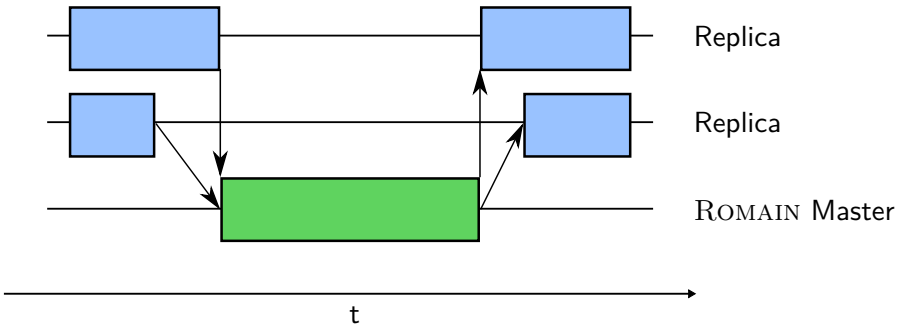

Figure 2.4: ROMAIN Synchronization
Based on: [12, Figure 3.3]

Thus ROMAIN cannot only compare the output, but also guarantee that all replicas get the same input exactly at the same point in their execution.

ROMAIN replicates the task in n-replicas. This allows ROMAIN to provide soft-error resilience on COTS hardware, because ROMAIN can compare, thus find differences, in the output. Since each replica runs in its own distinct address space, the resilience against soft-errors in System Memory (MEM) is not limited to ECC-RAM. As ROMAIN runs the n-replica it cannot only detect SEUs, but also correct soft-errors by majority voting. This requires to execute at least three replicas, because it is not possible to determine which output is correct if each replica has a distinct output.

3 Design and Implementation

Equally to other processes, device drivers have to be protected against soft-errors, especially since they are an important component of each OS. As device driver constitute a big part of the OS[30], it is preferable to exclude them from the RCB. In a microkernel system like Fiasco.OC drivers run as a service in usermode. This should permit Romain to replicate drivers and remove them from the RCB of the ASTEROID OS architecture.

Device drivers share a lot of properties with other applications. However, they have some traits which make them unique, because they have to communicate with hardware. This is done with four major mechanisms:

- *Port-mapped I/O (PMIO)*
- *Memory-mapped I/O (MMIO)*
- *Direct Memory Access (DMA)*
- *Interrupt Request (IRQ)*

This chapter explains the necessary modifications of Romain and ASTEROID to remove I/O device drivers from the RCB of ASTEROID. Also it describes the design and implementation of non-replicated access to the different I/O mechanisms. At the end an overview of the composed design is given.

Since the ideas were tested on an x86-microprocessor and x86 supports all these features, the discussion of the design and implementation decisions will be based on x86.

3.1 Port-mapped I/O

Port-mapped I/O (also called isolated I/O) is a way to access device registers by using a specific address space. This address space is separate and distinct from normal memory address space, thus special instructions are needed for accessing the device. On Intel's x86 processor these instructions are IN and OUT for 1,2, or 4 Bytes at a time.

If a process tries to access a port, the CPU will check whether the process is allowed to access the port. This is done by checking the IOPL flag in the EFLAGS register, which grants access to all ports if set, alternatively, if not set, by checking the *I/O permission bit map*. This bit map is located in the *Task State Segment (TSS)*[22] and contains one bit for each addressable port. Therefore, the I/O permission bit map is a port-by-port accessing-right handling mechanism for each task. If a task tries to access a port and has neither the IOPL flag set nor the permission for the port in the TSS

set, the x86-microprocessor will signal a general-protection exception. Both the I/O permission bitmap as well as the general-protection exception are used for replicating device drivers using PMIO in my design.

In L4Re a special service handles all resources and access rights of the hardware, the *I/O server*[18]. Thus if a task wants to receive access rights to hardware resources, the task has to request the resources from the I/O server. The I/O server then decides if the task is allowed to access the hardware resources. The decision policy is defined in configuration files and bound to IPC Gates. Fiasco.OC provides IPC Gates as means to open secure communications between tasks as described earlier in section 2.6[18]. Because Romain functions as a process loader for the replicas, it has the rights to open communication to the I/O server and request hardware resources. The request of the device drivers to grant access to the hardware resources will be intercepted by Romain and Romain then makes a single request. Romain requests the the hardware resources for the replicas from the I/O server. Therefore, Romain recieves all the access right from the I/O server.

In the case of PMIO the replicas request access rights for ports from the I/O server. This is trapped by Romain which then itself will request the access rights. The I/O server will then update the I/O permission bitmap of Romain.

Since the replicated drivers' I/O permission bitmap never changes, the replicas will trigger a general protection fault on each I/O port access. The fault is handled within Romain because Romain provides the fault-handler. In Romain the fault-handlers are implemented in an observer pattern for the different fault reasons. A fault observer was added to recognize the general-protection fault triggered by port access and handle it correctly.

On x86 microprocessors the set of different port I/O instructions is very limited[23, 24]. Thus the decision was made to implement a disassembler for this instruction set. Now the added fault-handler is able to determine if the trapped general-protection fault was triggered by a replica trying to access a port. The trapped port access is then executed by Romain. Since the accesses go directly to the device at PMIO, this means that the device answers directly and even read instruction can change the state of the device, respectively influence the subsequent reads and reactions to writes. To account for this possible state change the fault handler executes this operation only once and uses Romain's features to copy the result to all replicas.

3.2 Memory-mapped I/O

In contrast to Port-mapped I/O, Memory-mapped I/O uses the same address space and instructions set as memory operations and is protected by the processor through segmentation and paging. As a result the view of tasks of an MMIO area is the same as for normal memory. This is illustrated by figure 3.1

Romain could be made MMIO aware by recognizing the MMIO areas. As explained in section 3.1 the device driver has to request the rights to the MMIO region from the

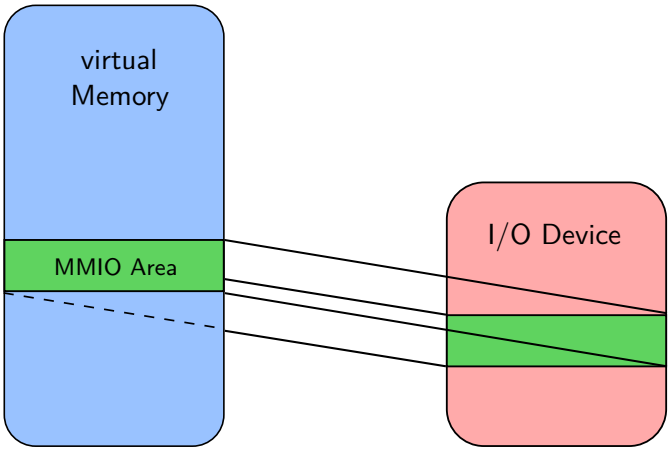

Figure 3.1: Memory-mapped I/O

I/O server. The difference to PMIO is that the I/O server will not change the TSS but will give the driver the pages containing the MMIO area. Since all communication between the driver and other processes is intercepted by Romain, Romain could map the page to its own address space. Because Romain needs full access to the replicas to compare states, this is already done. Also the detection of MMIO areas is possible by comparing the memory layout provided by the BIOS[7, 5] to the physical address of every memory page, the replicas want to map to their address space. Furthermore, the access to the MMIO area by the replicas can be managed by applying the same shared memory between replica idea as introduced later in section 3.3.

This approach is appealing, as this means that every driver running on Fiasco.OC could use MMIO when replicated by Romain. But a working implementation of this proposal would have meant major changes to the page fault-handler, memory manager, and shared memory access emulator of Romain. This was outside of the scope of this work. Moreover, this approach means that Romain needs more knowledge of the underlying hardware and has to have more system depend logic, hence the portability of Romain to other systems would be reduced.

Another design is to make the drivers conscious of Romain. Thereby, the device drivers would signal Romain of the existence of MMIO as well as each access of the device register via MMIO. This could be done either by IPC messages from the replica to Romain or by using interrupts, which signal Romain about MMIO accesses.

In contrast to the first discussed proposal, this is attractive because no considerable changes are necessary. However, it would introduce a completely new layer of direct communication between the replicas and Romain. It would also destroy the transparency of Romain from the viewpoint of the drivers, since the drivers have to directly interact with Romain. This also means that device drivers are dependent on the existence of a Romain instance, either the device driver cannot run outside the replicated environment or the drivers need addition logic to recognize if they run replicated.

The third proposal is to implement a device independent MMIO access layer for the MMIO operations outside of the replicated application. In this solution the device driver hands the MMIO memory page over to the access layer after receiving it form the I/O server. The access layer then maps the page to its own address space. All subsequent MMIO read or write operations are transformed into IPC messages to the access layer. Since every MMIO access is an IPC message, which is a system call, it is also a synchronization point for the replicated application.

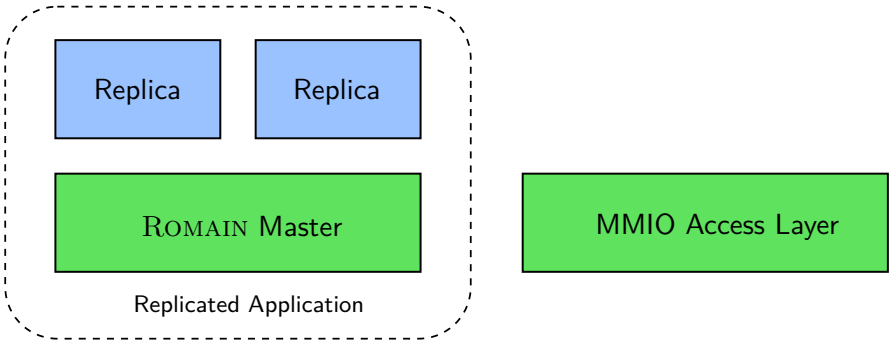

Figure 3.2: MMIO access layer

Like the first approach, this does not have the problem that the device drivers have to be specially prepared to run replicated, because the MMIO access layer is outside of the replicated application. Besides this, ROMAIN does not need additional knowledge about the hardware, which makes this design more portable. This design is similar to microdrivers[16], as the logic of the device driver is separated from a secondary part, which in case of microdrivers are performance critical functions, and in case of the MMIO access layer, the logic how to access the registers of the device.

Since the MMIO accesses of the device drivers have to be altered into IPC messages, an API was created. It takes the MMIO address, and in case of writes the value, and creates the IPC messages for the device driver. Comparably the Linux kernel also has special accessor functions for MMIO[29]. Thus, it is common to use an interface for MMIO accesses, and the necessary source changes for existing drivers should be small. Furthermore, it would be fairly easy to integrate PMIO accesses into this solution and eliminating the fact that the device drivers have to be aware of the different address spaces for PMIO and MMIO.

The MMIO access layer does not need any knowledge about the device except where the MMIO area is mapped into its address space. Therefore it is device independent.

An MMIO read access takes place as seen in Figure 3.3. The device driver calls the MMIO read function which then translates the read into an IPC message to the MMIO access layer and waits for a response. The message is received by the MMIO access layer. On reception it reads at the MMIO address and returns the read value back to the driver per IPC message. Afterwards the driver can resume its execution.

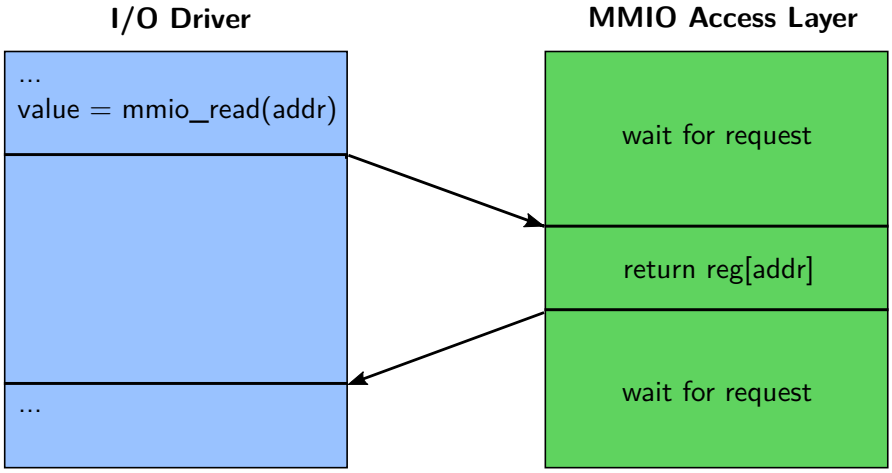

Figure 3.3: Memory-mapped I/O Access

ROMAIN intercepts the IPC message of each replica and compares the content against each other. After comparison ROMAIN sends the IPC message to the MMIO access layer. Since it only sends the message once, the access to the MMIO happens in a non-replicated way.

Because the MMIO access layer runs outside of the replicated application, it has to be inside of RCB. This increases the RCB in code and memory size as well as elevates the non-replicated data accesses and executed instructions. As the access layer is fairly small, less than 300 lines of additional code, does few data accesses during a call, and executes few instruction, it may not be the biggest contributor to the RCB. Still section 5.2 will shortly discuss how to secure the MMIO access layer against soft-errors.

3.3 Direct Memory Access

Direct Memory Access is distinct from Port-mapped I/O and Memory-mapped I/O in that it does not constitute a direct communication between CPU and device, but rather the communication takes place via the MEM of the computer system. Thereby, there is no difference between memory area used for DMA and all other memory. DMA is mainly used for high load I/O operation[19]. This is mainly due to the fact that the memory access from the device is not handled by the CPU, but special DMA controllers. The CPU and DMA controllers are sharing the system memory and I/O bus, this allows the CPU to execute while the DMA controller accesses the MEM. In contrast to the other two mechanisms this does not block the CPU. Through caches, the CPU can even access a subset of the MEM, while the DMA controller blocks the memory bus. Hence, DMA permits for a very fast transfer between I/O device and CPU.

Because the DMA area used by a device exists only once in the RAM, the problems for a replicated execution of device drivers are similar to MMIO(see 3.2). The first

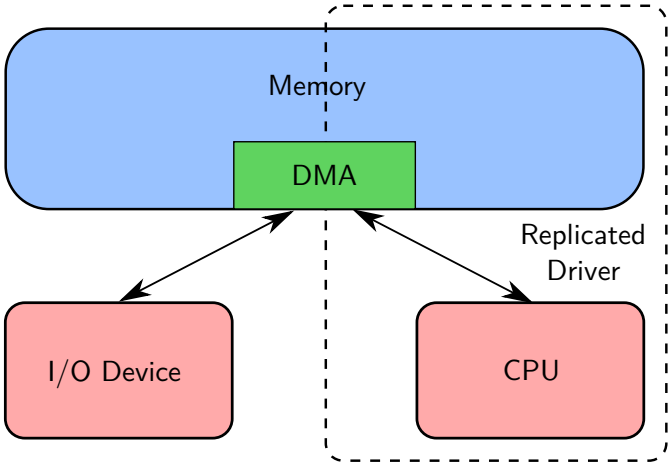

Figure 3.4: DMA area access

suggestion would be to reuse the MMIO access layer and extend it to provide DMA access over IPC messages as well.

As the access layer does not have any knowledge of the device and only processes memory accesses on request, this would be simple to implement. However, as DMA is designed and used for high data throughput, IPC messaging would quickly become a throughput limiting factor, as the UTCB only has a message buffer restricted in size. This would require to send a lot of IPC messages between I/O driver and access layer. Additionally Fiasco.OC has to copy the content of the UTCB of one thread to the UTCB of another thread. This is critical as it means that the data to be transferred have to be copied multiple times for a successful DMA transfer. As IPC messages are synchronous in a Fiasco.OC based system, for every IPC message the system has to perform task switches, decreasing the possible performance of this solution further.

The second solution is to use the shared memory feature of Romain.

To make shared memory accesses replicable, Romain does not attach the shared memory pages into the address spaces of the replicas, but rather provoke page faults when replicas try to access the memory pages. The trapped accesses are executed once by Romain. All replicas get the same input, since the memory is read only once. It also ensures that the replicas do not override anything already changed by another process between the writes of the different replicas. Romain implements two different algorithms to handle these type of page faults:

trap + emulate
: similar to the method of handling PMIO instructions (in Section 3.1), the instruction is disassembled and then emulated by the page fault-handler

copy & execute
: which copies the trapped instruction into a function which is called by Romain.

As a DMA area is a normal memory space, Romain cannot simply detect the pages used for DMA.

The simplest solution would be to share the memory pages used for DMA between the replicas and the MMIO access layer. This requires an IPC message to transmit the page capability. The message can be detected by Romain, consequently the following memory accesses to the DMA area would be trapped.

As straightforward as this would be, it appears odd to give a process resources it never needs and even must not touch at all.

This contradiction could be solved by extending the MMIO access layer with a DMA pool ability. In this case the device driver would not give the capability of shared memory to the MMIO access layer, but rather would request a memory area for DMA from the access layer.

As this would not change anything for Romain, it has some advantages compared to the first solution. The MMIO access layer could manage all DMA area, thereby providing additional features, like limiting the maximal used DMA space in the system or giving a uniform address translation of the bus address used by the devices[9] to the virtual address.

However, this would add additional logic to the access layer, which, since it runs inside of the RCB, has to be protected by mechanisms other than Romain. It also would require to change drivers, as they would have to request the necessary memory from another process. This would make it harder to port the driver to be used in ASTEROID.

Another approach would be to observe all I/O accesses by Romain and to detect when a driver gives a memory address to a device. Romain would then detach the memory pages from the replicated drivers and handle them as shared memory. Similarly the MMIO access layer can do the monitoring and then signal Romain. However, for this to work the access layer have to be extended to handle the PMIO accesses.

No changes to the device drivers would be necessary and it would be possible to prevent the driver giving addresses to the device to which the driver has no rights. But Romain or the access layer would need to have knowledge about most of the DMA-able devices, because of the lack of one central DMA controller. This would add a lot of logic to the RCB. Even worse this logic would have to be extended with every new device.

The in this work proposed solution is to mark the memory region which the replicas want to use as DMA area by the replicas, with the attaching request. This way the device drivers signal Romain that it should handle this region as shared. As flags are already applied to indicate how the regions should be available for the task, for example read only, or at a fixed address, this is simple to implement. The used bitmap for this flags is not fully employed, thus a free flag was available. This flag should not change the behavior of memory manager outside of the replicated application, thus the drivers using this flag are not restricted to running replicated. Romain on the other hand can react accordingly by sharing the page between the replicas.

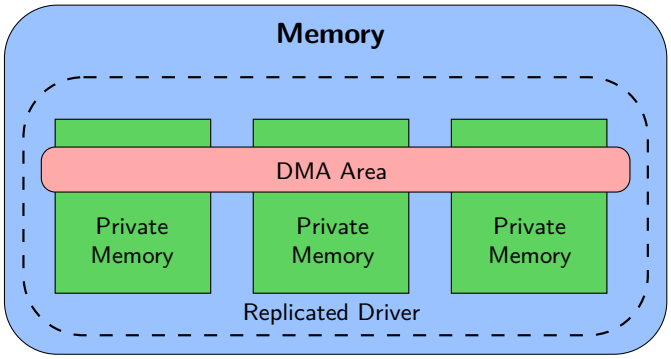

Figure 3.5: Shared memory between replicas

Adding very little to the RCB, while only tiny changes to the device drivers are necessary, this approach is preferable to the others discussed here. Furthermore, as a consequence of employing an unused attach-flag, the ability to run the drivers unreplicated is not altered. At the same time Romain can detect DMA memory pages, thus trapping all accesses. In addition no knowledge of the devices has to be implement into Romain or the MMIO access layer making this solution device independent.

3.4 Interrupts

The last way of communication between device and driver are Interrupt Requests. They are unlike the mechanisms discussed in the previous sections, as they do not represent a bidirectional communication, but rather the device signals the driver that something happened (e.g. received network packet). The driver has then to check the source and decided how to react.

Since IRQs are represented by kernel objects in Fiasco.OC, Romain already has the capabilities for the IRQs. Moreover, owing to the fact that software IRQs are employed for signaling between tasks, and no difference between software and hardware IRQs exists for user level processes[18]. Romain proxies IRQs already.

3.5 Overview

In figure 3.6 my whole design is presented for accessing I/O from a replicated driver.

The driver is replicated by Romain, which will trap and emulate all PMIO accesses through an added fault-handler, therefore normal operation is possible on PMIO.

MMIO is executed by an MMIO access layer, which lies outside of the replicated application. The drivers have to transform their MMIO accesses into IPC messages. These will then get compared by Romain when leaving the replicated application. The answer following afterwards from the MMIO access layer is copied to the replicated drivers by Romain.

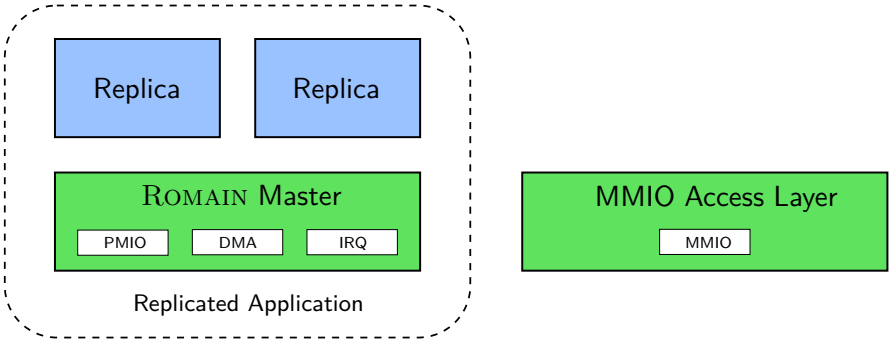

Figure 3.6: Replicated application with I/O extension

To use DMA, the drivers have to mark the memory pages, signaling ROMAIN that it should trap all accesses to this memory area, and execute the memory instructions by ROMAIN.

IRQs are already handled by ROMAIN, through the system call proxy feature. ROMAIN will get the interrupt and relay it to the replicas.

4 Evaluation

Evaluation of the design changes for replicated device drivers was done in two steps: performance and fault resilience.

This chapter will first present the reasoning for choosing the test set of drivers. This is followed by a presentation of the results of the measured performance. At the end of this chapter the reached failure resilience is discussed.

4.1 Prerequisite

The lack of available I/O device drivers for L4Re meant that some drivers had to be implemented. The decision was made to write a Universal Asynchronous Receiver/Transmitter (UART) driver for UART 16550[31] devices and the Intel e1000[32] 1 GBit/s Ethernet devices, because these are wildly supported, for example by QEMU[3], which was used for debugging, and Bochs[28], which is employed by the fault-injecting Fail*[37] framework. Furthermore, both devices together apply all I/O mechanisms discussed in chapter 3. The UART devices operate with PMIO and IRQs, the e1000 device communicates over MMIO, DMA, and IRQs.

4.2 Performance

The performance of the e1000 driver was evaluated on a system with an Intel Core i7-4770 processor running with 3.4 GHz and equipted with 8 GB of System Memory with a transmission rate of 1333 MT/s. As network device a 82540EM (rev2) PCI card was used and lwIP[14] as the network stack.

In the following diagrams and tables of the results the subsequent terms will mean the following:

native:
: The device driver and lwIP run without a Romain instance.

Romain:
: One instance of the device driver and lwIP run with Romain.

4.2.1 Round-trip Times

The first test was to compare round trip times of the driver running replicated against an unreplicated execution. The Linux's e1000 driver and Linux's network stack were used as baselines. Internet Control Message Protocol (ICMP)[34] echo request packets were send from a Linux machine to the test machine to measure the round trip

	Native	ROMAIN			Linux
		1 Replica	2 Replicas	3 Replicas	
Mean / ms	0.152	0.206	0.254	0.299	0.123
Min / ms	0.046	0.080	0.141	0.194	0.051
Max / ms	0.266	0.450	0.694	2.110	0.545

Table 4.1: Round-trip times

times. The time between echo request and ICMP echo reply was recorded. The commandline tool ping was used for this. For every tested setup 10 000 echo request packets were send and the same amount of replies was received, thus no request got lost.

In mean the Linux's driver and network stack are faster than the e1000 driver and lwIP (Table: 4.1). This was to be expected, as Linux is a well established kernel in a wide variaty of devices, therefore the network stack is more suitable for the test machine than the lwIP stack, which is developed for embedded systems. Thus lwIP cannot use the same optimizations as Linux. Moreover, the e1000 driver of Linux does not need to switch tasks to be able to access the MMIO, which does slow down the e1000 driver. Additionally years of optimization for the e1000 Linux driver give it a performance advantage over the for L4Re's e1000 driver.

Because ROMAIN has to get all externalization events and compare the state of the replicas, an increased round-trip time was presumed. Similarly, as every replica has to compute the ICMP request, the overall computation is multiplied by the amount of replicas. This should increase the answer times with a growing number of replicas of the driver. Both effects were observed. The maximal time of ROMAIN running 3 replicas stands out, as it is substantial higher than would be assumed by the other results. A closer look shows that this value was only measured once and the second slowest round-trip had a time of 0.914 ms. The second worst time is inside the expected range, thus the maximum was probably influenced by multiple unlucky circumstances, for example scheduling decisions and cache misses. However, as this value only appeared once, it can be assumed that it is very unlikely to strongly influence the overall result.

The distribution of the round-trip times was compared (Figure: 4.1,4.2). Linux's round-trip times have a smaller distribution than any of the other tested variants. This is because of the task switches in the L4Re driver, as scheduler decisions have to be made. Interestingly, the L4Re driver version running without ROMAIN had 695 round-trip times which were faster than any of the Linux's round-trip times. Two different optimizations in the driver may be responsible for this. The first optimization is the driver assuming if it received one packet, it will probably receive another packet shortly thereafter. In this case it does not wait for the next IRQ, but rather polls the device for a short time. This is a lot faster than to wait to get woken up again by the next IRQ. The second optimization effects the way the driver signals the device that the data is already read. As signaling this is slow, the driver informs the device only when the receive buffer of the device is nearly full. This could also explain the wider distribution of round-trip

times, as the driver behavior is depended on gap between and amount of previously received packets.

The expected distribution of round-trip times in ROMAIN should be similar to running native, except a small offset, which should depend on the same factors as discussed before. This expectation was confirmed by obseravtion. However, the set of very fast answer times, seen in the native execution, does not show running on ROMAIN. Since ROMAIN traps every system call, it also traps the polling requests of the driver optimization.

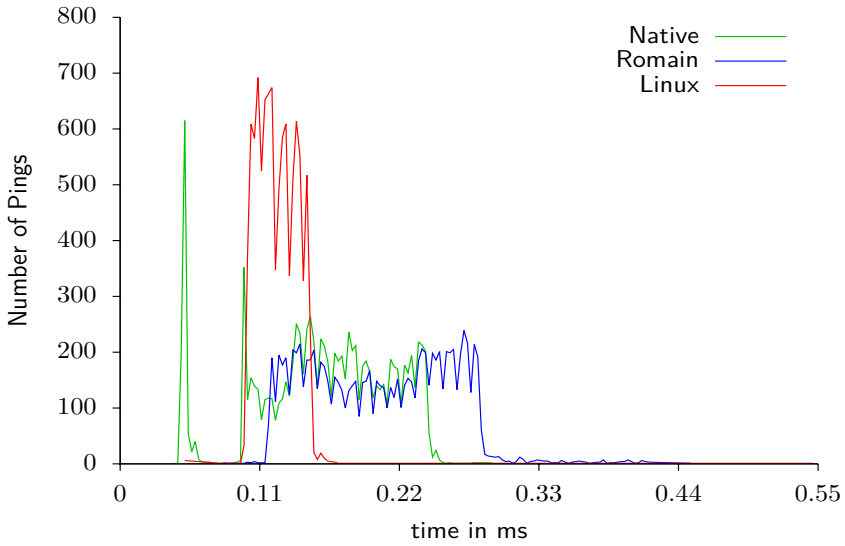

Figure 4.1: Time Distribution: Linux, ROMAIN, Native

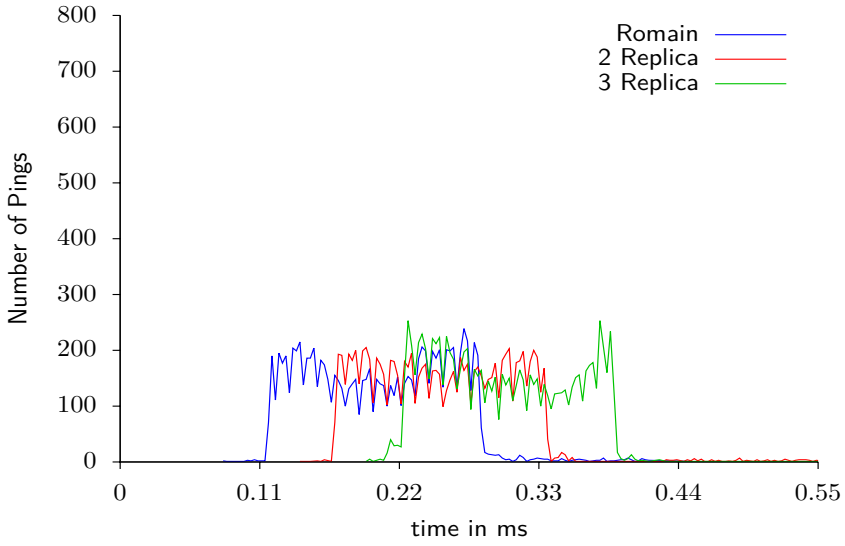

Figure 4.2: Time Distribution: ROMAIN

This makes this optimization less efficient than a native execution, and reduces the advantage over waiting for an IRQ. Since no unexpected increased execution times were noted, optimizations done for unreplicated device drivers do not seem to hurt execution times in a replicated environment.

4.2.2 Throughput

The throughput was determined by sending a file with the size of 21 MB over TCP/IP[8, 35] from a Linux computer to the test computer and measuring the time until the file was completely transmitted. For comparison a 1 GB file of zeros was also send. No difference regarding the throughput was observed between these files.

As Linux is able to fully utilize the 1 GBit/s connection, Linux is not used as baseline in this test, because none of L4Re variants achieves this.

The difference in throughput between the variants is clearly visible in figure: 4.3. As throughput in all cases is limited by the CPU, a reduction by a factor equal to the number of replicas is expected. Comparing Romain with two, respectively three, replicas to Romain only running a singular instance of the driver, the reduction ratio is exactly as assumed. A better result could be attained by running each replica on its own CPU. Since the current system call proxy for Romain does not support hardware IRQ relay to different CPUs, this was not tested.

While the limitation by the CPU explains the throughput differences between the amount of replicas, it does not describe the gap between the native run and the execution of one instance run by Romain. As described in section 3.3, DMA is used for high throughput environments, thus high speed for DMA areas is an important factor. In

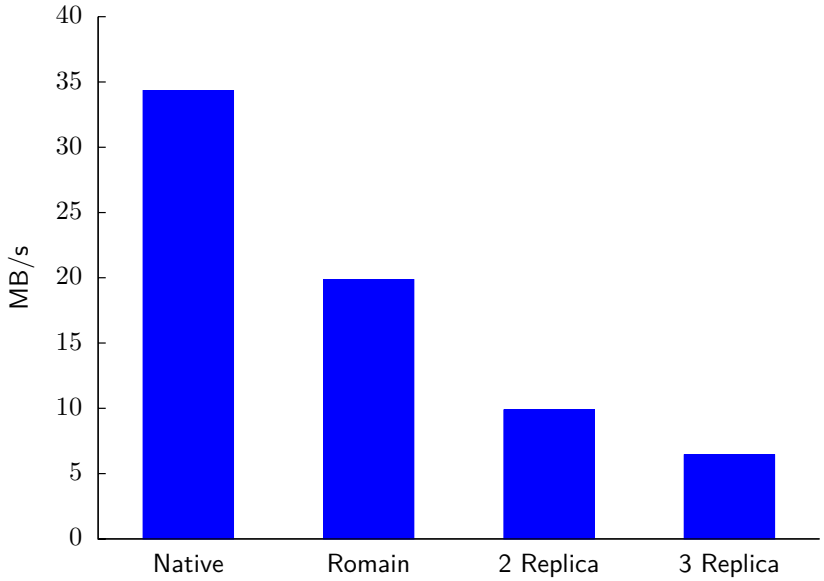

Figure 4.3: Throughput

	Memcopy in MB/s
Native	4 247
ROMAIN	88.78

Table 4.2: Memcopy on DMA area

table 4.2 the degeneration of accessing speed on DMA through the use of ROMAIN is shown. This data correlates with the throughput found by Döbel for shared memory. The difference in accessing speed accounts for the reduced throughput of the device driver running on ROMAIN.

4.3 Fault Resilience

One criteria for choosing these devices was them being supported by Bochs[28] as mentioned in section 4.1. This is important as it should allow to make automated fault injections tests with the Fail* framework, which uses Bochs as an x86 simulator.

The Fail* fault injection framework is designed for "lage-scale dependability evaluation and system analysis campaigns on various target-platform backends"[37, p.4]. This is by an abstraction of the fault injection infrastructure, thus allowing "portable experiment implementation"[37, p.4], while enabling fault injection experiments to change states in the target.[37]

However, the Bochs version 2.4.6 currently employed by Fail* does not have support for e1000 devices, this was only added later to Bochs(version 2.6)[4]. The attempt to port the e1000 simulation back to the used Bochs version failed, since the new network simulation design differs considerably to the one used in earlier versions.

Furthermore, a bug in Fail* prevented a golden run. A golden run is a run of the process, in which all necessary information is recorded to be able to compare a failure free run to all subsequent fail injected runs. This bug appeared while accessing a register over PMIO.

Because the automatic failure injection through Fail* does not work for the tested drivers, some failures were injected manually into the device driver. This was done by using GDB[1] in connection with QEMU. Only a small set of all possible failures could be tested this way, due to the immense effort the manual injection takes.

The injected failures show, that three separate regions in memory have to be differentiated. These memory regions are illustrated in figure 4.4.

Private Memory is the memory only accessible by one replica and ROMAIN. All injected failures into these memory areas were detected and corrected by ROMAIN, while they had catastrophic consequences running native. Private memory was found by Döbel to be fully protected by ROMAIN against SEUs by detection, and 99.6% of injected failures could be recovered.

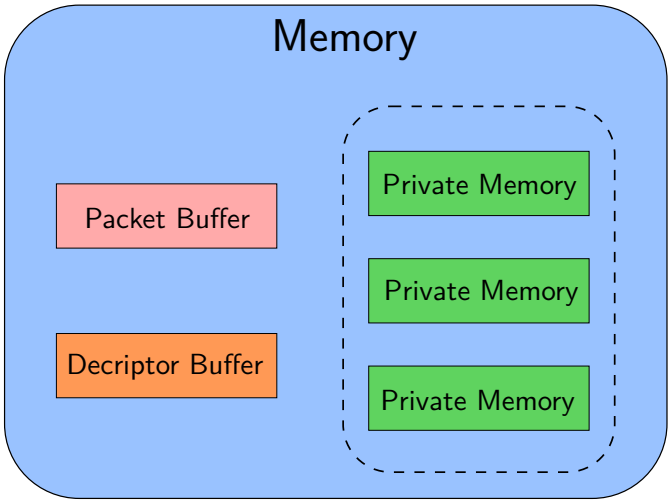

Figure 4.4: Different Memory Regions

The other two distinct regions are located in DMA space. Because these regions are not replicated by ROMAIN, a failure injected into the DMA area cannot be detected or corrected by ROMAIN. Furthermore, the failure is present in all replicas, thus the replicas will react identical to the failure. This makes it impossible for ROMAIN to identify the failure in the outputs following this.

However, the packets being transmitted over the network are secured by checksums. Consequently, the data in the *Packet Buffer* is protected via the checksums against failures. Failures introduced into this memory area, therefore are detected by the network stack. The network stack reacts to these failures as if a transmission error has occurred. This happens regardless of the driver being replicated or not.

In contrast to the packet buffer, the *Descriptor Buffer* does not have any failure detection features. This is especially problematic as the descriptors contain metadata, for example the size of the packet and if the packet is complete. Hence, induced failures in this area can not be detected by neither the replicated driver nor the native version. The failures injected into the descriptor buffer have very different effects on the device driver, for example sending a to large network packet.

However, the risk of getting an SEU into a DMA area can be greatly reduced with the tested device, since the size of the packet buffer and descriptor buffer is not fixed, but defined by the driver. A smaller memory area decreases the probability, owing to the fact that it means less transistors are responsible for the data. This shrinks the used physical size, too. As discussed in section 2.2 these are the major factors for SEUs. Furthermore, through the use of DMA for high throughput situations, most information stored in this area should only be valid for a short period of time.

5 Conclusion and Future Work

This work showed necessary changes to Romain and the ASTEROID OS's RCB for replicating device drivers. Doing this, it presented design approaches, which are reducing the required alterations to the driver, while limiting the increase of the RCB and retaining a maximum of portability to other hardware architectures.

5.1 Conclusion

As PMIO uses a completely different set of instructions and provides a fine grained access control structure, Romain can trap and emulate the PMIO instructions, similar as it traps and emulates shared memory accesses.

The optimum between portability, implementation effort, and increased RCB for MMIO operations was discussed to be an additional layer inside the RCB. For the communication between the driver and this access layer, the Fiasco.OC's IPC feature is used, thus Romain can check the output of the replicas. This also introduces the need to alter the drivers for running replicated. However, these transformations are expected to be reasonably in scope, as argued in section 3.2.

Since a replication of DMA area is not applicable, section 3.3 reasoned that the best way is to mark the DMA memory pages by the driver. This is done by extending the existing attach flags. Thereafter, Romain can trap and emulate all memory accesses to the marked memory pages by reusing the shared memory mechanisms.

Romain already implements a system call proxy, thus the IRQ functionality was already supported completely.

Replication adds a significant runtime overhead for device drivers. This was observed via the round-trip times, testing mainly the overhead of MMIO and IRQ, as well as via throughput, showing the expected overhead for DMA areas as discussed in 4.2.2, using a e1000 card.

In the failure injection discussion it was found, backed by some manual failure injections tests, that the non-replicated access of PMIO, MMIO, and IRQ does not add any additional undetectable failure potential.

In contrast DMA memory does provide a single point of failure, if it is not protected by other means, for example checksums. However, since data stored in DMA is mostly used for high throughput, it can safely be assumed most of these data is only valid for a very short time, reducing the risk of getting an non-benign fault into the memory area.

Therefore, replication of device drivers can increase device drivers resilience dramatically, while having a feasible slowdown for device drivers, which are not limited by CPU throughput.

5.2 Future Work

Drivers only consisting of private memory can be replicated by Romain. Moreover Romain should find 100% of SEUs inside these drivers. Since neither IRQ, PMIO nor MMIO introduce shared memory, a replication of drivers using these mechanisms results in the wanted increase of reliability. The same does not apply for DMA areas. As these are inside the replicated driver, but are not protected by replication, they do not profit from any protection provided by Romain, respectively the RCB.

In section 3.3 it was proposed to use the MMIO access layer as DMA pool, too. This would allow to restrict the DMA memory to especially hardend physical DRAM, for example ECC-RAM. Therefore, the DMA area would be protected by hardware if available, with neither adding additional logic to every driver nor have to use the hardened DRAM for the whole system, as the remaining MEM is protected by replication through Romain.

The current design of MMIO access allows an execution on different hardware, but does increase the required alteration of device drivers for use with the MMIO access layer. An improvement would be to use the shared memory feature similar to DMA. This would greatly reduce the necessary code changes for the driver, to only setting a specific bit, while making the MMIO access layer obsolete, as Romain would do all accesses. If a reuse of the flag introduced by DMA is possible, the entailed alteration to the driver would be reduced to a minimum, while all benefits of the MMIO access layer would be preserved.

As the MMIO access layer lies inside of the RCB, it is not protected by replication and has to be secured against SEUs by other means. A possible solution would be using compiler based solutions like SWIFT, introduced in section 2.4, as the access layer's source code is available and its only a single program. Furthermore, the MMIO access layer does not have any shared memory or a multithread behavior. However SWIFT does not suit the use-case, as load operations on the MMIO memory area must be executed only once.

In the tested drivers, only the descriptor buffer of the e1000 device is completely unprotected. As this buffer is fairly small, and most of the containing data is fixed, it should be possible to implement a memory scrubber for these memory areas. This would renew the information saved in the descriptor buffer frequently, thus reducing the effects of soft-errors.

Bibliography

[1] Pedro Alves, Joel Brobecker, Doug Evans, Tom Tromey, and Eli Zaretskii. *GDB: The GNU Project Debugger*. July 6, 2015. URL: `http://www.gnu.org/software/gdb/` (visited on Aug. 5, 2015).

[2] Robert Baumann, Tim Hossain, Shinya Murata, and Hideli Kitagawa. "Boron Compounds as a Dominant Source of Alpha Particles in Semiconductor Devices." In: *Reliability Physics Symposium, 1995. 33rd Annual Proceedings., IEEE International*. Apr. 1995, pp. 297–302.

[3] Fabrice Bellard. "QEMU, a fast and portable dynamic translator." In: *Proceedings of the USENIX Annual Technical Conference, FREENIX Track*. 2005, pp. 41–46.

[4] *bochs think inside the bochs*. May 3, 2015. URL: `http://bochs.sourceforge.net/` (visited on Aug. 5, 2015).

[5] Erich Boleyn. *INT 15h, AX=E820h - Query System Address Map*. 2004. URL: `http://www.uruk.org/orig-grub/mem64mb.html` (visited on July 14, 2015).

[6] S. Borkar. "Designing Reliable Systems from Unreliable Components: The Challenges of Transistor Variability and Degradation." In: *Micro, IEEE* 25.6 (2005), pp. 10–16. ISSN: 0272-1732.

[7] Ralf Brown. *15E820*. 2004. URL: `http://www.delorie.com/djgpp/doc/rbinter/id/50/17.html` (visited on July 14, 2015).

[8] V. Cerf, Y. Dalal, and C. Sunshine. *Specification of Internet Transmission Control Program*. RFC 675. Internet Engineering Task Force, Dec. 1974.

[9] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. "Memory Mapping and DMA." In: *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005. ISBN: 0596005903.

[10] Robin Degraeve, Guido Groeseneken, Rudi Bellens, Jean L. Ogier, Michel Depas, Philippe J. Roussel, and Herman E. Maes. "New Insights in the Relation Between Electron Trap Generation and the Statistical Properties of Oxide Breakdown." In: *Electron Devices, IEEE Transactions on* 45.4 (1998), pp. 904–911. ISSN: 0018-9383.

[11] Anand Dixit and Alan Wood. "The Impact of New Technology on Soft Error Rates." In: *Reliability Physics Symposium (IRPS), 2011 IEEE International*. Apr. 2011, 5B.4.1–5B.4.7.

[12] Björn Döbel. "Operating System Support for Redundant Multithreading." PhD thesis. 2014.

[13] Björn Döbel, Hermann Härtig, and Michael Engel. "Operating system support for redundant multithreading." In: *EMSOFT*. Ed. by Ahmed Jerraya, Luca P. Carloni, Florence Maraninchi, and John Regehr. ACM, 2012, pp. 83–92. ISBN: 978-1-4503-1425-1.

[14] Adam Dunkels. "Design and Implementation of the lwIP TCP/IP Stack." In: *Swedish Institute of Computer Science* 2 (2001), p. 77. URL: http://static2.wikia.nocookie.net/__cb20100724070440/mini6/images/0/0e/Lwip.pdf.

[15] Michael Engel and Björn Döbel. "The Reliable Computing Base-A Paradigm for Software-based Reliability." In: *GI-Jahrestagung*. 2012, pp. 480–493.

[16] Vinod Ganapathy, Matthew J. Renzelmann, Arini Balakrishnan, Michael M. Swift, and Somesh Jha. "The Design and Implementation of Microdrivers." In: *ASPLOS*. Ed. by Susan J. Eggers and James R. Larus. ACM, Apr. 8, 2008, pp. 168–178. ISBN: 978-1-59593-958-6.

[17] Dresden Operating System Group. *Fiasco - Overview*. Oct. 2014. URL: https://os.inf.tu-dresden.de/fiasco/ (visited on July 14, 2015).

[18] Dresden Operating System Group. *L4Re - L4 Runtime Environment*. Jan. 2015. URL: http://l4re.org/doc/ (visited on July 14, 2015).

[19] A. F. Harvey and Data Acquisition Divison Staff. *DMA Fundamentals on Various PC Platforms*. URL: http://cires1.colorado.edu/jimenez-group/QAMSResources/Docs/DMAFundamentals.pdf.

[20] Jorrit N. Herder, Herbert Bos, Ben Gras, Philip Homburg, and Andrew S. Tanenbaum. "Failure Resilience for Device Drivers." In: *DSN*. IEEE Computer Society, June 26, 2007, pp. 41–50.

[21] V. Huard, M. Denais, and C. R. Parthasarathy. "NBTI degradation: From physical mechanisms to modelling." In: *Microelectronics Reliability* 46.1 (Aug. 23, 2007), pp. 1–23.

[22] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 1: Basic Architecture*. Intel Corporation. June 2015.

[23] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2A: Instruction Set Reference, A-M*. Intel Corporation. June 2015.

[24] *Intel 64 and IA-32 Architectures Software Developer's Manual - Volume 2B: Instruction Set Reference, N-Z*. Intel Corporation. June 2015.

[25] John Keane and Chris H. Kim. *Transistor Aging*. 2011. URL: http://spectrum.ieee.org/semiconductors/processors/transistor-aging (visited on July 14, 2015).

[26] Yoongu Kim, Ross Daly, Jeremie Kim, Chris Fallin, Ji Hye Lee, Donghyuk Lee, Chris Wilkerson, Konrad Lai, and Onur Mutlu. "Flipping Bits in Memory Without Accessing Them: An Experimental Study of DRAM Disturbance Errors." In: *SIGARCH Comput. Archit. News* 42.3 (June 2014), pp. 361–372. ISSN: 0163-5964.

[27] Adam Lackorzynski, Alexander Warg, and Michael Peter. "Virtual processors as kernel interface." In: *Twelfth Real-Time Linux Workshop*. 2010.

[28] Kevin P. Lawton. “Bochs: A Portable PC Emulator for Unix/X.” In: *Linux J.* 1996.29es (Sept. 1996). ISSN: 1075-3583.

[29] Torvalds Linus. *Being more anal about iospace accesses.*. Sept. 15, 2004. URL: `https://lwn.net/Articles/102240/` (visited on July 14, 2015).

[30] Nicolas Palix, Gaël Thomas, Suman Saha, Christophe Calvès, Julia Lawall, and Gilles Muller. “Faults in Linux: Ten Years Later.” In: *ACM SIGARCH Computer Architecture News*. Vol. 39. 1. ACM. 2011, pp. 305–318.

[31] *PC16550D Universal Asynchronous Reciver/Transmitter With FIFO*. Texas Instruments. May 2015.

[32] *PCI/PCI-X Family of Gigabit Ethernet Controllers Software Developer's Manual*. Intel Corporation. Mar. 2009.

[33] Ilia Polian and John P Hayes. “Selective Hardening: Toward Cost-Effective Error Tolerance.” In: *IEEE Design & Test of Computers* 3 (2010), pp. 54–63.

[34] Jon Postel. *Internet Control Message Protocol*. RFC 777. Obsoleted by RFC 792. Internet Engineering Task Force, Apr. 1981.

[35] Jon Postel, ed. *RFC 791 Internet Protocol - DARPA Inernet Programm, Protocol Specification*. Internet Engineering Task Force. Sept. 1981.

[36] George A. Reis, Jonathan Chang, Neil Vachharajani, Ram Rangan, and David I. August. “SWIFT: Software Implemented Fault Tolerance.” In: *Proceedings of the International Symposium on Code Generation and Optimization*. CGO '05. Washington, DC, USA: IEEE Computer Society, 2005, pp. 243–254. ISBN: 0-7695-2298-X.

[37] Horst Schirmeier, Martin Hoffmann, Rüdiger Kapitza, Daniel Lohmann, and Olaf Spinczyk. “Fail*: Towards a Versatile Fault-Injection Experiment Framework.” In: *ARCS Workshops (ARCS), 2012*. IEEE. 2012, pp. 1–5.

[38] Dieter K. Schroder. “Negative bias temperature instability: What do we understand?” In: *Microelectronics Reliability* 47.6 (2007), pp. 841–852.

[39] Bianca Schroeder, Eduardo Pinheiro, and Wolf-Dietrich Weber. “DRAM Errors in the Wild: A Large-scale Field Study.” In: *SIGMETRICS Perform. Eval. Rev.* 37.1 (June 2009), pp. 193–204. ISSN: 0163-5999.

[40] Alex Shye, Tipp Moseley, Vijay Janapa Reddi, Joseph Blomstedt, Daniel Connors, et al. “Using Process-Level Redundancy to Exploit Multiple Cores for Transient Fault Tolerance.” In: *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*. IEEE. 2007, pp. 297–306.

[41] SiliconFarEast.com. *Hot Carriers*. 2004. URL: `http://archive.is/txuW5` (visited on July 14, 2015).

[42] Michael M. Swift, Muthukaruppan Annamalai, Brian N. Bershad, and Henry M. Levy. “Recovering Device Drivers.” In: *ACM Trans. Comput. Syst.* 24.4 (Mar. 6, 2007), pp. 333–360.

[43] Quming Zhou and Kartik Mohanram. “Gate Sizing to Radiation Harden Combinational Logic.” In: *Trans. Comp.-Aided Des. Integ. Cir. Sys.* 25.1 (Nov. 2006), pp. 155–166. ISSN: 0278-0070.

[44] J. F. Ziegler and W. A. Lanford. “Effect of Cosmic Rays on Computer Memories.” In: *Science* 206.4420 (1979), pp. 776–788. eprint: `http://www.sciencemag.org/content/206/4420/776.full.pdf`.