Diplomarbeit

L4ReAnimator

A Restarting Framework for L4Re

Dirk Vogt

2. Dezember 2009

Technische Universität Dresden
Falkutät Informatik
Institut für Systemarchitektur
Lehrstuhl Betriebssysteme

Betreuender Hochschullehrer:	Prof. Dr. rer. nat. Hermann Härtig
Betreuender Mitarbeiter:	Dipl.-Inf. Björn Döbel

Erklärung

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

Dresden, den 2. Dezember 2009

Dirk Vogt

Technische Universität Dresden
Fakultät Informatik

AUFGABENSTELLUNG FÜR DIE DIPLOMARBEIT

Name des Studenten:	**Dirk Vogt**
Studiengang:	Informatik
Immatrikulationsnummer:	2869070

Thema: *Entwurf eines Frameworks zur Wiederherstellung von Software-Komponenten nach Systemabstürzen*

Bearbeitungszeitraum:	03.06.2009 - 02.12.2009
Institut:	Systemarchitektur, Professur Betriebssysteme
verantwortlicher Hochschullehrer:	Prof. Dr. Hermann Härtig
Betreuer:	Dipl.-Inf. Björn Döbel

Aufgabenstellung:

Da in der Praxis fehlerfreie Software nahezu ausgeschlossen ist, sind Mechanismen notwendig, auf Abstürze zu reagieren und deren Folgen möglichst automatisiert zu beheben. Verschiedene Arbeiten hierzu haben sich mit diesem Thema bereits im Kontext anderer Betriebssysteme (Minix3, Linux, CuriOS) beschäftigt. Ziel der Aufgabe ist es, das L4 Runtime Environment (L4Re) hinsichtlich der Wiederherstellbarkeit nach dem Absturz einer oder mehrerer Komponenten zu untersuchen und ein Framework zu schaffen, welches Wiederherstellung nach dem Auftreten von
Fehlern unterstützt.
Die Softwareumgebung des L4Re besteht aus Komponenten, die

- sowohl Client- als auch Serverrollen einnehmen
- in beiderlei Hinsicht Zustand besitzen und verwalten, sowie
- über Invokation statisch (beim Start der Anwendung) oder dynamisch (per IPC) erlangter Capabilities miteinander kommunizieren.

Das zu entwickelnde Framework soll allen diesen Bedürfnissen Rechnung tragen.
Bei der Entwicklung des Wiederherstellungs-Frameworks sind insbesondere folgende Teilaufgaben zu erfüllen:

- Analyse existierender Architekturen (Minix3, CuriOS, ...) hinsichtlich Wiederherstellbarkeits-Anforderungen
- Analyse des L4Re hinsichtlich der Umsetzbarkeit der Anforderungen
- Entwurf und prototypische Implementierung eines Frameworks zur automatisierten Wiederherstellung von Softwarekomponenten für L4Re
- Evaluation der Arbeit, Vergleich mit den analysierten Architekturen

Aufgrund der prototypischen Natur des L4Re ist zu erwarten, dass bei der Entwicklung des Frameworks Schwächen im Hinblick auf diese Ziele aufgedeckt werden. In Abstimmung mit dem Betreuer und unter Berücksichtigung der zeitlichen Einschränkungen einer Diplomarbeit sollen für solche Probleme Lösungsansätze aufgezeigt werden

[illegible]

Dresden, 02. 06. 2009 (Prof. Dr. H. Härtig)

Abstract

As failures in hard- and software are likely to prevail, there is a need for operating systems that offer failure resilience. In this work, I introduce L4ReAnimator, a framework for restarting failed operating system components.

L4ReAnimator helps to restart failed components and to reintegrate them into the system. For that purpose, I designed and implemented a capability fault mechanism. I demonstrate that recursive resource management is also applicable for capabilities. Furthermore, an application-transparent userlevel checkpointing mechanism is part of L4ReAnimator, helping to restore a component's state after its restart.

Acknowledgments

I would like to thank Prof. Hermann Härtig for the opportunity to write my diploma thesis at the TU Dresden Operating System Group and all members of the group who contributed to this work. Especially, I thank my advisor Björn Döbel, for his professional support, encouragement and all the fruitful discussions during this work. Further, I would like to thank Adam Lackorzyński for his enthusiastic work on L4Re and his inconceivable patience while doing rail travels and correcting other peoples work. I am grateful to Michael Jahn, who helped me to fight bees and witches that were hiding in my thesis and all friends, particulary Konrad Heisig, who accompanied and supported me during the last months.

I really enjoyed the working athmosphere at the student's laboratory and like to thank each of the students in the lab for the great time we had.

Finally, I would like to thank my parents, who where allways there when I needed them and who always believed in me. *Thank you!*

Contents

List of Figures

1 Motivation

Blue-screens of death, sad macs and guru meditations have become a metaphor of unstable software systems and nearly everybody had to fight with lost data due to a system crash or application failure. So »save often, save early« was not only topical for players of old Sierra adventure games, but also for current users of every kind of software, e.g. word processors, drawing programs or database systems, whereas at the last one »save often« stands for »backup often«.

1.1 Why do systems fail?

Although there exist many tools that help software developers to detect bugs during the process of writing software, we are far away from getting failure free software. Indeed studies [25] show that the bug density was decreasing in the last years with the help of these tools. On the other hand, the complexity and the amount of code of software projects is increasing continuously. Hence the amount of failures in software projects seems to keep the same over the time.

But software is not the only reason for unstable systems. Hardware is getting more complex and error prone [29]. As the structures of chips are getting finer, bit flips due to cosmic radiation are getting more likely, which then can lead to software failures.

1.2 What can be done about it?

Verified software could be a way to prevent this situation, but it showed out that this approach is very complex and needs a lot of time. One exemplary project of building a verified system is seL4 [22], which is a verified L4 Microkernel. Verifying this kernel, which only consists of 8000 lines of C-code, took more than five years. So, verification of complex software seems to be impracticable, and further does not help against hardware faults.

Further, seL4 is only the fundament of the software stack. Having a verified kernel does not prevent legacy applications running on top of this kernel from crashing.

It seems likely that one cannot change that software will fail, but we can aim of confining the consequences introduced by this failures. One possible direction to reduce the impact of software failures are software systems that are aware of the possible occurrence of bad things and are somehow able to deal with those failures.

This work aims to lay the foundation for such a robust software system based on TUD:OS. I will introduce L4ReAnimator, a framework for restarting failed system components, that should be versatile and easy to use by system programmers. With this framework it should be possible to restart failed operating system components in a application transparent manner.

1.3 Structure of this Document

This document is structured as follows. In the second chapter I present an overview about failures in today's hard and software systems and further present mechanism that can help to build failure resilient operating systems. Further, I introduce TUD:OS, the operating system, this work relies on.

The third chapter contains three case studies of existing operating system providing failure resilience. Chapter four presents the design of L4ReAnimator and in chapter five I present how this design was realized and describe the problems that occurred during the implementation. In the sixth chapter I analyze L4ReAnimator, regarding aspects of effort and performance. Finally, in Chapter seven I summarize my work and suggest further improvements.

2 Preliminary considerations

In this chapter I first give an overview of failures that can occur in toady's computer systems and further present mechanisms that can be used to recover from these failures. Finally I introduce TUD:OS, a microkernel based operating system developed at the TU Dresden, which forms the foundation of this work.

2.1 What Can Go Wrong?

Where people act, let it be politics, software engineering or hardware design, things can go wrong and believing Murphy's law, they will go wrong. In this section I give a short systematical overview of possible failures in toady's hard- and software systems.

2.1.1 Fault – Error – Failure

Avizienis et al. describe a correct service as follows:

> *Correct service* is delivered when the service implements the system function. [7]

A system means an entity interacting with other entities, its environment, like hardware, software, humans and the physical world. The service in the contrary refers to the behavior expected by users of the system.

They further go on to describe a *service failure* as:

> [...] an event that occurs when the delivered deviates from correct service. [7]

The deviation of the correct service is called an *error*, which is caused by a *fault*.

Faults that deterministically remain in the system after a restart are called *non-transient* faults, also Bohr-bugs [18]. Such faults may be introduced to the system during its development phase, but can also be caused by defective hardware.

Faults that are not permanent, and which are likely to disappear due to an restart of the system are called transient faults, or Heisen-bugs [18].

However, a fault does not necessarily trigger an error. A fault existing in a system not triggering an error is *dormant,* whereas a fault that actually causes an error is *active* [7].

Another helpful classification of faults is according to their activation. If the activation of a fault is reproducible the fault is called *deterministic*, or a *hard fault.* In contrast, if the fault is only activated because of a »complex interaction of internal state and external requests«, or just bad timing of a request, it is called *soft* [7].

2.1.2 Error Propagation

Errors of one component may lead to errors in other components. This behavior is called error propagation. An example for error propagation is a hardware fault causing an error in a device driver. This error propagates as input fault into an application using the device driver and causes this application to crash.

Another example for error propagation is a bit flip in a pointer of a component, which causes memory corruption in other components. However, this is only possible if these components are running in the same address space or share memory and can be avoided by address spaces separation.

2.2 Survive Operating System Errors

In order to analyze the fault resilience of real-world systems in section 2.3 I now discuss three strategies that can be employed to achieve this.

2.2.1 Fault Containment

It is obvious that failures can only be handled, if there is something left to handle it. So, if a software failure leads to a crash of the whole operating system, including the kernel, there is no way to recover. Decomposition of the system into multiple components can therefore help to keep the consequences of a failing component limited.

Nooks [34], which will be presented in detail in the next chapter provides fault containment against driver crashes by lifting drivers into lightweight and separated kernel protection domains.

Microkernels offer this protection by design. In microkernel based operating systems services run as user-land processes, so that a crashed component will not lead to a whole system failure. If the crashed component was not a critical part of the system, it might be started again.

2.2.2 Detecting Faults

If a component crashed, it is obvious that this component contained a fault. But not every fault will cause a component to crash and these faults can be hard to detect. Further, silent faults may lead to error propagation into other components. To prevent that, it is necessary to detect faults as soon as possible.

To achieve this, Gray [18] proposes to construct *fail-fast software*. Fail-fast software is achieved using defensive programming techniques, and checking each input, output or immediate result during computation against soundness rules.

Another way to detect software faults early, i.e. before they can cause an error, is *periodic code checking* [11]. Thereby, the code segment of an application is periodically checked for unwanted modification. This can be done in software and there also exist microprocessor designs supporting a Run Time Integrity Checker[16].

2.2.3 Restarting Components

Indeed fault containment and fault detection are important, but they do not prevent components from crashing. It is still necessary to deal with failed components. In the simplest case the component can just be restarted, whereas decomposition can help, to keep the restart times as small as possible. This mechanism is also known as micro-rebooting [21].

However, simple restart only helps if the component is stateless or if the state can be easily recovered. In case the component is stateful, the state has to be made persistent and needs to be recovered during the restart of a component.

Checkpointing [28, 13] can be a solution to that kind of problem. Curi-OS [12], does that by holding the state in special regions. This mechanism will be described in the next chapter.

2.3 TUD:OS

TUD:OS is a microkernel-based operating system, developed at TU Dresden. It consists of the Fiasco.OC microkernel and the L4Re user-level runtime environment. Before I describe these two components in detail, I give a short overview on microkernel based operating systems.

2.3.1 The Microkernel Approach

Microkernels, in contrast to monolithic kernels like Linux, only offer fundamental primitives such as threads, virtual address spaces and communication mechanisms. In contrast policies, for instance memory management, but also device drivers have to be implemented as user-land applications, commonly called servers running in isolated address spaces, which communicate through *inter-process communication* (IPC) and shared memory and *can easily be replaced at run-time*.

First generation microkernels like MACH [5] had a disappointing performance, compared to monolithic kernels and were still complex. Second generation microkernels [36] offer a better IPC performance and are less complex than the former. Examples for second generation microkernels are the microkernels of the L4 family like OK.L4 [3], Pistachio [1], Fiasco [15]. Minix 3 [19] is another true microkernel system.

During the last years a third generation of microkernels, such as seL4 [22], Fiasco.OC [24] and NOVA, has evolved. These microkernels use new approaches to resource management, such as capabilities and support access to virtualization technologies of modern CPUs.

As only the minimal microkernel runs in privileged CPU mode, the probability of failures in privileged mode is minimized. All other parts of the system run in unprivileged CPU mode. Hence, a crash of a user-land component does not necessarily lead to a whole system crash. In the best case the failed component can be replaced and the system is fully functional again. But, as I show later in my work, simple restart is often not enough to bring back full system functionality.

2.3.2 Fiasco.OC

As mentioned before, Fiasco.OC, from now on called Fiasco, is the microkernel TUD:OS relies on. It implements the L4F specification and implements a set of kernel objects. The main kernel objects Fiasco offers are the following[24]:

Task A task implements a protection domain. This domain includes a virtual address space, and a capability space for holding object references.

Thread A thread implements temporal isolation. Threads belong to one task so that multiple threads belonging to one task share the same address space and capabilities.

IRQ IRQs implement asynchronous notifications, which can be received either in a blocking or non-blocking fashion. They represent either hardware interrupts or user-level notifications.

IPC-Gates An IPC-gate provides a communication channel to a certain thread. As it is transparent to a client, with what kind of kernel object it interacts, IPC-Gates can be used to act as proxy to other kernel objects. Further, user-level objects can be created with help of an IPC-Gate.

Factory The factory object creates new objects of a specific type.

User Level Paging

Like other kernels of the L4 family, Fiasco.OC does not implement memory management policies. Address spaces are constructed outside the kernel in a recursive manner using user level pagers. Upon start-up, all physical memory is given to σ0, the root pager. Other address spaces are created by user level pagers, using the following operations:

Map When a page is mapped from one address space to another address space this page will be accessible in both address spaces after the map operation.

Grant When a page is granted to another address space, this page removed from the granter's address space and mapped to the other address space.

Unmap A page which is unmapped will not be accessible in the unmapper's address space anymore. Further the page will be unmapped from all other address spaces, which obtained mappings from the original page.

For providing the unmap functionality the kernel organizes all mappings in a mapping database. This database is holds information of mappings in a tree, representing the mapping hierarchy.

Capabilities

Capabilities are used to address objects and can be seen as pointers to kernel objects. For transferring capabilities also the grant, map and unmap operations are used. Capabilities mapped to a capability space are accessed by the application through the capability index, which refers one entry in the capability space.

User applications interact with kernel objects by performing the only system call Fiasco.OC offers, the invoke system call, on a mapped capability. The parameters of this system call are stored in the CPU registers and special message registers stored in the *Userlevel Thread Control Block* (UTCB) of a thread.

2.3.3 TUD:OS' Run-time Environment

TUD:OS' user land comprises the *L4 Run-time Environment* (L4Re), which offers a set of servers implementing objects with specific interfaces. The interfaces provided by L4Re are:

Dataspace A dataspace [6] is a container of unstructured memory, let it be physical memory, a file or IO-memory. A dataspace provider implements methods like `map` and `unmap`, mapping and unmapping parts of a dataspace to the receivers address space.

Region Mapper The region mapper manages a task's address space, providing functions to attach and detach dataspaces to/from an address space or reserve areas of an address space.

Memory Allocator The memory allocator is providing memory in form of dataspaces. If supported by the implementer, this return memory can have special attributes, such as continuous and pinned memory

Name space The name space object offers a way to register objects under a human-readable name. As result of a name query the registered object will be returned.

Parent The parent object represents the creator of a task. The parent object is used to deliver signals to the parent, for example to inform the parent of the termination of the child.

Frame buffer The frame buffer object offers a virtual frame buffer device to an application. It offers a call to retrieve the graphics memory in form of a dataspace and a refresh function, used to update parts of the frame buffer.

Event The event object offers a simple way to implement a producer-consumer scenario. It offers a shared memory region, where events of any type (e.g., input events) can be stored in and a IRQ-object used to inform the consumer of new events.

Console The console combines the frame buffer and the event interface and thereby offers a virtual graphical console.

Log A log object offers a way for textual output without using a console.

Service Using the service protocol a client specific communication channel can be created.

Besides the interfaces, L4Re offers server and client libraries implementing these interfaces. Additionally, it offers skeletons helping to implement own servers.

In order to be able to run, a task needs an initial set of capabilities, which form the L4Re environment of a task.

The L4Re environment consists of the following initial capabilities:

1. Its parent,
2. A region mapper,
3. A memory allocator,
4. A name service,
5. A log object,
6. The task's main thread,
7. The task's task objects,
8. A factory, and
9. A scheduler object.

The L4Re Kernel

The L4Re kernel is a binary running inside each L4Re task. It loads the task's binary and starts the task's main thread. The L4Re kernel also implements the task's region mapper and acts as pager for all other threads of the task. Although running in the same address space, the task and the L4Re kernel communicate through IPC.

3 Case Studies

Researchers have already implemented systems focusing on fault resilience and self-healing. In this section I present three noteworthy examples and discuss their strengths and drawbacks.

First, I present Linux shadow drivers, an add-on to the Linux kernel providing protection against defective device drivers and restartability of crashed device drivers. The second system focusing on robustness is Minix 3, a microkernel-based operating system with self-healing–capabilities. CuriOS, also a microkernel-based system, will be the last system presented in this section. At the end of this section I summarize what was learned during the examination of these three systems.

3.1 Linux Shadow Drivers

Linux Shadow Drivers (LinuxSD) [32] are an extension to the Linux kernel providing restartable device drivers. As Linux device drivers run directly in the kernel, the first step was to isolate device drivers from the rest of the kernel. This was done in LinuxSD's predecessor, Nooks.

3.1.1 Isolating Linux Device Drivers — Nooks

In their work on *Nooks* [33], the authors introduce *Lightweight Kernel Protection Domains* (LKPD), in which kernel extensions run , offering virtual memory protection against the rest of Linux kernel. The kernel and the kernel extensions use special calls, *Extension Procedure Calls* (XPC), in order to safely transfer control between the kernel and the LKPD. Figure 3.1 illustrates two LKPDs inside the Linux kernel.

Extension Procedure Calls

The XPCs include `nooks_driver_call` and `nooks_kernel_call`, whereas the first one is called by the Linux kernel to transfer control to an LKPD and the latter called by the extension, to transfer control from an LKPD to the

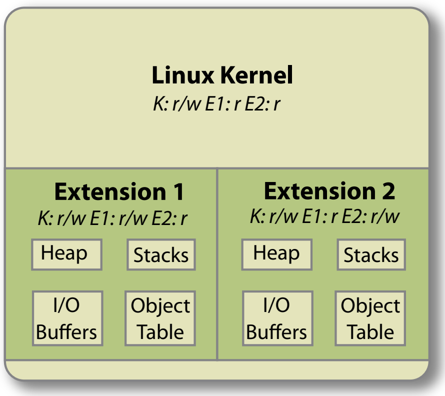


Figure 3.1: Linux kernel with two LKPDs and their memory access rights (K for kernel, E1 for extension 1 and E2 for extension 2).

kernel. These calls take a function pointer to the API function that should be called, a list of arguments and a LKPD identifier as parameters.

An LKPD has to maintain copies of all kernel objects that should by writable in an LKPD. During an XPC Nooks has to check which kernel objects were modified and synchronize their state.

In order to let the kernel and the device driver transparently use the standard kernel/driver interfaces, wrappers are used. The wrappers check the validity of parameters, synchronize kernel and LKPD versions of kernel objects and then perform an XPC.

The wrapper stubs are generated by an external tool, but the wrappers main functionality has to be written by hand and knowledge on how parameters are used is required. However, the authors propose, this process can be automated using meta-compilation [14]. Overall the wrappers include 14K *Source Lines Of Code* (SLOC), which is more than the half of Nook's code-base.

Benchmarks, done by the Nooks' authors show that the relative performance to an unmodified Linux kernel tends to be between nearly 100 % (playback of an MP3) and 44 % (serve a simple web page).

This performance degradation is mainly caused by TLB-flushes that occur on the x86 architecture when the protection domain is switched. However, the authors claim, that on other architectures, supporting tagged TLBs, this performance costs can be mitigated.

Nooks' Reliability

Although this section deals with LinuxSD, it is interesting to look at the reliability enhancement which is gained by the fault containment Nooks provides.

Nooks' authors used synthetic fault injection to test the reliability of their system. The fault injection tool used was the same tool that was used to test the Rio File-cache [10].

The five kernel extension tested were the Sound Blaster 16 sound card driver (sb16), two ethernet drivers (e1000 and pcnet32), a file system driver (vfat) and the in-kernel web-server (kHTTPd). The authors injected 400 faults. On native Linux this fault injections produced 317 system crashes from which 313 could be prevented by Nooks. In the remaining 4 cases the system deadlocked.

Nooks offers a simple recovery mechanism that is unload, reload and restart a crashed kernel extension. It turned out that this simple recovery mechanism is introducing several problems. First, a user application accessing a device will get a failure, because the accessed device will disappear when the kernel extension is unloaded. Second, the recovery of the VFAT kernel extension resulted in on-disk corruption.

3.1.2 Restarting Device Drivers

As Nooks' simple recovery mechanism caused applications to receive erroneous results, Nooks was extended by a new mechanism called shadow drivers.

Shadow Drivers

A shadow driver is a layer interposing the kernel and the driver interface using taps. It can be in active or passive mode and has to be written for each Linux device driver class.

During normal operation the shadow driver is in *passive* mode, monitoring all communication between the Linux kernel and the device driver, as illustrated in Figure 3.2. In this illustration we can see a shadow driver for the Linux sound driver class. Whenever the Linux kernel is calling a function of the sound driver class interface, the tap will first call the original function and then invoke the shadow driver's equivalent. This way, all communication between the kernel and the device driver can be monitored. The sound driver class shadow driver, for example, keeps a log of all `ioctl` calls to devices. In

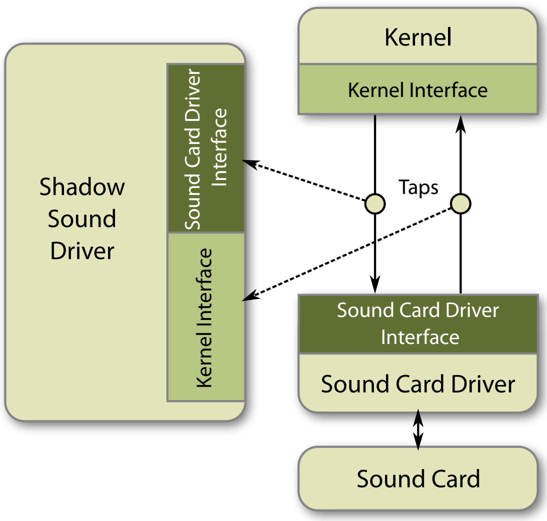


Figure 3.2: LinuxSD sample scenario. Shadow driver is in *passive* mode.

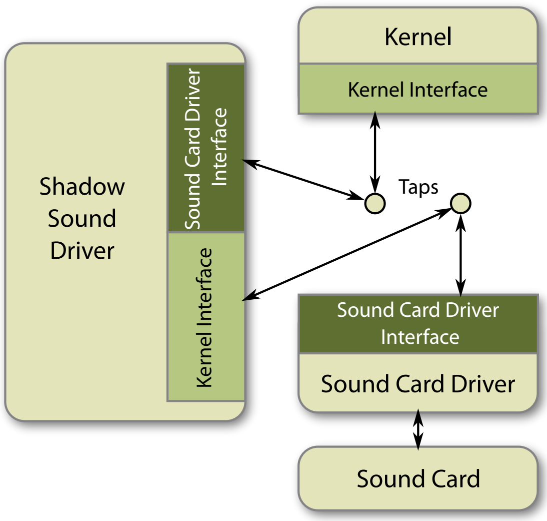


Figure 3.3: Same Scenario as in Figure 3.2, but shadow driver is in *active* mode.

the recovery case this information can be used to reset the sound card to the state it had before the driver crashed.

In the failure case the shadow driver switches into *active* mode, illustrated in Figure 3.3. In this mode the shadow driver prevents all communication between the kernel and the device driver. Further, it takes the device driver's role and responds to all kernel requests during the recovery process. After the recovery of the failed driver has finished, it switches back into passive mode and the system functions as before.

Recovery in Detail

Recovery is managed by a *Shadow Manager,* which is also responsible for loading the shadow drivers. When a failure in a device driver is detected, the shadow manager locates the corresponding shadow driver and instructs it to recover the failed driver. In case the failure is not automatically detected, the recovery mechanism can also be triggered manually from a program.

In the recovery case the shadow driver will:

1. *Stop the failed driver.* This includes disabling the device, and garbage collecting all resources, held by the failed driver. However, the kernel objects representing the driven devices are retained, so that the kernel will not notice the device driver is reloaded.

2. *Reinitialize the device driver.* During reinitialization of the failed driver, the driver's initial data section is restored. After that the shadow driver will replay the kernel's initialization sequence and reattach the driver to the kernel objects that were retained when the driver was stopped.

3. *Transfer old state to the device driver.* During this step the shadow drivers replays the configuration calls recorded in passive mode and for open device file descriptors the device driver's open function is recalled.

During the whole discovery process the shadow driver is handling all the kernel's requests to the driver. Depending on the situation the shadow driver can respond in different ways:

1. Respond with previously recorded information,

2. Drop the request,

3. Queue the request,

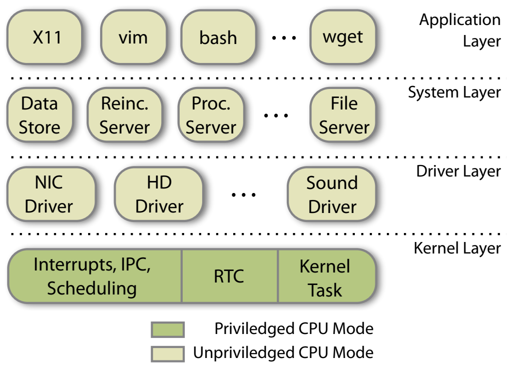


Figure 3.4: Overview of Minix 3's system architecture.

4. Block until recovery has finished, or
5. Answer with `-EBUSY` or `-EAGAIN`.

3.1.3 Evaluation of LinuxSD

As LinuxSD is based on Nooks, which provides the failure isolation, the two systems are behaving comparable in terms of failure recognition and performance.

However, it showed out that LinuxSD was able to recover from most of detected failures in an application transparent manner, whereby most of this failures caused the system to crash on native Linux. In contrary although Nooks was able to detect most of the failures, the system was malfunctioning after recovering the failed driver. So the authors of LinuxSD showed that their kernel extension is useful to improve Linux' fault tolerance.

3.2 Minix 3

Whereas LinuxSD is based on a monolithic system, Minix 3 [19, 35] is a microkernel-based operating system, designed with focus on robustness requirements. During the examination of Minix I concentrated on the self-healing ability of this system. First, I present an overview about the overall system architecture of Minix, after which I explain Minix' self-healing mechanism.

3.2.1 Minix 3 System Architecture

As mentioned before, Minix 3 is microkernel based operating system on top of which run userlevel device drivers and system servers providing a POSIX interface to applications. Figure 3.4 illustrates this architecture. Minix' system architecture can be divided in 4 layers: the (1) kernel layer, (2) the driver layer, (3) the server layer and (4) the application layer, which are described below.

Kernel Layer

In the *kernel layer* the microkernel and two processes are running. The first process is the only device driver running in kernel mode, the clock driver. Because of the tight coupling with scheduling, which is done inside the kernel, the authors decided to keep this driver running in privileged mode for the benefit of higher performance.

The second process running in kernel mode is the system task. This process is implementing the kernel call interface. Kernel calls are system calls to the microkernel and are named this way to ease the distinction from POSIX system calls.

Device Driver Layer

Minix device drivers are user processes, running in separate address spaces. They are communicating with the system task using IPC and the kernel call interface. To support device driver functionality the system task offers special kernel calls for accessing IO-ports, attaching to IRQs, etc.

System Server Layer

In the system layer, the operating system servers reside. These servers provide the actual POSIX functionality to applications. There are mainly two important servers, for offering this functionality, which are the *file server* and the *process server.* Beyond these two servers there exist further servers, for example the *data store* and the *reincarnation server,* which together provide Minix' self-healing capabilities.

File server The file server offers access to the file system and implements POSIX system calls like `open`/`close` and `read`/`write`.

Process Server The process server manages processes. This includes the creation of processes and process IDs, and tracking the system's process hierarchy. Further, the process server is responsible for delivering

POSIX signals to processes and also contains the *memory manager* responsible for allocation and deallocation of memory.

Reincarnation Server The reincarnation server is monitoring all servers of the driver layer and system server layer. If the reincarnation server notices a failure of a server, it will restart the failed server.

Data Store The data store is a small database with publish/subscribe functionality providing two functionalities. First, this server is used as the system's name server, providing a way to find communication partners and second, as persistent storage for the internal state of servers.

3.2.2 Self Healing

As mentioned before, the reincarnation server is the base component required for Minix' ability to recover from a component failure. In this section, I describe the process from the recognition of a failure to restarting the failed component.

Fault Recognition

There are several ways for the reincarnation server to get notice of a failed component. First, as the reincarnation server is the parent of all other servers, it will be informed of the server's termination by the POSIX signal `SIGCHLD`. Second, the reincarnation server requests heartbeat messages of other servers. In case of an absence of a response the reincarnation server will assume a server failure. Further, other processes may inform the reincarnation server of abnormal behavior of servers and a restart of a failed component may also be triggered from user-land applications.

Recovery from Failures

After recognition of a failure the reincarnation server will restart the failed component. After restarting the failed component, the reincarnation server publishes the new address of the restarted server to the data store. The data store will then forward this new address to all servers that have subscribed to the name of the failed component.

As the description of the restarting progress reveals, restarting in Minix is stateless. However, with the file system server the infrastructure for stateful restarts is given.

Limitations

Minix' failure resilience is limited by the dependency on the reincarnation server, the data store and the file system server. The first two servers are involved in the recovery process and the last one is required to reload the binary.

Further, a simple restart of the file system server is not enough, because the server's state will be lost after restart. The problem of restarting the file system server was solved by Veerman [37]. Thereby the state of the file system server is held in a shared memory region, which will survive software failures. A copy of the binary is stored by the reincarnation server, so the binary of the file system server has not to be read again from disk. After restart, the file system attaches the shared memory region again and verifies the state using checksums. However, restart fails if the state is corrupt.

3.2.3 Evaluation of Minix 3

The microkernel approach used by Minix offers effective fault containment. By running device drivers as user processes, a failure of a device driver cannot lead to data corruption of other processes.

Further, the authors of Minix showed that it is possible to recover from server failures by restarting them. As an example, they recovered from a failed network card driver, while a download using `wget` was not interrupted. The possibility of stateful server restart was demonstrated by Veerman, but until now Minix 3 lacks an infrastructure for applications to make use of this stateful restart in a comfortable manner.

It's hard make propositions on Minix' performance because there were no benchmarks done by its authors, comparing Minix with other systems. However, the authors compared Minix 3 with its predecessor Minix 2 and it showed that the microkernel based design of Minix 3 introduced 10 % overhead compared to the monolithic Minix 2.

3.3 CuriOS

CuriOS [12] is an operating system that not just offers fault resilience, but also an improved mechanism for protecting against error propagation, isolating client related state in *Server State Regions* (SSRs). It is based on Choices [8] and can be classified as microkernel-based operating system.

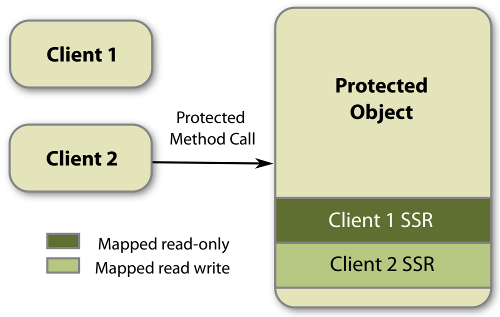


Figure 3.5: Illustration of server state regions in CuriOS. Client 2 is calling a protected method so the SSR belonging to client 2 is mapped writable.

3.3.1 System Architecture

CuriOS runs on top of the CuiK kernel, which offers IRQ-dispatching, context switching and threads. In the same address space, but protected through segmentation and with own stack and heap, run protected objects. These protected objects offer operating system services. Thus they play the same role as operating system servers in traditional microkernel operating systems and will be called server objects here.

Calls to server objects, called protected method calls, are wrapped. The wrappers are similar to the wrappers in Nooks. They take care of switching to the private stack and changing the protection domain.

Server State Regions

A specific feature of CuriOS are Server State Regions (SSRs). These SSRs hold client-specific state and are managed by the SSR manager.

They are created during the first call to a server object and are not accessible to the client.

In order to prevent failure propagation over several client states, the server object only gains write access to the SSRs of the clients currently performing a protected method call to that object. After the protected method returns, write access to the SSR is revoked.

The use of SSRs results in a solution, in which, in case of server failure, only the states related to clients currently interacting with the server may be corrupted. However, it is still possible that an undetected error may propagate into other client-related states, when the server holds a global state that can be corrupted.

3.3.2 Recovery in CuriOS

Failures in server objects are recognized using C++ exceptions. Whenever an exception is raised in a server object, e.g. because of an illegal instruction or illegal memory access, and this exception is not handled by the object itself, the exception is forwarded to the protected method call wrapper. The wrapper will then destroy and recreate the object. After the re-instantiation of the server object, a recovery method will be called, which each protected object has to implement. In this recovery method the server object has to gather all of its SSRs and reconstruct its local state from them.

It is possible that several SSRs, namely all SSRs of clients that were interacting with the server object, were corrupted. In order to detect these inconsistent SSRs, the server object uses magic numbers and checks if the data in the SSRs is in a consistent state.

During recovery the whole system is suspended and when performing recovery a retry mechanism in the protected method wrappers will again send the request to the server object. If the recovery fails several times, an exception is raised at the component using the server object.

3.3.3 Evaluation of CuriOS

CuriOS offers a good way of protecting against failure propagation. As only the SSRs of clients, which are currently calling a protected method are mapped at server objects, only these SSRs can be corrupted. The SSRs are also providing persistent server state.

Fault Recovery

In experiments CuriOS' authors injected two kinds of faults into CuriOS' server objects. First, they injected memory access faults. CuriOS was able to detect and recover from all of these faults. Further, they injected bit flips in register operands of several instructions. The faults were not leading to an error in 100 % of the injections: 5–13 % of the fault injections did not lead to an observable error. In the rest of the cases CuriOS was able to recover.

In an experiment testing the timer manager's reliability, 6 % of the injected bit-flip faults led to a malfunction of the whole system. The authors claim that this happens due to error propagation into the rest of the system and propose to check the arguments and the results of protected method against soundness rules to minimize this problem.

Performance

The interesting point when examining the CuriOS' performance is how much performance overhead is introduced due to the SSR mechanism. The authors of CuriOS measured a performance overhead of about factor three for an protected method call when SSRs are used.

3.4 Conclusion

Table 3.1 shows a comparison of the systems presented in this chapter. Each of them provides three core-features:

Fault Containment The effect of crashed components is reduced in all three introduced systems. In CuriOS and LinuxSD OS services run in a protected environment. Minix implements OS functionally in user-level servers running in dedicated address spaces. This address space separation also offers effective fault containment

Reintegration After restart, failed components are accessible again in all three solutions. In LinuxSD, the kernel objects representing devices are still at the same address. Also in CuriOS the protected objects' addresses are not changed during recovery. Minix in contrary uses a global name service with publish/subscribe functionality to reintegrate restarted OS servers into the system.

Persistence to a certain degree, the three presented systems offer persistence for applications. LinuxSD uses shadow drivers to track and restore the state of device drivers and CuriOS uses server state regions to store the state of protected objects. Minix offers a data store, but this data store has to be used explicitly by the servers.

Only two of the presented works offer true client transparency which are LinuxSD and CuriOS. In Minix 3, queries to server that are currently recovered will fail. Full transparency is achieved at the costs of higher complexity.

3.4.1 Consequences for L4ReAnimator

The three core features, fault-containment, reintegration and persistence should be also present in the framework presented in this work. Also transparency is a desirable goal, but as I will show in later it is necessary to find a acceptable trade-off between complexity and transparency.

	LinuxSD	*Minix 3*	*CuriOS*
Fault Containment	Memory Protection	Processes	Memory Protection and SSRs
Reintegration	Same addresses of data structures	Global names Publish/subscribe	Protected object restored at same address
Persistence	State (re-)stored by shadow driver	(partially) Data store	SSRs
Client Transparency	Yes	No	Yes

Table 3.1: Comparison of the three systems presented in this chapter.

CuriOS offers with SSRs a nice protection against fault propagation through client states. However, for that purpose also other techniques, like guard pages are possible. Although this would not make cross state corruption impossible, it would be more unlikely.

4 Design

In this chapter I present the design of L4ReAnimator, a framework for transparent restart of failed components for L4Re. First, I present the design goals for my work, after which I present how L4ReAnimator restarts failed components. Further, I show how restarted components are reintegrated into the system and present a capability fault mechanism, used for this reintegration. In the last part of this chapter, I describe a checkpoint/restore solution for L4Re.

4.1 Design Goals

Before I discuss the design of the L4Reanimator framework, I present the design goals that have been taken into account during the design of L4ReAnimator.

L4Reanimator strives to be a framework for transparent restarting of failed operating system components.

Although a fault recognition system is desirable, e.g to prevent error propagation by restarting/replacing the faulty component as soon as possible, this will not be part of this work. However, the framework shall be easy to combine with a fault recognition system.

Nooks achieved full transparency with a complex mechanism synchronizing objects in kernel space and inside the LKPDs. Minix in contrary does not offer full transparency which keeps complexity low. In this work I strive for good trade-off between complexity and transparency. Adding complexity to the kernel should be avoided, or at least, the added complexity should be minimal.

During normal system operation the overhead of the framework shall be negligible. However, during the recovering process, I consider performance as a secondary goal, assuming that a user prefers a system that is unresponsive certain amount of time over an unusable system.

Listing 4.1: Simple loader config file starting a hello-world sample.

```
1 [config]
2         dbg = info;
3
4 [namespace:hello]
5         rom=rom;
6
7 rom/ex_hello
8         ns = ns/hello;
9         log = logger->open("hello", red);
```

4.2 Restarting Failed Applications

In this section I point out the necessary steps for restarting failed components. These steps include adaptions to the component that is responsible for loading applications. I first describe the unmodified loading mechanism. Thereafter, I present strategies to determine if a restart of a component is necessary. In the last part of this sub-section I describe, which modifications to the current loading mechanism are necessary.

4.2.1 Starting L4Re Applications

L4Re applications are started by a dedicated component, called loader. The loader is configured using config files that describe which tasks should be started. These config files contain descriptions of name spaces, the command line that should be used to start a certain task, and the local name space of this task. In Listing 4.1 we can see a simple config file. In line two, the debugging level of the loader is defined, lines four and five define a name space called »hello« only containing one entry for the ROM file system. The rom file-system is a read-only file system containing all modules loaded by the boot loader. Line seven contains the command line for the `hello` task. After this, other attributes of the task are defined. In our example the name space of the task is set to hello and a new log object is created, using the the log object of the loader, adding the prefix »hello«.

When starting an application, the loader first creates the name spaces defined in the config file. If the application's name space is incomplete the loader defers the start of this task until the name space is complete. This can be the case if an entry in one name space is linked to a another name, under which another application will later register a service.

The loader creates the application by creating a task, the initial L4Re-kernel thread, including its stack and an initial region map. The command line defined in the config file is pushed on the L4Re-kernel's stack. Then the loader creates the new application's L4Re environment and maps it to the new application's capability space. After this setup procedure, the loader starts the new application. The loader does this by setting the stack and instruction pointer of the L4Re-kernel thread and instructs the scheduler to run the L4Re kernel thread.

4.2.2 When to Restart

It is important to know, in which cases a restart of an application should occur. In this section, I discuss some strategies, used to decide if a application should be restarted.

On Crashes

On termination of a task a signal is send to the loader in the exit routine of the failed task including a return value similar to sending a SIGCHLD signal in POSIX, as it is used in Minix 3. When receiving such a signal, the loader may evaluate the return value and decide if the application should be restarted.

If the application is no more able to send this signal, for example in case of an unresolved page fault, the signal can be sent by the L4Re kernel. This is possible because the L4Re kernel is the application's pager. So a good starting point seems to *restart tasks on abnormal termination.*

Further it is possible to introduce *heartbeat messages* between the loader and its child, similar to Minix 3's mechanism. In case of the the absence of heartbeat messages the loader then has to terminate its child and restart it.

On Malfunction

Not every failure leads to a crash of the affected component. It is possible that an application will accidentally stay in an infinite loop. As the L4Re-kernel and the application (in the common case) run in the same address space, an application may override the L4Re kernels application code. Then it might be possible for an application to crash without the possibility to inform its parent, i.e. the loader.

Here, I shortly discuss two possible solutions to this problem. First, it would be possible to introduce a *blaming mechanism*, that allows applications that depend on a service of a malfunctioning component to inform the system that the process is not acting as expected. If enough trusted applications blame one component the system can react. This mechanism is also used in Minix 3.

Further, *runtime integrity checks* of the child's code can be used like described in section 2.2.2. Using this mechanism, a component, e.g. the loader, periodically checks the applications code for unexpected modifications. This can be done by using checksums. If the check-sum of the applications code does not match the expected value, the application can be restarted using the original binaries. However it has to be assured, that the application is not able to execute code outside its code segment. This can be achieved by using hardware architectures providing a flag to mark memory pages as not-executable.

Of course it should also be possible for the user or other subsystems, that may have identified a malfunctioning component, to *manually trigger the recovery procedure* for that component. For this purpose the loader has to offer an interface that allows to trigger the restart of an application.

However, this work just aims to lay the foundation for a robust system and I limit the restarting of applications to their abnormal termination. Nevertheless the system can be easily extended to include the discussed mechanisms.

4.2.3 How to Restart

When an application is restarted, first the resources this application used, need to be freed. This can be done by unmapping the application's task capability. As the loader is the creator of the task capability the kernel will destroy this task object, and with it, all objects allocated by this task.

The application's local name space may remain. After that the config-file has to be re-parsed to reload the parameters of the task. Then the start-up procedure as described in section 4.2.1 is repeated.

It is desirable that not every application that terminates abnormally is restarted. In order to prevent unwanted restarting, a `l4reanimator` option is added to the loader config file, which can have three values:

`normal` In this case the application will not be restarted on abnormal termination.

`stateless` This setting means that the application should be restarted in case of abnormal termination.

And for completeness:

`checkpointed` In this case the application will not be restarted and will be set up to use checkpointing as described in section 4.5.

4.3 Retaining Access to Restarted Applications

After an L4Re server is restarted, all kernel objects created by this task are destroyed by the kernel. This means that whenever an L4Re server has created an IPC-gate, which was mapped to another task, this IPC gate will be destroyed and the mapping deleted. Thus after the restart all communication channels are lost and the restarted component is not accessible by other components. In this section I present a mechanisms that helps to re-establish these lost communication channels.

4.3.1 Initial Situation

In figure 4.1 we can see a typical L4Re scenario including two loaders. The two loader instances are necessary as the loader is also the page-fault handler for its child's L4Re-kernel. Because the loader is single threaded for complexity reasons, this would lead to a deadlock when the L4Re-kernel is page-faulting during the loader is waiting for an answer on a session open call of the page-faulting application. The loader in this case is not able to handle the page fault because it is waiting for the application to response.

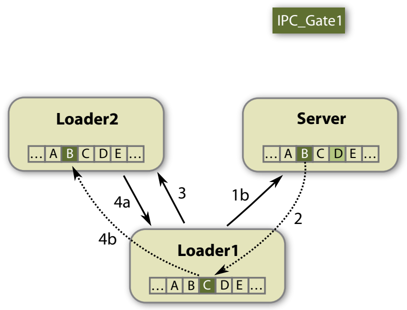
This can be mitigated by running an own thread inside the loader for each child, how it is done by the Genode operating system framework [2]. However, in the current design a second loader is needed to open the session for the client.

The start-up of the L4Re sample scenario is as follows:

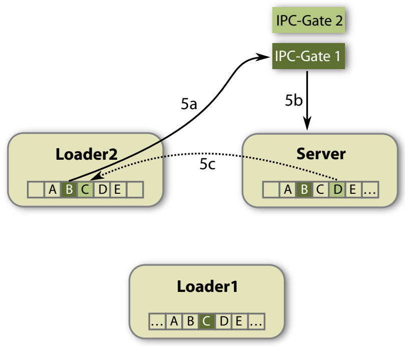
1. The first loader starts the server, which will create an L4Re service object represented by IPC gate 1. This IPC gate is mapped to the capability index *B* in the server's capability space.

2. The server registers its service in its local name space, which is managed by the loader. Thereby IPC gate 1 is mapped to the capability space of the first loader at the capability index *C*. This will make the name-space of the second loader complete.

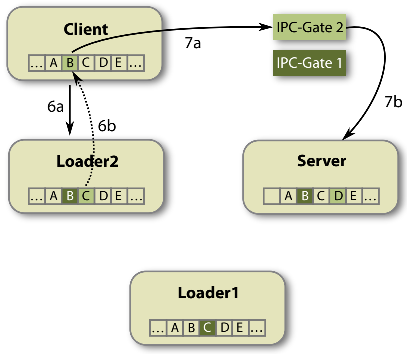
3. As now the name-space of the second loader is complete, the second loader is started by the first loader.



(a) Loader 1 is starting the server, which creates IPC_gate_1 (1), the server registers at its service (2). loader 2 is started by loader 1 (3), loader 2 queries service of server (4).

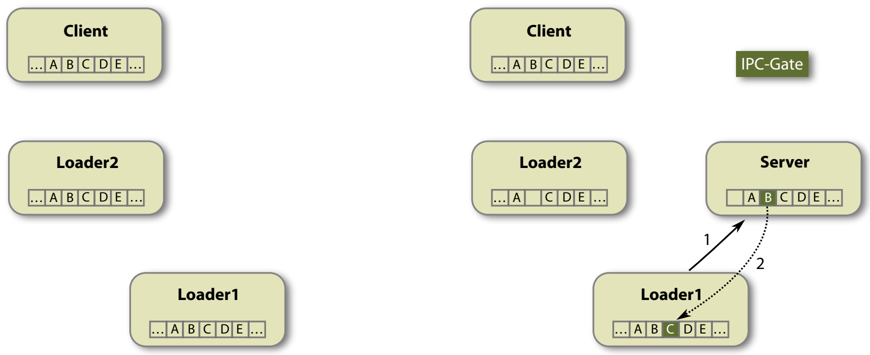


(b) Loader 2 is opening a session at server 5.



(c) Loader 2 is starting client (6), client is requesting server session (7), client communicates with server (8).

Figure 4.1: A typical L4Re scenario. Dashed lines represent mappings of capabilities.



(a) The server is terminated, all kernel objects created by server and their mappings are destroyed.

(b) The first loader restarts the server, which recreates the session IPC gate and registers its service in its local name space.

Figure 4.2: Situation before and after restart.

4. The second loader asks its local name-space for the service object of the server. As result the service object of the server will be mapped into the capability space of the second loader at capability index *B*.
5. In order to complete the name-space of the client, the second loader will open a session of the service by invoking the capability mapped at capability index *B*. This invocation will be answered by the server by creating another IPC gate identifying the session (mapped at capability index *D*) and mapping this capability to the second loader.
6. Now the client is started by the second loader.
7. As seventh step, the client will ask for the session represented in its name-space by a given name and thus will get the capability representing the session mapped (IPC gate 2) by the second loader.
8. Finally this capability can be used by the client for direct communication with the server.

4.3.2 Situation After Restart

Assuming the server was terminated for some reason, the scenario of figure 4.1 will change to the one represented in figure 4.2a. The kernel objects created by the server are destroyed and all mappings are revoked. When a capability is unmapped in one task's capability space, the capability will also be unmapped at all other tasks that had received a mapping of this capability from this task. Further, when a capability is unmapped by the creator of the kernel object, represented by this capability, the kernel will destroy this kernel object.

After restart (figure 4.2b) the server will recreate the service object and register its service at the first loader. However, the session and the mapping of the service object at the second server are lost.

4.3.3 Transparent Recovery of Lost Communication Channels

As shown in the previous section, it is necessary to the reestablish the communication channel between client and server. In this section I describe mechanisms that allows to recover lost communication channels while offering full transparency.

For this purpose two approaches are considerable. The first approach is using a proxy, to avoid unmapping of capabilities at the client. The second approach is to introduce a kernel mechanism helping to retain the communication, when a component is restarted.

Transparent Recovering by Delegating IPC Calls

The idea of this mechanism is to proxy all communication between server and client through a third party. The proxy has to create an IPC-gate representing each kernel object the client interacts with. So when the creator of a kernel object disappears, the IPC-Gates representing those kernel objects will stay mapped at the client.

However this approach has several disadvantages:

- First the problem of re-establishing the communication channel is still present. It has just moved from the client to the proxy.
- Further, there will be two additional context switches during every IPC. Because an IPC will first goes to the proxy and then is redirected to the server these two additional context switches will have a negative effect on the performance.
- It might be necessary for the proxy to introspect the IPC between the client and the server in order to know when a capability was mapped or unmapped. This will also influence the overall performance of the communication between client and server in a negative way.

Kernel Mechanism

Further, it is thinkable to use a kernel mechanism to recover lost capabilities. Such a mechanism could offer persistent kernel objects, that survive the termination of their creator. However, such a mechanism is likely to be complex and thus contradicts the design goals of L4ReAnimator.

Further, a restarted component will not know about the communication channels that were used before the restart of the components, as its state is lost. The developers of CuriOS showed that such a mechanism is feasible, but as I will show in section 4.4 this mechanism would require changes in the whole architecture of L4Re.

Conclusion

The proxying mechanism does not solve the problem of getting capabilities back and further will lead to a noticeable performance degradation and a kernel mechanism will increase the complexity of the kernel, contradicting the design goals of L4ReAnimator. In the next chapter I show that abandoning full transparency will lead to a design that keeps complexity low without a significant performance degradation.

4.3.4 Semi-Transparent Recovery

In this section I present a mechanism that semi-transparently recovers lost communication channels. Semi-transparent recovery here means that an application using a service will not notice that the server was restarted. The library abstracting the access to a service in contrary has to help reestablishing the communication channel. I show that the complexity that has to be added to the client-library of a service is rather small.

The central idea is that the client knows where a capability came from. This information can be stored and used in the recovery case to re-obtain the capability.

Capability Faults

There are two ways for an application to recognize that there is no capability mapped to a certain capability index. First, the application may proactively check if a capability is mapped to the capability index. This can be done by performing an IPC call to the task object with a special protocol-tag and the capability index as argument. The return value of the IPC call contains information about the mapping state of this index.

Second, the application may just use capability behind a certain index in normal fashion. If there is no capability mapped will fail. This is from now on called a *capability fault (CF)*.

Capability Fault Handlers

For handling these capability faults, my design introduces *capability fault handlers (CF-handler)*. A capability fault handler is an object that stores all necessary information for re-obtaining a capability. If recovery fails, the CF-handler will retry the recovery a certain number of times.

CF-handlers are stored in a *capability registry (CR)*. The CR is a storage for information on a specific capability index, such as CF-handlers. However, the CR can also be valuable for other tasks, e.g. garbage collection of user level objects.

There exist several types of CF-handlers, as they have to be implemented for each service that returns capabilities in response to a request, e.g. the name service. In order to support semi-transparent recovery, this service's client library has to be extended. This extension includes the instantiation of a CF-handler for returned capabilities and storing this CF-handler in the CR.

Capability Fault Handler Invocation

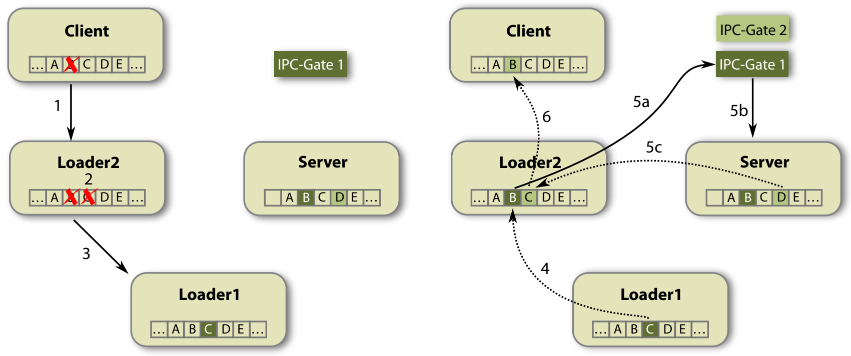
CF-handlers are called whenever a capability fault occurs. This is done by wrapping the `invoke` system call and interpreting its return value. In case the `invoke` system call fails because of a not-mapped capability, the system call returns `L4_IPC_ENOT_EXISTENT`, and the CF-handler is called.

Further, an application has to ensure that a capability is still mapped in its own capability space when another application requests it. To achieve this, the application can proactively check if the corresponding capability is still mapped before mapping it to the other application. However, as the application has to create a flex-page describing the capability that should be mapped, the checking can also take place automatically during the creation of this flex-page.

When all applications implement this policy the CF mechanism will propagate recursively down to the origin of the capability. This is comparable to page-fault handling in recursive address spaces.

Example

Looking back at the example from section 4.3.2 two CF-handler types are necessary: One to recover capabilities gained by name queries (names–CF-



(a) Capability mechanism propagates down to the first loader.

(b) Capability faults are resolved from bottom to top.

Figure 4.3: Illustration of the capability fault mechanism.

handler) and one for capabilities gained by opening service sessions (service–CF-handler).

A names–CF-handler stores the name that was used during the name service query and will repeat the query if it is invoked, whereas a service–CF-handler stores the capability index of the service object and the arguments used to open the session.

In figures 4.3 we can see the process of recovering lost capabilities. The loss of the capability will be recognized, when the client tries to invoke the lost capability (at index *B*) the first time after the restart of the server.

1. Now the names-CF-handler of the client is invoked and re-queries its name space for the session object.

2. The name-service implemented by the second loader, checks if the session capability is still present in its own capability space (at index *C*), and because this is not the case, it invokes the CF-handler for capability index *C*. This CF-handler in turn is a service–CF-handler which tries to open a session of the service object at capability index *B*.

3. As this invocation fails, the CF-handler of capability index *B* is invoked. This CF-handler is a names–CF-handler and will query the first loader for the service object.

4. This query now can be answered by the first loader, because the server has again registered its service at the first loader.

5. Now, as the service object is again mapped to the second loader at index *B*, the service-CF-handler can finish its request and re-open a session. The server will recreate a session (IPC gate 2) and this session is mapped to the second loader at index *C*.

6. Now the name query of the client's names-CF-handler can be answered and the client can again communicate with the server.

Limitations of Capability Fault Handling

Capability fault handling also has some drawbacks, which I discuss in this section. First, it is not possible to recover capabilities for which no CF-handler is implemented. The service developer has to implement an additional component, i.e. create the CF-handlers for his service. However, as I show in section 6.1.1 these CF-handlers are rather small and this additional effort is negligible.

Further, recovering capabilities belonging to the L4Re environment is problematic. For some capabilities, like the name service, recovery would be possible by asking the applications parent to map the name service capability again to the application. However, if the parent capability is lost too, there is no one to ask for this capability. A pager capability can indeed by requested at the parent, but a page fault occurring during the recovery mechanism can not be handled. To forgo this issue, the use of pinned memory for the CF-handler is imaginable. Anyhow, it is questionable whether it is reasonable to recover from such serious faults or if it would not be better to restart the whole affected subsystem.

4.4 Server State Regions for L4Re

For offering persistence for L4Re servers a mechanism like the SSRs of CuriOS is thinkable. In this section, I outline how this mechanism can be realized in TUD:OS. As described in section 3.3.1, the benefit of server state regions is that only the client-related server states of clients currently interacting with the server can be corrupted.

Because in the common case the client should not be able to read or even write its server-side state, the memory for the SSR may not come from the client. Also if the client grants a page to the server, it is possible that it just granted a second mapping of the page, and still has full access to the first mapping. So a third instance is needed to manage the memory for SSRs.

This third instance can be a proxy, that also initiates a session to a server. When the session is initiated by the proxy, it may allocate the memory for the SSRs and whenever the client makes a request, the SSR can be mapped writable to the server's address space. The proxy would have the same role as the server state manager in CuriOS.

The SSR may be allocated from an allocator object the client previously has given to the proxy, so that the memory is accounted to the client. The server has to check that the allocator is trustworthy, for example by asking its parent to affirm the trustworthiness of the allocator object. If the parent task cannot confirm the trustworthiness of the allocator it has to call *his* parent, and soon, until the first parent can confirm the trustworthiness or the root of the subsystem is reached. This way it can be verified that the allocator came from a trusted sub-tree in the task hierarchy.

Although it is possible to implement a mechanism offering SSRs for L4Re, all L4Re servers would have to be adapted to use this technique. Because these adaptions are too big to be done in this work and the proxying mechanism would introduce a considerable performance overhead, I chose checkpointing over SSRs to offer persistence to L4Re applications.

4.5 Checkpoint/Restore for L4Re

Checkpointing is a mechanism that allows to store the whole state of a running application. This checkpoint can later be used to restore the application's state to the state when the checkpoint was taken. Thus, checkpointing can be used to create failure-tolerant systems.

Checkpointing can also be useful for other use cases. It can be used to implement suspend/resume functionality. Further, Gao et al. [17] use checkpointing and replay in order to generate run time patches to prevent a bug, causing an application crash, to reoccur. Carlyle et al. [9] proposed to store the application state on a non-volatile memory in order to accelerate the restart of an application.

Plank [27] proposes the categorization of checkpointing solutions into (1) OS-Checkpointing, (2) transparent user-level checkpointing *and* (3) non-transparent user-level checkpointing.

Using *OS level checkpointing* the operating system will take the checkpoint. This process is transparent for the applications running on the OS. Examples for this approach are EROS [30] and Zap.

Eros takes periodic checkpoints of its whole system state, which is then written to disk. However, in most cases it is enough, to checkpoint only several subsystems of a system, as not all OS components are stateful. Laadan et al. [23] have demonstrated with ZAP a checkpointing solution that is able to transparently checkpoint and restart multiple processes on a Linux kernel.

Transparent user-lever checkpointing means that the state is stored by the application itself, which is done transparently by a library linked to the application. An example for this technique is libcktp [28].

An application using *non-transparent user-lever checkpointing*, will proactively use mechanisms offered by the checkpointing solution to store its application state.

Skoglund et al. [31] proposed a checkpointing mechanism for L4. Their proposal introduced a dedicated checkpointing server that acts as a pager for the rest of the system. The system is divided in transient tasks and persistent tasks. Transient tasks are device drivers, as those are not checkpointable, and the checkpoint server itself. The checkpoint server also holds the *thread control blocks* (TCBs) of the persistent tasks. These TCBs are mapped to the kernel and periodically copied to a persistent storage (e.g. hard disk).

The checkpoint server is the pager for all physical memory and takes a consistent copy by revoking write access in all other address spaces during the checkpoint is taken. The page faults of task trying to write to the memory are handled after taking the checkpoint.

However, holding TCBs outside the kernel requires total trust in the checkpoint server. A more decentralized solution is desirable.

Because of the lower complexity of a user-land solution I decided to implement a transparent user-level checkpointing solution for L4Re.

4.5.1 Application State in L4Re

First it must be clarified what belongs to the state of an L4Re application. Further, it is necessary to know which parts of this state have to be stored in order to be able to recover this state later. These two problems will be discussed in this section.

An application's state is represented by the following:

Memory All mapped dataspaces as well as the address space layout of the checkpointed application belong to the applications state. This includes the stack, the heap and the BSS. Also shared memory regions have to be counted to an applications state, but they only have to be stored when the checkpointed task was the creator.

Threads Further, the number of threads and their execution context belongs to the state that has to be stored when taking a checkpoint.

Capabilities At this point one has to distinguish between the capabilities the task created itself and those it got mapped by other applications.

Acquired Capabilities The initial set of capabilities, i.e. the L4Re environment has to be recreated by the restoring instance. Dataspaces obtained by the allocator are implicitly stored when memory is stored. All other capabilities can be restored by the capability fault mechanism described in section 4.3.4.

Created Capabilities Capabilities created by the application itself can also be restored using the capability fault mechanism, whereas CF-handlers have to be written for capabilities obtained from the factory.

4.5.2 Checkpoint Process

In this section I will describe the process of taking a checkpoint for a single application.

The address-space layout of the application is stored by the L4Re-kernel in form of a region map. Even though the L4Re kernel and the application run in the same address space, the application itself does not know about the memory location of the region map, and should not be accessed by application threads. For this reason, it seems practicable to instruct the L4Re kernel to store the address-space layout.

The application's memory that should be stored in contrary is known to the application. Indeed, the L4Re kernel knows about all mapped dataspaces as they are part of the region map, but it does not know which dataspaces have to be stored when a checkpoint is taken. In case the dataspace contains I/O-memory or shared memory created by another application, this dataspace does not have to be stored. Also dataspaces containing files do not have to be stored.

The dataspaces that have to be stored in the common case are the ones that are received through allocator objects and the writable segments of the binary.

Possible solutions are to explicitly inform the L4Re kernel which dataspaces to store, let the L4Re proxy calls to the allocator to keep track of all allocated dataspaces, or let the application store the dataspaces itself.

The same situation exists for *threads*. The application itself has knowledge about its threads, so the thread state can be stored can be stored by the application. If the serialization of the thread state should take place in the L4Re kernel, the threads have to be previously registered at the L4Re kernel.

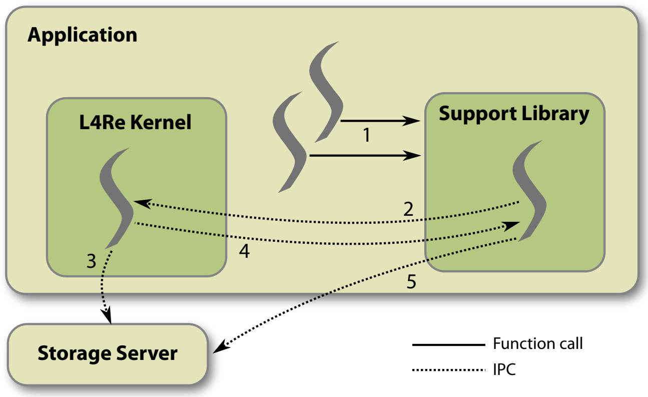


Figure 4.4: Illustration of partitioned checkpointing: (1) Clients register at checkpointing library, (2) checkpointer thread initiates a checkpoint, (3) L4Re kernel takes snapshot of memory a (4) sends a response to checkpointer thread. Finally, (5) Checkpointer thread stores thread state.

Further, the L4Re kernel thread may proxy calls to a factory to get knowledge about the creation of new threads.

For *capabilities*, the capability registry can be used. The information necessary to restore them is available in form of CF-handlers, which will be invoked lazily when the capability is the first time used after a checkpoint restore. This information is part of the memory checkpoint and thus already stored.

In the the next two sections, I discuss two possible solutions, for the question where the checkpointing process should take place. The first one is to partition the checkpoint process over the L4Re kernel and the application. The second one is to let the L4Re kernel perform the whole checkpointing process.

Partitioned Checkpointing

Using partitioned checkpointing a library has to be linked to the application that keeps track of all existing threads in the application. This method is illustrated in Figure 4.4. The threads have to register at this library (1), which has to be done during the creation of the thread, An extra thread, created by the library, periodically initiates the checkpointing process and serialize the thread states.

The L4Re kernel in contrary is responsible for storing the address space layout and the data spaces and has to offer a way to register dataspaces that should be stored when taking a checkpoint.

The actual procedure to take a checkpoint is:

- The checkpointer thread stops all threads.
- The checkpointer thread sends an IPC to the L4Re kernel that it should store the dataspace and address space layout (2).
- L4Re kernel sends a serialized version of the region map and a copy of all dataspaces that have to be stored at a storage server (3).
- The L4Re kernel signals the checkpointer thread via IPC that storage process finished (4).
- Finally, the checkpointer thread copies the thread state of the application threads and then lets the threads continue. The gathered thread state is also stored at the storage server (5).

L4Re Kernel Only Checkpointing

If the checkpointing process should take place solely in the L4Re kernel, the L4Re kernel not only has to know all dataspaces that have to be stored, but also needs knowledge about all threads.

The L4Re kernel can keep track of all dataspaces the client allocates by proxying calls to the applications allocator. This is not possible if the application uses multiple allocators. The L4Re kernel could also keep track of the creation of threads by proxying calls to the factory of the application. As the L4Re kernel also needs the address of the thread's UTCB it is also necessary to proxy calls to the thread objects, in order to get knowledge of the UTBC's address when the thread is bound to it.

The checkpointing process itself would be very similar to the one described above, except that the L4Re kernel is responsible to stop the threads, gather the threads' state and to let the threads continue.

4.5.3 Restore Process

The restore process takes place in the L4Re kernel and is identical in both, the partitioned and the L4Re kernel solution.

The L4Re kernel will be started by the loader as usual, but is signaled that a checkpoint restoration should take place. If this is the case, the L4Re kernel asks the storage server for the checkpoint and then

- Restores the address space layout, attaching all regions and reserved areas stored in the checkpoint,

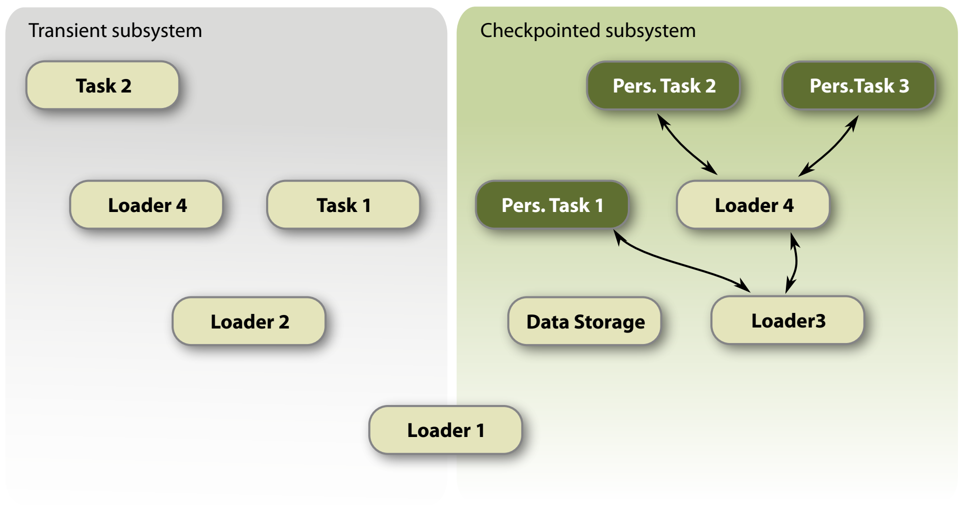


Figure 4.5: Illustration of checkpointing whole subsystems. The dark components implement the checkpointing functionality described in section 4.5.2. The loader instances are stateless.

- Maps all dataspace capabilities to the stored index and
- Recreates all threads and sets their state to the one stored in the checkpoint.

This process is described in more detail in the next chapter.

4.5.4 Checkpoint/Restore for Subsystems

As I have chosen user-level transparent checkpointing, only single applications can be checkpointed, at once. When multiple applications are to be checkpointed, it has to be assured that state shared by the applications is in a consistent state when a checkpoint is taken.

This can be achieved by adding the possibility to initiate the checkpoint process from outside the application. The initiation of the checkpoint process has to be two-staged. In the first phase all application threads are suspended and in the second phase the actual checkpoint is taken.

This way, it is possible to recursively checkpoint a whole subsystem as shown in figure 4.5. On the right side we can see a persistent subsystem, which is checkpointed periodically. As loaders hold no state which has to restored during a checkpoint restore, they don't have to be checkpointed. Tasks that are actually checkpointed are the dark ones.

The checkpointing process is initiated by the loader in the lowest layer of the subsystem, i.e. loader 3. First, this loader will stop all checkpointed applications, which is done by sending an IPC to the tasks L4Re kernel. If the task is a loader, it has to forward the request to its child's. After all threads are suspended, the loader starts the checkpoint process of its child also by sending an IPC message to the applications L4Re kernel. Again, this IPC will be forwarded to all other persistent tasks in the sub-tree. When all children replied to the IPC the checkpoint process has finished and a consistent checkpoint of a whole subsystem is taken.

5 Implementation

In this chapter, I describe the implementation process of L4ReAnimator and elaborate problems that occurred during the implementation.

First, I explain how I reached link-time transparency and second, I describe the implementation of the capability registry and the capability fault mechanism. In the last part of this this chapter I explain the implementation of the checkpoint/restore mechanism, which is part of L4ReAnimator.

5.1 Link-Time Transparency

L4ReAnimator consists of three libraries, which can be linked to applications, if the functionality of L4ReAnimator shall be used.

The first library contains the capability registry and a set of capability fault handlers, covering most of L4Re's services. The second library implements the capability watching mechanism, as described in section 5.3.4, whereas the third library supports checkpointing as described in section 5.4.

All libraries provide their functionality by linking them to the application. This is done by declaring the init functions of these libraries as constructors, which are invoked by the C-runtime before the main function of the binary is called.

L4Re was modified to use L4ReAnimator, by placing L4ReAnimator calls inside the L4Re libraries. All public functions of the L4ReAnimator libraries are declared as weak symbols. The header files of the libraries contain wrappers that will first check for the existence of the libraries and then call the actual library function. If L4Reanimator is not linked to the application, this modifications do not have any functional effect. However, it is to expect that a small performance degradation, caused by the checking for the existence of L4ReAnimator, will occur.

5.2 Capability Registry

As described in section 4.3.4, the application needs a place to store information on capabilities, for which I created the *Capability Registry* (CR).

The CR is represented by the abstract C++ class `cap_registry`, offering an interface for synchronized access to a hash table containing objects of the type `Cap_registry_entry`. By inheriting from this class, specialized registries are created, holding specific information. An example for a specialized version of the CR is the `Capfault_handler_registry`, holding objects of the type `Capfault_handler_entry` which in contrast is derived from `Cap_registry_entry`. This special CR is used for storing CF-handlers.

5.2.1 Creation of Capability Entries

The entries of the capability registry are instantiated whenever the application allocates a new capability index. An L4Re application has to keep track of used and unused indices. This functionality is offered by a small utility library of L4Re including the functions `cap_alloc` and `cap_free`. A call to `cap_alloc` returns a previously free capability index, whereas `cap_free` frees the index.

For each call to `cap_alloc`, a new entry in the CR has to be created. This is done via a method call to the CR. However, this approach introduced difficulties when the application runs out of heap memory. In this case the C-runtime will try to enlarge the heap by allocating new memory from the applications allocator object. As the memory is returned in form of a dataspace, a free capability index is needed for the dataspace capability, which in turn will require a new CR-entry. In order to instantiate the CR-entry also a memory allocation call to the C-runtime is necessary, which will again try to enlarge the heap causing the application to run into an endless loop.

In my implementation, I solved this problem by the introduction of a dedicated C++ memory allocator for L4ReAnimator. This allocator is used to get the memory of CR-entries and always has enough memory for at least one CR entry required to enlarge the heap of the application.

5.2.2 Synchronization and Initialization of the Capability Registry

As multiple application threads may simultaneously look-up, create, or modify CR entries, all accesses to the CR have to be synchronized. This synchronization is implemented by using a pthread mutex. A thread accessing the CR has to acquire this mutex.

The init process of the CR takes place in a special function, which will be called before the main function of the application. In this function the allocator described in the previous section, and the CR itself are instantiated. The mutex synchronizing access to the CR is also created here. The instance of the CR is stored in a global variable, which is accessed by a static method of the CR base class.

5.3 Capability Fault Handling

In this section, I describe how the CF-handling mechanism is realized in L4Re. First, I show how CF-handlers are built and invoked, after which I describe the implementation of selected capability fault handlers. In the last part of this section I present problems that occurred during the implementation of the capability fault mechanism and how I solved them.

5.3.1 Anatomy of a Capability Fault Handler

A CF handler is an object inheriting from the base class `cap_fault_handler`. The base class implements the functionality that is common to all CF-handlers, i.e. the number of retries, the period of retries and logging. Further, it is necessary to store the content of UTCB, because recovering a capability involves an IPC operation which will invalidate the content of the UTCB. After the recovery of the CF-handler restores the UTCB.

When writing a new CF-handler, the virtual method `specific_handle` has to be implemented. This method has to contain the functionality for recovering a capability and returns a boolean value, specifying if the recovery process was successful.

IPC Call Wrapping

As the CF-handling mechanism should be transparent for application developers, a mechanism had to be found to transparently handle capability faults. This is done by checking the return value of the invoke system call. This check has been added to the function `l4_ipc_call`. If the system call is returning the error code `L4_ENOT_EXISTENT` L4ReAnimator will initiate the capability fault handling. After the successful recovery of the invoked capability, L4ReAnimator repeats the system call. Thereby, the application will not notice that a capability fault occurred.

Invoking a Capability Fault Handler

Capability fault handling is initiated by calling the C-wrapper function `handle_cap_fault`, which takes a capability index as argument. This function will look up the CF-handler registry object and call its method `handle_cap_fault`. When this method is called, the registry looks up the corresponding entry containing the CF-handler. The CF handler then is invoked and tries to recover the capability.

5.3.2 Implemented Capability Fault Handlers

During this work, I implemented CF-handler for several L4Re services. Each of them enables L4ReAnimator to recover capabilities that were mapped to the application by the corresponding service. I describe four of them in this subsection.

Names

All the names-CF-handler has to know to recover a capability, is the name that was used to obtain the capability from the name service. The constructor of the names–CF-handler takes the name as argument and stores a copy of the name inside the handler.

In the `specific_handle` function the handler repeats the name query with the given name. The handler is created by the name service library whenever a name query succeeded and no CF-handler for the corresponding capability index already existed.

On the server side a minor modification was necessary. The name server was modified to check if a capability is still mapped in its own capability space before mapping it to the client. If the capability is not mapped, it will initiate the CF-handling for this capability.

Service

The service–CF-handler is able to recover capabilities that were obtained by invoking the open call of a service object. This CF-handler has to store the capability of the service object that is invoked, as well as the arguments used to open the session. The arguments are represented by a byte array of a given length. During the instantiation, the service–CF-handler stores the capability index of the server object and the arguments. The `specific_handle` method of this handler repeats the open call to the session capability with the stored arguments.

As this CF-handler relies on another capability it is an example for the propagation of capability faults: If the invocation of the service capability fails its CF-handler is initiated automatically as described in section 5.3.1.

Framebuffer

The framebuffer object provides a shared memory region represented in form of a dataspace. The dataspace capability has to be recovered if the server providing the framebuffer is restarted. The framebuffer CF-handler has to store the capability index of the capability representing the framebuffer and has to request the shared memory region again in its `specific_handle` method.

Event

The event service uses two capabilities. The first capability is an IRQ to notify the client about new events and second, a dataspace that holds the events. This handler stores the capability index of the event object and in its `specific_handle` method it has to re-query the IRQ and the dataspace capability. A specialty here is, that the same CF-handler instance is registered for two capability indices, because it cannot be determined, which capability invocation leads to recovery.

5.3.3 Dataspaces vs. Memory Mappings

When a dataspace capability is unmapped, it seems logical that memory mappings of this dataspace are also unmapped. but this is not the case. As illustrated in figure 5.1 a dataspace is only the communication channel used to get memory mappings. A loss of a dataspace only means that an application has no possibility to request new mappings from the dataspace. This was leading to problems during the recovery of a scenario involving framebuffers.

Scenario

While running an application using a virtual graphical console offered by `con`, `con` was restarted. During the next invocation of the framebuffer object, which took place in order to redraw the framebuffer, the capability fault mechanism successfully recovered the framebuffer capability. However, the

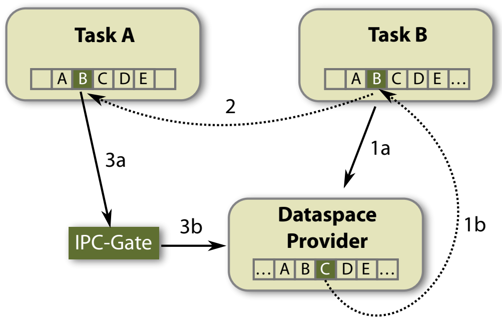


Figure 5.1: Dataspaces in L4Re: Task B (1a) allocates a dataspace at the dataspace provider, which creates an IPC gate representing the dataspace and (1b) maps the capability to this IPC gate to task B. The dataspace capability is (2) mapped to task A by task B. When task A (3) invokes the map operation on the dataspace, the memory is mapped by the *dataspace provider* to task A.

system did not notice that the dataspace describing the framebuffers memory was also lost. The old framebuffer memory was still mapped as the dataspace provider was not con and the application kept on drawing in the old framebuffer's memory.

Solution

The application has to be notified that the old framebuffer is not valid anymore. One way to achieve this is to unmap the old framebuffer memory. Then the next access to this memory would cause a page fault, which can be handled by the L4ReAnimator framework, by recovering the dataspace capability. First, I present mechanisms that can be used to unmap the old framebuffer from the application's address space and second, I describe how the resulting page fault is handled by L4ReAnimator.

Garbage Collection

If a task is destroyed, it is desirable that all resources the task exclusively used are freed. This also includes dataspaces that were allocated by the task at its allocator. However, the dataspace provider does not necessarily know when an application terminates and the resources used by this application can be freed.

In L4Re garbage collection should be used to clean up resources. To support this, the kernel offers a mechanism that allows a tasks to check if a certain capability is still mapped to other tasks. If this is not the case, it may clean up the associated resources of this capability, i.e. in case of a dataspace, unmap the memory at all other tasks. However, garbage collection is not yet implemented in L4Re.

Watching Capabilities

Another possibility is to let the application itself unmap the memory. In order to do this, the application has to notice when a dataspace capability is revoked. In the case of revocation, all mappings received by this dataspace have to be unmapped. This introduces two sub-problems: First, gain knowledge about the revocation and second store all mappings of the dataspace in order to be able to unmap them later.

The first problem is solved, introducing an extra thread, the *capability watcher thread*. This thread periodically iterates over a list of watch objects containing a capability index and an `action` method. For each object, the watcher thread checks if there is still a capability mapped to the given index and if not, calls the action method of the object.

The second problem is solved by creating special dataspace watch objects storing all mappings of a dataspace. The dataspace–watch-objects are stored in the capability registry and are created when the first mapping from a dataspace occurs. Whenever a mapping takes place, this mapping will be stored in the watch object. When the action method of a watch object is called, all stored mappings are unmapped.

Further it would be possible to inform the L4Re kernel about the loss of a dataspace. As the L4Re kernel already has the information about all mappings of a dataspace in its region list, it could iterate over all regions, and remove all mappings of the lost dataspace.

The capability watching mechanism is implemented as an additional library. This library takes care of creating the watcher thread and the periodic checking of capabilities.

Handling Page Faults

After making sure that memory originating from a revoked dataspace is unmapped, the application will run into a page fault when it is trying to access this memory the next time. Here, I describe how this page fault is handled by L4ReAnimator.

As mentioned before the L4Re kernel is the pager of all other application threads. The L4Re kernel has been modified in a way that it will translate a page fault into an exception, if for the page fault location a region is attached, but the dataspace capability for this region is not mapped.

This exception is handled by a dedicated exception handler thread: The exception handler receives the thread state of a faulting thread, including the memory location where the page fault occurred, and then performs the following steps:

1. Query the L4Re kernel for the index dataspace capability affiliated with the page fault memory address. The L4Re kernel offers a look up call, which takes a memory address as argument and, if a region exists for this address, returns the dataspace's capability index stored in that region.

2. After getting this information the exception handler starts the capability fault mechanism in order to get the dataspace capability mapped again.

3. It will now answer the exception IPC. The thread state of the faulting thread is not modified.

As the thread state of the faulting thread is not modified, the thread will trigger the same page fault, but now, as the dataspace capability is mapped again, the L4Re kernel can resolve the page fault and the new dataspace's memory, i.e. the new framebuffer, is mapped into the application's address space.

5.3.4 Waking Up Waiting Threads

Another problem is that under certain circumstances a caller may not notice that a callee disappeared during an IPC-call. An IPC call consists two steps: First the caller performs a send to the callee and then waits for an reply.

When the callee disappears, e.g. because he crashed, during the caller is waiting for the reply, the caller will wait for ever.

There are three possible solutions to forgo this problem, whereas two of them are userlevel-only solutions and the third one requires a kernel patch. First I present the two userlevel-only solution and then present the kernel-level solution.

Timeouts

The first solution is to use timeouts for each IPC operation. Using timeouts the client will wait for a reply at most for the timeout duration. IPC operations that should not have a timeout have to be wrapped in multiple IPC operations with timeout. If the timeout expires, because the callee is not there anymore, the next try will fail and the capability fault mechanism can recover the capability.

Watching Capabilities

The capability watching mechanism, described in section 5.3.3 can also be used to solve this problem.

Before a thread performs an IPC-call, it has to inform the capability watcher thread, with which capability the thread is currently interacting with. The watcher thread will then periodically check if this capability is still mapped and wake up the calling thread if necessary.

I implemented this functionality by adding a thread list to the capability watcher library. This list contains a watch object for each thread, which contains the capability's index, the thread is currently interacting with and the index of the thread's capability.

A thread can access its entry by using a pointer in its *Thread Local Storage* (TLS). This allows fast and asynchronous access to the watch objects when the thread wants to set the capability index it is currently interacting with.

The watcher thread periodically iterates over the thread list and checks for each entry if the capability at the stored index is still present and if not, it wakes up the thread by performing an `ex_regs` call to the thread capability stored in the list entry.

Kernel Modification

In the kernel this problem is solved by checking if the callee thread is still holding a reply capability when it is destroyed. The reply capability is a capability that is implicitly mapped during an IPC representing the caller thread. If the reply capability is still mapped, the caller thread is woken up and the IPC_CANCEL flag of this thread is set.

Summary

The kernel modification solves the problem described in this section and should introduce no performance overhead. However, as the capability watcher already exists for dataspaces, I also extended this mechanism to watch threads. As with capability watching a solution already exists that is completely implemented in the user-land I did not implement the timeout solution.

5.4 Checkpointing

In this section I describe the implementation of the checkpoint/restore mechanism included in L4ReAnimator. First, I present the dsstorage, a small server offering a way to store dataspaces and then explain how a checkpoint is structured. Thereafter I describe how a checkpoint is taken and how the restore mechanism works.

5.4.1 Dataspace Storage

As TUD:OS does not contain a writable file system at the point of this writing, a facility is needed to store data persistently. For this reason, I implemented the dsstorage server, which stores copies of dataspaces under human-readable names. The interface offered by the dsstorage is the following:

`store`	Stores a dataspace under a given name. The dataspace will be stored in a copy-on-write fashion. The call will return a failure if the name is already used.
`get`	Returns a copy of the dataspace previously stored under a certain name.
`link`	Creates an entry for a given name pointing to an entry of another name. This call can be compared to the creation of symbolic links in a file system.
`del`	Deletes an entry for a given name.

5.4.2 Structure of a Checkpoint

A checkpoint consists of multiple dataspaces, which are stored at the dsstorage. These dataspaces include:

The checkpoint dataspace This dataspace contains the sequence number of the checkpoint, a serialized version of the region map and the capability indices of the dataspaces that are part of the checkpoint, i.e. the applications data. The name used to store this checkpoint is »`ckpt_N`«, wheres N is the sequence number of the checkpoint. This dataspace is created by the L4Re kernel.

The thread dataspace This dataspace contains the thread state of all application threads and is created by the application level checkpointing thread and stored under the name »ckpt_N_th«, whereas N is the sequence number of the checkpoint.

Further, the application dataspaces containing the actual memory content are stored at the dsstorage. They are stored under the name »ckpt_N_ds_ID«, whereas N is the sequence number of the checkpoint and ID is the dataspace's capability index.

5.4.3 Taking a Checkpoint

I implemented the checkpointing method described as partitioned checkpointing in section 4.5.2. This method involves an extra application thread, here called checkpointer thread, serializing the state of all other application threads.

The checkpointing functionality required inside the application is implemented as a library. In order to be able to store all thread states, the library has to know about all threads of the application. For this reason, the library offers functions to register and unregister threads. In the register function, the UTCB address of the new thread and the thread's capability index are stored in a thread list.

The initialization of the checkpointing library takes place in a function, which registers the main thread and creates the checkpointer thread. The checkpointer thread will sleep for the a certain amount of time, the checkpoint period, and then take checkpoint.

Gathering Client State

As mentioned before, the memory and the address space layout are stored by the L4Re kernel. Only the thread state is stored by the application level checkpointer thread. However, the checkpointer thread has to take care that the memory is in a consistent state when a snapshot is taken, what his is done by halting all threads.

After all threads are stopped, the checkpointer thread performs an IPC call to the L4Re kernel. During this call the L4Re kernel stores the address space layout and the application data. As result of this call the L4Re kernel returns the name of the checkpoint dataspace.

When the IPC call returns, the checkpointer thread will take a snapshot of the thread state of the application. The checkpointer thread performs the following steps:

1. *Allocate a dataspace* that is big enough to hold all thread states. This dataspace is called *thread dataspace*. here
2. Iterate over all threads and do *for each thread*,
   a) *Store its UTCB* in the thread dataspace.
   b) *Set the checkpointer thread as its exception handler.* This is done using the `control` method of the thread object. The checkpointer thread will store all values returned by the `control` call in the thread dataspace. These values include the capability indices of the thread's pager, exception handler and scheduler.
   c) *Trigger an exception* using the `ex_regs` method of the thread object.
   d) *Let the thread run again.*
   e) *Handle exception.* During the exception IPC the checkpointer thread receives the CPU state of the checkpointed thread. This state is stored also in the thread dataspace. Afterward1 the old exception handler is restored.
3. *Store the thread dataspace* containing the thread states at the dsstorage.
4. Link the name returned by L4Re kernel to a name that refers to the last taken checkpoint, e.g. »`checkpoint_latest`«

Checkpointing Inside the L4Re Kernel

In this section I describe the modifications to the L4Re kernel that were necessary to get checkpointing running. These modifications include the actual checkpoint/restore functionality, as well an extension to the IPC interface offered by the L4Re kernel.

Further, the L4Re kernel has to maintain a list containing the indices of all dataspaces, that have to be stored during a checkpoint. The indices of the dataspaces containing the BSS and the main thread's stack are added to that list.

IPC Interface The IPC interface of the L4Re kernel was extended by the `checkpoint` protocol, offering the following calls:

`add_ds`	Adds a dataspace to the list of checkpointed dataspaces.
`remove_ds`	Removes a dataspace from the list of checkpointed dataspaces.

`ckpt`	Takes a snapshot of the address space layout including all checkpointed dataspaces. This call takes a dsstorage session as argument and returns the name under which the checkpoint dataspace is stored at the dsstorage.

`add_ds` and `remove_ds` are called by the checkpointing library whenever a new dataspace is allocated/freed from the applications allocator. The ckpt call is invoked by the checkpointing thread, after all threads are stopped.

Storing the Address Space Layout The checkpointing process inside the L4Re kernel works the following way:

1. Allocate a dataspace for the region map. This dataspace also contains some meta information, including the sequence number of the checkpoint, the capability indices of stored dataspaces and the number of region map entries. The size of this dataspace is determined by the number of region map entries and the number of stored dataspaces. The dataspace will later be stored under the name »`ckpt_N`« in the dsstorage, whereas n is the sequence number of the checkpoint.

2. The L4Re kernel iterates over the region map, serializes all region map entries and stores them in the previously allocated dataspace.

3. Now the L4Re kernel iterates over the dataspace list stores the dataspace's capability indices in the previously allocated dataspace. Further, it will deposit a copy of the dataspaces at the dsstorage under the name »`ckpt_N_ds_ID`«, where N is the sequence number of the checkpoint and ID is the capability index of the stored dataspace. Using this naming scheme, the names of the dataspace copies can be reconstructed during the restore process, using the information stored in the checkpoint dataspace.

After performing these steps the L4Re kernel will return the name of the checkpoint dataspace to the calling thread, i.e. the checkpointer thread of the checkpointed application.

5.4.4 Restoring a checkpoint

After I showed how checkpoints are taken, in this section I describe how a checkpoint is restored. During the start up the L4Re kernel queries a dsstorage session from its name space. If the dsstorage session is found, it requests the dataspace with the name »`checkpoint_latest`«. If this dataspace exists this checkpoint will be restored.

Restore Process

After loading the binary, the L4Re kernel reconstructs the region map stored in the checkpoint dataspace. Thereafter it requests the dataspaces whose capability indices are stored in the checkpoint dataspace from the dsstorage. The dataspace capabilities are mapped to the same indices they were when taking the checkpoint.

After the address space layout is restored and the dataspace capabilities are mapped to their previous capability indices, the threads have to be restored. The L4Re kernel will query dsstorage for the thread dataspace in which the threads state is stored in. To restore the threads the L4Re kernel will iterate over the thread list, stored in the thread dataspace and perform for each entry in this list:

1. Create a new thread object, using the task's factory.
2. Copy the old UTCB content to the UTCB address stored in the entry.
3. Perform a `control` on the new thread in order to:
    a) Bind the thread to the task and the UTCB stored in the entry.
    b) Set L4Re kernel thread as pager and exception handler.
4. Perform an `ex_regs` to trigger an exception in the new thread.
5. Let the thread run. Now the thread will send an exception IPC to the L4Re kernel.
6. Wait for the exception of the new thread. The exception is handled the following way.
    a) Stop the thread again, to make sure that there is only one application thread running.
    b) Set pager and scheduler to the values stored in the entry. Send the CPU context stored in the entry as answer to the exception IPC.

After performing these steps for each entry in the thread dataspace, the L4Re kernel will iterate again over all entries and let the threads run.

6 Evaluation

In this chapter I first give an overview about the code size of the L4ReAnimator framework and second analyze the performance overhead introduced by the mechanisms the framework offers.

6.1 Complexity Analysis

L4ReAnimator consists of several components providing the capability fault mechanism and a checkpointing library. In this section I give an overview about the complexity of those components and describe the amount work that was needed to create L4Reanimator.

6.1.1 Capability Registry and Capability Fault Handlers

The major part of the work was the implementation of the capability fault handling mechanism. The main components offering the CF-handling are the capability registry, the capability watcher and of course the CF-handler. Also adaptions to L4Re were necessary, to let the L4Re libraries install CF-handlers. The complexity of these components can be seen in table 6.1.

The capability registry compromises ca. 700 *Source Lines Of Code* (SLOC) code[1] and a small amount of C code wrapping the C++ functions so they can

[1]The amount of code measured using David A. Wheeler's »SLOCCount«.

Component	*SLOC*
Capability registry	771
Capability watcher	288
Capability fault handlers	494
Modifications to L4Re and L4Sys	130

Table 6.1: SLOC of the components offering the capability fault mechanism. If components were just modified only the amount of modified SLOC is specified.

Component	*SLOC*
Checkpointing Library	476
Modified L4Re kernel	405
dsstorage	494
L4Re	2

Table 6.2: SLOC of the components offering the checkpointing mechanism. If components were just modified only the amount of modified SLOC is specified.

be used by programs, implemented in C. The capability watcher consists of 300 SLOC and the capability fault handlers together amount to 494 SLOC. A CF-handler for one service is rather small. The handler for the capabilities acquired by the name server for example counts 40 SLOC. Circa 130 SLOC were modified in the L4Re itself. It is planned to move the source code out of L4Re and put in into utility functions which are part of the CF-handlers.

Binary Complexity

For performance reasons, I implemented some of the functionality in header files, so that the functions can be inlined by the compiler. Moreover some of the functionality is still directly implemented in L4Re. This together leads to a growth of the binaries, also if the L4ReAnimator libraries are not linked to the library. For example, the statically linked binary of con without L4ReAnimator, increased by 170 kilobyte. This circumstance can be mitigated by using dynamic linking or putting the functionality into the L4ReAnimator libraries. However, some growth cannot be avoided.

6.1.2 Checkpointing

Checkpointing for L4Re consists of a library linked to the application and further a modified version of the L4Re kernel. As there existed no writable file-system for L4Re at the time of this writing, a facility was need to store data beyond application restarts. For this purpose I created the dsstorage.

The complexity of this components can be seen in Table 6.2. In the current implementation, a mechanism to store the thread's floating point unit state is missing and needs to be added. I expect this to cost 20–40 SLOC in both, the modified L4Re kernel and the checkpointing library.

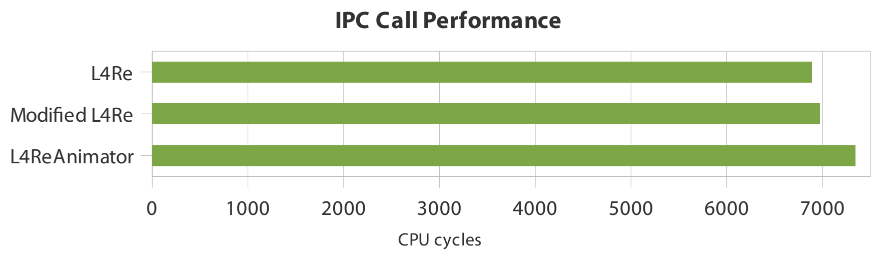


Figure 6.1: Duration of an IPC call in the L4Re, modified L4Re, and L4ReAnimator.

6.2 Performance Analysis

In this section I analyze L4ReAnimators performance. First, I analyzed the overhead introduced by the capability fault mechanism and then I investigated the costs of the checkpoint mechanism. Further, I examined if the mechanism used to reach link-time transparency has a negative impact on performance.

For my experiments I used a 2.8GHz Pentium 4 based computer equipped with 512 MB system memory. All benchmarks were done with no output during the measurements. After the measurements completed, the results were dumped to the serial console. The time was measured reading the CPU's time stamp counter.

In the rest of this section I describe the experiments I performed.

6.2.1 IPC Call Performance

As I described in the Sections 5.3.1 and 5.3.4, L4ReAnimator wraps IPC calls to achieve two things: First it checks for the success of any IPC operation and further the calling thread registers itself at the capability watcher thread, to be woken up if the callee disappears.

Wrapping the IPC operation is likely to have a negative consequence on the IPC performance. Due to the link time transparency mechanism, there should also be an observable performance decrease when L4ReAnimtor is not used.

In order to measure this overhead, I micro-benchmarked the IPC performance. I wrote a test program performing 1000 IPC calls to a server without payload. The clock cycles necessary for these 1000 calls were measured using the time stamp counter. This experiment was repeated 10 times and done for the unmodified L4Re and the modified version of L4Re with and without the L4ReAnimator libraries.

System	*Clock-cycles*	*Relative to unmodified L4Re*
L4Re	6 887	100 %
Modified L4Re w/o L4ReAnimator	6 972	101.23 %
L4ReAnimator	7 341	106.58 %
L4ReAnimator with server unmap	66 031	829,86%

Table 6.3: IPC call performance with and without L4ReAnimator.

Figure 6.1 and Table 6.3 show the results of this experiment. As one can see the overhead introduced by the weak linking mechanism is 1.23 %, which is negligible. With the L4ReAnimator framework libraries linked against the client library the costs are higher With 6.58 % overhead seems to be high, but these measurement were done with a micro-benchmark and in a real-world scenario this overhead would be much lower.

The result shown in the last column of table 6.3 is the result of a modification of this experiment. In that case the server unmaps the session capability of the client during each IPC.

The session capability will get unmapped, both at the loader and at the client. When the client invokes the session capability the next time, the capability fault mechanism recovers the capability. This results in a call to the loader, as the session capability was received through the name service. The loader has to reopen the session, after which it can reply the name query of the client. Then the client transparently retries the invocation of the session capability. In consequence there are five context switches instead of two, leading each time to a TLB-flush. Further the server has to recreate the IPC-gate representing the session. To get this IPC-gate the server has to make a call to his factory, which in this case is the kernel factory. This together explains the performance degradation in this scenario. However, in my opinion these costs are acceptable, as this should only happen in the failure case.

6.2.2 Allocating Capability Indices

In this experiment, I measured the cost of creating capability registry entries. As I mentioned in section 5.2.1, these entries are created whenever a capability index is allocated. This has only to be done, when a certain index is allocated the first time, as the capability entry will stay in the system when the capability index is freed. Therefore it is to expect, that when the same capability index is allocated again, the allocation should take shorter as the first time.

System	*Clock-cycles*	*Relative to unmodified L4Re*
L4Re	187	100 %
Modified L4Re w/o L4ReAnimator	188	100.55 %
L4ReAnimator	3 927	2 105,86 %
L4ReAnimator 2nd allocation	385	206,39 %

Table 6.4: Capability index allocation performance.

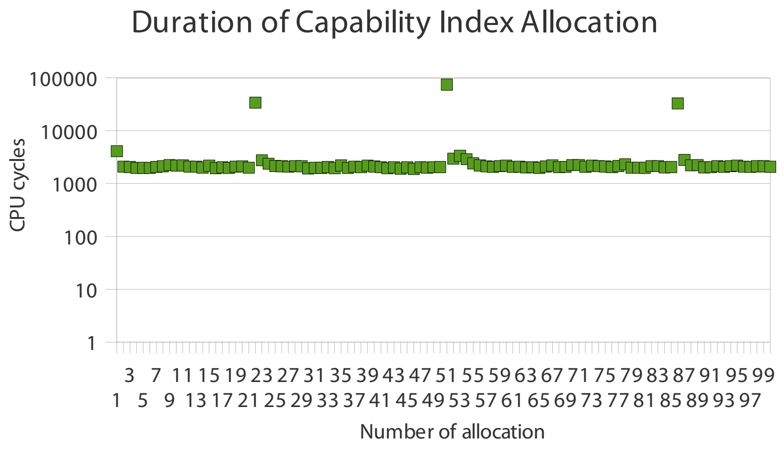


Figure 6.2: Duration of capability index allocation measured in L4ReAnimator for 100 Capability allocations.

I measured the allocation of 1000 capability indices for the unmodified L4Re, modified L4Re and for the modified L4Re together with the L4ReAnimator libraries. I measured the duration of each capability index allocation. The results of these measurements are shown in Table 6.4.

As we can see, the overhead introduced by the weak symbols is not noteworthy. Further the results show that, as predicted, the second allocation of a capability index lasts factor 10 shorter than the first one, when L4Reanimator is used.

However, with the L4ReAnimator framework, the first allocation of a free capability index takes ca. 20 times longer. The main reason for that is, that capability index allocation is such a simple process and adding any kind of complexity is striking. In a real world scenario, this overhead would be insignificant, as most application allocates capability indices only during their initialization.

Operation	*L4Re*	*Modified L4Re*	*L4ReAnimator*
1st attach	329.38	333.09	728.70
2nd attach	344.94	326.21	562.21
1st detach	1 069.20	1 094.17	1 491.18
2nd detach	1 026.41	1 087.57	1 478.04

Table 6.5: Time in microseconds needed for attaches and detaches of 100 dataspaces.

A noteworthy observation was that the performance data taken under L4ReAnimator contained some outliers, illustrated in Figure 6.2. The small spikes are caused by page faults which occur when a CR-entry is written. The larger one is caused by L4ReAnimator's allocator, allocating a new memory. The memory is allocated in 8KB chunks. In the version of L4ReAnimator used in this measurements, 144 bytes are needed for one CR-entry. So a page fault should occur every 28th capability index slot allocation, what the measurements confirm.

6.2.3 Attaching Dataspaces

As mentioned in section L4ReAnimator stores every mapping of a dataspace in order to be able to unmap it later, if the dataspace capability is lost. For this reason, a call to L4ReAnimator is necessary for each attach and detach operation. I determined the overhead introduced by these calls by measuring the time that is needed to attach and detach 100 dataspaces to and from the application's address space. I expected that the second attach operation of a dataspace would be faster than the first, so I repeated attaching the dataspaces a second time and measured the time needed for the second attach operation.

This experiment was done for the unmodified L4Re, and the modified L4Re without L4ReAnimator and with L4ReAnimator. The results of this experiment are shown in Table 6.5 and Figure 6.3.

The results again show that the weak linking mechanism used to achieve link-time transparency does not have any significant effect on the system performance. Further they show that L4ReAnimator leads to an performance degradation of the attach call by factor 2.2 compared to the unmodified L4Re when a dataspace is the first time attached. If a dataspace is the second time attached this performance degradation is reduced to factor of 1.5.

The reason for this overhead is that L4ReAnimator has to allocate a dataspace watch object, whenever the dataspace is attached the first time. Further

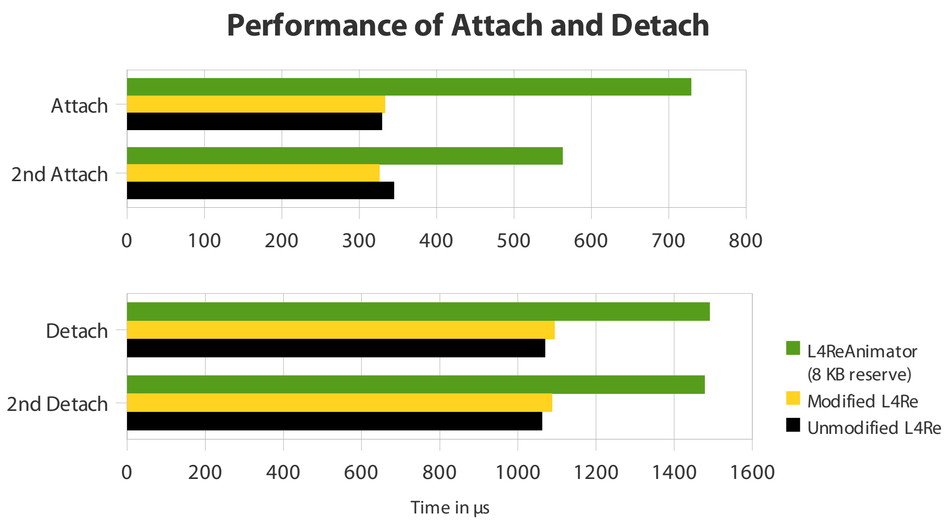


Figure 6.3: Time in microseconds needed for attaching and detaching 100 dataspaces.

an entry in the mapping list of the watch object is needed and also has to be allocated. When the dataspace is attached the second time the watch object already exists and only the mapping list entry has to be created.

Single Attaches

In order to evaluate the consequences of L4ReAnimator's memory allocator mechanism, described in Section 5.2.1, I also measured the time needed for each single attach operation. This experiment was done for the unmodified L4Re and for two versions of L4ReAnimator. In the first version L4ReAnimator's allocator enlarges its memory pool with chunks of 8 kilobytes and in the second with chunks of 4 kilobytes.

The result of this experiment is shown in figure 6.4. As one can see in the graph, the count of needed CPU cycles increases with each dataspace attached to the applications address space. The reason for this lies in the L4Re kernel, which has to add a region to its region map. The duration of this process is proportional to the number of regions in the region map.

Further it is observable, that the overhead introduced by L4ReAnimator is relatively small and stays constant over time, besides some outliers. For the version of L4ReAnimator that enlarges its memory pool with 4 kilobyte chunks, there are outliers after each 50th attach, which are caused by the enlargement of the memory pool. In the 8 kilobyte version there are also

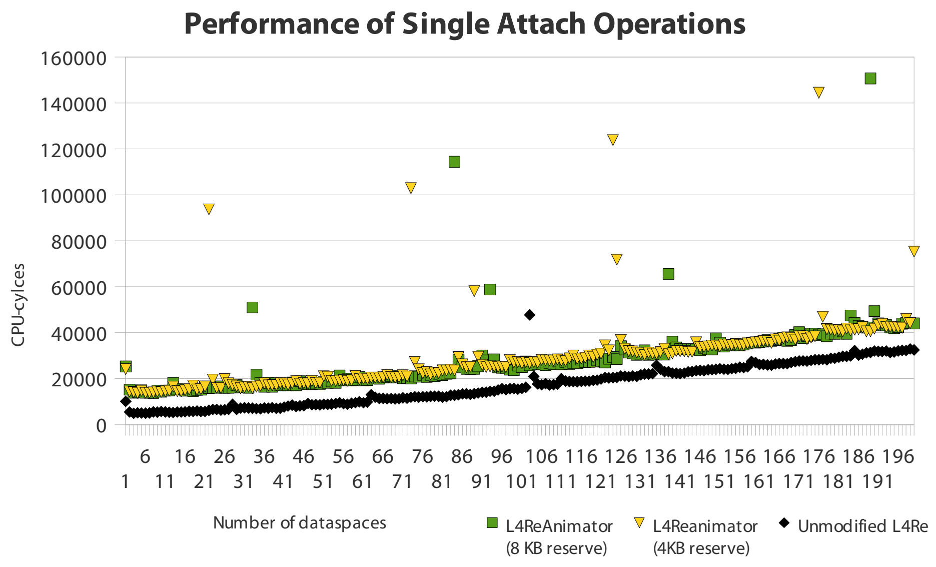


Figure 6.4: Time in CPU-cycles needed for single-dataspace attaches.

outliers at every 50th attach operation. But in this case a small maverick alternates with a larger one. The cause for the small outlier is a page fault in the application occurring when a new page of the memory pool is touched and the cause for the bigger one is the enlargement of the memory pool.

There is one outlier common to all three measurements between the 86th and the 106th attach operation, which is caused by a page fault int the L4Re kernel that occurs when the a new region is inserted in the region map.

6.2.4 Capability Watching

In this experiment measured the overhead introduced by the capability watching mechanism. I measured the time needed to do 50000 matrix multiplications for different watching intervals. After that, I increased the number of watched capabilities, by attaching 100 dataspaces and repeated the measurement. The reference measurements for the unmodified and modified L4Re can be seen in Table 6.6. As one can see the overhead of 0.01 % introduced by the modifications of L4Re is negligible . Also watching of seven capabilities each 20 ms does not have a noticeable negative consequences on the performance.

The results of the measurements using different watching intervals and a different count of watch objects are shown in Figure 6.5.

System	Time in ms	Relative to unmodified L4Re
L4Re	3 772.02	100.00 %
Modified L4Re w/o L4ReAnimator	3 772.43	100.01 %
L4ReAnimator (7 capabilities, 20ms)	3 773.18	100.03 %

Table 6.6: Duration of 50000 multiplications of a 200x200 matrix.

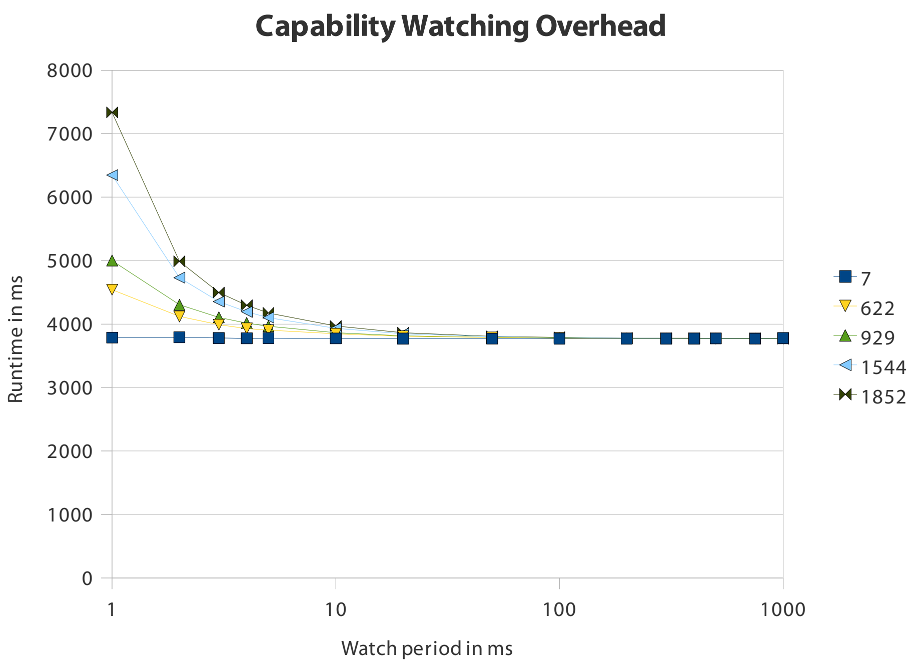


Figure 6.5: Consequences of different watching intervals on the performance of 50000 multiplications of a 200x200 matrix. Different lines represent different count of watched dataspaces.

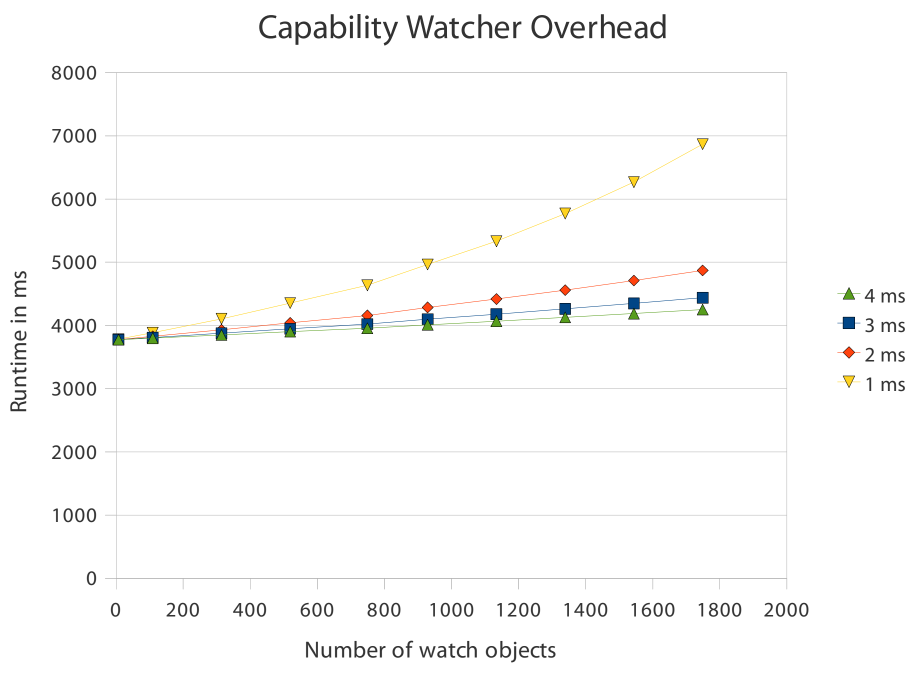


Figure 6.6: Overhead of capability watching with different number of watched capabilities. Different lines represent different watch intervals.

On the x-axis we can see the interval between the capability watching and on the y-axis the runtime of the benchmark in milliseconds. The overhead is relatively constant for intervals longer than 20 ms. For these intervals the overhead lies around 0 %–1 %. When the interval gets smaller the overhead increases exponentially. How much the overhead grows depends on the number of watched capabilities. For 1852 watched capabilities with an interval of one millisecond the overhead reaches nearly 100 %. In Figure 6.6 we can see the growth rate for the execution time of the benchmark against number of watched capabilities. The execution time grows nearly linear with the number of watched capabilities.

The results show that applications using L4ReAnimator should be designed carefully to use as few capabilities as possible. The less capabilities have to be watched, the higher the watch frequency can be. A kernel mechanism offering revoke notification is desirable. This would make it possible to watch large number of capabilities without a big performance loss.

6.2.5 Further Experiments and Checkpointing

I measured the overhead of taking checkpoints using two benchmarks. The first is again a matrix multiplication. This time one million multiplications of a 200x200 matrix were performed. The second is a C++ implementation of the n-queens problem, which I ran with n=13. I measured the execution time of this benchmarks for the modified L4Re and L4ReAnimator with and without checkpointing. In the checkpointing experiments I changed the checkpoint frequency. It can be expected that a lower checkpoint interval leads to a higher performance decrease.

n-Queens Problem

The results of this measurements for the 13-queens problem can be seen in table 6.7. It is noticeable, that the performance overhead of L4ReAnimtor is slightly higher than in the last experiment. The reason for this is that during the processing more than 8 megabytes of memory are allocated by the test program. As the memory is allocated page-wise, this results in 2000 dataspaces and each of this dataspaces has to be watched by the capability watcher. This introduces extra overhead, as described in the last section. However, with 1.66 %, the overhead is still in an acceptable range.

In the last column of Table 6.7, we can see the test results of the benchmark when checkpointing is used. As checkpointing relies on the capability

Checkpoint interval	*Modified L4Re*	*L4Reanimator w/o Checkpointing*		*Checkpointing*	
—	138.86	140.99	1.66 %	—	—
10 seconds	—	—	—	142.00	2.39 %
5 seconds	—	—	—	142.54	2.78 %
2 seconds	—	—	—	143.36	3.37 %

Table 6.7: Execution time of the 13-queens problem with L4ReAnimator and Checkpointing in seconds. The percent values represent the overhead relative to modified L4Re.

fault mechanism, the overhead of this mechanism is also included in the measurements.

As expected, the performance decreases when the checkpoint interval gets lower but the overhead is still in an acceptable range. The source of the performance decrease is on the one hand the communication between dsstorage and the applications L4Re kernel. For each stored dataspace one IPC operation is necessary. Second, as the dataspaces are stored in a copy on write fashion page faults will occur in the application when the memory is accessed the next time.

To increase the performance several optimizations are considerable. First, incremental checkpointing could be used, which means that only the modified dataspaces are copied. This would lead to less page faults after a checkpoint is taken. Second, the L4Re kernel could aggregate the dataspaces to be stored into one single dataspace, which then is sent to the dsstorage server.

Matrix Multiplication

The results for the matrix multiplication benchmark can be seen in Table 6.8. The overhead in this scenario is much smaller. For L4ReAnimator, the overhead is 0.06 %. This is a result of the low memory usage of this benchmark. In this benchmark only 7 dataspaces have to be watched.

When checkpointing is used the overhead is slightly higher (0.20–0.23%), but still one order of magnitude lower than in the n-queens benchmark. This can be attributed to the low memory usage of this benchmark: The benchmark allocates 960 kilobytes in chunks of 320 kilobytes. Hence, the heap of the application is not increased page wise and hence less dataspaces are used.

As the memory consumption of this benchmark is low, it was possible to further reduce the checkpoint interval. The overhead for further check-

Checkpoint interval	*Modified L4Re*	*L4Reanimator w/o check-pointing*		*Checkpointing*	
—	75.4	75.45	0.06 %	—	—
10 seconds	—	—	—	75.56	0.20 %
5 seconds	—	—	—	142.54	0.21 %
2.5 seconds	—	—	—	143.36	0.23 %

Table 6.8: Execution time of the matrix compilation benchmark done with the modified L4Re, L4ReAnimator and checkpointing in seconds. The percent values represent the overhead relative to modified L4Re.

pointing intervals can be seen in figure 6.7. In this figure we can see, that the overhead grows exponentially with smaller checkpointing interval. With all intervals the overhead is in an acceptable range.

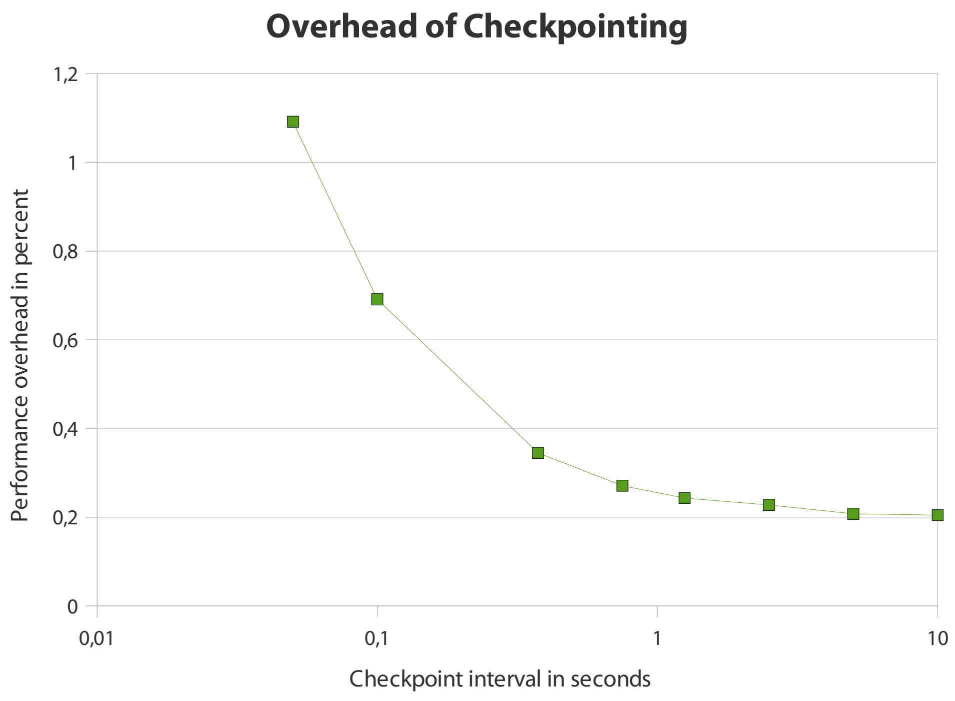


Figure 6.7: Overhead of checkpointing against checkpoint interval for one million multiplications of a 200x200 matrix.

7 Conclusion and Future Work

In my thesis I introduced L4ReAnimator, a restarting framework for L4Re. In the design phase of this work it appeared, that it is necessary to recover communication channels that were lost due to a server restart. For this purpose I designed a *capability fault mechanism* that can be implemented entirely in user-land, although kernel support could help to improve its performance.

Capabilities are recovered using capability fault handlers, which have to be implemented by service providers and are initialized in the server's client library. The capability fault mechanism is transparent for application developers just using these services. The recovery mechanism is also transparent for applications, as in case of an IPC failure the IPC is repeated by the L4ReAnimator framework. The recovery process is recursive and comparable to userlevel page fault handling in L4. Hence, I showed, that recursive resource management is also applicable for recovering capabilities. I implemented and evaluated a prototype of this mechanism. It turned out that the performance overhead introduced by this mechanism acceptable.

To store an application's state, I proposed an application-transparent user-level checkpointing mechanism for L4Re, also part of L4ReAnimator. This mechanism was prototypically implemented, but due to the limited amount of time not fully evaluated. However, first tests have shown that this mechanism is able to take a checkpoint in an application-transparent manner.

7.1 Future Work

In this section, I present work remains to be done and propose topics for further investigation.

Checking L4ReAnimators Real World Suitability

L4ReAnimator was tested in simple use cases such as restarts of the server providing virtual framebuffer consoles and other simple client–server scenarios. Although these tests were successful, it would be interesting to bring this mechanism to complexer scenarios, such as L4Linux [20], a userlevel port of the Linux kernel.

Kernel Support for Capability Watching

In its current implementation the capability watching mechanism works as described in Section 5.3.3, i.e. by periodically checking the capability indices of all watch objects. This periodic checking involves an overhead that can be minimized with a little assistance from the microkernel. In order to achieve this, the kernel has to send revoke notifications to the application. A revoke notification is a notification informing the application that a capability has been unmapped. After receiving such a notification the watcher thread may iterate over all watch objects for finding out which capability was unmapped and call this object's `action` method. Therewith, it is not necessary to spin over all watch objects all the time. A further optimization would be also to send index of the unmapped capability, which would make the iteration over all watch objects superfluous.

Checkpointing

The dsstorage server is currently the only way to store application data beyond application restarts and used to store checkpoints. It is desirable to store checkpoints on a real persistent storage, e.g. the hard disk. The implementation of a storage server that is able to store checkpoints to a file system is a considerable goal.

Further, it is interesting to investigate the performance gain of the dataspace aggregation described in Section 6.2.5. Incremental checkpointing is also a desirable feature, as it would reduce the memory consumption of a checkpoint.

For now, checkpointing does not store the state of the *Floating Point Unit* (FPU), consequently applications using the FPU are not checkpointable. This functionality has to be added to the checkpointing library and the L4Re kernel.

In Section 4.5.4, I proposed a mechanism to take checkpoints of whole subsystems, which would be a valuable extension of L4ReAnimator.

Further, Carlyle et al. [9] proposed to use checkpoints to accelerate the start-up of applications. If this technique is applied to whole subsystems an instant-on OS, like splashtop [4], can be created, offering limited functionality, such as web browsing. In contrast to existing solutions the normal boot process may go on in the background. This way the users don't have to wait until the operating system has fully booted for being productive.

Finally, a further performance analysis needs to be performed, including micro-benchmarks of the checkpoint and restore process.

Kernel Support for Checkpointing

In the current checkpointing implementation the thread state is gathered using the Fiasco.OC's exception mechanism. This mechanism offers a comfortable way to get the execution context of a thread, but has an serious disadvantage: Ongoing IPC operations have to be canceled every time a checkpoint is taken. This is no problem for idempotent function calls. For non-idempotent calls the server should be able to rollback its state to the state before the IPC call. This has to be done every time, a non-idempotent IPC call was interrupted by the the checkpointer thread.

For the same reason there is a problem with the l4_sleep call. This call sleeps for a certain amount of time by performing an IPC receive operation with timeout on an invalid capability. This IPC operation will also be canceled when a checkpoint is taken. Thus, this IPC call has to be wrapped in order to sleep again, if the sleep duration was not long enough.

In the long term a kernel mechanism for getting a thread's execution context in a transparent manner is desirable.

Further Ideas

L4ReAnimator only offers a simple fault detection mechanism: Application are restarted in case of abnormal termination. As described in Section 4.2.2, also other situations may require a restart. For this purpose a fault detection system is necessary.

Another considerable research topic is how checkpointing can help to debug programs. For example, it is supposable to use periodic checkpoints to restore an application to the state before an error occurred. The checkpoint may then be restored in a debugging environment like Valgrind [26]. This would reduce the overhead of the debugging process as the application has not to run in Valgrind all the time, but only in the failure case.

For now the CF-handlers have to be written by hand, but a mechanism for generating these handlers automatically is desirable. To generate the CF-handlers it might be useful to model the protocols used by the service. These models could be either created by hand or by analyzing the source code of server and client. This model could then be used to generate the CF-handlers automatically.

Bibliography

[1] Homepage of L4Ka::Pistachio. http://l4ka.org/projects/pistachio. 6

[2] Homepage of the Genode Operating System Framework. http://genode.org. 29

[3] Hompage of Open Kernel Lab's. http://wiki.ok-labs.com. 6

[4] How splashtop works. `http://www.splashtop.com/how_splashtop_works.php`. 74

[5] Accetta, M., Baron, R., Bolosky, W., Golub, D., Rashid, R., Tevanian, A., and Young, M. Mach: A New Kernel Foundation for UNIX Development, 1986. 6

[6] Aron, M., Liedtke, J., Elphinstone, K., Park, Y., Jaeger, T., and Deller, L. The sawmill framework for virtual memory diversity. In *ACSAC '01: Proceedings of the 6th Australasian conference on Computer systems architecture* (Washington, DC, USA, 2001), IEEE Computer Society, pp. 3–10. 8

[7] Avizienis, A., Laprie, J.-C., and Randell, B. Dependability and its threats - A taxonomy. In *IFIP Congress Topical Sessions* (2004), pp. 91–120. 3, 4

[8] Campbell, R. H., and mong Tan, S. Choices: An Object-Oriented Multimedia Operating System. In *In Fifth Workshop on Hot Topics in Operating Systems, Orcas Island* (1995), IEEE Computer Society, pp. 90–94. 19

[9] Carlyle, J. C., M., D. F., and H., C. R. Back in a Flash! - Fast Recovery using Non-Volatile Memory. In *37th IEEE/IFIP International Conference on Dependable Systems and Networks* (June 2007), pp. 422–423. 37, 74

[10] Chen, P. M., Ng, W. T., Chandra, S., Aycock, C., Rajamani, G., and Lowell, D. The Rio file cache: surviving operating system crashes. In *ASPLOS-VII: Proceedings of the seventh international conference on Architectural support for programming languages and operating systems* (New York, NY, USA, 1996), ACM, pp. 74–83. 13

[11] David, F. M., and Campbell, R. H. Building a Self-Healing Operating System. In *DASC '07: Proceedings of the Third IEEE International Symposium on Dependable, Autonomic and Secure Computing* (Washington, DC, USA, 2007), IEEE Computer Society, pp. 3–10. 5

[12] David, F. M., Chan, E., Carlyle, J. C., and Campbell, R. H. CuriOS: Improving Reliability through Operating System Structure. In *OSDI* (2008), R. Draves and R. van Renesse, Eds., USENIX Association, pp. 59–72. 5, 19

[13] Duell, J. The design and implementation of Berkeley Lab's linux Checkpoint/Restart. Tech. rep., 2003. 5

[14] Engler, D., Chelf, B., and Chou, A. Checking system rules using system-specific, programmer-written compiler extensions. pp. 1–16. 12

[15] Fakultät Informatik, Technische Berichte, Hohmuth, M., and Hohmuth, M. The Fiasco Kernel: Requirements Definition. Tech. rep., 1998. 6

[16] Freescale Semiconductors. *i.MX31 Multimedia Applications Processor.*, 2.4 ed., 12 2008. 5

[17] Gao, Q., Zhang, W., Tang, Y., and Qin, F. First-aid: surviving and preventing memory management bugs during production runs. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*. ACM, New York, NY, USA, 2009, pp. 159–172. 37

[18] Gray, J. Why Do Computers Stop and What Can Be Done About It?, 1986. 3, 4, 5

[19] Herder, J. N., Bos, H., Gras, B., Homburg, P., and Tanenbaum, A. S. MINIX 3: a highly reliable, self-repairing operating system. *SIGOPS Oper. Syst. Rev. 40*, 3 (2006), 80–89. 6, 16

[20] Härtig, H., Hohmuth, M., and Wolter, J. Taming linux. In *In Proceedings of the 5th Annual Australasian Conference on Parallel And Real-Time Systems (PART '98* (1998). 74

[21] Ishikawa, H., and Nakajima, T. Micro-Reboot Support for Multi-Server Operating Systems. In *The 1st International Workshop on Dependable Ubiquitous Nodes* (2007). 5

[22] Klein, G., Elphinstone, K., Heiser, G., Andronick, J., Cock, D., Derrin, P., Elkaduwe, D., Engelhardt, K., Kolanski, R., Norrish, M., Sewell, T., Tuch, H., and Winwood, S. seL4: Formal verification of an OS kernel. In *SOSP 2009: Proceedings of the 22nd ACM Symposium on Operating Systems Principles* (Big Sky, Montana, October 2009), ACM. 1, 6

[23] Laadan, O., and Nieh, J. Transparent checkpoint-restart of multiple processes on commodity operating systems. In *ATC'07: 2007 USENIX Annual Technical Conference on Proceedings of the USENIX Annual Technical Conference* (Berkeley, CA, USA, 2007), USENIX Association, pp. 1–14. 38

[24] Lackorzynski, A., and Warg, A. Taming subsystems: capabilities as universal resource access control in L4. In *IIES '09: Proceedings of the Second Workshop on Isolation and Integration in Embedded Systems* (New York, NY, USA, 2009), ACM, pp. 25–30. 6

[25] Li, Z., Tan, L., Wang, X., Lu, S., Zhou, Y., and Zhai, C. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *ASID '06: Proceedings of the 1st workshop on Architectural and system support for improving software dependability* (New York, NY, USA, 2006), ACM, pp. 25–33. 1

[26] Nethercote, N., and Seward, J. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *PLDI '07: Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation* (New York, NY, USA, 2007), ACM, pp. 89–100. 75

[27] Plank, J. S. An overview of checkpointing in uniprocessor and distributed systems, focusing on implementation and performance. Tech. Rep. CS-97-372, University of Tennessee, July 1997. Also published as "Program Diagnostics", to appear in the Encyclopedia of Electrical and

Electronics Engineering, John G. Webster, editor, published by John Wiley & Sons, Inc. 37

[28] Plank, J. S., Beck, M., Kingsley, G., and Li, K. **Libckpt**: Transparent checkpointing under Unix. In *Usenix Winter Technical Conference* (January 1995), pp. 213–223. 5, 38

[29] Schroeder, B., Pinheiro, E., and Weber, W. D. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems* (New York, NY, USA, 2009), ACM, pp. 193–204. 1

[30] Shapiro, J. S., and Adams, J. Design Evolution of the EROS Single-Level Store. In *ATEC '02: Proceedings of the General Track of the annual conference on USENIX Annual Technical Conference* (Berkeley, CA, USA, 2002), USENIX Association, pp. 59–72. 37

[31] Skoglund, E., Ceelen, C., and Liedtke, J. Transparent Orthogonal Checkpointing through User-Level Pagers. In *POS-9: Revised Papers from the 9th International Workshop on Persistent Object Systems* (London, UK, 2001), Springer-Verlag, pp. 201–214. 38

[32] Swift, M. M., Annamalai, M., Bershad, B. N., and Levy, H. M. Recovering device drivers. *ACM Trans. Comput. Syst. 24*, 4 (2006), 333–360. 11

[33] Swift, M. M., Bershad, B. N., and Levy, H. M. Improving the reliability of commodity operating systems, 2003. 11

[34] Swift, M. M., Martin, S., Levy, H. M., and Eggers, S. J. Nooks: an architecture for reliable device drivers. In *EW10: Proceedings of the 10th workshop on ACM SIGOPS European workshop* (New York, NY, USA, 2002), ACM, pp. 102–107. 4

[35] Tanenbaum, A. S., Herder, J. N., and Bos, H. Can We Make Operating Systems Reliable and Secure? *Computer 39*, 5 (2006), 44–51. 16

[36] Toward, J. L., and Liedtke, J. Toward Real Microkernels. *Communications of the ACM 39* (1996). 6

[37] Veerman, T. Dynamic Updates and Failure Resilience for the Minix File Server. Master's thesis, Vrije Universiteit Amsterdam, May 2009. 19