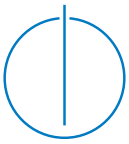
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Design and Development of real-time capable Checkpoint/Restore Mechanisms for L4 Fiasco.OC/Genode

Denis Huber



DEPARTMENT OF INFORMATICS

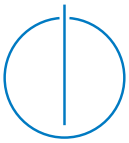
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Design and Development of real-time capable Checkpoint/Restore Mechanisms for L4 Fiasco.OC/Genode

Konzeption und Implementierung von echtzeitfähigen Checkpoint/Restore Mechanismen auf Basis von L4 Fiasco.OC/Genode

Author:	Denis Huber
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Sebastian Eckl, M.Sc., Daniel Krefft, M.Sc.
Submission Date:	15. December 2016



I confirm that this master's thesis in informatics is my own work and I have documented all sources and material used.

Munich, 15. December 2016 Denis Huber

Abstract

In the automotive domain the Electronic Control Unit (ECU) count increased to a magnitude of hundred devices. The high number results from a high number of functions provided as well as from a redundant ECU model used for fault tolerance. This practice is limited by the effort to reduce the weight and energy consumption and reaches its maximal feasibility in the near future.

A trend coming from the desktop/server domain intends to counter this problem. The idea is to replace many, low-performance devices by few, high-performance devices. Together with hardware consolidation and virtualization the high safety and mixed-criticality requirements of the automotive domain are satisfied.

This hardware upgrade enables the utilization of another technique coming from the desktop/server environment which guarantees fault tolerance: Checkpointing and restoring. Unfortunately, there is only little active research for Checkpoint/Restore algorithms suitable for an embedded and real-time system.

Taking a step in this direction, this thesis focuses on transferring state-of-the-art algorithms to the embedded real-time domain. By analyzing the work of other contributors the thesis demonstrates the lack of embedded real-time algorithms. Therefore, promising algorithms were evaluated by considering embedded and real-time requirements. The result is a real-time capable Checkpoint/Restore mechanism which implements a shared resources approach to access the state of a target application and an incremental checkpointing mechanism to optimize the checkpointing of memory.

Contents

1. Introduction

Checkpoint/Restore mechanisms provide fault tolerance to operating systems and processes where failures imply high operating costs. For example, Libckpt [1] can restart a long running process with a previously checkpointed state after a system crash occurred. Thus, Libckpt saves time which would be necessary to recompute the results until the crash-time.

Checkpointing and restoring are activities to access and store the state of a target application and to restore it from a stored state, respectively. Its main application domains are desktop and server systems, especially virtual machines that natively implement Checkpoint/Restore functionality, and high performance computing where uncaught failures imply wasting of operation time. Among others, it is used for fault tolerance by periodically checkpointing a target application and for load balancing by checkpointing, migrating, and restarting a target application on another node. Unfortunately, the Checkpoint/Restore research has yet to reach the safety-critical, embedded real-time domain that demands high stability and fault tolerance from the system. This thesis will elaborate on the topic of bringing Checkpoint/Restore to the embedded real-time domain.

1.1. KIA4SM Overview

This work is embedded in the KIA4SM (Cooperative Integration Architecture for Future Smart Mobility Solutions) project [2]. This project unifies the cooperation of Intelligent Transportation Systems (ITS), Smart Mobility devices, and vehicles under a common run-time environment. The environment uses universally applicable Electronic Control Units (ECUs) and utilizes hardware consolidation to dynamically distribute tasks – specific to the before mentioned participants – on the available ECUs. It provides virtualization techniques to isolate the tasks from each other to guarantee strong isolation, especially for vehicular systems. The project focuses on flexibility by allowing dynamic reconfiguration of a task distribution during the run-time of the system. The task distribution is based on an offline activity, which uses organic computing paradigms to work out possible configurations for specific scenarios, and on an online activity, which evaluates the current scenario and changes the configuration based on the offline data.

1.2. Motivation and Problem Statements

The current practice for fault tolerance of car manufacturers is to provide an identical, but inactive ECU for each ECU responsible for a critical task [3] (e.g. brakes, airbag). It is also common to realize one such task per ECU. Hence, integrating a new functionality into the vehicle basically requires double the amount of ECUs. The KIA4SM project avoids this approach by using universally applicable ECUs which host several tasks isolated by virtualization.

To provide fault tolerance a Checkpoint/Restore design, lent from the desktop and server domain, seems to bring promising results. It replaces redundancy of hardware by redundancy of memory through copying the state of an application to another memory region. The long term goal of this thesis is to replace static redundancy of ECUs by dynamic checkpoints of software components. The idea of this thesis is to transfer the Checkpoint/Restore methodology from the desktop/server domain to the embedded real-time world.

Therefore three challenges, formulated in form of problem statements and questions, arise:

- Are there any state-of-the-art approaches targeting embedded real-time systems and do they satisfy the strict requirements of the vehicle industry?
- What requirements are imposed by an embedded and real-time system to a Checkpoint/Restore mechanism?
- Are there suitable candidates of Checkpoint/Restore algorithms which can be transferred to the embedded real-time domain?

1.3. Overview

This thesis gives answers to the above mentioned questions. In chapter 2 the related work of Checkpoint/Restore contributors is presented. It points out the lack of real-time capable Checkpoint/Restore mechanism in the overall research. In chapters 3 and 4 the requirements for a Checkpoint/Restore mechanism are elaborated in regards to real-time systems and microkernel concepts. Chapter 5 describes the programming environment based on Genode and Fiasco.OC. The design chapter (chapter 6) elaborates on transferring suitable Checkpoint/Restore mechanisms by considering embedded and real-time requirements from the before mentioned chapters. Chapter 7 illustrates the implementation of a shared resources approach and an incremental checkpointing mechanism. Finally, chapter 8 concludes the thesis with a showcase, shows up limitations of the design, mentions further work, and summarizes the work.

2. Related Work

Checkpoint/Restore is an activity which allows to save a process and restore it at another time and/or on another location. Thereby the memory, thread data, and kernel state is stored and restored with the goal of running the process as if nothing happened. The peak stage of Checkpoint/Restore research was between 1987 and 1997 [1, 4–13] when distributed operating systems used this concept for migrating processes from one node to another. Specialized operating systems that implemented the concept in their kernels were used. Nowadays the trend goes to user-level approaches implemented on general purpose operating systems like Linux. The approaches are used in various use cases like fault tolerance, process migration, memory rejuvenation, etc. Furthermore, there are only a few papers which scratch the topic of real-time checkpointing. The following sections present checkpointing techniques and current work in this field. For further reading the survey of Milojičić et al. about process migration [14] and two survey's from Sancho et al. and Maloney and Goscinski about Checkpoint/Restore [15, 16] are recommended.

2.1. Kernel-level Checkpoint/Restore

Kernel-level Checkpoint/Restore approaches modify the underlying operating system to enable the access to kernel structures from the user space. They can simply access the memory regions, CPU registers and kernel state (management data structures, open files, identifier) of a process. The down side of a kernel-level implementation is the portability from one operating system to another. The interior structures from a system can vary significantly and small kernel updates can break the whole Checkpoint/Restore mechanism.

2.1.1. BLCR

A popular representative is the Berkeley Lab's Linux Checkpoint/Restart (BLCR) [17] project which introduces a kernel-level Checkpoint/Restore approach. It has three main application areas:

- Gang scheduling: Groups of applications are scheduled to run at specific times. For example, long running or computation heavy jobs are designated to run at

night. Therefore these applications are checkpointed when their designated time slot ends and restarted when it starts again.

- Process migration: An application is checkpointed on one node and restarted at another node. It is useful, if an imminent failure is about to occur, e.g. CPU fan does not work anymore.
- Periodic backup: The applications are periodically checkpointed. If the application crashes, it can be restarted again without losing interim results.

The project provides a kernel module and a library which has to be linked with the target application. A target application consists of several processes which in return consist of several threads. The checkpoint is initiated by sending signals to the threads of the target application to freeze them and force them to store their states (memory, shared memory, open files, PIDs, registers and pending signals) to disk. During the whole checkpoint the threads cannot execute their normal functionality. Because BLCR does not checkpoint network states, it provides user-level callback functions for the target application. A callback is called before the checkpoint to quiesce the network or after a restart to recreate the network state. The restart is initiated by a process which creates the target application's processes and their threads. Again, the target application's threads are forced to restore their states.

The BLCR is a sophisticated implementation of a Checkpoint/Restore approach in the desktop and cluster environment. Because it is implemented at kernel-level, it is inherently efficient by accessing kernel state directly at checkpoint time. BLCR is not meant to run in real-time critical application areas. It does not optimize the checkpoint time or the checkpoint overhead opposed on the application's threads. These threads are not working on their task, but are stopped and forced to store their states. It is also not transparent to the application, because it has to bring itself to a checkpointable state through the use of callback functions in the target application.

2.1.2. Zap

Zap [18] is another kernel-level Checkpoint/Restore approach which introduces a virtualization layer between the operating system and the target application. It addresses the problem of consistent resource names, resource conflicts after migration, and resource dependencies in Linux. The virtualization layer encapsulates a group of processes in a so called pod (PrOcess Domain). A pod has two tasks:

- Provision of virtual, consistent names: A process inside a pod receives pod-unique IDs. To migrate a process, the whole pod has to be migrated.

- Masking out resources not contained in the pod: Processes inside a pod are not allowed to communicate over IPC with processes in other pods or processes without pods. But, they are allowed to communicated between each other and also to special processes outside the pod. To communicate to outside processes they use network communication or shared files.

Both simple mechanisms provide the target application to have consistent names, and a conflict free, resource independent migration. They are realized by providing the same API to the target application as the operating system. The so-intercepted function calls are used to track the state of the kernel. The state includes process' IDs and IPC state, memory state, file system state (e.g. open files), I/O device state, and network state.

This Checkpoint/Restore mechanism is only used in conjunction with migration. Therefore, the application's threads are frozen before checkpointing. Then, Zap saves the virtualization mappings, the memory, CPU registers, and other kernel states and transfers it to the new node, where the whole pod is recreated. Janakiraman et al. [19] extends Zap in Cruz and introduces a coordination protocol for creating consistent checkpoints for distributed applications.

The Zap project focuses on virtualization of consistent resource naming, and decoupling of resources. Thus, this Checkpoint/Restore mechanism is fairly simple and does not provide any real-time capable approaches.

2.1.3. Proteus

Groesbrink [20] presents a real-time aware concept to migrate virtual machines (VMs) between ECUs in his PhD thesis and implements it in Proteus, a bare-metal hypervisor designed for his application domain. His application domain is an automotive-like environment with several ECUs which communicate via a network. Each ECU runs a hypervisor for hosting VMs. In turn, each VM hosts a dedicated task. The target for migration are single VMs, which are transferred to other ECUs.

The migration is triggered by a partial hardware fault on an ECU. Such faults still allow the storing and transmitting of the VMs. The hypervisor supports a real-time aware migration, which guarantees that no VM involved in the migration misses its deadlines. This concept is implemented in the migration test which is performed during a migration and consists of an acceptance and a downtime test. The former test checks whether the target machine has enough resources for accepting the VM and whether it can produce a feasible scheduling[1]. The latter computes the downtime of the VM during the whole migration process to determine, if the VM can be migrated without missing its deadline. When the migration test succeeds the VM is suspended by the hypervisor,

[1] A feasible scheduling is a scheduling which does not miss any deadlines.

checkpointed, transferred to the destination node, restored and finally rescheduled by the destination hypervisor.

Groesbrink's real-time migration concept is based on the implementation of Proteus' VM scheduler. This scheduler runs a Rate Monotonic algorithm [21] and checks, whether the deadline requirements of the scheduled VMs are satisfied. He does not focus on an efficient implementation of a Checkpoint/Restore mechanism which gives away a lot of potential. Although the usage of VMs facilitates the implementation of a Checkpoint/Restore mechanism, it also introduces overhead by the need to checkpoint the complete VM. Groesbrink's work is a solid foundation for further work, where the migration could be optimized to allow higher utilization of used hardware.

2.1.4. Speculative Memory Checkpointing

Speculative Memory Checkpoint (SMC) [22] is a high-frequent checkpointer for Linux presented by Vogt et al. It provides a kernel-level module, which exports kernel data to the user-level. This design circumvents high overhead cost by managing memory accesses and the MMU bits. It also provides a user-level library which is linked against the target application, thus, it checkpointer threads have access to the same environment as the target threads. The checkpointer threads and the target threads form one process.

SMC has the goal to optimize the incremental checkpointing mechanism (see section 2.2.1) which only checkpoints changed memory pages. The problem of an incremental checkpointing mechanism is the overhead induced by the monitoring system, which uses page faults, hardware, or software dirty bits. The monitoring system observes the memory access during the runtime of the target threads. Vogt's idea to reduce the costs is to introduce a writing working set model [23] which divides the memory into hot and cold pages. Hot pages are accessed frequently, while cold pages are accessed seldom. His proposition is to monitor only the cold pages while checkpointing the hot pages at each checkpoint.

Vogt has to cope with the challenge to efficiently find out the hot pages per target's run. Because marking pages during target's runtime has the same costs as marking pages for the incremental checkpointing mechanism, Vogt introduces a blackbox optimization technique from genetic computing [24]. The blackbox approach uses the information from the memory marking of the incremental checkpointing mechanism to collect a set of hot pages during the runtime of the target. After having several sets, these sets are evolved by combining two parent sets to a child set. The strategy is to pick those parents which were the most successful from the their generation.

The advantage of SMC is that the writing working set optimization reduces the overhead induced by the incremental checkpointing mechanism during the runtime of the target. Its genetic approach supports a high-frequent checkpoint interval of 2 seconds

(tested). Unfortunately, Vogt does not name a special use case where his approach is best suited. He also does not concern real-time capabilities, and the checkpointing algorithm uses kernel-modules to reduce memory and MMU bits management costs to further reduce the overhead imposed on the target.

2.2. User-level Checkpoint/Restore

A user-level approach uses only the mechanisms provided by the operating system. A Checkpoint/Restore mechanism in user-space is possible, if the operating system provides enough information to restore a process. Often the process has to be "well-behaved" in order to be checkpointable, meaning it cannot use all tools the operating system offers. For example, a process shall not be aware of its process ID. If the process is aware of its process ID, the restoration on another node has to also restore the same process ID to guarantee the correctness of the process. If this process ID is already in use, then the restoration has to wait or simple be canceled. The advantage of a user-level approach is the portability to other operating systems with the same interface (e.g. UNIX derivatives) and also a better support for updating the operating system.

2.2.1. Libckpt

Libckpt [6] is a user-level Checkpoint library for Unix, which claims to be real-time capable. The library is meant for periodic snapshots of a long running, single-process application without network activity. If the application crashes, it can be restored from a previous snapshot, thus, a large amount of invested time is saved. The library has to be linked against the target application, thus, the checkpointing mechanism runs in the context of the target application. Consequently, the checkpointing mechanism has direct access to memory, threads, and system call invocation by intercepting them in order to track the kernel state.

The authors Li et al. claim that their checkpointing mechanism is real-time capable. After the checkpoint is initiated, the library thread freezes all threads, saves the state of the threads, marks all memory pages as read-only, and unfreezes the threads. Now, the library spawns a copier thread, which copies the memory to a secondary memory location concurrently to the threads of the target application. After copying a memory page, it is set to read-write. Whenever a thread of the target application wants to write to a read-only memory page, a page-fault is triggered. The page-fault handler handles the fault by copying the memory page and setting it to read-write. Also, the thread is restarted with the state at the time of unfreezing the thread.

This mechanism is called copy-on-write. In the successor papers [1, 7] this mechanism was realized with Unix's fork(). With fork the copy-on-write mechanism is already

implemented in the operating system and does not require a restart of a thread when a page fault occurs. Instead of a restart, a new process is created which receives a copy of the faulting memory page for copying, while the original memory page is set to read-write causing the thread of the target application to continue its work.

The real-time aspect of the mechanism lies in imposing little downtime to the target application's threads, because the copier thread and application's threads are allowed to run concurrently. In desktop environments with a round robin scheduling policy it is sufficient to run the applications concurrently to simulate real-time behavior. But the concurrent execution does not lower the total checkpoint time. In embedded systems with a priority based scheduling policy, where an application is meant to run until it finishes its work, the total checkpoint time is more important than a concurrent execution.

In its final version Libckpt [1] also implements an incremental checkpointing mechanism, which stores only changed memory pages in comparison to the last checkpoint. This mechanism has the potential to reduce the total checkpoint duration. Another optimization is the compression of the checkpointed data, which can reduce the file system usage or the transfer duration to another node. The latest version also introduces user directed checkpointing, where the user tells the application when to use checkpoint and what memory pages shall be checkpointed. This feature can greatly increase the checkpoint performance, when large memory pages are excluded from checkpoint, because their content is not needed anymore. But it also breaks the transparency to the application, thus, the programmer has to deal with checkpoint logic.

2.2.2. CRIU

The CRIU project [25] stands for Checkpoint/Restore In Userspace and is implemented for Linux. Primarily, CRIU was meant to be used to migrate an application to another node. Now, it is also used to speed up slowly booting application, by checkpointing them after a long initialization phase and just restarting them from the checkpoint. CRIU also provides fault-tolerance by periodically checkpointing an application and restoring it after a crash.

CRIU is a standalone application which does not need the linking to another application. It retrieves its information of a process or a process tree through the usage of the /proc file system, where a large amount of process metadata is stored. For a checkpoint CRIU stops the threads of the application. Now it gathers information about the processes and their child processes, the used memory regions, threads, and file descriptors. To save the memory regions of the application's processes it injects its own code to the main memory of each process and sets the instruction pointer of one of the process's threads to the new code. This thread is resumed and executes the new code.

This "parasite code" works in the context of the process, thus, it has access to the main memory protected by the MMU and stores the content of the process to a file. Finally, this code is removed from the main memory, the original code and instruction pointer are restored and the threads are resumed.

The restore mechanism runs in an own process which is designated to morph to the target application's main process. It recreates all child processes and their memory pages from within the context of each child process by injecting parasite code similar to the checkpointing code. After the state of the processes is recreated, the code of the restore mechanism is deleted from each process.

This hacking-style mechanism is used to access the target application's memory regions. It is transparent to the target, but it requires Linux to enable the parasite code injection by the ptrace system call. Allowing such a mechanism in an embedded and Internet-accessible system is a huge security issue, which is not applicable to the KIA4SM project.

2.2.3. Checkpoint/Restore in KIA4SM

There is also research available regarding Checkpoint/Restore mechanisms in the KIA4SM project [2]. David Werner was working on porting Checkpoint/Restore concepts from CRIU [25] to the Genode OS Framework in his Bachelor's Thesis [26]. He focused on accessing the target application's process' resources by intercepting its environment through a parent process. The target process is the child of the Checkpoint/Restore process and receives resources which are controlled by the parent. Consequently, the parent keeps track of the resource-activity of the child and is able to restore the resources at a later time. As an optimization, his checkpointing mechanism implements copy-on-write mechanism which allows to run the Checkpoint process concurrently to the target process.

David Werner's Checkpoint/Restore mechanism does not claim to be real-time capable. In fact, his effort was to test currently available Checkpoint/Restore mechanisms for their feasibility in Genode OS. Unfortunately, he could not finish his implementation due to unforeseen difficulties with the underlying kernel Fiasco.OC.

2.2.4. PointStart

In his diploma thesis [27] Kulik presents a Checkpoint/Restart mechanism called PointStart. It runs on Fiasco.OC/L4Re and provides a restart mechanism for L4Re-server-applications which are processes providing services for other processes. The checkpoints are taken periodically. When a failure occurs, which leads to the crash of the target application, the target application is restarted with the state of a previous

checkpoint.

A checkpoint consists of three parts: Storing memory state, thread state, and target process intern capability list[2]. Although the mechanism runs in user-space, the kernel had to be extended to support reading and writing from and to the capability list and the thread state. The thread state consists of the CPU and FPU registers, as well as of IPC and scheduler state. The memory is retrieved from a L4Re-native, user-level region manager named Moe. The checkpointer asks for copies of the memory chunks attached to the target process from Moe. Moe returns capabilities to these memory chunks, but does not perform a copy. As long as the original chunk or the virtual, copied chunk are not modified, they do not need to be copied. As soon as one of them is tried to be modified, the virtual, copied chunk is created.

This implementation of Checkpoint/Restore resembles an optimization mechanism in Libckpt [1] discussed earlier. It also lacks real-time capabilities with priority-based schedulers. Still, his thesis is worthwhile reading, because it describes Fiasco.OC's internal mechanisms. These information are hard to get, because the system misses an in-depth documentation.

[2]Capabilities are access rights to kernel objects.

3. Real-time Systems

The introduced Checkpoint/Restore mechanisms from chapter 2 do not focus on real-time constraints. Their primary target platforms are desktop or cluster environments to provide fault tolerance or load balancing. Although Groesbrink [20] and Li et al. [6] introduce real-time concepts, they also introduce sever constraints for them. The former uses VMs to encapsulate tasks. The checkpoint cost of the VMs is high regarding memory and CPU usage. The latter introduces a Checkpoint/Restore mechanism which reduces the time the target application is halted by the checkpointer by allowing the target application's threads to run concurrently with the checkpoint thread. This mechanism does not affect the overall checkpoint duration, which is more important in embedded real-time systems. The following sections will introduce the term real-time system and also the requirements for a real-time capable Checkpoint/Restore mechanism.

3.1. Basics

A real-time system is a system which has a timing constraint while fulfilling its native functionalities [28]. Such a task has a deadline to finish its work. There are two types of real-time constraints: *hard* and *soft*. If a task exceeds its deadline in a hard real-time system, the results are no longer useful and the system is considered faulty. In contrast, if a task exceeds its deadline in a soft real-time system, the results are degraded in value, but they can be still used.

The fulfillment of real-time constraints is the task of the operating system. An operating system aware of these constraints is called real-time operating system (RTOS). It has to provide usual operating system mechanisms like threads, memory access, hardware interrupts, interprocess communication etc. but with guaranteed response times.

Furthermore, it has also to provide a scheduler which is aware of real-time constraints. In an embedded real-time system the scheduling algorithm is preemptable and priority/deadline-based. Each task has a priority and the task with the highest priority is executed. Consequently, the tasks have be interruptible to allow a higher-priority task to be executed. The main goal of a real-time scheduler is to produce a *feasible scheduling*. It has to arrange the execution of tasks in a way, that no task will miss its deadline.

Therefore, it considers task's start time, deadline, execution time, and period after which duration the task is started again.

3.2. Deterministic Memory Protection

There is one hardware device which fell into disgrace in embedded real-time systems: The memory management unit (MMU). It is considered in-deterministic, when it is used for swapping/paging of memory pages to secondary storage [29]. But the MMU can be used for memory protection which forbids the direct access of a process' memory pages from another process.

For example, a process A has allocated a memory page at address $addr_A$. Another process B, which has not allocated a memory page at $addr_A$, wants to access the memory and tries to access it by dereferencing $addr_A$ in its address space. The CPU finds the address in the instruction and asks the MMU for the content. The MMU looks up the address in its translation lookaside buffer (TLB) and finds no mapping of process B's $addr_A$ and a cached memory page. It queries the page table by a page table walk and finds no mapping of process B's $addr_A$ and a physical memory page. It returns an error code to the CPU which triggers a page fault exception and sends it to the microkernel or OS. The page fault has to be answered by a page fault handler thread. Which has to be called deterministically by the microkernel/OS. The page fault can be resolved by allocating memory to process B or put the process into a fault state, thus, excluding it from execution.

The memory protection is deterministic, if the TLB walk, page table walk, page fault emission, and handling is deterministic. A naive approach could search the TLB and page table in a linear way, instantly emitting a page fault exception after the MMU error, and handling the page fault by a thread which inherits the priority of the faulting process. The memory protection provides a clear memory layout of a process which is independent of other processes. Furthermore, it provides protection against unwanted memory access from one task to another, thus, increasing system stability by avoiding bugs and malevolent intrusion.

3.3. Requirements for Checkpoint/Restore

The last section described, that the responsibility for a real-time system lies in the RTOS. But there are still real-time requirements for a task to support the RTOS. A real-time task has to fulfill the following requirements [30]:

- Reliability: The task shall be free from logical mistakes.

- Predictability: The task shall be free from timing failures. An algorithm shall be able to compute its guaranteed worst execution time.
- Performance: The task shall be implemented in an efficient way. This is also an important requirement for a soft real-time systems.
- Compactness: The task shall contain only the necessary components in order to fulfill its goal.
- Scalability: The task shall support a growing number of work. For example, a Checkpoint/Restore mechanism shall support a growing number of target applications efficiently.

The priority of real-time requirements regarding Checkpoint/Restore lies in performance and scalability. The higher the performance the lower the impact of the mechanism on the real-time system. The lower the impact the more tasks can be handled by the system. The same consequence is put on scalability. The higher the scalability the more tasks can be handled efficiently.

3.4. Checkpoint/Restore in Conjunction with Migration

Although an RTOS can produce a feasible scheduling for its own system, a migration involves at least two systems. Thus, the migration has to be aware of the deadlines, execution, and idle times resources of the destination system (compare [20]) and the migrating application. When performing a migration the task has to find a suitable destination which has enough time resources to produce a feasible scheduling with the target task, which is migrated. Also, the serialization, transportation and deserialization times has to be considered for the target task to not miss its deadline. The Checkpoint/Restore mechanism shall be implemented to allow the integration of a migration task without breaking the real-time constraints of the Checkpoint/Restore or the migration mechanism.

4. Microkernel Concepts

Embedded systems most often use microkernels, because they are more fault tolerant than macrokernels and scale better regarding resource consumption. In the KIA4SM project there are also microkernels used for microcontrollers which are connected between each other to form the target automobile. This chapter will introduce basic microkernel concepts and how they benefit a Checkpoint/Restore mechanism. It will also create a basis for the concepts in the design chapter (chapter 6).

4.1. Basics

The basic idea of a microkernel is to provide a minimal set of functionality from which other components can be build. Liedtke [31] and Tanenbaum et Woodhull [32] present basic concepts which cannot be removed out of the kernel without breaking its fundamental functionality: Providing an abstraction from hardware and a multitasking environment.

First, an *address space* is a mapping of physical to virtual addresses, usually in memory page size. It hides the underlying hardware concept of the memory to provide protection. It means, that the user cannot map arbitrary physical pages, which belong to other address spaces, into its address space. An address space is just a working space where objects can be created, read, updated, and destroyed.

A *thread* is an entity which works on an address space. It is an activity which executes a specific task or a part of a task. The threads can be grouped to *processes* or *tasks*[1]. Each task has a designated address space and its threads share this address space.

Usually, the threads are not independent from each other, e.g. a thread from task A could provide a service to a thread in task B. Therefore, the concept of *inter-process-communication* (IPC) allows communication between two threads with usually different address spaces. It is realized by message passing. The microkernel copies a message from one designated address space location to another one's.

Because a system may have more threads than CPU cores, the microkernel provides a *scheduling* algorithm. In a desktop environment it uses a round-robin strategy, where

[1]The term "task" is more common in embedded systems.

all threads may get a fair time slot. In embedded and especially in real-time systems, a priority-based strategy is used to allow a fast processing of critical tasks.

An important difference of microkernels to monolithic kernels is the exclusion of hardware drivers from the kernel. In fact, the drivers are implemented as user-space tasks using the standard kernel functionality. Especially IPC calls are used extensively, thus, they have to be implemented efficiently, what Liedtke showed in the implementation of the L3 kernel [33].

All other important operating system functionality can be tailored from this small basic set. For example, a hardware device can be interpreted as a thread (called hardware thread) which sends IPC messages with no payload (like a signal) to subscribed threads. A file system is a driver for a disk space. It is implemented as a user-level task which uses a hardware thread and interprets the IPC messages.

4.2. Capability System

The basic mechanisms provide a foundation to build common applications, but such an application could access any arbitrary kernel object. Kernel objects are objects in the kernel address space which usually manage the hardware of the underlying system, for example a thread represents an activity which can be executed through the CPU. The capability system provides means to limit the access to arbitrary kernel objects to a minimum what an application needs to perform its task. Each process of an application is associated with a *capability list* residing in the kernel address space. This list contains *capabilities* which are access rights to kernel objects. The tasks are allowed to access kernel objects through capabilities. During the life-time of a task, it can receive further capabilities from other tasks or by requesting them from the kernel.

Although a capability system is not necessary to build applications, many microkernels use them to implement security. Some representatives are microkernels from the L4 family like Fiasco.OC, seL4, NOVA, OKL4, and microkernels from The EROS Group, the Johns Hopkins University, and the University of Pennsylvania: KeyKOS, EROS, CapROS, and Coyotos. The KIA4SM project uses the Fiasco.OC microkernel from TU Dresden. Therefore, the Checkpoint/Restore mechanism has to deal with capability lists in Fiasco.OC.

4.3. Checkpoint/Restore in a Microkernel

The microkernel concept facilitates a user-level Checkpoint/Restore mechanism by reducing the usually unaccessible kernel state to a minimum. A large part of the kernel

state can be tracked by monitoring the activity of the target application to other user-level applications (e.g. drivers, servers).

5. Genode OS Framework

The Genode OS framework [34] (called Genode) is an open source project lead by Genode Labs GmbH in Dresden, Germany. It is a tool kit for building operating systems ranging from embedded, special-purpose systems with as little as 4 MB memory to desktop/server, general-purpose systems. By focusing on security aspects, the framework instrumentalizes small, low-complexity building blocks from which new applications are created. The applications are structured hierarchically through a parent-child relationship and receive only as much resources as they need to fulfill their task. Genode runs on kernels, which range from monolithic kernels like Linux, over microkernels like Fiasco.OC, seL4 to hypervisors like NOVA, and supports the x86 and ARM architecture. These properties make Genode a promising operating system framework for the KIA4SM project, which uses Genode in combination with the Fiasco.OC microkernel on an ARM chipset like a PandaBoard ES.

This chapter will give a condensed introduction to Genode's concepts which will be used in the design chapter 6. For further readings, the Genode guide [34] is recommended.

5.1. Capability-based System

In Genode processes are called *components*. They live inside their dedicated *protection domain* what isolates their execution environment to other components. A component interacts with another one by using *Genode capabilities*. A Genode capability uses the kernel specific capabilities (see section 4.2) to access a dedicated *RPC object*, which is an object that can be called via IPC. The RPC object implements a specific *RPC interface*. A capability can be compared to a pointer in C/C++: Similar to a pointer which points to an object, a capability points to an RPC object. With the capability the component can invoke a function of the RPC object. The important difference to pointers is that capabilities can also point to objects in other processes, but pointers cannot.

Figure 5.1 illustrates a component's capability and its own list of usable capabilities called *capability space* in the kernel. A capability space cannot be accessed by the component itself, because it is protected by the kernel. In the example, the capability has the name 3 and the kernel stores a so-called *object identity* in slot 3 of the capability space. An object identity is an identifier for the capability, it denotes the owner of the

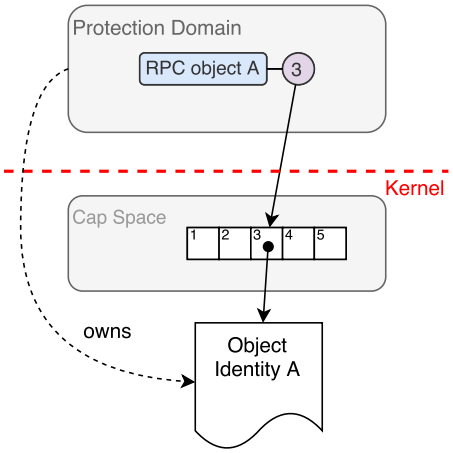


Figure 5.1.: Relationship of a capability, its cap space and object identity.

RPC object, which is useful for finding the RPC object from another component and for deleting it. Only the owner is allowed to delete the RPC object with its object identity.

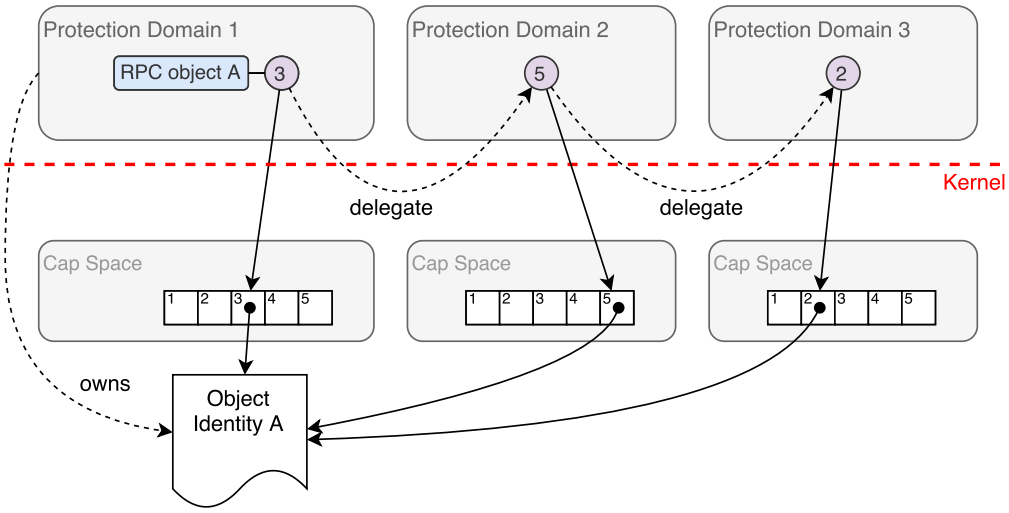


Figure 5.2.: Delegation of a capability to different protection domains.

Capabilities are most useful when they are used from a process other than the owner of the RPC object. Therefore, the owner sends or *delegates* a capability from its protection domain to another. Figure 5.2 depicts the delegation of a capability from the owner to a 2nd protection domain, and from the 2nd to a 3rd protection domain. When delegating a capability from a capability space to another, the capability has to receive

a free slot in the new capability space. The free slot does not have to be the same number as the one in the originating capability space. A capability is a security concept, where an arbitrary component cannot create a valid capability to an existing RPC object accidentally or malevolently by itself, because the kernel manages the capability space and the delegation is also performed by the kernel through IPC calls.

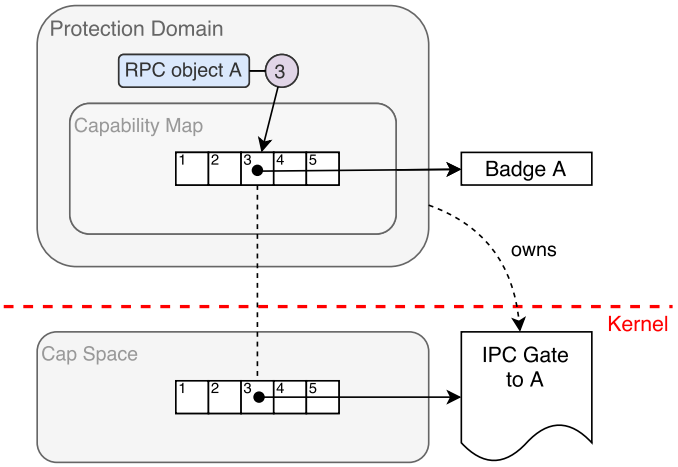


Figure 5.3.: Fiasco.OC specific capability system.

In Fiasco.OC the concept could not be implemented directly as described above. Figure 5.3 shows the real implementation of an object identity and an extra, component intern capability space. Fiasco.OC does not provide an efficient way of traversing the capability space to find a specific capability. Therefore, each component has a *capability map* which mirrors the capability space to add additional information for each slot. The capability map introduces a system-global identifier called *badge*. It tackles the problem of duplicate kernel capabilities pointing to the same object identity. For example, if a delegated capability already exists in the destination protection domain, the protection domain shall not copy the capability to a new slot, but use the found slot. In Fiasco.OC an object identity is an IPC gate. An IPC gate is a kernel object and is associated with a thread and a badge. If an IPC message is send to the IPC gate, it is forwarded to a thread. The thread receives the message and the badge. With the badge it can identify the RPC object to which the message is destined. Each RPC object has its own IPC gate. All in all, an RPC object is associated with an IPC gate, which forwards IPC messages to it, which consequently resembles an RPC mechanism.

5.2. Client/Server Concept

Genode uses small building blocks for creating new components. These building blocks are in the form of services. An arbitrary component can provide a service for other components. The provider is called *server* and the service consumer is called *client*.

A service is implemented as a *root RPC object*, which can create, upgrade, and destroy *session RPC objects*. A session RPC object is a service instance which provides the service specific methods. The difference between normal RPC object and session RPC objects is, that session RPC objects can be requested through the parent component. Normal RPC objects are obtained from methods of a session RPC object.

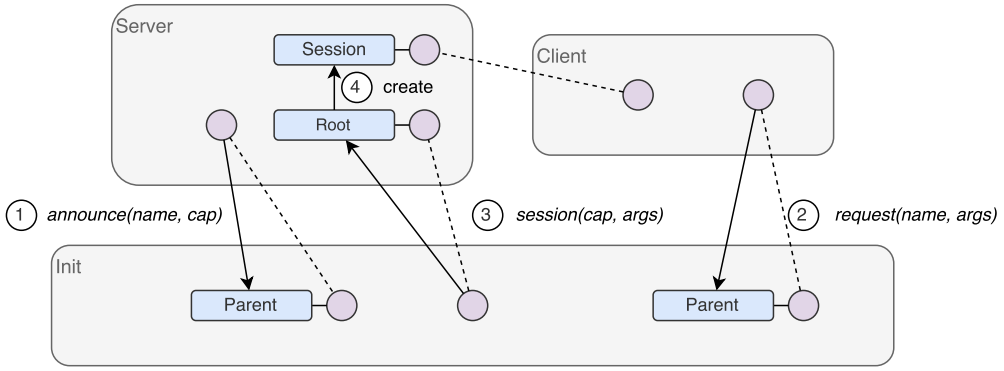


Figure 5.4.: Client/Server interaction: Announce and request.

Before a client can access a service, the server has to announce its service to other components. This is realized through the parent component. Figure 5.4 shows the steps from announcing a service to requesting this service. The client and server components have a common parent, the init component. This is a Genode specific component which creates other components and provides system resources to them. In the first step, the server announces its service to its parent by providing the service name and the root capability (= capability to a root RPC object). After that init knows the root capability of the server component which is illustrated by dashed line between both circles. When a client wants to use the service it requests a session capability (= capability to a session RPC object) with the service name and arguments which are needed for the creation of the session RPC object. In step two the parent receives a session request and can react in three ways:

- It can simply ignore the request and return an invalid capability.
- If the parent knows the service (i.e. it has a capability to the root RPC object), it can request to create a session RPC object and return the capability to it.

- If the parent does not know the service, it can ask the grandparent and return the result to the child.

In step 3 of the example the parent takes second option. It issues the method "session" from the root RPC object to create a new session RPC object at the server. Consequently, the server creates a new session RPC object and delegates the capability to it to init and init delegates the capability to the client.

This mechanism forms a *recursive system structure*, where root RPC methods are handled recursively. The parent decides what services the child is allowed to use, thus, each parent implements its own policy. Contrary, the session RPC methods are handled directly from client to server and vice versa. This enables a fast processing of session RPC methods without involving a lot of IPC calls.

5.3. Core's RPC objects

Core is the first user-level component created directly by the kernel. It receives all hardware resources from the kernel which are passed to its only child component, init. Init spawns new components and hands out resources to them. Furthermore, core provides services to utilize the resources of components. The following sections describe the services provided by core:

- *Dataspace* and *RAM session*: A dataspace represents a memory chunk. It can be attached into an address space of a component and then read or written. Dataspaces are allocated through an RAM session. A component can create dataspaces, if it has sufficient RAM quota. This is a budget of available memory space, which the component receives from its parent component at creation time. For example, init creates a child component and transfers RAM quota from its RAM session to the one of the child. Components can establish shared memory by attaching the same dataspace to their address space. The second component could have received the capability of the dataspace through an RPC function.

- *Region map* and *RM session*: A region map is a mapping of physical to virtual address spaces just like the page tables in an MMU. Each component has three pre-allocated region maps, the address space, stack area, and linker area. The address space represents the complete virtual address space of a component. The stack area contains the stacks of all component-intern threads, while the linker area contains shared libraries. The latter two areas are attached into the address space as so-called managed dataspaces which were converted from region maps (see section 5.5). The RM session can create further region maps, which can be attached as managed dataspaces into other region maps.

- *PD session*: The PD session defines the execution environment of the child. It comprises a capability space (with a capability map in Fiasco.OC) and the three pre-allocated region maps: address space, stack area and linker area. The capability space is a kernel-intern data structure and is transparent to the component. The PD session also allows to create new capabilities for component-own RPC objects (including object identity) and has an API for receiving and emitting signals.
- *CPU thread* and *CPU session*: A CPU thread represents a thread. It can be used to read the state of a thread and to change it. Each CPU thread has an own stack which resides in the stack area. A CPU session allows to create new CPU threads.
- *ROM session*: The ROM session provides dataspaces where compiled source code resides. When components are bootstrapped, they receive a dataspace containing their source code through the ROM session. This dataspace is attached into their address space. Furthermore, the ROM session provides dataspaces with an XML configuration for a component.
- Access to I/O devices: Core provides services for memory-mapped I/O regions, I/O ports, and the reception of device interrupts. These services create *IO_MEM*, *IO_PORT*, and *IRQ* session RPC objects for its clients. With these sessions user-level driver can be realized.
- *LOG session*: The LOG session is used to provide diagnostic messages to the programmer.

5.4. Child's Resources

As mentioned earlier, a creator of an component is called *parent* and the created component is called *child*. The parent also creates the environment of the child and provides it to the child in the form of capabilities inserted into child's capability space. Each child receives a PD, RAM, and CPU session capability. The PD session provides capabilities to three region maps: address space, stack area, and linker area. The child can use them to access dataspaces. With the RAM session it can create and destroy its own dataspaces. Before that, the parent has to donate RAM quota to this RAM session which is used as budget for creating dataspaces. With the CPU session, the child is allowed to create and destroy its own CPU threads.

A fourth essential capability in child's capability space is the parent capability. During child's creation, the parent creates an RPC object of the Parent interface and delegates its capability to the child. This RPC interface provides functions to request session capabilities from components offering a service. It also allows to transfer RAM quota

from child's RAM session to the RAM session of the server to allow it to allocate objects necessary for the service. Parent's close function informs the server to destroy the session RPC object and consequently close the session connection to the server. The donated RAM quota is returned from the server to the child. The parent interface also allows to announce child's services. With this function the parent receives the root capability of child's service and is allowed to create new session RPC objects in child and delegate the created session capability to the requester.

5.5. Managed Dataspaces and Page Fault Handler

A *managed dataspace* is a dataspace which represents a region map RPC object. Each component receives three region maps in its PD session. If the component needs extra region maps, it can create them through core's RM service. A managed dataspace is useful for manual management of a part of the address space. For example, the region map representing the stack area is meant for dataspace which are used as stacks and thread meta data.

Another use case is on-demand-populated dataspaces. A server could handout dataspaces disguised as region maps. Initially, the region map does not contain any dataspaces. When the client accesses a portion of the region map, the server attaches a dataspace backed by physical memory. This mechanism is realized by a user-level *page fault handler*. Each region map can associate a page fault handler to solve user-level page faults.

A User-level page fault is generated when a CPU thread accesses a portion of a region map which is not backed by a dataspace. If this happens, core creates a signal and submits it to the thread which handles the page faults for this region map. A component has a main region map, the address space. Every memory address (e.g. pointer) in a component refers to the address space. Whenever a CPU thread accesses the address space (or any other region map) by a memory address there are three options what could happen:

- If a dataspace is attached to this position, then the CPU thread accesses the physical memory associated with the dataspace.
- If a managed dataspace is attached, then the CPU thread queries the content of this region map.
- Else core generates a page fault signal and submits it to the thread responsible for the region map. The thread resolves the page fault by attaching a dataspace to the faulting address. If it does not attach a dataspace, then the CPU thread is blocked forever.

5.6. Inter-Component Communication

Genode provides three ways of communication between two components: Synchronous RPC function invocations, signals as asynchronous notifications, and shared memory for bulk information exchange. At the beginning of an *RPC function call* the client sends a request message with the method identifier and the method arguments over the IPC mechanism of the underlying kernel. The server receives the request message, dispatches it, processes the request, and answers with another IPC message to the client. During this process the client is blocked from execution.

The *signal* system is realized with the IPC mechanism and a dedicated signal thread for each component. A signal is a mere notification which does not contain any payload. It is used to inform the receiver that a specific event occurred, e.g. a page fault or a keyboard input occurred. Genode implements its own signal system, which uses core as a proxy component between sender and receiver.

Before a component can receive signals, it invokes the RPC function block_for_signal of core's signal source RPC object from its signal thread. This function blocks the thread until core receives a signal for the component and returns it through the return mechanism of the RPC function. The signal thread dispatches the signal and forwards it to the component's thread which waits for this signal.

The signal submission is realized by the submit method of the PD session in core. A sender invokes the method which sends an IPC message to core. Core processes the message and answers to the receiver by returning the signal information from the RPC function block_for_signal.

The third communication mechanism is *shared memory* between two components. It is realized by delegating a dataspace capability from one component to another. Shared memory allows the components to communicate a bulk of information. If the it is used for bi-directional communication the access to the memory has to be managed in order to not override the changes of one component.

These three mechanisms can also be used in combination to exploit their advantages and circumvent their disadvantages. For example, in Genode a page fault is a signal which is send from core to a page fault handler. The page fault handler uses an RPC function call to the faulting region map to receive the page fault information. To propagate the page fault, core does not need to rely on an RPC function call to the receiver. If core would use an RPC function call, the receiver could be malevolent and block core indefinitely, thus, disabling any other component from using core's services. Instead, core just submits a signal to interested receivers. If there are any, they can invoke the state method of a region map RPC object.

6. Design

There are many possible ways to design a Checkpoint/Restore mechanism. Each design depends on the hardware and software environment where it shall operate. The environments of this thesis are an embedded real-time system and Genode/Fiasco.OC, respectively. This chapter will discuss the goals and requirements of a Checkpoint/Restore mechanism suited for these environments. It will introduce different designs for major tasks in a Checkpoint/Restore mechanism and discuss their advantages and disadvantages. Eventually, suitable designs are chosen and contributed to the implementation of a real-time capable Checkpoint/Restore mechanism called *Rtcr*.

6.1. Goals and Requirements

The KIA4SM project is settled in a time critical environment where hard *real-time* requirements are present. Therefore, the Checkpoint/Restore mechanism shall support these requirements by following the real-time requirements in section 3.3. It shall illustrate the full potential of a fast mechanism. Consequently, the main focus lies on performance and also scalability, because it indirectly supports the performance requirement. Scalability means that the Rtcr component will impose only a small overhead increase while increasing the amount of information to checkpoint (e.g. more memory regions).

The implementation will base on Fiasco.OC/Genode. Nevertheless the design will depict a general approach for a *minimal microkernel* with the addition of a capability space. Thus, other researchers can take this approach as a design basis.

Another goal is to *not bind* components on different CPU cores. The Rtcr and target components are allowed to run on the same as well as on different cores. Furthermore, the Rtcr component shall be *transparent* to the target component. This has the advantage of using already developed components which are unaware of a Checkpoint/Restore mechanism. Also, a developer has not to bother about using a Checkpoint/Restore library. Rtcr shall follow the microkernel-paradigm (see section 4.1) and realize a *user-space* mechanism. It shall use the Genode API which is based on the microkernel and which can be broken down to microkernel basic functionalities (see table 6.1).

6.2. Design of Rtcr's Mechanisms

The Rtcr component is composed of several subtasks. First, the component has to acquire *access* to the data of the target component. The access is used to read the data for the checkpointing mechanism and write the checkpointed data back to a target for restoration. Second, Rtcr has to periodically and time-efficiently *checkpoint* the data of the target by blocking the target component as little as possible. Then, the *restore* mechanism has to recreate the target component without any system and logical errors. The Rtcr component shall support a custom migration module, which can transfer the raw checkpointed data from one node to another. Consequently, the data has to be checkpointed and *serialized* on the source node, and transferred to the destination node, where it is *deserialized* and restored.

6.2.1. Target's Resources

A Checkpoint/Restore mechanism has to checkpoint data and restore them on a fresh component. Which data is required depends heavily on the restoration mechanism. The basic idea is, that the restored target component produces the same results as the checkpointed target component would do. This idea poses two questions: What data are needed for a checkpoint, and how is the access established to the data during the runtime of the target component.

Data to Checkpoint and Restore

To know which data of a process has to be restored, it is necessary to know how a process is defined on the lowest level, i.e. kernel-level. A process has an address space, it is comprised of at least one thread, has scheduling parameters, and uses IPC for communication (compare section 4.1). Genode abstracts these concepts and provides services and inter-component communication mechanisms (compare section 5.3). The idea is to implicitly restore the microkernel state by explicitly restoring the state of Genode's objects.

Basically, all information has to be checkpointed which is needed to restore the computational environment of the target component. Target's environment is composed mainly by capabilities and their referenced RPC objects. The capabilities are stored in the capability space which is managed by the kernel. Each RPC object is associated to an IPC gate (compare figure 5.3). Some RPC objects also create further kernel objects during their creation: A CPU thread creates a thread kernel object, a PD session creates a task kernel object. Consequently, the state of a kernel object can be restored indirectly by creating an RPC object and restoring its state. Therefore, the RPC object's creation

arguments and the RPC object's method calls which change its state have to be stored.

To restore the state of a target component, all session RPC objects and their created normal RPC objects have to be restored. In particular, these are the three session RPC objects provided at the creation of a component: PD, RAM, and CPU sessions. And consequently their created normal RPC objects: Region map, CPU thread, Signal context, and Signal source. Additionally, all session RPC objects and their normal RPC objects, which are known to the target component, have to be restored, e.g. an RM session or Timer session.

To restore the capability space not only the kernel objects have to be restored, but also the locations in the capability space which point to them. Additionally, the capability map which resides in the address space of the target has to be adjusted. Each recreated RPC object will have a new badge, but the checkpointed memory content will still reference the old badges. Thus, the new badge has to be replaced with the corresponding old badge in the capability map.

The inter-component communication is also a possible subject of restoration. It stands for three mechanisms which are signals, RPC function call, and shared memory. The next paragraph will analyze both incoming and outgoing communication for restoration.

For restoration the information about *any incoming communication messages* to the target component are not needed. The reason is, when the target is restored, the other components which issued the communication messages are not there on the same node. *Outgoing signals* are not necessary to checkpoint, because the target does not expect any answer. *Shared memory* is established by trading dataspace capabilities through RPC. Thus, it will be restored when the session RPC object with its created normal RPC objects are restored. *Outgoing RPC function calls* block the execution of target's thread. The thread expects an answer which changes the user thread control block (utcb) which resides in target's memory. These calls can be avoided by Rtcr, if it only uses the checkpointing mechanism, when no RPC function call is in use. Otherwise, the instruction pointer of the thread has to be changed to the next command after the RPC function call.

To summarize the required data, it is needed to restore all session RPC objects and their created RPC objects, the capabilities in the capability space and map, and potentially the outgoing RPC function calls. Table 6.1 provides an overview of microkernel concepts mapped to Checkpoint/Restore data or restore-avoidance mechanisms.

Accessing Data to Checkpoint and Creating Data for Restoration

The access to the checkpointing data can be realized by different components in Genode. Possible candidates are an arbitrary component, the core component, the parent component, or the target component itself. Each component has to read/checkpoint the state

Microkernel Concept	Checkpoint/Restore Data
Address space	Region map in PD session
	RAM session incl. dataspaces
Thread	CPU session incl. CPU threads
Process	PD session
IPC	Avoiding RPC function calls
Scheduling	CPU threads

Table 6.1.: Mapping of microkernel concepts and Checkpoint/Restore data

of the target component and also write/restore the state. The following paragraphs will discuss a possible way of accessing required session RPC objects. Furthermore, they elaborate on the the advantages and disadvantages of each approach, especially considering real-time requirements. Additionally, one approach for accessing the capability space and one for an RPC function call is presented.

Access to Target's Resources *Arbitrary component*: An arbitrary component has no access to the RPC objects known by the target component, due to security reasons. For transparency, the arbitrary component has to query the information from the service providers offering the session RPC objects. Therefore, each server component has to implement an interface which returns the state of an RPC object belonging to the target component. The advantage of this approach is that no special component is needed to checkpoint and restore the target. But its disadvantage is the lack of security, because any component can read the state of RPC objects of any component. Also, each server component has to implement an additional service which provides state information for its created RPC objects. The approach adds IPC calls for asking the server about whether they have information about a specific component and the information itself. This adds further overhead to the checkpointing mechanism.

Core component: The core component knows every capability which was created in the system and also provides access to fundamental services which are required by a component. A process requires a PD, CPU, and RAM session to implement simple functionality. Unfortunately, a process could use services of other components. The core component does not have access to RPC objects created at other service components, e.g. a Timer server. Therefore these services have to actively provide information about their created RPC objects to other components. In the end, the core component approach requires the same extra services like the arbitrary component approach. The benefit

of this approach is that core knows the states of its own RPC objects. If the target component would be limited to only use core's services, then this approach would be beneficial. Core could checkpoint all states directly from itself and restore it easily. If the target component is allowed to use the services of other components, then the approach has to follow the arbitrary component approach by implementing an additional service per server to query the state of the RPC objects used by the target component. Like the arbitrary component approach, this approach has the same benefits and issues. Additionally, extending core imposes conflicts then upgrading Genode and core to a new version.

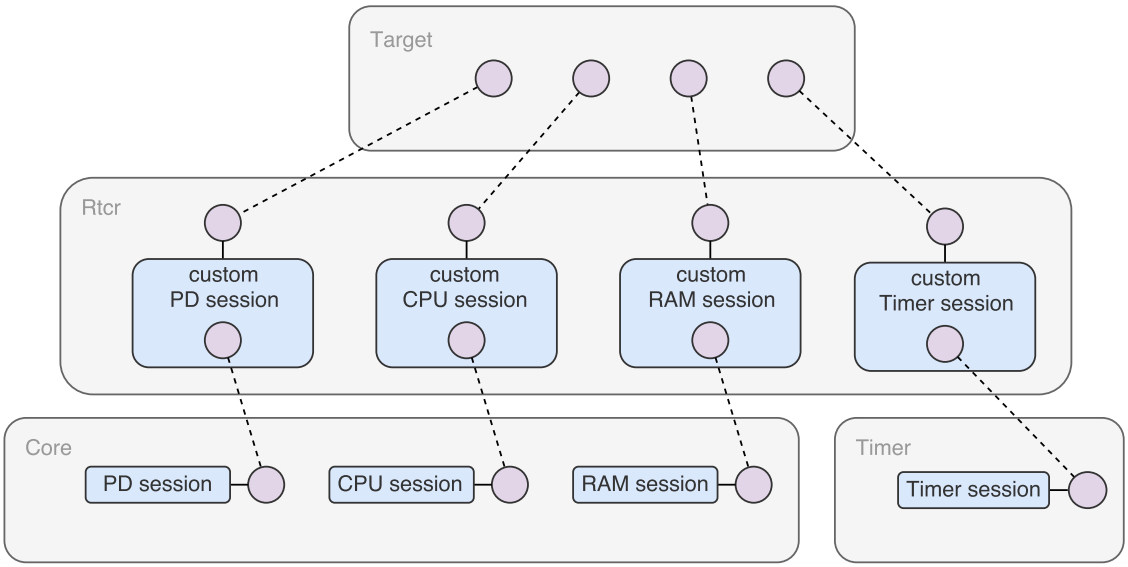


Figure 6.1.: Intercepted services of Rtcr for the target.

Parent component: The parent component is responsible for providing essential resources to a new child process. The idea of this approach is, the parent component checkpoints and restores a child component, because it knows the resources which it provides to a child at creation time or when requested by the child. Therefore, the parent component shall provide custom session RPC objects which intercept the usage of the child component to the real RPC objects. Instead of providing core's PD, CPU, and RAM session at the creation time of the child, the parent component provides its own PD, CPU, and RAM session which internally use core's services. Furthermore, whenever the child requests a session to an arbitrary service, the parent provides a custom session of its own which also uses the real service internally. Figure 6.1 shows how parent intercepts the sessions of the child component. The Rtcr component is the parent component and uses custom RPC objects which monitor the access of the child

to these objects. By monitoring the access, Rtcr can recreate the internal state core's and Timer's RPC objects by reissuing the same calls as the child.

The advantage is that a parent component is natively responsible for the resources of the target. By simply intercepting all session requests and invocation of session functions, the parent can restore the state of the RPC objects used by child. The disadvantage is, although the services are not needed to be extended, Rtcr has to implement custom services. Fortunately, it implements them in its own component. Also, an RPC function invokation of an intercepted session induces an additional RPC function invokation from parent to service. This imposes a performance disadvantage during the runtime of the target by extending its execution time by doubling the waiting time for an RPC function call.

Target component: It is possible to let a target component checkpoint and restore itself and also ensure transparency. This is possible by linking a Checkpoint/Restore library to the binary of the target component. For example, the approach of Libckpt [6] requires to link a library to the executable of a target process in Linux (compare section 2.2.1). By doing so Libckpt has access to the resources of the target process because it runs in the same execution environment. In Genode a library can also be linked against the executable file. The library will intercept all invocations of used RPC objects and checkpoints its state to an unused dataspace. To restore the RPC object's state the library reissues these method calls. The advantage of this approach is that no further component is needed to checkpoint and restore the target. Thus, it saves IPC calls to retrieve state information like in the other approaches. The disadvantage of this approach is, that it needs an own bootstrap code to first start the library. Genode's init component has to be modified to start target components with the new bootstrap code.

Access to the Capability Space Reading the capability space of the target component can be realized by reading the capability map stored in target's memory. The capability map stores badges with their corresponding capability space slot (see section 5.1). Thus, for each RPC object used in target the corresponding capability space slot can be checkpointed. For systems without capability maps, the capability space slot of an RPC object can be found out by comparing a known capability to each capability space slot of the target until the capability is found or until the end of the capability space, which is set statically for each component.

When restoring the capability space, Fiasco.OC's asynchronous mapping system call can be used. It maps a capability from a capability space slot of an arbitrary component to a capability space slot of another arbitrary component. To support a microkernel without such a method, the target's memory could be altered by replacing all pointers from old to the new capabilities. However, this task is not trivial and requires the

memory locations of all capabilities. Otherwise, the microkernel could be extended with this functionality while ensuring the requirements of the kernel, e.g. security.

Handling of RPC Function Calls The RPC function calls are handled by avoiding them during the checkpointing phase. They are avoid by implementing a lock which is set when an RPC function method is called and unset when it returns. In case of the arbitrary and core component approaches, the lock has to be implemented for each used RPC object. The other approaches implement the lock themselves. Avoiding RPC function calls is a simple approach and allows to a first implementation of the Checkpoint/Restore mechanism. Later other approaches can be designed to handle them.

Discussion All four approaches are generally real-time capable by using functions which are guaranteed by the microkernel to be real-time capable. Nevertheless low overhead imposed from the checkpointing component onto the system supports high scalability of a real-time system. The arbitrary and core component approaches extend the checkpoint time by requesting the state of RPC objects from server components. They also require to change and extend current components with checkpoint/restore logic. The parent component approach doubles the time of target's RPC function calls by intercepting the calls. But it takes the responsibility to implement the checkpoint/restore logic to its custom RPC objects. It also uses Genode's native mechanisms to read and manipulate the state of the child (= target) component. The target component approach eliminates the need to issue IPC calls to other components when intercepting RPC object methods or when checkpointing the target. It is the favorable candidate for an optimal real-time approach. But its complexity of integrating it into Genode is high and not suited for a master's thesis. Furthermore, it requires to extend the init component to be aware of checkpoint/restore capable components.

Decision Although the target component approach seems to be the best suited real-time capable approach, it imposes high implementation complexity and extending Genode's init component. The arbitrary and core component impose IPC calls to servers and also lack portability. The parent component also imposes extra IPC calls during the runtime of the target, but allows portability and extensibility. It is interesting to find out whether adding new RPC function calls to servers or doubling the execution time of target's RPC function calls extends the execution time more as the other approach(es). In this work the parent approach will be used, because it is portable, extensible, and is managable during a master's thesis.

6.2.2. Checkpointing

The checkpointing mechanism has to copy target's resources to a new memory location. From there it can be copied to disk or sent to another node via network. As described in one of the last sections, target's resources consist of PRC objects and the capability space. The RPC function calls shall be avoid by locking the checkpointing mechanism when RPC methods are used. This section assumes that a parent approach as described in the previous section is used.

To create a consistent checkpoint, the threads of the target will be paused during the checkpoint. If they would be allowed to run and potentially change the state of RPC objects before they are checkpointed, then the checkpointing data will contain data which is valid at checkpointing time and other data which is valid after the checkpointing time.

The amount of data of various RPC object types is not similar, especially the dataspace RPC object which represents a memory region. While most RPC objects store only a few flags, integers or capabilities (which is a memory address to a capability map slot) which ranges from 4 to 40 bytes, the state of dataspace RPC objects is at least 4 KiB. To optimizing a checkpointing mechanism towards performance and real-time support, the time for checkpointing the memory has to be lowered.

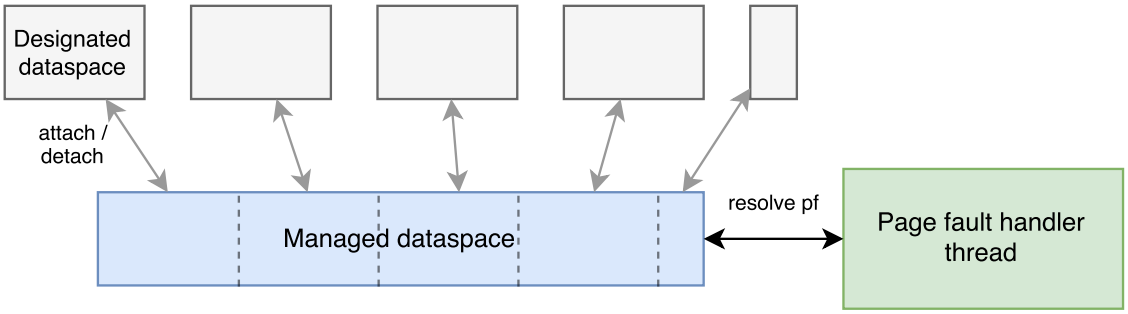


Figure 6.2.: Structure of a managed dataspace provided by the custom RAM session

Incremental Checkpointing The incremental checkpointing mechanism was inspired by plank et al. [1]. The idea is to copy the dataspace content of changed dataspaces. Therefore, the algorithm has to detect the write access triggered by the child component. To accomplish this, the custom RAM service, which is provided to the child from Rtcr, allocates managed dataspaces instead of normal dataspaces. The structure of one such managed dataspace is depicted in figure 6.2. Each managed dataspace is associated with designated dataspaces. Designated dataspaces are dataspaces which occupy a specific location in the managed dataspace. These dataspaces are usually detached from the

managed dataspace, and attached when the target uses a certain region of the managed dataspace.

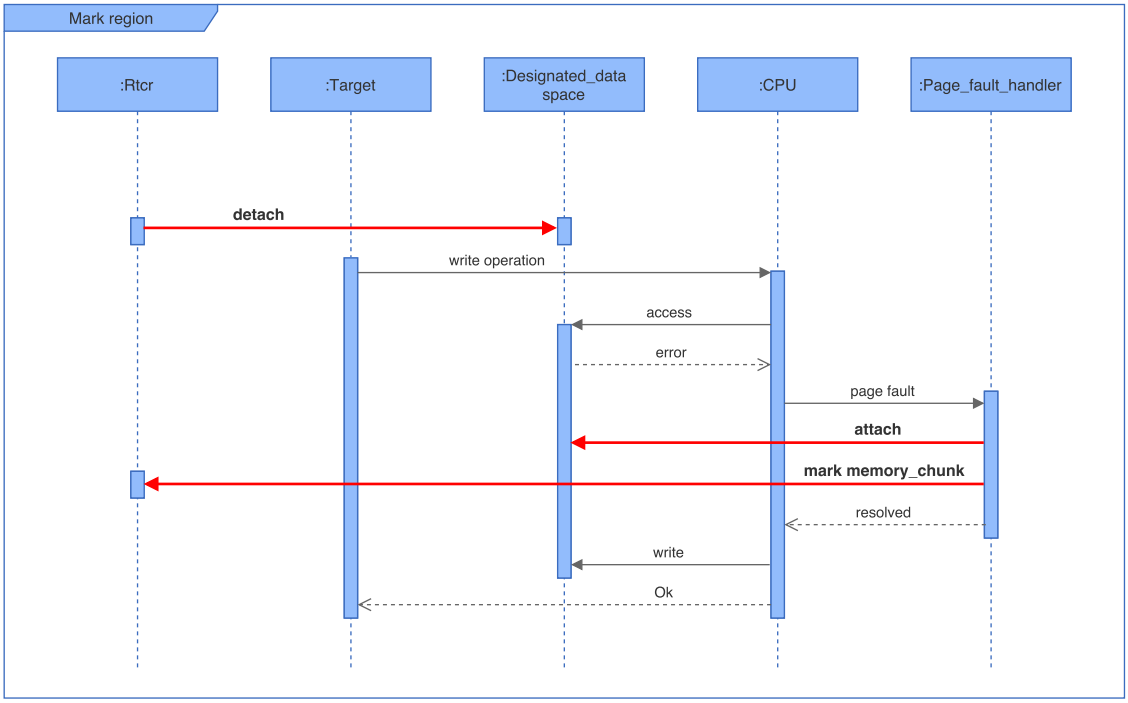


Figure 6.3.: Incremental Checkpoint: Marking an accessed dataspace

Figure 6.3 depicts the marking of accessed regions in the managed dataspace. First the designated dataspace is detached from the managed dataspace. When the target component tries to write to the region of the designated dataspace, the CPU triggers a page fault. This page fault is handled by Rtcr's page fault handler thread by attaching the designated dataspace back to its location and marking this dataspace as accessed. The attaching method resolves the page fault and the CPU can read the content.

The unmarking is realized by the checkpointing mechanism and is depicted in figure 6.4. For a consistent checkpoint the target's threads are paused. In the next two steps the Rtcr component copies the content of the designated dataspace to its own dataspace. Next, it unmarks the designated dataspace and detaches it in order to trigger the next page fault to mark the dataspace. Eventually, target's threads are resumed.

The size of the designated dataspaces can be used to control the granularity of the accessed region in the managed dataspace. The higher the granularity the higher the amount to copy data for an accessed region. Contrary, the higher the granularity the lower the overhead imposed by the Rtcr component during the runtime of the child.

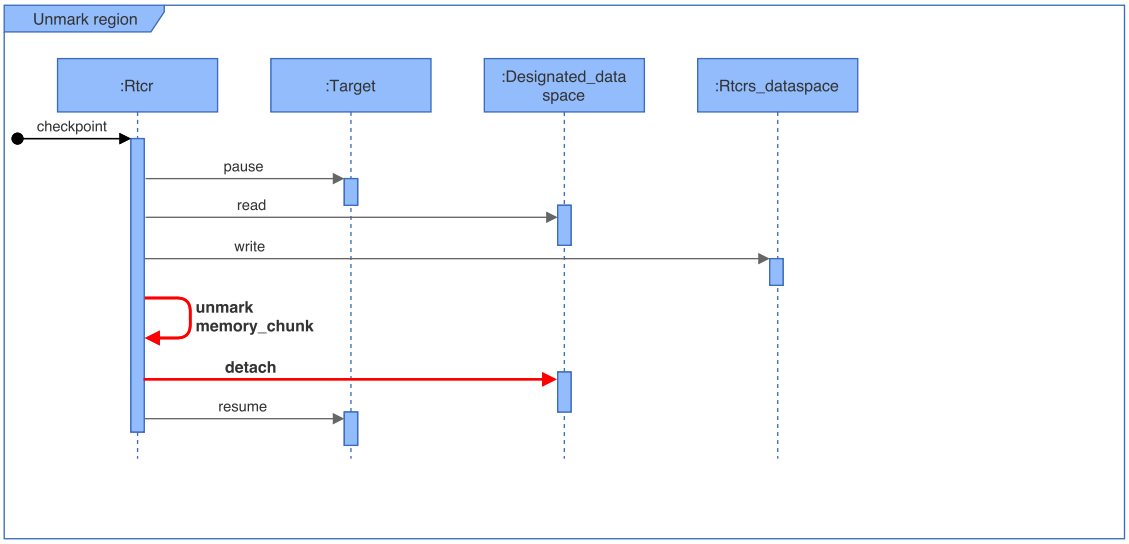


Figure 6.4.: Incremental Checkpoint: Unmarking an accessed dataspace

The lower overhead comes from a smaller number of designated dataspaces (which can trigger a page fault). The optimal granularity depends on the application and its usage of the memory locations.

The incremental checkpointing mechanism has the advantage of reducing the amount of dataspace content which has to be copied. But it uses page faults to mark the access to dataspace. The usage of page faults causes a small and but controlled delay to the target component during its execution. Unfortunately, Genode does not provide any mechanisms to mark regions of dataspaces except this one. Another disadvantage is the inability to only mark write accesses to the dataspace. In order to solve any page fault a dataspace has to be attached to the managed dataspace. If a read page fault occurred, theoretically the dataspace could be attached and detached shortly after. Unfortunately, there is no mechanism to immediately detach the dataspace after the read access or even a mechanism to know it was read by the target. Furthermore, it is possible that one designated dataspace could trigger many read page faults without triggering a write page fault. Thus, to optimize the incremental checkpointing mechanism in Genode, read access shall not trigger the marking mechanism.

Copy-on-Write Mechanism A copy-on-write (COW) mechanism was already used by Li et al. in Libckpt [6]. The basic idea of the COW mechanism is that target's threads are allowed to run while the memory content is checkpointed. First the threads are paused in order to create a consistent list of dataspaces to checkpoint. Then, all

dataspaces from target's address space are detached, thus, accesses to the address space will trigger a page fault just like the mechanism introduced in the incremental checkpointing mechanism before. After that, threads are resumed and are allowed to process their workload. Now, a checkpointer thread starts and copies the detached dataspaces and attaches them back to the address space. When one of target's threads tries to access a memory region where a dataspace was detached before, a page fault is triggered and handled by a page fault handler thread. This thread copies the content of the accessed dataspace, reattaches them, and implicitly resolves the page fault. The COW mechanism lowers the checkpointing overhead and allows a concurrent execution of checkpointer and target's threads. This approach suits well for desktop systems where a round robin scheduling algorithm is used. Unfortunately, in embedded real-time systems a priority-based algorithm is used, thus, prohibiting an interleaved execution of checkpointer and target's threads.

Multi-Core Approach A multi-core approach is an approach, where a CPU core is designated for the checkpointer which can run in parallel to the target components. This approach synergizes with a COW approach, because the COW approach provides consistent memory checkpointing while target's threads run in parallel. It allows to utilize the COW approach with a priority-based scheduling algorithm. Currently, there is a trend (e.g. [35]) of using microprocessors which provide a cluster of high performance CPU cores and a cluster of low performance CPU cores for balancing power consumption. Such a microprocessor can be utilized to run the Rtcr component on a low performance CPU core while running the target components on the high performance ones.

Mirror Copy Mechanism The mirror copy mechanism is heavily inspired by Staknis's sheaved memory concept [36]. The idea is, when writing to a page frame, the hardware writes also to one additional page frame simultaneously. Both page frames are sheaved together and form a 2-way sheaf frame. The page frames in a sheaf frame all contain the same data.

During the runtime of the target component, the target component writes data to a page frame and to another page frame which belongs to the same sheaf frame. The other page frame is known by the Rtcr component. Hence, the Rtcr component has always the newest copy of target's memory content. If the Rtcr component decides to checkpoint or migrate the component, it has to detach its page frames from the sheaf frames for consistency of the data. If migrating the component the content of the page frames can be sent directly and the target component killed. If just checkpointing the state to another node (e.g. a node which stores checkpointed data), the page frames have

to be reattached to the sheaf frames afterwards and synchronized. This can be realized by marking changed page frames from the target component during the checkpointing process. Then synchronizing the page frames in each sheaf frame the target component has to be stopped to guarantee a consistent synchronization. This approach has to be realized in the kernel, because Genode's and Fiasco.OC's API does not provide a sheaved memory concept. The advantage of using a kernel realization is that the kernel manages the memory directly. If the MMU supports direct memory-to-memory copy, the memory copy during a synchronization could be handled without occupying the CPU. But the target component still has to wait until the memory regions are copied before accessing them. This approach postpones the memory copy from the checkpoint to the synchronization phase. The disadvantage of this approach is the extension of the microkernel and loosing the microkernel paradigm to implement only the essential mechanisms. For the realization an in-depth knowledge of the underlying microkernel and hardware mechanisms are necessary.

Discussion The introduced approaches vary a lot in their properties. While the incremental checkpoint and COW mechanisms allow a Genode native implementation, the mirror copy mechanism requires the extension of the microkernel and Genode API. Thus, mirror copy requires in-depth knowledge about kernel and hardware mechanisms and effort to implement the mechanism and extend the APIs of the microkernel and Genode. Although the checkpointing phase is very short, the synchronization phase has to copy memory regions. Due to the short checkpointing phase, the memory region for synchronization are rather small. Theoretically, the algorithm poses the shortest memory-to-memory copy phase of all introduced algorithms. Although the COW mechanism supports the concurrency between Rtcr and the target component, it does not affect the overall checkpoint duration, which is crucial in the optimization of checkpointing mechanisms. COW together with a multi-core approach seems promising, because there is a chance, that the target component runs without interruptions during the checkpointing mechanism. But, theoretically, it can be interrupted at each memory access of the target component reducing the benefits of the algorithm. Furthermore, the implementation of a COW algorithm was already started by Werner [26]. The incremental checkpointing mechanism does reduce the overall checkpointing duration and thus, increasing the performance of the checkpointing mechanism.

Decision In this thesis the incremental checkpointing mechanism will be implemented. It can be implemented with Genode's native API and does not extend the microkernel. Although it uses page faults which might increase the execution time of the target component, the checkpoint duration will be reduced significantly when the target

component needs a lot of memory.

6.2.3. Serialization, Transfer, and Deserialization

The serialization and deserialization of checkpoint data is needed for migration. The serialized data is transferred from the source node to the destination node, where it is deserialized. To lower the transfer duration, the data shall be sent in binary form.

6.2.4. Restoring

The restore mechanism has to use the checkpointed data and recreate a new component which yields the same results as the checkpointed component would do. It can be designed in two ways. The restore mechanism can recreate all RPC objects with their stored states by manually and inject their capabilities into target's capability space. This approach has full control over the target's environment, but it imposes implementing an own bootstrap mechanism to start components from a checkpointed state.

Another approach avoids implementing a bootstrap mechanism by using the mechanism provided by Genode's API. The component is bootstrapped as it would be, when there is no checkpointed state. Just before the main component code is about to be executed, the restore mechanism is executed which alters the state of the already created RPC objects and which creates new RPC objects with their state from the checkpoint data. This approach loses the full control over the created RPC objects. Thus, the approach has to identify the RPC objects which were created by the bootstrap code and data of the RPC objects in the checkpoint data.

The second approach will be used for the restoration, because it reuses the bootstrap code which already exists. It has to implement a identification phase which does not impose high overhead. Furthermore, the identification phase is not crucial for a real-time capable Checkpoint/Restore, because also the dataspace content takes significantly more time.

The restore mechanism is summarized in the following steps:

1. When the restoration mechanism starts, it has to identify RPC objects that were created by the bootstrap mechanism and associate them to the checkpointed RPC object data, and update their state to the checkpointed state.

2. After that, it has to recreate RPC objects which were not automatically created by the bootstrap mechanism and to change their states.

3. The capabilities of the manually recreated RPC objects have to be inserted into the capability space.

4. Finally, the capability map has to be adjusted to new badges of the new RPC objects.

6.3. Design's Portability to other Microkernel

The design is based on Genode's API. The operating system/microkernel which shall run the mechanisms chosen in this chapter has to provide the basic functionality of a microkernel, namely processes, address spaces, threads, IPC, scheduling, and capabilities. Furthermore it has to support some of Genode's paradigms:

- The parent approach utilizes Genode's concept of hierarchical creation of processes including the hierarchical provision of resources.
- The incremental checkpointing mechanism uses page faults to mark the accessed memory regions. Page faults are a concept from the Memory Management Unit (MMU). Thus, the target operating system/microkernel has to support an MMU.
- The microkernel has to provide functionality to compare capabilities from own capability space to the capability space of a target process. Furthermore, it has to provide a functionality to manipulate the capability space of an arbitrary process. These mechanisms are used for accessing the capability space during checkpointing and restoring.

7. Implementation

This chapter gives an overview of the system, and depicts implementation solutions of the design chapter.

7.1. Programming Environment

The implementation of Rtcr is written in C++11. It targets Genode 16.08 with the Fiasco.OC kernel and is meant to run on an ARM chipset like PandaBoard ES. It is developed and tested on Ubuntu 16.04 with Qemu. For these tasks, Genode provides a convenient build system to leverage the development of new Genode components [37]: It provides g++/gcc cross-platform compiler, and a run script system to automate the start of compiled Genode OS instances, composed of chosen components, in Qemu.

The Rtcr component is developed to be agnostic on which CPU core it runs. The target and Rtcr component can run on the same CPU as well on different. During the checkpointing phase the target is stopped and the memory is checkpointed. Hereafter, the target is resumed and is allowed to access its memory regions again. Furthermore, Rtcr is designed to support an extension which allows the parallel execution of Rtcr and target threads, for example when using a multi-core approach.

7.2. Overview

The main purpose of the Rtcr component is to monitor the state of a child and periodically checkpoint its state onto a separate memory location. Figure 7.1 shows the structure of Rtcr. The *Target Child* represents the target process. It offers an interface to start, pause, and resume. It requires RPC objects to fulfill its tasks. The *Intercepting* component offers these RPC objects. Besides their native tasks, e.g. a RAM session RPC object offers dataspace RPC objects, they intercept and monitor the method invocation initiated by its interface user. The Intercepting component requires a storage to save the monitored data. The *Online Storage* and the *Offline Storage* provide interfaces to retrieve and store data. The former is meant for the Intercepting component and stores the live values of RPC objects. The latter is for the Checkpointer/Restorer and stores process' data from a certain time (i.e. from last checkpoint). It is basically organized in lists

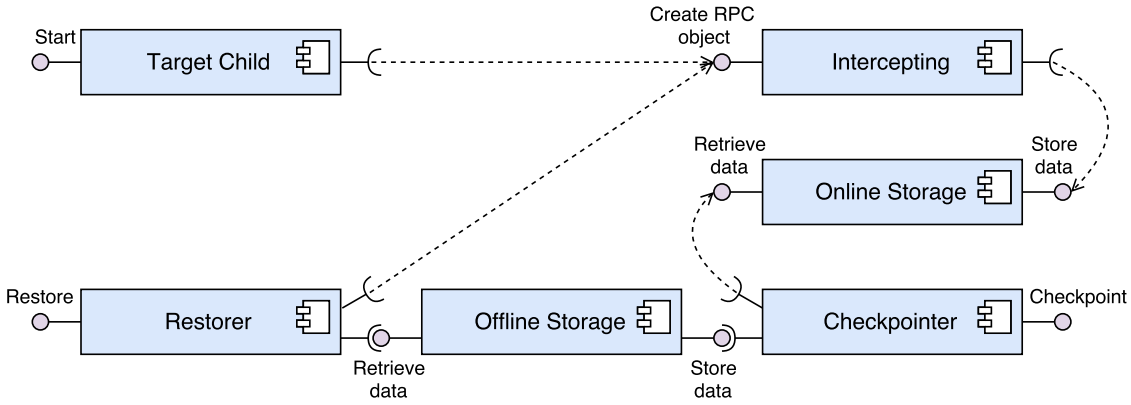


Figure 7.1.: Overview of Rtcr

containing session RPC objects which in turn contain lists of normal RPC objects. The *Checkpointer* provides an interface to checkpoint a Target Child. It basically processes data from the Online Storage and stores them in the Offline Storage. The *Restorer* provides an interface to restore a process from a stored state. Therefore, it requires the data from the Offline Storage and access to create RPC objects, which is provided by the Intercepting component.

The next sections provide a deeper understanding of these components.

7.3. Target Child

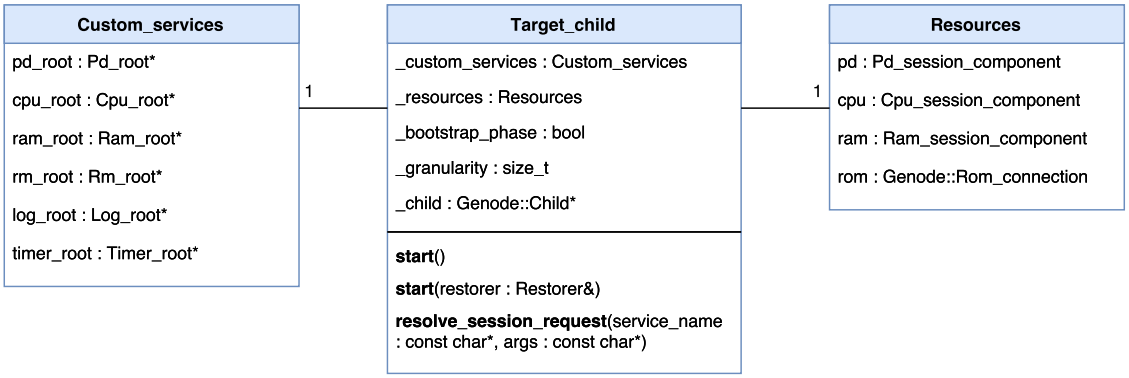


Figure 7.2.: Structure of Target_child

The `Target_child` class represents the child process of the Rtcr component. Figure 7.2 illustrates its member attributes. The Online Storage is the `_custom_services` object and

it contains all services, which shall be intercepted and monitored by Rtcr, in form of valid root RPC object pointers. `Pd_root`, `Cpu_root`, and `Ram_root` are mandatory, because the child requires a session from each of them at its creation time. These sessions are encapsulated in `_resouces`, which is an object of the class `Resources`. Together with the ROM session they form a group of mandatory sessions which are needed to startup/bootstrap a child. The ROM session is not required to be intercepted by Rtcr, because the child needs only its dataspace containing the compiled source code.

The `_bootstrap_phase` member is an indicator showing whether the child finished its bootstrap/startup phase. It is used in the restore phase, where a child is bootstrapped with Genode's native startup code using `Genode::Child class`. `Genode::Child` creates RPC objects for the child, which shall be reused in the Rtcr's restore mechanism. The `_bootstrap_phase` member allows to identified already created RPC objects and to restore their state from the checkpointed data. Therefore, each root RPC object, its session RPC objects and normal RPC objects get a reference to this indicator. Whenever an RPC object is created the Intercepting component passes the indicator which is stored in a constant member variable (see figure 7.4).

When a `Target_child` is created, the `_bootstrap_phase` indicator is set to true. It is set to false, when the first LOG session request arrives at Rtcr through the method `resolve_session_request`. A LOG session request is triggered whenever the child issues a log method for the first time. Therefore, I have inserted a log method invocation at the end of the bootstrap phase. The `resolve_session_request` method native task is to find a session for child's request. This method returns a `Genode::Service` object which provides access to a custom root RPC object. The root RPC object creates a custom session RPC objects through its `session` method and the capability of the session RPC object is passed to the child.

The `_granularity` attribute is a value for the incremental checkpointing mechanism (see section 7.5). It specifies a quantity of pages which are grouped together to form a monitoring unit. When the child accesses one byte of this group, the whole group is marked for checkpointing. If `_granularity` is zero, then no pages are monitored and, thus, the incremental checkpointing mechanism is disabled.

Finally, the `_child` attribute is a pointer to a `Genode::Child` object which performs the bootstrap/startup task and which represents the child component. When a `Genode::Child` object is created, it creates a child and automatically starts it. To decouple the starting time from the creation of a `Target_child` object, the pointer is initialized to a nullptr by the constructor of `Target_child`. When the `start()` or `start(Restorer&)` method is invoked, a `Genode::Child` object is created and `_child` is assigned to it.

7.4. Intercepting and Monitoring Child's RPC Objects

`Target_child` creates the *root RPC objects* for child's session requests. In turn, the root RPC objects create *session RPC objects* which create *normal RPC objects*. All three types can have a state which has to be checkpointed. The state of a root RPC object is a list of created session RPC objects. On the contrary, the state a session RPC object varies a lot between different session types. Usually, the session RPC object offers the creation of normal RPC objects. A part of its state is a list of its created RPC objects. In addition, session RPC objects can have further attributes which belong to its state, e.g. a `Signal_context_capability` for signal reception.

There are two types of normal RPC objects. One type has to be intercepted to monitor its method invocations which change the state of that object. The other type does not provide methods which change its state (e.g. a dataspace RPC object provides methods to only read the state), thus, it does not need to be intercepted. Its state is constant after the creation. Nevertheless, the capability, creation arguments and the return value has to be checkpointed by a simple storage object.

7.4.1. Intercepting a PD Session

Figure 7.3 depicts an example of how PD sessions are intercepted and monitored. The interception of other sessions works analogous to the PD session. The `Pd_root` class derives from `Genode::Root_component` and is implicitly an RPC object. But, the child does not know the existence of these objects, because it has no capability to it. The `Pd_root` class contains a reference to the `_bootstrap_phase` indicator of `Target_child`, which is simply passed to a `Pd_session_component` to mark its creation time and to also use it to mark the normal RPC objects it creates. The `Pd_root` has a list of `Pd_session_-components` which were requested by the child. It uses three methods which manipulate its list:

- A `Pd_session_component` is created by the `_create_session` method and the creation arguments are passed to the object which are stored in its `Pd_session_-info` object, named `_parent_state`.
- The `_upgrade_session` method extends the ram_quota of a `Pd_session_component` and is also stored in `_parent_state`.
- A `Pd_session_component` is closed by the `_destroy_session` method which removes the object from list of `Pd_root`.

A `Pd_session_component` is derived from `Genode::RPC_object` and is a session RPC object. It has a reference to the `_bootstrap_phase` indicator to mark normal RPC objects,

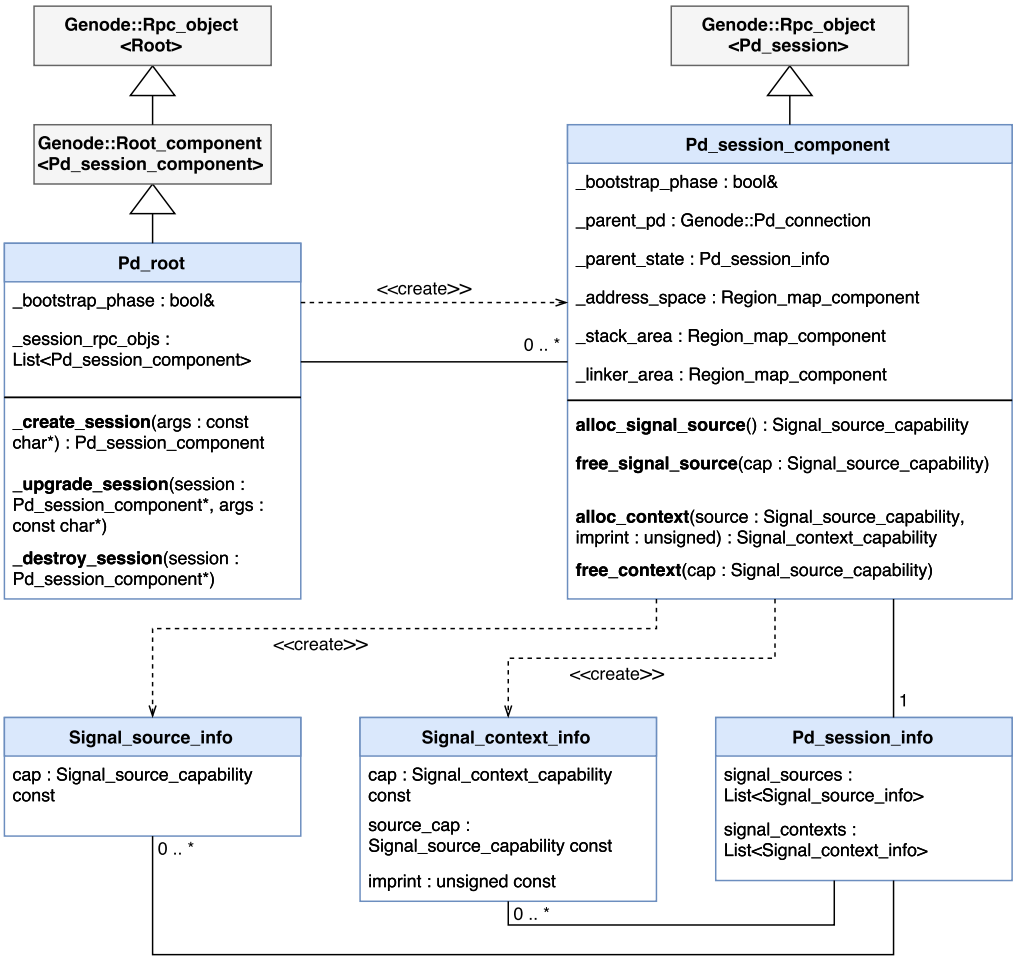


Figure 7.3.: Creation order of RPC objects related to the PD session (Region maps are omitted due to simplicity)

like Signal_contexts or Signal_sources. It has a Genode::Pd_connection attribute called _parent_pd which is used to implement the session interface of a PD session. The parent PD session is requested throught the parent component of Rtcr, and is usually the init component which returns a session from core. It is used to realize the functionality of Pd_session_component's methods: For example, when the child invokes the alloc_context method, the method implicitly invokes the alloc_context method of the parent PD session to create a Signal_context RPC object. Additionally, this method stores the passed parameters and the return value of _parent_pd.alloc_context in the list signal_contexts in _parent_state. A Pd_session_component inhabits 3 Region_map_components which are custom normal RPC objects used to intercept the attach and detach methods. For simplicity, they are not described here. They are intercepted like a PD_session_component.

A Pd_session_component has to implement the interface of a PD session. In figure 7.3 are four methods depicted which belong to the interface. They create and destroy normal RPC objects, namely Signal_context and Signal_source. In an alloc method an RPC object (e.g. Signal_context) is created and information about its state (in a Signal_context_info) is stored in a list residing in Pd_session_info (signal_contexts). Complementary, a free method destroys an RPC object and the corresponding list element is removed from its list. A Signal_context or Signal_source are RPC objects which do not need to be intercepted, because their state is constant after their creation. Thus, the alloc methods return the capability from _parent_pd, instead of the capability of the intercepted RPC object. An example where the capability of an intercepted RPC object is returned is the address_space method of Pd_session_component (not depicted in figure 7.3). This method returns a capability of an Region_map_component managed by Rtcr.

7.4.2. Storing the State of a PD Session in the Online Storage

The interception of a PD session requires different types of Online Storage classes. Figure 7.4 illustrates what information are stored in a Pd_session_info, Signal_context_info, and Signal_source_info. All attributes which are denoted with a const qualifier are attributes which do not change and are usually the creation arguments. All three classes have one property in common: They have to store the time, when they were created, i.e. during or after the bootstrap phase. This is realized in General_info, from which all Online Storage classes inherit. The Session_rpc_info stores the creation and upgrade arguments, because a session RPC object was created and possibly upgraded by an argument string. The Pd_session_component contains, like all other session RPC objects, a member attribute (i.e. _parent_state) of a type which inherits the Session_rpc_info class.

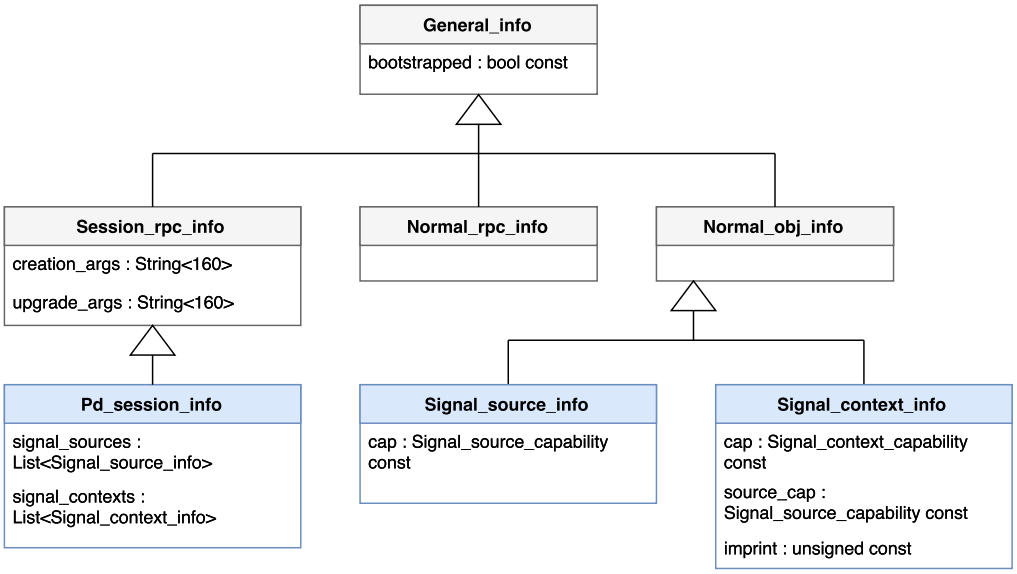


Figure 7.4.: Structures for storing PD session's online state

As mentioned earlier, a normal RPC object is categorized into an intercepted and non-intercepted RPC object. The former is represented by a `Normal_rpc_info` and the latter by a `Normal_obj_info` class. Right now, they do not contain any attributes, but can be extended, if desired. The normal RPC objects `Signal_context` and `Signal_source` are not intercepted and thus restored in a class derived from `Normal_obj_info`. Both have to store the parent's capability which the child component is aware of. Contrary to `Normal_obj_info` and equivalent to a `Session_rpc_info`, a `Normal_rpc_info` belongs to an intercepted RPC object and thus, is implemented by a `*_component` class. For example, a `Region_map_component` contains an object of the type `Region_map_info` which is derived from `Normal_rpc_info`. A `Normal_rpc_info` does not need to store the capability of its parent, because the capability is implicitly stored in each `*_component` by deriving from `RPC_object`. For a summary, table A.1 contains a mapping of RPC objects and specifies which storage type they need, and whether they are intercepted.

7.4.3. Storing the State of a PD Session in the Offline Storage

When checkpointing the RPC objects, the information of the Online Storage objects and the capability of the RPC object has to be processed and stored into Offline Storage objects. Figure 7.5 depicts the inheritance tree of the Offline Storage classes for the example with a PD session. Like in the Online Storage `const` attributes usually denote creation arguments in the Offline Storage. The class `Stored_general_info` stores information

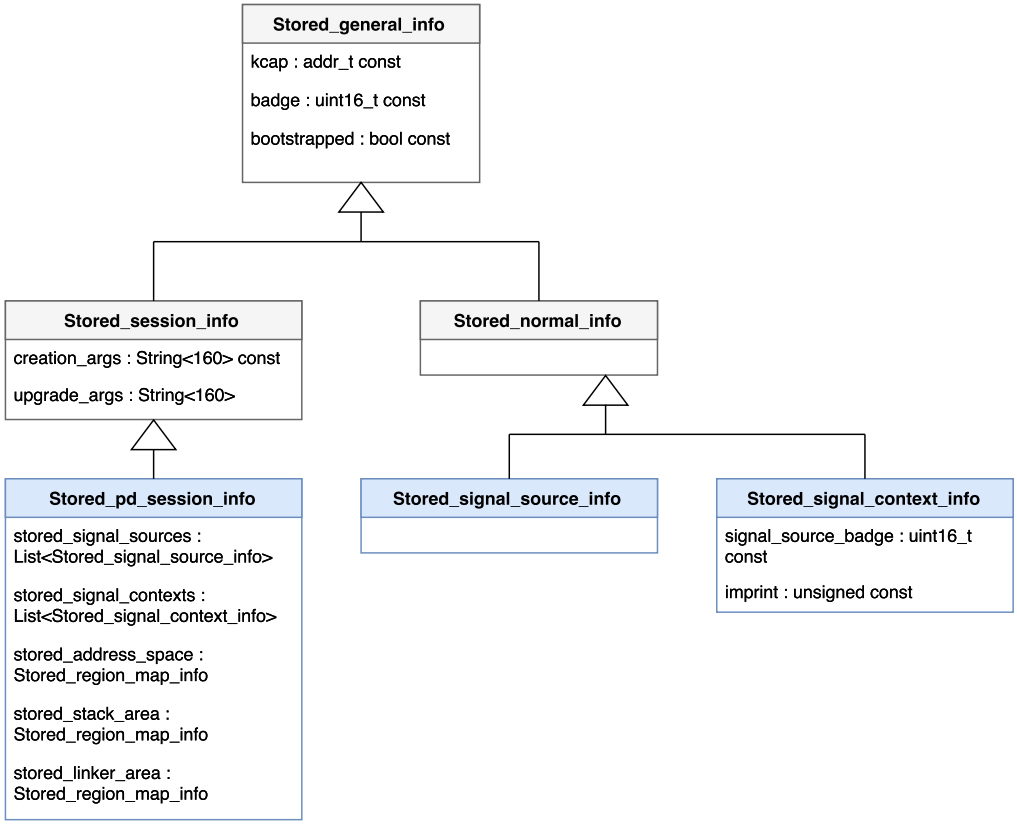


Figure 7.5.: Structures for storing PD session's offline state

which is common to all stored RPC objects: The kcap address, the badge identifier, and the bootstrapped indicator. Fiasco.OC defines a specific format for the kcap address: The first 20 bits specify the index of the capability space where the capability resides. The last 12 bits are reserved for IPC calls. The badge identifier is a system global identifier of an RPC object. It serves as an identifier of RPC objects during the checkpoint and restore phases identify RPC objects referenced by other RPC objects. The bootstrapped indicator is required by the restore mechanism and is described in section 7.7.

The Offline Storage classes do not differentiate between intercepted and non-intercepted normal RPC objects, because both classes do not have to store the capability of the RPC object. Actually, the capability is processed into a kcap and badge component. Therefore, there is one class for session RPC objects and one for normal RPC objects.

`Stored_session_info` class contains the creation and upgrade arguments which are directly copied from `Session_rpc_info`. The `Stored_pd_session_info` inherits from this class. It contains two lists for `Stored_signal_source_infos` and for `Stored_signal_context_infos`. Additionally, it has to store the information about the three `Region_map_components` of a PD session in `Stored_region_map_info` objects.

The `Stored_normal_info` class is meant for normal RPC objects. It does not contain any attributes, but can be extended, if desired. A `Signal_source` is stored in `Stored_signal_source_info`. The only information about this RPC object are the three attributes from `Stored_general_info`, because it is created without parameters. A `Stored_signal_context_info` stores a `Signal_context`. It contains a badge identifier of a `Signal_source`, which is used to reference a `Stored_signal_source_info`, and the `imprint` argument, which was used to create a `Signal_context` RPC object.

Analogous to the PD session, other services can be intercepted, monitored, and stored in this way.

7.5. Incremental Checkpointing Mechanism

The incremental checkpointing mechanism is an optimization to the normal checkpointing approach. It marks accessed regions in managed dataspaces which will be stored in the next checkpointing phase. The design chapter describes the functionality of the incremental checkpointing mechanism in section 6.2.2. This section will describe the class structure of the implementation of this mechanism in Genode.

Figure 7.6 depicts the involved classes. The `Ram_session_component` plays a central role, because it provides RAM dataspaces to the child. Like other intercepted session RPC objects, it has a reference to the bootstrap phase indicator, a parent RAM session, and a parent state. In addition, it has an attribute `_parent_rm` which is a connection to a parent RM session. With an RM session, this class can create region maps and return their

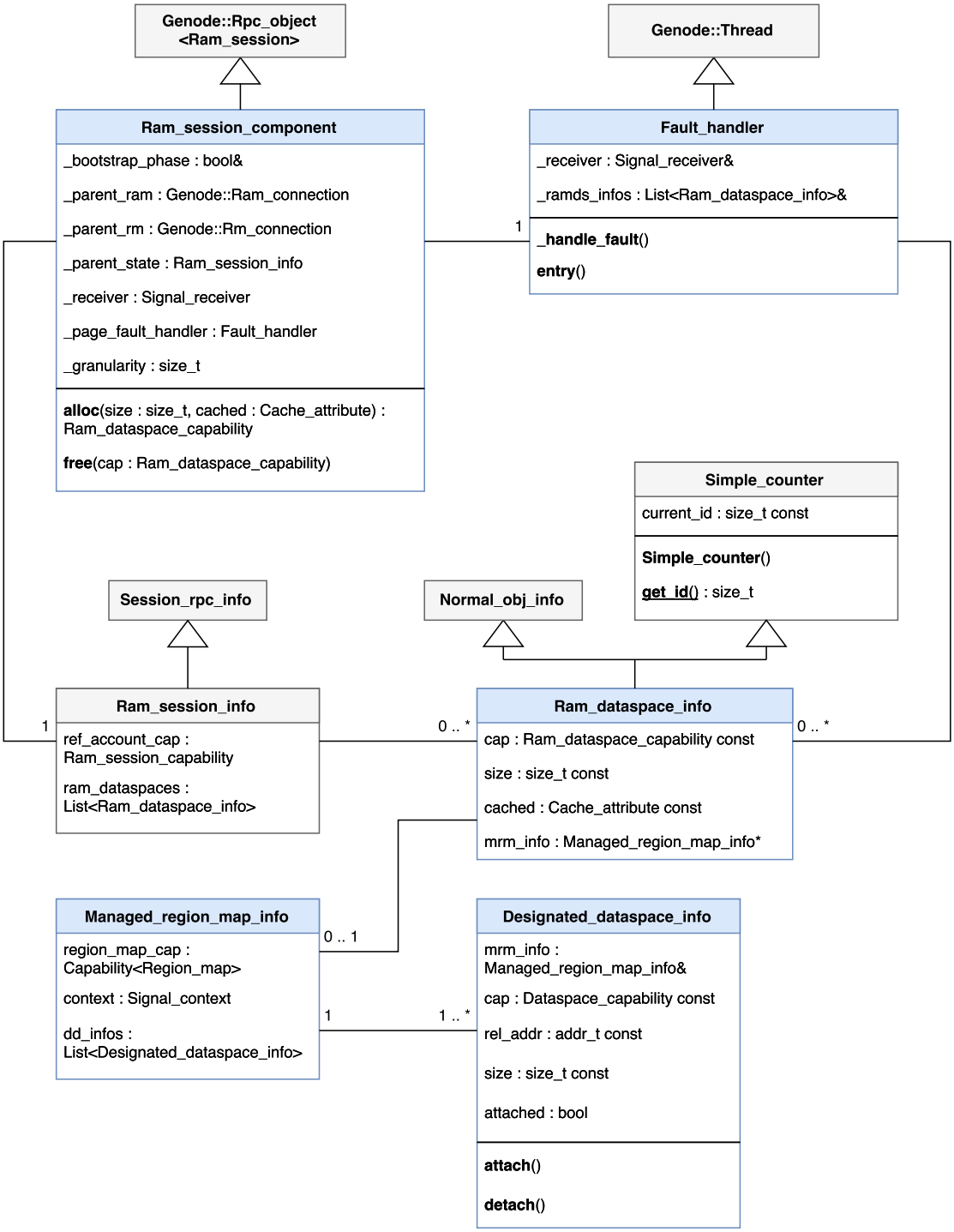


Figure 7.6.: Structure of the incremental checkpointing mechanism

dataspace capabilities to the child process in the `alloc` method instead of capabilities of real dataspaces. Furthermore, it has an attribute `_granularity` which denotes the granularity of the incremental checkpointing mechanism. It is also responsible for creating a thread which can handle page faults for its created region maps. Therefore, the page fault handler requires a `Signal_receiver` object, called `_receiver`, which will receive page fault signals. With the parent RAM and RM session connection, the granularity, and the page fault handler with its `_receiver` object, the `Ram_session_component` can create a region map and fill it with dataspaces of the size of $\_granularity * Pagesize$. It can detach the dataspaces from the region map to cause a page fault signal and attach the dataspaces to resolve the page fault.

The `alloc` method provides the child process with RAM dataspaces. If the Rtcr component is configured to not use the incremental checkpointing mechanism ($\_granularity = 0$), this method returns capabilities of real dataspaces allocated from the parent RAM session and it stores a `Ram_dataspace_info` with a `nullptr` to a `Managed_region_map_info` in `_parent_state`. Otherwise ($\_granularity > 0$) the `alloc` method creates a region map from the parent RM session and a `Ram_dataspace_info` object whose `cap` attribute is set to the dataspace capability of the parent region map. In the next step, the `alloc` method creates a `Managed_region_map_info` whose `region_map_cap` attribute is set to the capability of the created parent region map and whose `_context` attribute is associated to the `_receiver` object of a `Ram_session_component`. Furthermore, it has a list of `Designated_dataspace_infos` which represent the dataspaces with the size of $\_granularity * Pagesize$ which are designated to specific, non-overlapping locations in the region map. The `attached` attribute of a `Designated_dataspace_info` object denotes whether its dataspace (referenced by the `cap` attribute) is attached to the region map. The methods `attach` and `detach` are self-explanatory and implicitly change the attribute `attached`.

A `Ram_dataspace_info` class represents a non-intercepted RPC object, thus, it is derived from `Normal_obj_info`, and also has a constant timestamp value for each object which is realized by inheriting from the `Simple_counter` class. `Simple_counter` has a constant attribute `current_id` which created with the default constructor. The default constructor uses the method `get_id` which contains a static variable which is incremented every time it is called. The method returns the value of the static variable. Thus, every time a `Ram_dataspace_info` object is created, it has a unique timestamp which is used for ordering the creation of allocated RAM dataspaces. The timestamp is required for the restore phase, where the restore mechanism needs to identify bootstrapped RPC objects (of `Genode::Child`) with a stored RAM dataspace RPC object, because the bootstrap mechanism creates several RAM dataspaces with the same creation arguments. This mechanism helps to differentiate between bootstrapped RAM dataspaces and to associate each of them bidirectionally to one stored RAM

dataspace.

Finally, the `free` method of `Ram_session_component` destroys all management objects, if the incremental checkpointing mechanism is enabled, and frees the parent RAM dataspace. Or it simple frees the dataspace, if the incremental checkpointing mechanism is disabled.

The `Fault_handler` class represents a thread and has a reference to the `_receiver` object of the `Ram_session_component`. It has also a reference to the list of allocated `Ram_dataspace_infos`. This thread executes its `entry` method, which waits for signals to arrive from `_receiver`. When a signal arrives it calls `_handle_fault` which iterates through the `Managed_region_map_info` of each `Ram_dataspace_info` and queries each region map for its state. If the state type is not `READY` (e.g. the type is `READ_FAULT` or `WRITE_FAULT`), it uses the page fault address from the state object to find a `Designated_dataspace_info`. This `Designated_dataspace_info` is attached to the region map and marked implicitly by setting attached. Afterwards, the checkpointing mechanism will only store the attached `Designated_dataspace_infos` and detaches them from the region map.

7.6. Checkpointing

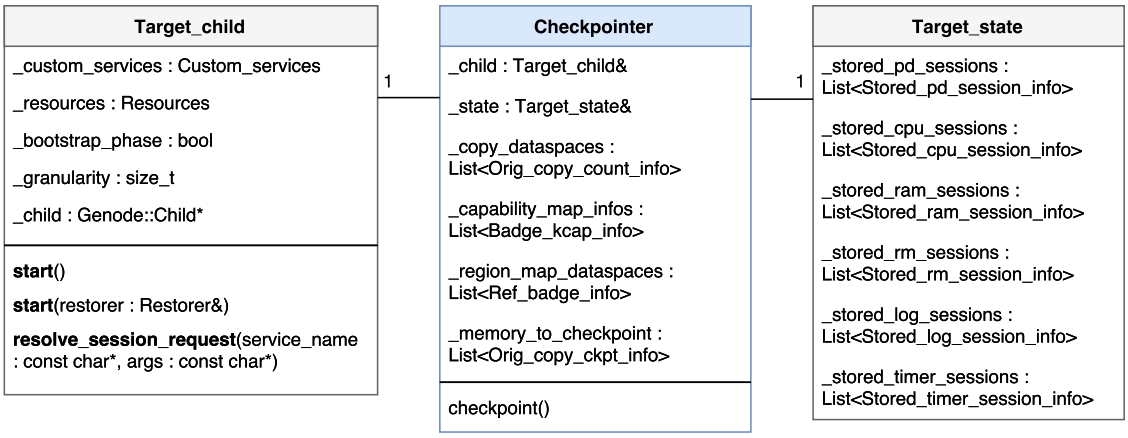


Figure 7.7.: Checkpointer stores online data into the Offline Storage

The checkpointing mechanism stores the volatile state of the child process to a persistent storage. The volatile state is the state of RPC objects held in the Online Storage, the persistent storage is the Offline Storage called `Target_state`. Figure 7.7 illustrates this relationship. The Checkpointer uses supplementary lists to support the checkpointing phase. The most important is the `_copy_dataspaces` list which stores

mappings of a child's dataspace and a corresponding stored dataspace. This list has to be managed between checkpoints to reduce allocating and freeing dataspaces from the RAM session and to support the incremental checkpointing mechanism by updating each stored dataspace whose corresponding child's dataspace was accessed. The other three lists play a supplementary role. They are created at the beginning and during the checkpointing phase and destroyed at the end.

- `_capability_map_infos` stores tuples of a kcap and a badge of an RPC object. This information is used to assign a capability space slot to a stored RPC object.
- `_region_map_dataspaces` is a list containing only badges which correspond to the badges of dataspace capabilities of region maps known to the child. It is used to identify region maps which are attached to other region maps. With this information the checkpointing mechanism will not copy the memory content of a region map falsely, because its attached dataspaces will be checkpointed directly.
- `_memory_to_checkpoint` is identical to `_copy_dataspaces`, but it has information whether the dataspace was already checkpointed to allow different threads to checkpoint the memory content. And it supports managed dataspaces from the incremental checkpointing mechanism, where `_copy_dataspaces` is not aware of (more information in section 7.6.2).

The checkpointing mechanism is divided into two parts: First, it has to interpret the capability map of the child and store the information. This information allows the restore mechanism to inject recreated capabilities into child's capability space at the right location, and allows also to adjust the capability map with badges of recreated RPC objects. Second, the state of RPC objects has to be stored.

7.6.1. Capability Map and Space

The capability map is a management structure for `Cap_index` pointers. A `Cap_index` holds a badge. It is stored in an array of `Cap_indexes`. The capability space slot can be determined by the relative position of a `Cap_index` in the array, because this array mirrors the capability space of Fiasco.OC. The array resides in one of the dataspaces attached to the address space of the child.

The start of the array is calculated by the address of an object which holds this array: The object is a static `Cap_index_allocator_tpl` object which can be retrieved by the global function `Genode::cap_idx_alloc`. To find out the position of the object, the child pro-actively propagates the information of the address to the native PD session of its PD session during the bootstrap/startup process. Therefore, the native PD session was extended by a getter and a setter method.

Listing 7.1 illustrates the process of creating the list of $(kcap, badge)$-tuples. Lines 5 to 11 compute the size of an array entry and also the start and the end of the array in Rtcr's address space. With this information the array can be traversed in lines 13 to 26: For each array entry, the badge is fetched and the kcap address is computed. In line 17 the badge is fetched by converting the address, plus an offset to the badge, into a pointer of 16 bit size and dereferencing it. In line 19 the kcap is computed by the difference from the start and current position of the array, divided by the array entry size to get the index of the array entry. Finally, this index is shifted by 12 bits to the left to obtain a valid kcap format (see section 7.4). Lines 21 to 25 check whether the badge is not invalid and not unused. If the check is positive, then a `Badge_kcap_info` object is inserted into the result list.

```
1  Genode::List<Badge_kcap_info> Checkpointer::_create_cap_map_infos()
2  {
3      Genode::List<Badge_kcap_info> result;
4      // Computing local address of the array
5      addr_t const cap_idx_alloc_addr =
           ↪ Foc_native_pd_client(_child.pd().native_pd()).cap_map_info();
6      size_t const array_ele_size = sizeof(Cap_index);
7      addr_t const child_ds_start = ar_info->rel_addr;
8      addr_t const local_ds_start = _state._env.rm().attach(ar_info->attached_ds_cap,
           ↪ ar_info->size, ar_info->offset);
9      addr_t const local_struct_start = local_ds_start + (cap_idx_alloc_addr -
           ↪ child_ds_start);
10     addr_t const local_array_start = local_struct_start + 8;
11     addr_t const local_array_end = local_array_start + array_size;
12
13     for(addr_t curr = local_array_start; curr < local_array_end; curr += array_ele_size)
14     {
15         size_t const badge_offset = 6;
16         // Fetching badge
17         uint16_t const badge = *(uint16_t*)(curr + badge_offset);
18         // Computing kcap
19         addr_t const kcap = ((curr - local_array_start) / array_ele_size) << 12;
20         // Inserting kcap, badge tuple into list, if the badge is valid
21         if(badge != UNUSED && badge != INVALID_ID)
22         {
23             Badge_kcap_info *state_info = new (_alloc) Badge_kcap_info(kcap, badge);
24             result.insert(state_info);
25         }
26     }
27     _state._env.rm().detach(local_ds_start);
28     return result;
29 }
```

Listing 7.1: Creating a mapping of kcaps and badges

7.6.2. RPC Objects

The basic idea of the checkpointing mechanism is to reuse Offline Storage objects (e.g. `Stored_pd_session_info`) from the last checkpoint. This synergizes well with the fact, that the child process does not destroy RPC objects often, thus, these objects can be reused without needing to allocate new memory from the heap. The consequence is, that new RPC objects (non-existent in the last and existent in the current checkpointing phase) have to be inserted to the Offline Storage. Also, old RPC objects (existent in the last and non-existent in the current checkpointing phase) have to be removed.

This considerations yield the following approach: For each RPC object in the Online Storage a stored RPC object is searched in the Offline Storage. If one is found, it is updated with new values. If none is found, a new stored RPC object is created and also updated. After that, each stored RPC object is traversed and a corresponding child RPC object is searched. If none is found, the old stored RPC object is remove and destroyed.

This process was called *preparing* an RPC object in this thesis. If an RPC object contains a list of created RPC objects (e.g. a RAM session contains a list of allocated RAM dataspaces), each RPC object in the list is also prepared. The prepare mechanism can be divided into 3 phases: Create, update, and destroy. When creating a new stored RPC object, a kcap value is searched by the badge value in `_capability_map_infos`. This value is passed to the stored RPC object. During the update phase the checkpointing mechanism gathers additional data about dataspaces, which are stored in an internal list. This list (`_copy_dataspaces`) contains all found dataspaces in the traversed RPC objects which is used to create a list of dataspaces to checkpoint. When destroying a stored RPC object which references a dataspace, the dataspace has to be removed from the `_copy_dataspaces` list, if and only if it is not referenced by any other RPC object. Therefore each list element contains a reference count which is decremented by each destroy phase. If the count reaches zero, then the list element including a dataspace is destroyed and the copy dataspace which contains the checkpointed memory content is freed.

After preparing the RPC objects, a list of dataspaces to checkpoint is created (`_memory_to_checkpoint`) from `_copy_dataspaces`. The next step resolves the managed dataspaces from the incremental checkpointing mechanism in `_memory_to_checkpoint`. This means, each entry representing a managed dataspace is removed, and new list elements for each marked designated dataspace are inserted.

Now, all attached designated dataspaces are detached which activates the marking mechanism of the incremental checkpointing mechanism. Furthermore, the dataspaces are checkpointed by using the information stored in `_memory_to_checkpoint` and all supplementary lists are cleaned up: `_capability_map_info`, `_memory_to_checkpoint`, and `_region_map_dataspaces`.

7.7. Restoring

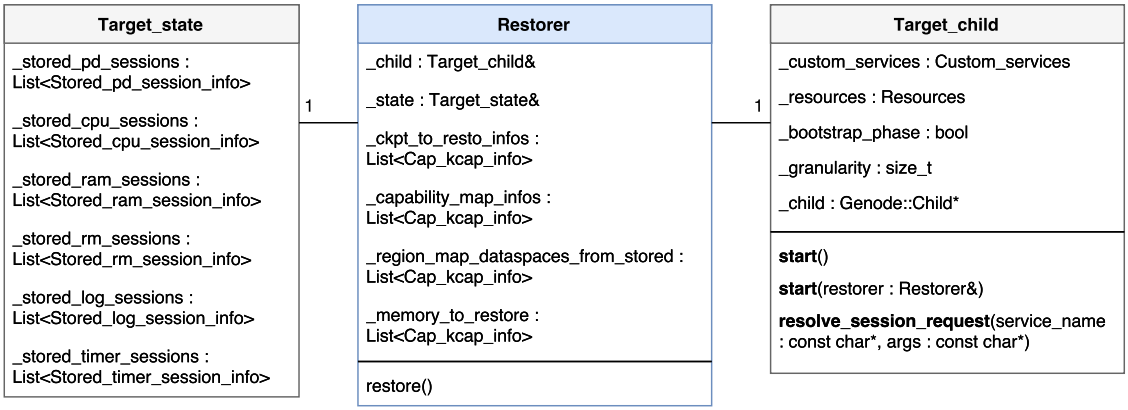


Figure 7.8.: Restorer identifies and recreates RPC objects and restores their state from `Target_state`

The restore phase has to recreate all RPC objects which are known to the checkpointed child through its capability space, and also restore the capability space and map. Contrary to the checkpointing phase, the restore phase has to restore the Online Storage from the Offline Storage. Figure 7.8 illustrates the usage of the `Target_state` object to recreate the RPC objects by the Restorer. The Restorer recreates the RPC objects by calling the RPC methods found in *Target_child::_custom_services*.

The restoration phase utilizes four supplementary lists. They are created in the `restore` method and destroyed at the end.

- `_ckpt_to_resto_infos` is like a dictionary which contains the badge of a stored RPC object and a capability of a corresponding, recreated RPC object. It is used to identify RPC objects which are referenced by stored RPC objects.
- `_capability_map_infos` contains a $(kcap, capability)$-tuples. The restore method uses it to insert a recreated capability into the capability space and map of the restored child.
- `_region_map_dataspaces_from_stored` is equivalent to the list `_region_map_dataspaces` in section 7.6. It stores the dataspace badges which refer to dataspace badges of region maps from the Offline Storage. It is used to identify region maps attached in other region maps when storing marking dataspaces for restoration.
- The list `_memory_to_restore` references destination and source dataspaces used for copying dataspace content. It associates a stored dataspace to a recreated

dataspace.

The restore mechanism is divided into three parts. The first part is Genodes's native bootstrap/startup phase which creates a new process and mandatory RPC objects. The second phase identifies and associates these RPC objects by stored RPC objects as well as recreates missing RPC objects, and restores the state of them. Finally, the restore mechanism restores the capability map and space by inserting missing badges and capabilities, respectively.

7.7.1. Bootstrap

As explained before, the child process is booted by the bootstrap/startup mechanism of Genode. The first child process which was created without a checkpoint state, was also bootstrapped by this mechanism. Because this mechanism yields the same output for the same input, the created RPC objects of the checkpointed child process will be the same as the one of the restored child process after the bootstrap. Furthermore, the restored child process has to be intercepted shortly after the bootstrap mechanism, thus, it does not execute its main function (`Component::construct`).

The main thread of the child process can be intercepted when it uses a log output function. This triggers a synchronous RPC function call to establish a LOG session which pauses the thread and causes it to wait for a response. In this time the recreation of the child process state is realized. Therefore, a Genode::log call is inserted into the code of the bootstrap mechanism. It is called when all mandatory RPC objects are created and just before the thread reaches the main function.

7.7.2. RPC Objects

Restoring child's RPC objects has to be realized in two steps. First, the bootstrapped RPC objects, which were created during the bootstrap phase, have to be identified by stored RPC objects. Usually, the bootstrapped RPC objects can be identified unambiguously by their creation arguments and the bootstrap indicator, e.g. a CPU thread is identified by the bootstrap indicator and its name. The RAM dataspaces require an additional tool, because the RAM session allocates several RAM dataspaces with the same creation arguments during the bootstrap phase. Thus, all arguments are the same. Therefore, a counter is used for each `Ram_dataspace_info`. This counter realizes a simple timestamp mechanism to order the creation sequence. With this mechanism a child's RAM dataspace can also be identified unambiguously to a stored RAM dataspace.

Another aspect of the first step is to recreate the RPC objects in the right order. The CPU session requires a PD session to create a thread, thus, all PD sessions have to be recreated before all CPU sessions.

During this step, the list `_ckpt_to_resto_infos` is filled, which contains translations from badges of stored RPC objects to capabilities of recreated RPC objects. Also, the list `_capability_map_infos` is filled when a new RPC object is recreated. The capabilities of these RPC objects have to be inserted into the capability map and space of the child process.

The second step restores the state of all recreated RPC objects of the child process. For each stored RPC object a corresponding recreated RPC object is found and its state updated. In this step, the _memory_to_restore list is created and afterwards all managed dataspaces from the incremental checkpointing mechanisms are resolved just like in the corresponding function of the Checkpointer (see section 7.6.2).

7.7.3. Capability Map and Space

Fiasco.OC provides an asynchronous mapping system call for capabilities. It allows to map a capability from one capability space to another. Because only core shall be aware of the underlying kernel, Rtcr does not use this method directly, but through the native PD session of the child which is managed by core.

An additional RPC method was implemented into the native PD session which takes a kcap and a capability. It maps the provided capability from core's capability space to the capability space to which the native PD session belongs. This means, the capability which was created by Rtcr is delegated to core through the RPC mechanism. Then, it is mapped from core to the target child.

The capability map is a management structure which lowers the read costs to the `Cap_index` array. This structure uses Genode's AVL tree which is balanced by the badge numbers. To restore the capability map, this management structure also has to be restored, thus, the child's threads can find the available capabilities. Because an AVL tree depends on the badges which are going to be replaced, the whole tree has to be re-balanced. This is not a trivial task and hinders the implementation of the restore mechanism. Thus, the AVL tree was replaced by a simple list provided by Genode. Now the nodes have only one pointer to the next node. This means, the position of the list elements is arbitrary and not ordered anymore.

When adjusting the checkpointed capability map, the bootstrapped capability map can be reused, because it already contains the badges of the bootstrapped RPC objects. Also, the bootstrapped RPC objects reside on the same locations as the stored RPC objects, because Genode's bootstrap/startup mechanism yields the same results as described in section 7.7.1.

The restore phase of the capability map is divided into the following subtask:

1. Find the dataspace of the child and the checkpointed dataspace containing the capability maps.

2. Attach both dataspaces into Rtcr's address space.
3. Compute the addresses of child's and checkpointed array.
4. Copy child's array into checkpointed array.
5. Insert badges of recreated RPC objects found in `_capability_map_infos` into the checkpointed array.
   a) Within the array, find a `Cap_index` with a valid next-node pointer and store its address into `previous_cap_index`. The `previous_cap_index` will be adjusted when inserting `Cap_indexes` corresponding to recreated RPC objects into the array.
   b) For each list element containing a `kcap` and a `capability` in `_capability_map_infos` do:
      i. Compute the address of the current `Cap_index` using the `kcap`. This `Cap_index` will be created.
      ii. Compute the address of the previous `Cap_index` using `previous_cap_index`.
      iii. Set the first 32 bit which represent the list pointer of current `Cap_index` to the address of the next `Cap_index` of `previous Cap_index`. The address is local to the address space of the child process.
      iv. Set the 16 bit which represent the badge of current `Cap_index` to the badge of the list element's `capability`.
      v. Set the 32 bit which represent the list pointer of previous `Cap_index` to the address of to current `Cap_index`. The address is local to the address space of the child process.

After the capability space and map are restored, the dataspaces from `_memory_to_restored` are copied and all supplementary lists are destroyed. Now Rtcr returns from the LOG session RPC function call to the child process with new RPC objects and new states of the RPC objects.

8. Conclusion

This chapter concludes the thesis by demonstrating the functionality of the implementation, discussing the limitations of the design, bringing ideas for future work, and by highlighting the contribution of the thesis.

8.1. Showcase

The implementation of the Rtcr component contains a simple showcase scenario which illustrates the fundamental functionality of the component. The scenario comprises a modified core component, a test-driver component for Rtcr's functionality, a Timer component, and a simple test component, called Sheep Counter. It is realized by a Genode run script file called `rtcr_restore_child.run` and is found in the GitHub repository of the operating system chair of the TUM[1].

The Sheep Counter is a simple program which creates a Timer connection, allocates a RAM dataspace, and uses its first 4 bytes for counting "sheep". It then goes into an endless loop, prints and increments the current sheep count each second.

The test-driver for Rtcr first creates a Sheep Counter process and starts it. It waits 3 seconds and checkpoints the Sheep Counter, but does not resume it afterwards. During the checkpoint the test-driver prints out diagnostic messages about the checkpoint progress. Finally, it creates a new component with the checkpointed state and lets the new child continue counting sheep.

Unfortunately, the new child does not start from where the first child ended. Instead, it starts the execution as if no restore phase was triggered. It seems that the manipulation of the instruction and stack pointer of the restored threads was not successful. This issue has to be debugged and the fault has to be corrected. Due to a fixed working time, it could not be realized before the end of this work. It is a task for future work.

8.2. Limitations

Throughout the conception and implementation phase several obstacles were encountered. They were solved by algorithms which have their benefits and also disadvantages.

[1] `https://github.com/702nADOS/genode-CheckpointRestore-SharedMemory`

This section briefly elaborates on the design decisions and additionally mentions unfinished secondary tasks belonging to this thesis.

Due to the limited implementation time, the serialization and deserialization module could not be implemented. This module is not necessary to show the basic functionality of a Checkpoint/Restore mechanism. To ensure real-time capability the serialization, migration, and deserialization has to be considered. In the end the priority was set to implement the Checkpoint/Restore functionality, and the serialization and deserialization has to be postponed to another work.

As described in the showcase section 8.1, the restore mechanism does not deliver expected results. In the end of the implementation, there was only little time to debug this issue, which did not suffice. There is also a discussion[2] regarding this issue with the developers of Genode on their Mailinglist [38]. The suggestion of Norman Feske, a developer and founder of Genode Labs, is to use a hidden functionality of Genode's bootstrap mechanism `Genode::Child` to bypass the automatic RPC creation and create all RPC objects manually. The advantage is, that the start of the threads is in the control of the Rtcr component, thus, a start from the stored instruction and stack pointers is possible. The current mechanism fails on changing the instruction and stack pointers after the threads were started.

The Rtcr component does monitor the RPC objects which are created outside of the child component. The child component has the ability to create its own RPC objects via the PD session. When a component provides a service to other components, it allocates capabilities from the PD session to provide access to the session RPC objects it created. Fortunately, restoring the capabilities of these session RPC objects is not required, because they are usually not used by the child, but by other components which do not exist on the same platform when the child is restored.

There is still one problem to be solved: The child creates an `Genode::Entrypoint` object, it creates a capability to a thread kernel object by the same method call as is used to create capabilities for session RPC objects. This capability has to be identified and restored to allow the child to maintain its service on the migrated node. Fortunately, Norman Feske proposed a solution to this problem[3]. Due to lack of implementation time this task belongs to the future work.

During the implementation of the incremental checkpointing mechanism a bug was found in the Fiasco.OC kernel[4]. A Genode scenario demonstrates the bug with the run script `concept_session_rm.run` in the implementation repository [39]. The bug seems to occur when pausing a target thread, detaching dataspaces from a region map attached in target's address space, resuming the thread, and then solving the page faults by attaching

[2] `https://sourceforge.net/p/genode/mailman/message/35529306/`

[3] Norman Feske, `https://sourceforge.net/p/genode/mailman/message/35413778/`

[4] Stefan Kalkowski, `https://sourceforge.net/p/genode/mailman/message/35391907/`

the dataspace back. This process goes through three cycles. After the second cycle, after the dataspace was detached and the thread was resumed, no page fault occurred. Stefan Kalkowski, a Genode developer, found out, that after detaching the dataspaces from the region map they were not not unmapped in the management structures of Fiasco.OC. He found a solution by using a newer kernel version. Kalkowski's used kernel was highly experimental and does not support Genode's functionality completely. Therefore, the kernel has to be patched for Genode or other mechanisms has to be found to support the incremental checkpointing mechanism.

The marking system of the incremental checkpointing mechanism cannot distinguish between read and write accesses, thus, it marks a memory region as modified on read and write accesses. The Genode OS Framework provides static dataspaces in regards to access protection. A read-write and read-only dataspace can be created, but one type of dataspace cannot be changed to the other one. By allowing the changeability, the incremental checkpointing mechanism could set all dataspaces of the target to read-only after a checkpoint. If the target writes one of the dataspaces, a page fault occurs which is resolved by setting the dataspace to read-write. Unfortunately, this mechanism imposes an extensive modification of Genode's interplay with the underlying kernel.

8.3. Future Work

The real-time capable Checkpoint/Restore project on which this thesis is based is not yet complete. This thesis could not process all its tasks and, furthermore, raises subsequent questions regarding the designed algorithms. This section presents further work which will contribute to the real-time capable Checkpoint/Restore project.

As described in the limitations section, due to limited processing time the implementation work on this thesis is not yet finished. The restore mechanism has to be debugged, the Fiasco.OC kernel has to be upgraded, and finally, the Checkpoint/Restore mechanism has to be extended to handle target's capabilities for service provision. After that, the Checkpoint/Restore mechanism can be evaluated for different types of processes. Among others, these types may differ in the amount of memory used, on different memory access patterns, e.g. frequent usage of same memory regions or chaotic usage of memory regions. For the KIA4SM project, the types of processes shall be chosen on the typical processes used in the automotive domain. Furthermore, work can be contributed to find an optimal granularity value for these types of processes. The rule is as follows: The higher the granularity, the lower the overhead of the incremental checkpointing mechanism, and the higher the overhead of the memory checkpointing.

Further work can be done to compare different checkpointing mechanisms with each other. The approaches introduced in the design chapter (see section 6.2.2) can be used for

that. Promising candidates are the mirror copy and multi-core with COW approaches. The three approaches can be compared in regards to real-time capabilities, the type and amount of overhead induced to the target component, and resource consumption.

In this thesis the parent approach was chosen to access the resources of the target component (see section 6.2.1). This approach can be compared to the *arbitrary component* approach whose IPC usage pattern differs vastly to the parent approach and to the *target component* approach, which does not add extra IPC usage, but extra overhead for intercepting the RPC methods. The parameter to compare to may be the amount of IPC calls, the resource consumption, and the induced overhead on the target component.

Another limitation of Genode, whose extension could improve the incremental checkpointing mechanism, is that it provides only the page fault mechanism to mark memory accesses. Other approaches like using the dirty bit of the MMU or implementing a software dirty bit could overcome the high overhead of using page faults.

8.4. Summary

This section concludes the thesis by giving a summary of the work and by highlighting its contribution. This work introduces Rtcr, a real-time capable Checkpoint/Restore component, implemented in user-space. Rtcr uses a parent/child approach to access the resources of the child through Genode's native mechanisms. This approach is also called shared resources approach, because all capabilities that a child is aware of are shared with the parent component. Rtcr guarantees and supports real-time capability by using mechanisms which are real-time capable and by optimizing the checkpointing mechanism, respectively. In particular, it implements the incremental checkpointing optimization to reduce the overhead during a checkpointing phase, by marking and checkpointing accessed memory regions using page faults.

This work elaborates on the requirements related to an embedded real-time system. It provides non-functional requirements of a real-time system and sets the focus on performance and scalability. Also, it names microkernel concepts whose state is subject to checkpointing and restoration: Process including address space, threads and their registers, IPC and scheduling parameters. Furthermore, the work presents different approaches to access the resources of a target component and discusses their real-time support. It elaborates also on state-of-the-art checkpointing optimizations by discussing them in the context of embedded real-time systems. At last, the thesis works on requirements needed for porting the presented design to another microkernel.

A. Tables

RPC objects	Storage type	Intercepted
PD session	Session RPC object	+
Signal context	Normal object	-
Signal source	Normal object	-
CPU session	Session RPC object	+
CPU thread	Normal RPC object	+
RAM session	Session RPC object	+
RAM Dataspace	Normal object	-
RM session	Session RPC object	+
Region map	Normal RPC object	+
LOG session	Session RPC object	+
Timer session	Session RPC object	+

Table A.1.: List of intercepted RPC objects

List of Figures

List of Tables

Bibliography

[1] J. S. Plank, M. Beck, G. Kingsley, and K. Li. *Libckpt: Transparent checkpointing under unix*. Computer Science Department, 1995.

[2] S. Eckl, D. Krefft, and U. Baumgarten. "COFAT 2015 - KIA4SM - Cooperative Integration Architecture for Future Smart Mobility Solutions." In: *Conference on Future Automotive Technology*. 2015.

[3] P. Schleiss, M. Zeller, G. Weiss, and D. Eilers. "SafeAdapt-Safe Adaptive Software for Fully Electric Vehicles." In: *3rd Conference on Future Automotive Technology, CoFAT*. 2014.

[4] F. Douglis. *Process migration in the Sprite operating system*. Tech. rep. DTIC Document, 1987.

[5] J. K. Ousterhout, A. R. Cherenson, F. Douglis, M. N. Nelson, and B. B. Welch. "The Sprite network operating system." In: *Computer* 21.2 (1988), pp. 23–36.

[6] K. Li, J. F. Naughton, and J. S. Plank. *Real-time, concurrent checkpoint for parallel programs*. Vol. 25. 3. ACM, 1990.

[7] K. Li, J. F. Naughton, and J. S. Plank. "Low-latency, concurrent checkpointing for parallel programs." In: *Parallel and Distributed Systems, IEEE Transactions on* 5.8 (1994), pp. 874–879.

[8] M. J. Litzkow. "Remote Unix: Turning idle workstations into cycle servers." In: *Proceedings of the Summer USENIX Conference*. 1987, pp. 381–384.

[9] M. J. Litzkow, M. Livny, and M. W. Mutka. "Condor-a hunter of idle workstations." In: *Distributed Computing Systems, 1988., 8th International Conference on*. IEEE. 1988, pp. 104–111.

[10] M. Litzkow and M. Livny. "Experience with the condor distributed batch system." In: *Experimental Distributed Systems, 1990. Proceedings., IEEE Workshop on*. IEEE. 1990, pp. 97–101.

[11] M. Litzkow and M. Solomon. "Supporting checkpointing and process migration outside the UNIX kernel." In: (1992).

[12] M. Livny and M. Litzkow. "Making workstations a friendly environment for batch jobs." In: *Workstation Operating Systems, 1992. Proceedings., Third Workshop on*. IEEE. 1992, pp. 62–67.

[13] M. Litzkow, T. Tannenbaum, J. Basney, and M. Livny. *Checkpoint and migration of UNIX processes in the Condor distributed processing system*. Computer Sciences Department, University of Wisconsin, 1997.

[14] D. S. Milojičić, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou. "Process migration." In: *ACM Computing Surveys (CSUR)* 32.3 (2000), pp. 241–299.

[15] J. C. Sancho, F. Petrini, K. Davis, R. Gioiosa, and S. Jiang. "Current practice and a direction forward in checkpoint/restart implementations for fault tolerance." In: *Parallel and Distributed Processing Symposium, 2005. Proceedings. 19th IEEE International*. IEEE. 2005, 8–pp.

[16] A. Maloney and A. Goscinski. "A survey and review of the current state of rollback-recovery for cluster systems." In: *Concurrency and Computation: Practice and Experience* 21.12 (2009), pp. 1632–1666.

[17] J. Duell. "The design and implementation of berkeley lab's linux checkpoint/restart." In: *Lawrence Berkeley National Laboratory* (2005).

[18] S. Osman, D. Subhraveti, G. Su, and J. Nieh. "The design and implementation of Zap: A system for migrating computing environments." In: *ACM SIGOPS Operating Systems Review* 36.SI (2002), pp. 361–376.

[19] G. Janakiraman, J. R. Santos, D. Subhraveti, and Y. Turner. "Cruz: Application-transparent distributed checkpoint-restart on standard operating systems." In: *Dependable Systems and Networks, 2005. DSN 2005. Proceedings. International Conference on*. IEEE. 2005, pp. 260–269.

[20] S. Groesbrink. "Adaptive Virtual Machine Scheduling and Migration for Embedded Real-Time Systems." University of Paderborn, 2015.

[21] C. L. Liu and J. W. Layland. "Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment." In: *J. ACM* 20.1 (Jan. 1973), pp. 46–61. ISSN: 0004-5411. DOI: 10.1145/321738.321743.

[22] D. Vogt, A. Miraglia, G. Portokalidis, H. Bos, A. Tanenbaum, and C. Giuffrida. "Speculative Memory Checkpointing." In: *Proceedings of the 16th Annual Middleware Conference*. Middleware '15. Vancouver, BC, Canada: ACM, 2015, pp. 197–209. ISBN: 978-1-4503-3618-5. DOI: 10.1145/2814576.2814802.

[23] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. "Live Migration of Virtual Machines." In: *Proceedings of the 2Nd Conference on Symposium on Networked Systems Design & Implementation - Volume 2*. NSDI'05. Berkeley, CA, USA: USENIX Association, 2005, pp. 273–286.

[24] M. Mitchell. *An introduction to genetic algorithms*. MIT press, 1998.

[25] *Checkpoint/Restore in Userspace*. [Online; accessed 31-May-2016]. 2016. URL: https://criu.org/Main_Page.

[26] D. Werner. "Porting an existing linux-based Checkpoint/Restore Mechanism (CRIU) to L4 Fiasco.OC/Genode." Technical University of Munich, 2016.

[27] J. Kulik. "Client-independent Checkpoint/Restart of L4Re-server-applications." Dresden University of Technology, 2015.

[28] J. J. Labrosse. *MicroC/OS-II: The Real-time Kernel*. 2nd. USA: CMP Media, Inc., 2002. ISBN: 978-1-57820-103-7.

[29] G. Rose. *Using the Microprocessor MMU for Software Protection in Real-Time Systems*. [Online; accessed 11-Nov-2016]. URL: http://pcbs13.informatik.uni-stuttgart.de/~lagally/ifi/bs/lehre/osp/MMU4rel.pdf.

[30] Q. Li and C. Yao. *Real-Time Concepts for Embedded Systems*. 1st. Boca Raton, FL, USA: CMP Media, Inc., 2003. ISBN: 978-1-57820-124-2.

[31] J. Liedtke. "On Micro-kernel Construction." In: *Proceedings of the Fifteenth ACM Symposium on Operating Systems Principles*. SOSP '95. Copper Mountain, Colorado, USA: ACM, 1995, pp. 237–250. DOI: 10.1145/224056.224075.

[32] A. Tanenbaum and A. Woodhull. *Operating Systems: Design and Implementation*. Prentice-Hall software series. Pearson Prentice Hall, 2009. ISBN: 9780135053768.

[33] J. Liedtke. "Improving IPC by Kernel Design." In: *Proceedings of the Fourteenth ACM Symposium on Operating Systems Principles*. SOSP '93. Asheville, North Carolina, USA: ACM, 1993, pp. 175–188. ISBN: 0-89791-632-8. DOI: 10.1145/168619.168633.

[34] N. Feske. *Genode Operating System Framework 16.05 Foundations*. [Online; accessed 14-Oct-2016]. May 2016. URL: http://genode.org/documentation/genode-foundations-16-05.pdf.

[35] ARM Limited. *big.LITTLE Technology*. [Online; accessed 07-Dec-2016]. URL: https://www.arm.com/products/processors/technologies/biglittleprocessing.php.

[36] M. E. Staknis. "Sheaved Memory: Architectural Support for State Saving and Restoration in Pages Systems." In: *SIGARCH Comput. Archit. News* 17.2 (Apr. 1989), pp. 96–102. ISSN: 0163-5964. DOI: 10.1145/68182.68191.

[37] Genode Labs. *The Genode build system*. [Online; accessed 03-Dec-2016]. URL: http://genode.org/documentation/developer-resources/build_system.

[38] Genode Mailinglist. *genode-main*. [Online; accessed 09-Dec-2016]. URL: https://sourceforge.net/p/genode/mailman/genode-main/.

[39] D. Huber. *Repository of the Rtcr component*. [Online; accessed 08-Dec-2016]. URL: https://github.com/702nADOS/genode-CheckpointRestore-SharedMemory.