DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Restoring of Memory Areas and Threads as Part of a Checkpoint/Restore Mechanism based on the L4 Fiasco.OC and the Genode OS Framework

Simon Himmelbauer

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Information Systems

Restoring of Memory Areas and Threads as Part of a Checkpoint/Restore Mechanism based on the L4 Fiasco.OC and the Genode OS Framework

Wiederherstellung von Speicherbereichen und Threads als Teil eines Checkpoint/Restore-Mechanismus auf Basis des L4 Fiasco.OC und des Genode OS Framework

Author:	Simon Himmelbauer
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	David Werner, M.Sc.
Submission Date:	10/15/2020

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

Munich, 10/15/2020 Simon Himmelbauer

Acknowledgments

I want to thank my advisor David Werner for his reliability and readiness to provide feedback at any time. I especially appreciate David for allowing me to freely decide on the direction of this thesis, while still giving helpful advice when needed. I also want to thank both David and Prof. Dr. Uwe Baumgarten for offering this demanding and interesting topic.

Furthermore, I would like to value the Genode community for their swift assistance on the Genode mailing list.

Needless to say, I want to thank my parents, brother, friends and my girlfriend for supporting me all the time.

Last but not least, thank you to mother Earth for bringing coffee to this land.

Abstract

The chair of operating systems at TU Munich has proposed to run safety-critical applications on general-purpose computers and implement redundancy by migrating the software during execution onto another machine. This concept is named Real-time Checkpoint/Restore (RTCR) and is based on Genode, a framework for developing secure isolated applications. Over the course of this research project, several functional and non-functional aspects have been investigated, especially in regards to checkpointing. The restore mechanism was often not in the scope of previous theses.

Thus, this thesis focuses on extending RTCR with the capability of recreating and resuming a process based on a checkpoint. The implementation lets Genode handle the bootstrap of a child process. Dataspaces are restored during allocation, while the *ep*-thread is explicitly reset by notifying RTCR via an RPC-call once bootstrapping the component has been completed.

Moreover, additional test cases have been constructed to evaluate the performance in different scenarios. These tests cover a variety of memory allocation strategies. Most notably, the relative overhead has been identified to be at around 77 % on average.

Finally, the architecture is overhauled by enabling easier adaptations to the design in the future.

Contents

1. Introduction

Today's automobiles contain numerous computers, also known as electronic control units (ECU). These modules are often specialized boards designed for a single task, including safety-critical ones, such as opening the air bag or the anti-lock braking system (ABS).

However, in recent years, cars have been equipped with more advanced driver-assistance systems (ADAS) and they are likely to become even more important in the future. Examples are adaptive cruise control (ACC), blind spot monitoring and especially, autonomous driving. These systems usually consist of many different and complex software components. However, passengers must still be able to rely on their functionality. One idea to implement redundancy can be achieved by installing multiple devices providing the same functionality. It goes without saying that cost efficiency and electricity consumption suffer from the fact that more and more types of ECUs are being installed.

Thus, the chair of operating systems proposed a different approach: Instead of using many simple but specialized modules, a few general purpose computers handle multiple tasks. Thanks to Genode, a framework for developing contained and secure programs, processes cannot interfere with the operation of others, even when an application malfunctions (e.g. leaking memory). However, a continuous operation of critical tasks is still necessary. If one computer node fails, another one must take over. Therefore, a process including its current state (memory, threads, etc.) must be migrated onto another node. This kind of functionality is already available on Linux (See chapter 3) but not within the Genode Framework. Therefore, the project KIA4SM was started with the goal to develop a Real-time Checkpoint/Restore (RTCR) system for Genode. [11]

While the checkpoint-phase has been thoroughly investigated by previous work, this thesis attempts to extend RTCR with a restore mechanism. In addition, the current architecture was found to be unsophisticated for future modifications, which must be expected due to the dynamic nature of research. Moreover, some aspects of the design were found to be unnecessarily complicated. This should be avoided in order to allow future students to more quickly become familiar with RTCR as the time span for a thesis is usually limited.

The following goals were specified for this thesis:

- Simplifying the current architecture
- Restoring a single dataspace
- Restoring all mandatory dataspaces created during the bootstrap
- Restoring a single thread (stack and instruction pointer)
- Restoring additionally allocated dataspaces (e.g. by a heap allocator)

1.1. Motivation

As mentioned above, the checkpointing part of RTCR has been thoroughly investigated at the chair of operating systems over the last couple of years [19, p. 67]. However, not many theses implemented a restore mechanism because necessary components to restore a child have not been identified yet [10, p. 59]. Denis Huber was able to develop a restore mechanism but critical bugs preventing full functionality could not be fixed [8, p. 61-62].

Therefore, this bachelor's thesis focuses exclusively on the restore side. Nevertheless, the research spent on checkpointing lead to a more stable functionality. Johannes Fischer's contribution where he merged previous efforts on RTCR into a modular architecture was particularly helpful [9]. This allowed this thesis to build on top a solid foundation and theoretically spend less time on the checkpoint-architecture.

1.2. Outline

This thesis starts with an introduction in chapter 2 to the Genode Framework and microkernels, specifically the Fiasco.OC-microkernel used. Next, the work accomplished until the current version of RTCR is summarized in chapter 3. In addition, Checkpoint-Restore-mechanisms on other operating systems will be presented. While chapter 4 discusses different approaches for the design of the restore mechanism, it also includes the architecture and drawbacks of the current version of RTCR and how these have been adapted. Chapter 5 talks about the implementation and especially issues that lead to deviations in the architecture described previously. Chapter 6 evaluates the performance of the restore procedure and eventually, chapter 7 summarizes the limitations and problems identified during the work on this thesis.

2. Foundation

2.1. Genode

The Genode OS Framework is a middleware for developing applications focusing on security and isolation. It follows the philosophy that no software in a system can be trusted so that one process shall not interfere with the operation of another [6, p. 54-55]. Therefore, processes in Genode are structured in a hierarchical tree (called recursive system structure [6, p. 13]), as is common in modern operating systems. However, unlike in commonly used operating systems, a process cannot request resources from the kernel directly. As seen in figure 2.1, a typical Linux process can allocate memory by directly issuing a system call. However, if a child wants to achieve the same in Genode, it must ask its parent to provide a dataspace. The parent can then decide to reject or handle this request. In the latter case, it can either explicitly donate part of its own memory budget to the child or pass the request on to the grandparent.

Thanks to this architecture, a rogue process (e.g. by leaking memory due to a bug) can only exhaust its own and its childrens' memory but cannot influence its parent or other processes on the system. [6, p. 46-53]

To identify, what a process is allowed to do, Genode provides a primitive named capability which is associated with a certain property or functionality. Therefore, a process holding a reference to a capability in its capability space can utilize it. The capability space is similar to a private virtual address space except for referring to capabilities.

Moreover, a capability can be transferred via IPC to other components. For instance, a RAM-dataspace capability represents a piece of physical memory, which a process can attach to its own virtual address space. The process can also decide to share that capability with other processes in order to enable access to shared memory. Note that ownership is still tied to the original process. If a capability is deallocated by the owner, all references to it are removed from all other processes. [6, p. 39-41, 64]

A component may also decide to trade a capability with another process. For example, when a client wants a server to process some data for it, the client could abuse the server by sending loads of requests, which will eventually exhaust the server's memory and affect its operation, potentially endangering the entire system as other processes might depend on the server application as well. In order to prevent this attack vector, a

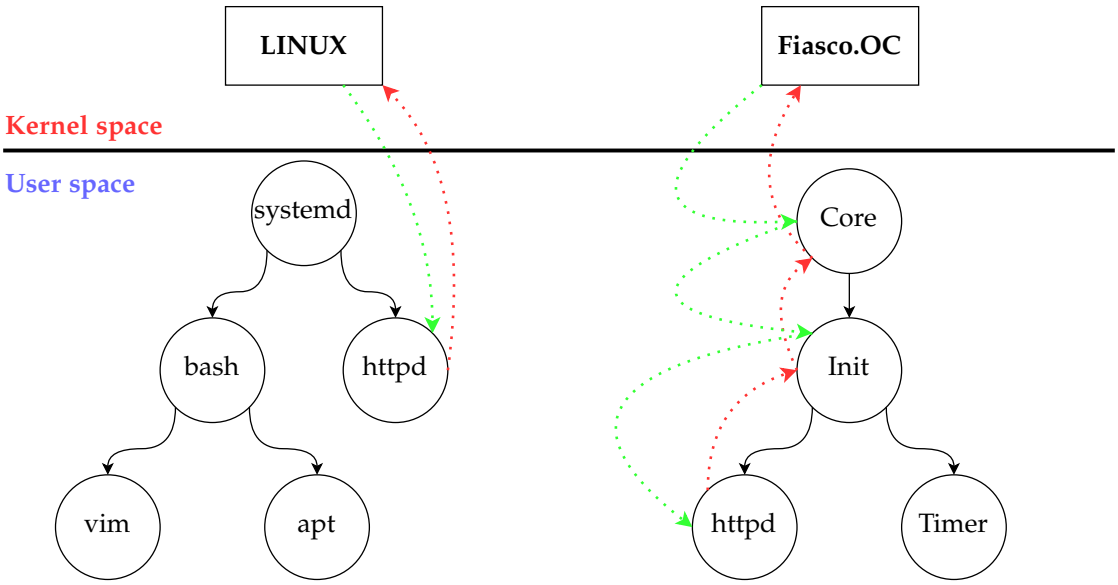

Figure 2.1.: Comparison of a typical process tree between a common GNU/Linux- and Genode-setup. The arrows demonstrate the flow of resource allocations.

client must donate some of its own memory to the server if it wants to use the provided service. The client achieves this by transferring a dataspace capability to the server, which is then able to utilize it for the duration of the session. [6, p. 58]

This communication between parent and child (and any other processes) is accomplished via Inter-Process Communication (IPC), most notably, synchronous Remote Procedure Calls (RPC). RPCs are tied to a specific service, which can be offered by any other process. The root-process, called Core, provides the most basic services (ROM, PD, RM, CPU, IO_MEM, IO_PORT, IRQ, LOG and TRACE) [6, p. 64-70]. An instance of a service-provider is called session while the consumer is simply named client [6, p. 64]. On a technical level, each process consists of at least three threads. The first one started during the bootstrap is named *ep* (entrypoint) and is responsible for initialization as well as RPC-communication in general. It also starts the *signal_handler* thread for handling signals, as the name suggests. Signals are another form of IPC provided by Genode. They allow for asynchronous communication in a *fire-and-forget*-manner. Signals are also unbuffered, meaning they cannot carry any payload and an often repeated signal may not necessarily lead to the execution of the handling method for each signal. [6, p. 204-205, 453]

Genode abstracts these features so that applications may run independently from the underlying kernel. In fact, the framework supports several types of the L4-family,

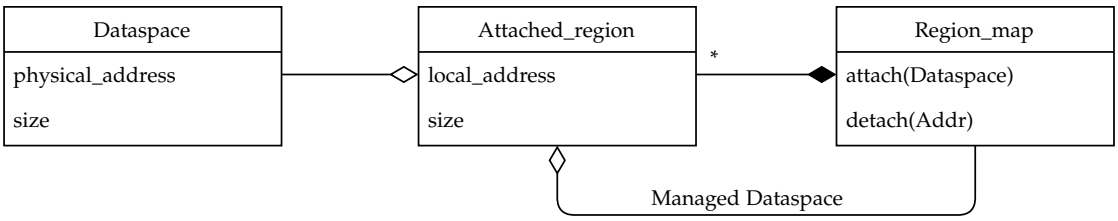

Figure 2.2.: Relationships between dataspaces and region maps.

NOVA and the Linux-kernel. This is achieved by providing common interfaces found in the *base*-repository and kernel-specific implementations of these interfaces in a *base-{kernel}*-repository. [6, p. 14, 21, 39]

When a Genode image is booted, the first process executed is the Core process mentioned before. It implements primitive and sometimes kernel-specific features including but not limited to task creation, memory and capability management and logging. Core then delegates almost all of its resources to the Init process, which is launched by Core as the second process [6, p. 64]. Init's task is to bootstrap the initial set of user-defined tasks. This phase of the boot process can dynamically be configured using XML. This allows to specify the initial set of processes, including services that can be provided, configure their hierarchy and routing when issuing a service request. Most notably, the configuration also defines the amount of resources (Memory, Capabilities, CPU) a task is allowed to use. [6, p. 176-189]

2.2. Memory Architecture of Genode

As this thesis is mainly concerned with restoring memory areas of a process, the memory architecture of Genode is particularly interesting. The following glossary should assist with understanding the different components involved in Genode:

- **Protection Domain (PD):** Each process (also named component) in Genode consists of exactly one PD-session. It contains the virtual address space, capability space and a budget for physical memory as well as capabilities. [6, p. 66]
- **Dataspace:** A dataspace represents a part of physical memory with a page-granular size (usually multiple of 4 KiB). Owning a capability to a dataspace allows a process to attach it to its virtual address space (region map) in order to access it. [6, p. 64]
- **Region Map:** A region map represents a virtual address space. Dataspaces can be attached to a region map in order to make them available to the process.

Additionally, region maps can be attached as dataspaces to other region maps as well allowing for nested virtual address spaces. This relationship is illustrated in figure 2.2. In fact, Genode makes use of this concept on every process as each one consists of at least three region maps (address space, stack and linker area for shared objects). The latter two are backed by physical dataspaces by default. Both the linker and stack area region maps are again attached to the *address space* region map. [6, p. 65, 67]

- **Attached Region:** As the name suggests, an attached region is either a dataspace or region map (managed dataspace) attached to a region map. Most importantly, it contains the virtual address local to the region map.
- **Region-map Management (RM):** The RM-session allows to manage region maps and their virtual addresses manually. However, it is not necessarily required as the PD-session implicitly allocates fundamental region maps. Thus, this bachelor's thesis will not attempt to restore the RM-session and its associated region maps. [6, p. 67]

2.3. Microkernel vs. Monolithic Kernel

A microkernel implements only the very basic features required to create an operating system environment unlike monolithic kernels, such as the Linux- and most of the BSD-kernels, which additionally run drivers for hardware, networking stacks and other more complex components in kernel space. Microkernels usually focus on process scheduling and memory management, while keeping drivers and any other more advanced features on the user land side with processes having to communicate via an IPC-mechanism. Figure 2.3 roughly illustrates the differences between these two kernel architecture. The advantage of this design is that the code run in privileged mode, part of the so-called Trusted Computer Base, is kept as small as possible, because a fault in the kernel can have an impact on the operation of the entire system. However, a rogue userspace process can more easily be contained (e.g. when accessing an unmapped virtual address, it can simply be killed by the kernel).

Of course, having drivers run in userspace instead of kernel space comes with additional overhead as communication must be done via kernel-provided IPC-mechanisms. However, this trade-off is usually acceptable, especially in safety-critical fields, what Genode is mainly designed for.

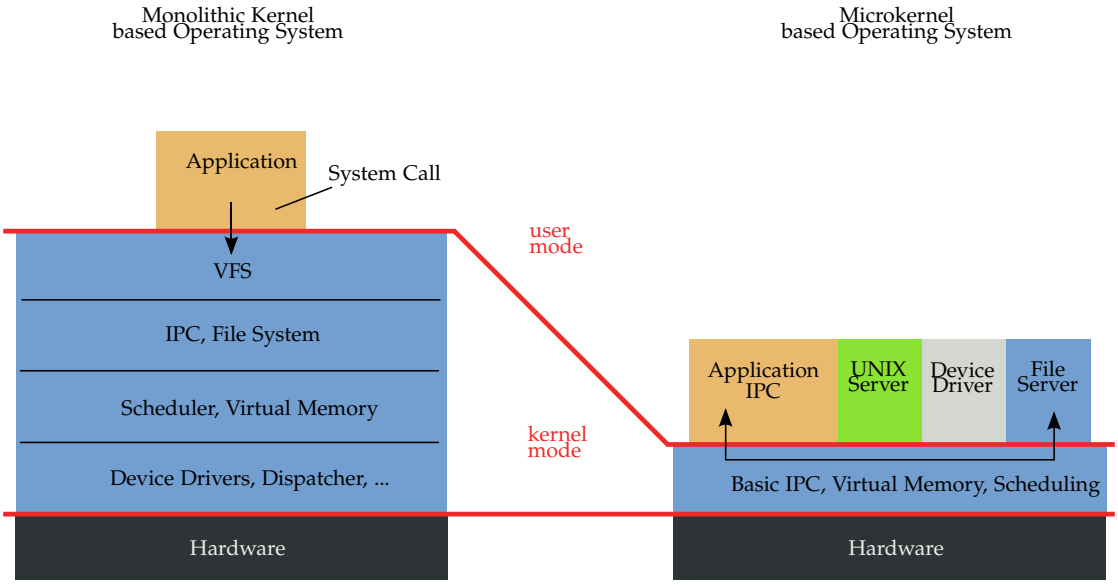

Figure 2.3.: Architectural difference between a microkernel and a monolithic one. Source: Wikipedia [14]

2.4. L4-Microkernel and Fiasco.OC

The L4 microkernel was originally developed by Jochen Liedtke. As he designed it with performance in mind, it was written in pure x86-assembly, which prevents its use on other architectures. Therefore, other implementations of the L4-ABI were developed over the years [24]. One of them is the Fiasco.OC microkernel written in C++. It features realtime scheduling and a powerful debugger within the kernel among other features [22]. In addition, it provides capabilities, which are an abstract construct to allow processes to make certain system calls. Processes can communicate via synchronous IPC calls [7].

The Fiasco.OC-kernel comes with the L4 Runtime Environment. This interface provides abstractions similarly to Genode so that applications do not need to communicate with the kernel directly [12]. When booting the Fiasco.OC-kernel, the root pager (called Sigma0) is started first and receives full access to all userland RAM and device memory. Sigma0 is mainly responsible for providing memory to Moe, the initial user space process (called task in the Fiasco.OC context) [20]. Moe is also started by the microkernel and implements the basic L4 runtime environment abstractions, such as dataspaces, memory allocators, namespaces and scheduler policies. At this stage, L4-compatible executables can be started as all necessary services are provided by Moe [16]. However,

in order to provide a flexible way of starting processes, L4 provides an init process called Ned, which includes a Lua-interface to bootstrap tasks and capabilities accordingly [15]. When using Genode, the first tasks started after sigma0 are Core and Init instead of Moe and Ned.

3. Related Work

This chapter presents previous theses ultimately leading to the current state of RTCR. In addition, other Checkpoint-Restore solutions, specifically for Linux, are introduced.

3.1. Realtime Checkpoint/Restore

Initial Resarch and Proxy Component: David Werner investigated the state of Checkpoint-Restore-solutions on other platforms, namely CRIU, which will be introduced in section 3.2. His initial idea was to port CRIU to Genode but he found that its design was not compatible with Genode's architecture. Thus, he developed a new component named *proxy*, which would sit inbetween the child and its parent and intercepts session requests for dataspace allocation and attachments. Other sessions were not supported and restore-functionality was not available. [23]

Design and Development of real-time capable Checkpoint/Restore: In the master's thesis written by Denis Huber an attempt to develop a real-time checkpoint/restorer was made. His idea was similar to Werner's who intercepted RPC-calls made to core sessions, such as PD and CPU. The parameters of these calls would immediately be stored in classes specified as *online storage*. Once a checkpoint is triggered, the data will be copied from the online to the offline storage. In addition, instead of copying all data each time a checkpoint is triggered, Huber implemented an incremental approach to only copy modified dataspaces based on page-faults. He also developed a restore, however, it did not yield the expected results. [8]

Hardware-assisted Memory Tracing: As mentioned above, Huber implemented an incremental checkpointer by attaching dataspaces on demand when a page-fault is triggered. This causes overhead, an issue that Sebastian Bachmaier attempted to solve. His idea was to develop an FPGA-component capable of tracing memory access and exposing this information to Genode via a driver he implemented as well. However, Bachmaier assessed that his hardware-accelerator led to a negative performance impact of 900 %. [2]

RTCRv2 (Realtime Checkpoint/Restore Version 2): RTCRv2 is the second version of the Checkpoint/Restore-mechanism named by Johannes Fischer in his master's thesis. His stated goals are support for performance measurements, kernel-independent design, uniform configuration interface, modularisation and multi-core checkpointing. His work attempted to unify and improve the previous work on RTCR such as incremental and hardware-assisted checkpointing. These different techniques were refactored into *modules* which share a common interface. This allows to select the appropriate module at runtime. [9, p. 10-20]

Another goal of Fischer is to improve checkpointing duration via parallelization. He conducted a dependency analysis of all checkpointed-related methods in RTCR developed by Huber and resolved these dependencies to be able to checkpoint all sessions in parallel. However, during the evaluation no significant performance improvement could be identified. Fischer concluded that this must be the result of the I/O-connection being the bottleneck. [9, p. 78-79]

Furthermore, Fischer extensively tested his implementation in terms of performance based on multiple hardware platforms, Genode versions, microkernels (Fiasco.OC and seL4) and other configuration settings. In particular, he identified that among all tested kernels the Fiasco.OC was by far superior in terms of performance. [9, p. 69-84]

Fischer's thesis is particularly important for this bachelor's thesis as some design choices and architectural changes are based on RTCRv2. Therefore, his work is discussed in more detail in section 4.4.

3.2. Checkpoint-Restore Support on Linux

Multiple Checkpoint-Restore solutions have been developed for Linux. A well-established project in this field is CRIU (Checkpoint and Restore in Userspace). It mostly utilizes the proc-filesystem, where CRIU extracts information on children, file descriptors and memory maps. After identifying all resources, the whole process tree is put into a frozen state and the checkpointing procedure is started. Some information, such as memory contents, are extracted from within by injecting a so-called *parasite code* into the target process. The restore-functionality happens in a similar fashion. It forks itself the required amount of tasks, restores file descriptors, sockets, memory maps and so forth. As a last step, the parasite code is removed and the instruction/stack pointer are reset accordingly. [5]

CRIU is mainly developed for Linux and as a consequence, it heavily utilizes kernel/POSIX-specific features. As these are not directly available on Genode, Werner concluded in his bachelor's thesis that porting CRIU is not a viable option. [23, p. 25]

4. Design

4.1. Glossary and Design Principles

The following sections explain the design of the updated Real-time Checkpoint/Restore (RTCR), based on Fischer's previous work [9]. All information regarding his architecture have been extracted from both his thesis and source code.

As Fischer refers to his version of RTCR as RTCRv2 [9, p. 10], this convention is continued in this bachelor's thesis and the updated design is referred to as RTCRv3. In addition, the following terms might occur occasionally:

- **Cold storage** consists of classes representing a checkpointed Genode object. Its values are only updated when a checkpoint is explicitly requested [9, p. 22-23]. In RTCRv3, they have been renamed to **checkpoint information** and are modified to be immutable. Thus, values are not updated anymore.
- **Functional classes** (RTCRv3) represent the owner of a certain Genode capability. They are responsible for the entire lifetime of the capability. For instance, the class *Signal_source* requests a Signal_source capability through the parent component.
- **Hot storage** (RTCRv2) consist of classes representing a checkpointable Genode object. Their values are updated immediately when a change is intercepted [9, p. 22-23]. This functionality has been removed in RTCRv3 and are thus called **functional classes**.
- **RTCR implementation** is a program which uses the RTCR-library to provide checkpoint/restore-functionality.
- **SP/IP** are shortcuts for stack and instruction pointer respectively.

The following goals were kept in mind while developing the design:

- **Avoiding Genode Modifications:** To offer high compatibility, RTCR should work with Genode from the upstream repository. This also decreases the maintenance burden of migrating to new versions.

- **Flexibility:** As RTCR is still under research and thus likely changes frequently, the design should leave room for making upcoming modifications more easily. Realtime capability in particular, which is not specifically considered in this bachelor's thesis, must be available in the future.
- **Full transparency:** Existing applications must work without requiring any modification or special support.
- **Simplicity:** The architecture should be as minimalistic as possible in order to support a basic functional memory restore and thus facilitate understanding of the code/architecture by new and upcoming developers and researchers. Features which would require more complicated heuristics have been intentionally avoided.

4.2. General Architecture of Restore-Procedure

In order to checkpoint a child process, RTCRv2 intercepts each RPC-call to its parent and tracks the information exchanged. Thus, RTCRv2 does not work with arbitrary services and must therefore implement an appropriate intercepting service, which is instantiated as a session for each individual child. RTCRv2 already provides the basic services CPU, LOG, PD, RM, ROM and TIMER.

Concerning the restore mechanism, the PD- and CPU-services are of main interest. The first one manages the virtual address space as well as all stack areas of the child, while the latter is necessary to restore both the stack and instruction pointer for each thread.

When restoring a child, the exact moment of triggering the restore procedure must be carefully chosen. One of the primary design goals of this thesis is restoring the child in a completely transparent way. Thus, the following possibilities have been evaluated:

- **Bootstrap component regularly and restore on demand:** The child is bootstrapped as usual by Genode and the intercepting PD-session restores the memory content on demand, when an allocation is requested. For instance, when the child allocates a dataspace, the PD-session checks whether it has checkpoint information matching the same criteria. This information is then used to copy the contents of the checkpointed dataspace into the newly created one.

  However, this implicit approach cannot be applied to thread creation (at least not for the *ep*-thread), because *ep* initializes the signal-context, -thread and -source in addition to properly setting up the stack and other dataspaces. Therefore, the restorer must know when it is allowed to reset the IP/SP. Thus, a thread of a child must use an explicit barrier to notify the parent that the restore should

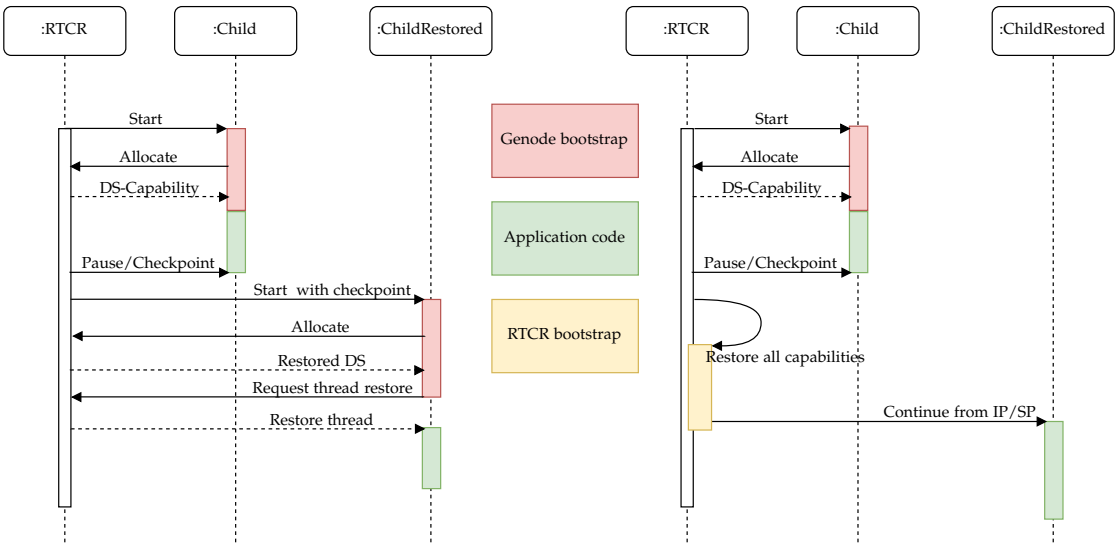

Figure 4.1.: UML-Sequence diagram of the on demand restore (left) and custom bootstrap process (right).

be commenced. This allows the parent to set the stack and instruction pointers accordingly in addition to repairing the stack as the contents before the stack pointer could be unwillingly modified. This is elaborated in section 4.3.

- **RTCR takes over bootstrap:** A checkpoint theoretically contains all necessary information to completely restore a child. Therefore, RTCR could handle the bootstrap process directly by creating all necessary threads, dataspaces, signal contexts and so forth and restoring them to their checkpointed information. This would normally be handled by the *ep* thread inside Genode.

Both designs are presented in figure 4.1 as UML-diagrams for better comparison. This thesis implements the first approach as it seemed more promising for an initial implementation. The second solution would lead to duplicated effort as RTCR needs to mimic Genode's bootstrap. This breaks one of the primary goals of RTCRv3 in keeping the complexity low. Additionally, future changes within the bootstrap process of Genode could likely require adjustments in RTCR as well. In addition, performance could theoretically be better in the first case, since RTCR only needs to recover the stack from in between the stack pointer from which the thread restore is requested and the stack pointer stored in the checkpoint.

At the end of working on this thesis, it was discovered that Denis Huber independently described two ideas similar to the approaches presented above. One of them was to let Genode bootstrap the process and then explicitly ask RTCR to restore the

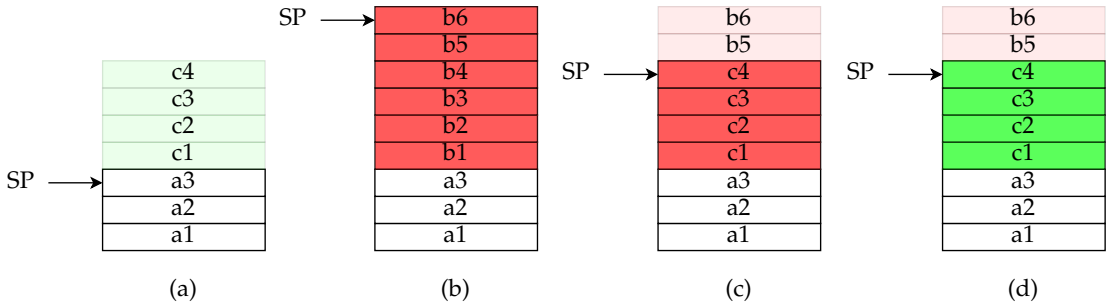

Figure 4.2.: Stack behavior during a restore. Inspired by: Structured Computer Organization [21, p. 259].

process before application-specific code is called [8, p. 39]. However, his mechanism differs in that **all** capabilities are restored at this barrier while the design proposed here only restores thread-relevant data during this phase. Dataspaces are already restored during allocation. Note that other capabilities are not in the scope of this thesis.

4.3. Stack Repair

In section 4.2, a solution is proposed to let Genode handle the bootstrap and restore dataspaces once they are allocated. However, this is not sufficient for the one used as a stack for the entrypoint-thread (*ep*). This thread is responsible for the entire bootstrap including calling the application-specific code in *Genode::construct*. This means that the thread cannot be restored until an explicit barrier is reached. At this point, the SP/IP could theoretically be set back to their checkpointed values and program execution could be resumed. However, the stack does not grow continuously and it can shrink as well, disrupting the restored data.

Figure 4.2 illustrates this behavior. Green and red bars indicate restored and overwritten values respectively. The example begins with function A (a) calling B, increasing the stack pointer (b). After B has returned, A initiates a thread restore. The SP/IP are set accordingly, however, the stack content does not match its checkpointed information anymore because it was overwritten by function B, leading to undefined behavior within function C (c). Thus, RTCR must specifically restore the area between the explicit barrier and the checkpointed stack pointer before the execution can be resumed (d).

4.4. Design Changes to RTCRv2

As described this bachelor's thesis builds on the work of Johannes Fischer's master's thesis. He refactored the original design of RTCR into a reusable, module based library. This allows using different mechanisms, such as incremental checkpointing or FPGA assisted memory tracing and copying, while sharing common functionality by inheriting from a general purpose module. Those individual modules are then managed by a factory, where the RTCR implementation can retrieve an instance of the module it wants to use for the checkpoint/restore-mechanism. [9, p. 13-21]

On a technical level, modules are registered via static allocations. The RTCR-application can then simply request an appropriate module by asking for a new instance via the Module_factory based on the name of the module. As mentioned above, this design requires the modules to be allocated statically. Typically, this is handled by the libc-runtime before calling the applications *main*-routine [13], which is why Fischer's implementation depends on a proper C-library (Genode optionally provides FreeBSD's libc) to offer this functionality. However, one of Genode's main goals is to keep the trusted computer base low and thereby, ensuring low complexity [6, p. 198]. Therefore, RTCRv3 follows the same principles by removing the module factory and all associated static allocations. This also eliminates the need for a proper libc-environment.

Nevertheless, modularity is a practical concept. In the following, several approaches to keep it in another form are outlined:

- **Break up inheritance between Base_module and Init_module and use C++-templates instead:** In Fischer's implementation, the Child-class takes an (abstract) *Init_module* as a parameter for the module to be used. The architecture can be changed to use a template argument instead, which has the advantage of avoiding unnecessary vtable-lookups and allowing the compiler to more aggressively apply optimizations, such as inlining function calls. On the other hand, binary size increases as the compiler has to generate all class-methods for each type used as a template argument. However, it is likely that a checkpointer will only use one module implementation in practice anyway, theoretically leading to an identical file size.

- **Instantiate modules directly:** Considering that RTCR is designed to be used in an automotive context, the target environment of RTCR is rather static, meaning the developer of an RTCR implementation will likely already know which RTCR-module is most suitable for the use case at hand. The implementation is straightforward as it does not require any modifications to the Child-class.

The first approach was chosen for this thesis as it allows for easier adaption of the architecture in the future and should theoretically benefit performance thanks to code generation specific to the template argument [25]. However, the main improvement lies in the reduction of complexity as discussed later in section 5.1.4.

4.4.1. Remove Destroy-Queues

In RTCRv2, objects (dataspaces, signal_contexts, etc.) destroyed via the intercepting sessions are freed/deallocated via the parent but the appropriate classes are not immediately deallocated from the heap. Instead, the associated storage class is marked for removal by being inserted into appropriate destroy-queues. Once a checkpoint has been triggered the objects will be deallocated. Fischer's idea behind this design was to avoid overhead caused by list lookups of $O(n^2)$ in RTCRv1 each time a capability is removed [9, p. 24].

However, considering RTCR is designed to run in realtime settings, the checkpointing process should be kept as short as possible which is why removing capabilites during this process should be avoided. In fact, this overhead of destroying during a checkpoint increases, the more objects have been marked for deletion, which additionally relates to the checkpoint frequency. Fischer assumes this frequency remains constant [9, p. 22], however, in the opposite case the current design could lead to unpredictable behavior, for example when checkpoints are only triggered in case of failures. In addition, these destroy-queues lead to more code and thereby makes the architecture unnecessary complex. For instance, each queue is also accompanied by a mutex that must be carefully handled. Therefore, all destroy-queues have been removed.

But still, Fischer's suggestion to reduce the worst case performance of $O(n^2)$ is worth upholding. When analyzing the source code of RTCRv2, it was noticed that the objects inside the destroy-queues are not used anywhere else apart from removing them during a checkpoint. Thus, the target capability can simply be removed immediately which theoretically leads back to an upper bound of $O(n)$ due to a list look-up. Unfortunately, Genode's list interface does not provide the possibility to find and remove at the same time. However, finding and removing the target capability separately would again lead to an undesired upper bound of $O(n^2)$. Thus, the following ideas for providing a find-and-replace functionality are presented:

- **Indirect extension:** The functionality can also be implemented inside RTCR. This requires additional design choices as C++ offers multiple ways such as inheritance or befriending external methods to extend existing functionality.
- **Modify Genode:** An appropriate *find_and_remove*-method can simply be added to Genode's linked-list implementation. Although this approach is likely the most

straightforward one, each modification to Genode must be carefully considered as this might make migrations to future updates increasingly difficult.

The first approach was chosen as it follows the design principle of avoiding modifications inside Genode. The specific implementation is discussed in 5.1.4.

4.4.2. Decouple *_info-Classes

Most classes in RTCRv2 share a similar inheritance construct where the base and inheriting class represent the cold and hot storage respectively. The hot storage class is constantly updated each time the state of the particular capability changes. When a checkpoint is initiated, the state is then copied into the cold storage, which is also the super class of the hot storage. The cold storage is publicly accessible to the RTCR implementation. A major advantage of this design is that it requires no additional allocations as the cold storage is simply reused for each checkpoint. [9, p. 22-23]

However, this architecture also puts multiple restrictions in place:

- **Global state:** The cold storage classes are shared which can accidentally lead to race conditions in certain situations. For example, some code areas holding a reference to the same cold storage might not expect it to change. However, when a checkpoint is triggered, the cold storage is overwritten, which might cause undefined behavior. This makes formal verification of the software much harder as well since the same memory location might be mutated from multiple locations. Finally, modern software engineering commonly declares global state as an anti-pattern.

- **High coupling:** Due to inheritance, future design changes are more difficult to apply. In addition, it opens a potential security hole, where a class with access to a cold storage can cast it to a hot storage, which should not necessarily be allowed. In the current design, this might not be a problem as downcasting the cold storage simply accesses the hot one. However, in RTCRv3, the hot storage is extended with additional functionality that are internal to the RTCR library.

- **Overhead for multiple checkpoints:** RTCRv2 can only hold one checkpoint state at the time, as the cold storage is overridden at each checkpoint. This limitation could be ignored in an automotive scenario, where a checkpoint is, in the majority of cases, only used for transferring to another computer unit. Nevertheless, other use cases should be kept in mind. Having multiple checkpoints would require transferring the cold storage to another location (memory or other type of storage) after each checkpoint, leading to additional overhead. In general, the inheritance design does not allow other means of storing checkpoints.

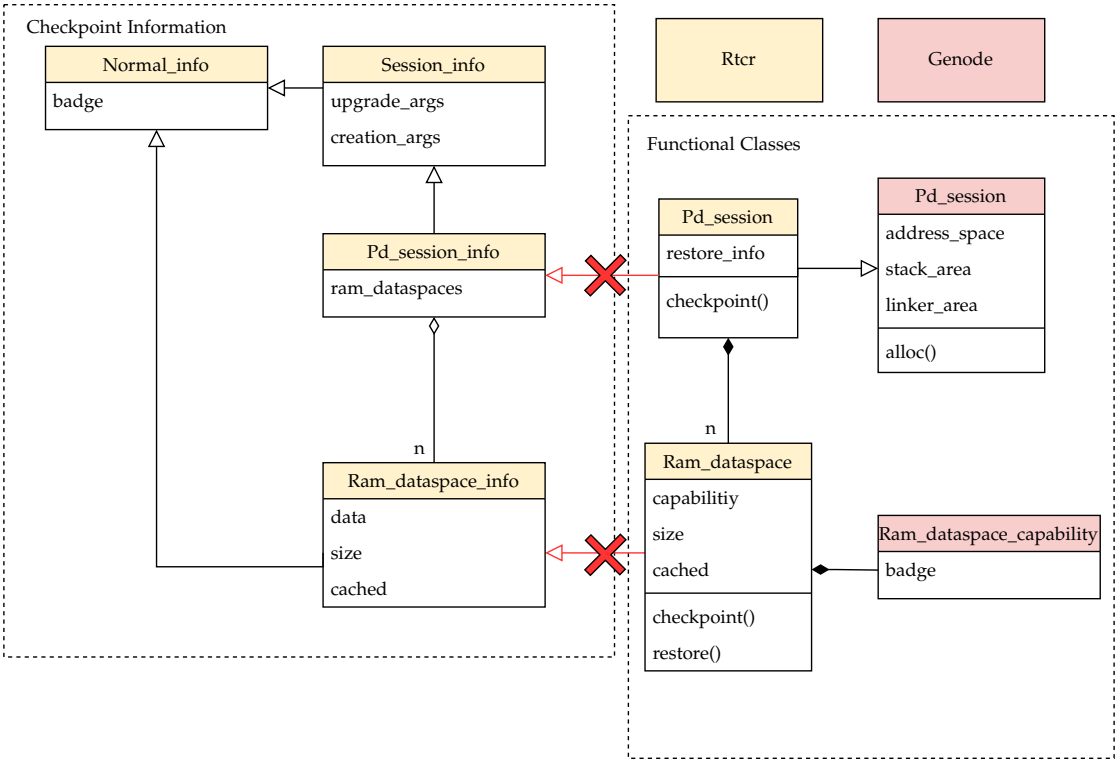

Figure 4.3.: Simplified class diagram of RTCR. Colors refer to their respective namespace.

Therefore, the design was changed to create cold storage structures during checkpointing and return them immediately to the RTCR implementation. Figure 4.3 presents these modifications on Ram_dataspaces and a Pd_session. The figure also highlights the decoupling between checkpoint information and application logic which used to exist in RTCRv2. Note that many but not all structures have been adapted to this architecture.

When compared to the original design, a drawback is the additional allocation required for each info structure. However, this can be avoided in the future by providing a pre-allocated buffer for RTCR to be used. Furthermore, this also open up other possibilites of storing checkpoints independent from RTCR, see section 7.1.

Most importantly, the design fixes the problems mentioned above:

- **No global state:** Checkpoints are created locally when requested. Due to the constant initialization and ownership transfer to its caller, accidental modifications are prevented.

- **Low coupling:** The functional classes are independent from their info pendants, fixing the aforementioned security problem of gaining access to RTCR internal classes.
- **Support for multiple checkpoints:** As the checkpointed data is immediately returned to the implementation, the caller is responsible for deciding on how to use the information. For instance, the implementation can decide to keep multiple checkpoints in memory.

4.4.3. Transfer Ownership to Associated Objects

In RTCRv2, when the allocation of a new capability (Signal context, RAM-dataspace, etc.) is requested, the responsible session creates the object by asking its parent. Then, it allocates a hot storage object in which the capability is stored. When a Genode capability is released, it is the responsibility of the session to deallocate it as well. However, the associated hot storage objects are not immediately removed as described in subsection 4.4.1. Hence, the lifetime of a capability is not tied to its hot storage object and tracking the lifetime of an object becomes more difficult.

RTCRv3 follows a different philosophy: The former hot storage class, now named functional class, is always associated with one specific capability. Therefore, new capabilities are allocated during the instantiation of functional objects. When the object is deallocated, the destructors handle the appropriate destruction of the associated capability. This couples the lifetime of the functional class with the one of its capability as can be observed in figure 4.3. Thus, it must also be ensured that there is no more than one owner of the same capability. This design is very similar to a *unique_ptr* in C++.

5. Implementation

The following sections discuss the implementation of the architecture described in chapter 4. Furthermore, the test program used during development and other changes independent from the presented design, such as improved debugging output, are presented.

5.1. Foundation Changes

5.1.1. Test Application

The development of RTCR requires a proper test application for verification purposes. RTCRv2 already provides a program called *sheep_counter* consisting of a timer connection, manual dataspace allocation and a variable within the dataspace, which is endlessly incremented in a loop. Each incrementation is printed on the console.

Affected source files:

- src/app/sheep_counter/main.cc

Since this bachelor's thesis is focusing on restoring memory, the original sheep_counter was adapted:

- Additional counter variables were added, each located in different memory areas. One is stored on the stack, one on the heap and one in a manually created dataspace.
- The timer connection has been removed as restoring a timer-session is not in the scope of this thesis.

The full source code can be inspected in figure 5.1.

```
1  void Component::construct(Genode::Env &env) {
2          //More complex data structure
3          auto heap = Heap(env.ram(), env.rm());
4
5          // Counter variable on stack
6          uint8_t n = 1;
7
8          // Counter variable on heap
9          auto heap_n = new (heap) uint8_t(1);
10
11         // Manually created dataspace
12         auto ds_cap = env.ram().alloc(4096 * 20);
13         auto ds_addr = env.rm().attach(ds_cap);
14         auto &n2 = *(uint16_t *)ds_addr;
15
16         log("Heap|Stack|Manual␣DS");
17
18         while (true) {
19                 log((*heap_n)++, "|", n++, "|", n2++, "␣sheep.␣zzZ␣");
20
21                 // Used as delay without requiring a timer
22                 for (long long i = 0; i < 10000000; i++) {
23                         __asm__("NOP");
24                 }
25         }
26 }
```

Figure 5.1.: Updated sheep_counter.

5.1.2. Enabling Compiler Warnings

Affected source files:

- lib/mk/rtcr.mk

While working on the implementation, it was noted that warnings were entirely disabled in the build configuration file (rtcr.mk). This might make bootstrapping development easier as GCC will abort only on critical errors, such as invalid syntax. However, GCC offers plenty of features to detect undefined or unwanted behavior in addition to "unclean" code, such as missing *override* declarations.

These warnings should be handled as early as possible since it becomes increasingly harder to fix them once the codebase has reached a larger scale. Therefore, warnings

were explicitly enabled (by adding the *-Wall* command line parameter) and all complaints by the compiler have been fixed. One noteworthy exception is the error type *non-virtual-dtor*. It states that a parent class has a non-virtual destructor, which could be problematic when polymorphism is used. (e.g. Invoking delete on a pointer of the parent type will only call the parent's destructor but not the overridden ones.)

This warning was disabled because it was often triggered due to Genode::List::Element not having said virtual destructor. Fixing this bug was not deemed necessary for the scope of this bachelor's thesis as modifications inside Genode are avoided. Thus, the error non-virtual-dtor was demoted to a warning by appending *-Wno-error=non-virtual-dtor* to the compiler flags.

5.1.3. Improving RTCR Output

Affected source files:

- include/rtcr/util/logging.h
- lib/mk/rtcr.mk
- lib/mk/spec/debug/rtcr.mk

Genode does not offer a *cerr/cout* or *printf* function as commonly used with C(++). Instead, it provides a function named log(...), taking variable arguments to be printed onto the serial console. Unfortunately, logging induces significant overhead and therefore, should be avoided for performance evaluation. However, log-messages are crucial for understanding and debugging the behavior of a program. Fischer solved this problem using a custom *DEBUG_THIS_CALL*-macro, which expands to *Genode::log(__PRETTY_FUNCTION__)*, when the compile-time macro *DEBUG* is defined. This leads to printing the function name from where the macro was invoked. If the aforementioned DEBUG-macro is not defined, *DEBUG_THIS_CALL* simply expands to nothing, avoiding any overhead.

Next, RTCRv2 produces only a few messages, even with *DEBUG* enabled. This makes it rather difficult for new developers to understand and debug the behavior of RTCR. Thus, more debug and informational messages inspired by the GStreamer framework were introduced over the course of this thesis [3]. However, since printing many messages can result in cluttered output, RTCRv3 implements support for different log levels. Similarly to version 2, individual log levels can be enabled by defining the appropriate macros (See *lib/mk/spec/debug/rtcr.mk*). Setting a certain log-severity also enables all levels above it. For instance, activating informational messages also prints warnings and errors but debug output is suppressed.

WARNING:init_module.cc:103:RM-Session is not available!

Figure 5.2.: Example warning message from RTCR. Structure inspired by GStreamer [3].

The following message types are provided:

- ERROR: A critical error occurred, program execution must likely be aborted.
- WARNING: A non-fatal error occurred. The program might still be working but it could hint to a logical programming error and/or might result in a critical error in the end.
- INFO: Notifying the user/developer that some functionality has been triggered.
- DEBUG: Debugging information of any kind, usually quite verbose.

By default, nothing is printed on the command line due to performance overhead caused by logging. The recommended workflow for enabling output is to add *debug* to the *SPEC*-variable found in *etc/spec.conf*. The appropriate level can then be configured in *lib/mk/spec/debug/rtcr.mk* by setting the *LOG_LEVEL*-variable to one of the described categories above (ERROR, WARNING, INFO, DEBUG).

Finally, to enable easy filtering of certain messages (e.g. by using grep), each message prints its type and filename as well. In addition, the output is color coded depending on the severity level, making spotting critical messages more easy. In general, a message consists of the following structure: [TYPE]:[FILENAME]:[LINE]:[MESSAGE]. An example warning message can be seen in figure 5.2.

Finally, Fischer's macro *DEBUG_THIS_CALL*, which prints the name of the executed function, is refactored into *LOG_CALL* and included in the appropriate header file.

5.1.4. Complexity Reduction

Removing *Module_factory*

Section 4.4 discussed the removal of the *Module_factory* while keeping the module-based architecture. The main idea behind this change is the reduction of complexity. When comparing commit 06c5db1 (Fischer's last commit) and 5250dcf2, the affected source files could almost be halved by 342 lines of code. Moreover,

Affected source files:

- include/rtcr/base_module.h
- include/rtcr/init_module.h
- src/rtcr/base_module.cc
- src/rtcr/child.cc
- src/rtcr/init_module.cc

RTCRv3 removes the dependency on a libc-runtime, eliminating further lines of code.

Removing Destroy Queues

Affected source files:

- include/rtcr/util/misc.h

In subsection 4.4.1 the design and its issues of how RTCRv2 handles deallocated capabilities was elaborated. The goal is to reduce the current complexity while still achieving an upper bound of O(n) by removing the capability immediately. However, Genode does not provide a proper find-and-remove-method. Since Genode's code shall remain untouched, this functionality was implemented inside of RTCRv3. The idea is to find the desired object and once the target is found, set the next value of its predecessor to its successor, thus removing the current object from the list.

However, manually modifying the list is not possible because the *_next*-attribute of a list element is protected. The following solutions have been evaluated:

- **Add method to target class:** Since the type used as list elements must inherit from *Genode::List<T>::Element*, it is able to access the protected *_next*-value.
- **Befriending:** C++ provides the keyword *friend* which enables external functions to access protected and private members even though they are not part of the class. The function must simply be added to the class declaration with the *friend*-keyword before the return type. [18, p. 272]
- **Custom Rtcr::List<T>:** Genode::List<T> can be extended with a find-and-remove-method via inheritance and thus, providing a custom list. Since *Genode::List<T>* is a friend of *Genode::List<T>::Element*, the custom list is able to access the protected *_next*-value.

The first solution proposed was not chosen because it would require duplicated code in each class that is supposed to support the find-and-remove functionality. This issue could be solved by refactoring it into a separate class. However, since this class would need to inherit from *Genode::List<T>::Element* as well, the complexity of the architecture increases. This option might be worth considering in the future in case more list-specific functionality needs to be added. The same applies to option three as a custom list could be useful once more features are required.

The second approach was chosen because it adds the least complexity. The function *find_and_remove* was added to *misc.h*. *Native_capability, Ram_dataspace, Signal_context*

```
1 class Rtcr::Ram_dataspace : public Genode::List<Ram_dataspace>::Element {
2         SUPPORT_FIND_AND_REMOVE;
3         // Expands to:
4         // template <typename T, typename F>
5         // friend T *find_and_remove(Genode::List<T> &list, F predicate)
6 }
```

Figure 5.3.: Example for adding support for *find_and_remove*.

and Signal_source received an appropriate friend declaration. In order to reduce duplicate efforts and easily add support for other classes in the future the macro *SUPPORT_FIND_AND_REMOVE* can be used which expands to said declaration. An example is provided in figure 5.3.

5.1.5. Separating Info-Classes

In Fischer's RTCRv2, a regular class would derive from an info class, separating it into hot and cold storage respectively. As mentioned in chapter 4.4.2, the cold storage is supposed to be publicly accessible but due to the inheritance structure, it would be indirectly possible to access the internal hot storage class. Thus, the derivation of the info class is removed. Instead, each functional class receives a *checkpoint*-method which allocates an info-class of the appropriate data structure. The checkpoint methods are invoked by each session and then added to the *Child_info*-class of the current checkpoint.

Affected source files:

- include/rtcr/cpu/*
- include/rtcr/log/*
- include/rtcr/pd/*
- include/rtcr/rm/*

In figure 5.4, a simplified structure shared by many functional classes is shown. The constructor allocates the associated capability via the provided *Genode::Env* and stores it in the corresponding *cap* variable. A *Genode::Allocator* is required for the dynamic creation of the corresponding info-class during a checkpoint.

The *DISABLE_COPY*-macro deletes the copy assignment and constructor and thus implicitly also the move assignment and constructor [18, § 15.8.1.8]. This ensures that only one functional class (Ram_dataspace in this example) references to the same capability. Consequently, the destructor, which is responsible for freeing the capability via its parent, will only be called once. Otherwise, having multiple owners of the same capability would lead to double frees.

As mentioned before the basic structure shown in figure 5.4 is shared by most classes

```
1  class Rtcr::Ram_dataspace : public Genode::List<Ram_dataspace>::Element {
2          DISABLE_COPY(Ram_dataspace);
3          Genode::Env &_env;
4          Genode::Allocator &_alloc;
5
6  public:
7          const Genode::Ram_dataspace_capability cap;
8
9          Ram_dataspace(Genode::Env &env, Genode::Allocator &alloc) : _env(env),
10         _alloc(alloc) { /*Allocate capability*/ }
11         ~Ram_dataspace() { /*Deallocate capability*/ }
12         Ram_dataspace_info *checkpoint() const { /*Create checkpoint of
               capability*/ }
13         void restore(const Ram_dataspace_info &info) { /*Restore capability from
                checkpoint*/ }
14 };
```

Figure 5.4.: Basic structure shared by most functional classes.

with the notable exception of the *Signal_source*-class as Genode will hang for unknown reasons, when allocating the capability within its constructor. This bug could not be resolved in this thesis. Instead, capability allocation is still handled by the PD-session, while deallocations occur within the Signal_source destructor.

Next, the *Child_info* class was also allocated before the child process is launched and reused for each checkpoint. In the new implementation, *Child_info* is constructed on demand at a checkpoint. In addition, the session information classes are marked *const* so that they cannot accidentally be altered. This ensures that the state of a checkpoint remains frozen once it is created.

5.2. Implementation of Restore Procedure

As described in the design chapter, the checkpointed memory contents are restored at the PD-session's *alloc* function. After it has allocated the actual dataspace through its parent, RTCR checks for available checkpointing information. If it is available, memory contents are restored back via the *restore* function of the newly

Affected source files:

- include/rtcr/pd/pd_session.h
- include/rtcr/pd/ram_dataspace.h
- src/rtcr/pd_session.h

```
1  void restore(const Ram_dataspace_info &info, const Genode::addr_t skip,
2          const long long amount) {
3          //Attach child dataspace
4          void *data = _env.rm().attach(cap);
5          auto size = amount < 0 ? this->size - skip : amount;
6
7          //Copy memory contents from checkpoint
8          Genode::memcpy(data + skip, info.data + skip, size);
9
10         //Detach child dataspace
11         _env.rm().detach(data);
12 }
```

Figure 5.5.: Simplified implementation of a dataspace restore.

allocated *Ram_dataspace*.

Figure 5.5 shows a simplified version of the restore implementation. Mostly log messages and size constraints checks have been omitted. Basically, the dataspace of the child is attached to the address space of RTCR, the memory content from the provided checkpoint is copied back and finally, the dataspace is again detached from the address space. Restore also takes *skip* and *amount* as additional arguments for specifying the amount of bytes that will be ignored and copied respectively. Setting the first to zero and the latter to a negative value will restore the entire dataspace. Both parameters are useful for restoring the stack more efficiently, which is discussed in chapter 5.2.2.

At last, *_restore_next_ram* is set to the next checkpointed dataspace in the list. This one will be used next for restoring a dataspace allocated in the future. The idea behind this algorithm is explained in the following subsection.

5.2.1. Heuristic for Locating Required Dataspace

Before RTCR can initiate the restore process, the appropriate checkpointed dataspace needs to be found. Capabilities are uniquely identifiable by comparing their badges, however, the newly created and checkpointed dataspaces both refer to two different capabilities. Therefore, a good heuristic algorithm is required to find the matching dataspace. The initial implementation used in this thesis simply compared sizes of the newly allocated and checkpointed dataspace until it has found one equally large. However, this heuristic only works as long as each dataspace created during the lifetime of the child has a different size. In practice, this is almost impossible to avoid as Genode itself already allocates dataspaces with equal sizes. In addition, the goal of RTCRv3 is

```
1 void Component::construct(Genode::Env &env) {
2         // Check whether we are supposed to restore this thread
3         if (env.cpu().restore_thread(Thread::myself()->name())) {
4                 // Restore will be initiated
5                 while (true) {
6                         __asm__("NOP");
7                 }
8         }
9 }
```

Figure 5.6.: Trigger for restoring the instruction and stack pointers.

to work transparently meaning that the child process must not care about restrictions of RTCR (e.g. when allocating dataspaces manually). Therefore, this implementation was not suitable in the long run but it helped confirm that the design of the memory restorer is working in its essence.

The next and current version of the algorithm did not compare any properties of the ckeckpointed dataspaces with the newly allocated one. Instead, dataspaces are simply restored in order of the list. This approach is based on several assumptions:

$$C := \text{Tuple of checkpointed dataspaces} \tag{5.1}$$

$$t_x := \text{Time of allocation of dataspace } x \tag{5.2}$$

$$\forall x, y \in C : \quad ord(x) < ord(y) \iff t_x < t_y \tag{5.3}$$

Equation 5.3 states that all checkpointed *Ram_dataspaces_info*s as part of the *Pd_session_info* (5.1) must be ordered by the time of capability allocation (5.2). In addition, the Genode bootstrap process must be deterministic in a way so that all Ram_dataspaces allocations during this phase **always** happen in the same order.

5.2.2. Restoring Stack and Instruction Pointer

Unlike restoring memory, the stack and instruction pointer cannot be reset directly, as the initial thread *ep* needs to run the bootstrap code first. In order to notify RTCR, that the init-code has completed, the thread needs to explicitly notify RTCRv3 that the thread can now be restored. Thus, the CPU-session was ex-

Affected source files:

- include/rtcr/cpu/cpu_session.h
- include/rtcr/pd/pd_session.h
- src/rtcr/cpu_session.h
- src/rtcr/pd_session.h
- lib/mk/rtcr.mk

tended to provide a method named *restore_thread*, as can be seen in figure 5.6.
The *restore_thread*-method asks RTCR whether the thread with the provided name needs to be restored. If RTCR returns false because no appropriate checkpoint information is available, the thread may continue its execution. If true is returned, the thread should prepare for a thread restore. In order to avoid undefined results, the affected thread should put itself in an endless loop. In theory, this should not be required because RTCR could simply reset the IP/SP during the RPC-call, which blocks the child's thread anyway. However, during implementation, it was found that Genode locks up, when trying to pause and reset the thread state within an RPC-call. In order to work around this problem, the parent thread starts another thread to take care of the restore procedure. In the meantime, the RPC routine immediately returns. The restore thread then performs the following steps:

- **Pausing the target child thread:** This is needed in order to avoid the interference of the child thread with the thread state and the stack which could lead to undefined results otherwise.
- **Restoring the stack dataspace again:** As explained in subsection 4.3, previous function calls may grow/shrink the stack (not to be confused with the dataspace used for the stack). Using the *sheep_counter* program as an example, the memory address of the counter variable could already have been used by a previous function invocation. Therefore, before the execution can be continued from the saved SP/IP, RTCRv3 must memcopy back the checkpointed data first. However, this process can be slightly optimized. Refer to subsection 5.2.3.
- **Allocating remaining dataspaces:** During the execution of a program, additional dataspaces can be freely added at any time as long as resources are available (e.g. when using a heap). However, RTCRv3 by default restores dataspaces when they are allocated, but for instance when a program uses a heap, which would normally allocate an additional dataspace during its construction, the allocation is skipped when restoring a thread. Thus, all checkpointed dataspaces that have not yet been used, will be allocated, restored and attached to the address space according to checkpointed information.
- **Restoring the state (registers, stack pointer, instruction pointer, etc.) to their checkpointed values.**
- **Resuming thread execution.**

As observed in figure 5.6 the call to *restore_thread* occurs within *Component::construct* which happens to be part of application-specific code. This breaks one of RTCRv3 main design rules to be fully transparent and not requiring any special code in the child process. During implementation an attempt to call *restore_thread* right before *Component::construct* (in $GENODE_DIR/repos/base/src/lib/base/entrypoint.cc) has been made. Interestingly however, Genode would lock up for reasons which could not be resolved. Thus, while the current implementation breaks the transparency principle, it should theoretically be possible to reverse this drawback.

5.2.3. Improving Stack Restoration

During development of RTCRv3, it was found that performance could be improved by skipping restoration of stack dataspaces during dataspace allocation and only restore necessary data. This idea is based on the following observations:

- As described in sections 4.3 and 5.2.2, the stack must be "repaired" anyways, as data restored at dataspace allocation could have been overwritten. Thus, the initial memory copy at allocation time is unnecessary.
- RTCRv3 is designed to let Genode regularly bootstrap the child process. This means there is an upper bound address for the stack pointer, until RTCR does not need to memcopy back as these data structures will be created during the Genode bootstrap phase anyway. In theory, RTCRv3 can assume that the upper bound stack address is at the time of thread restoration, meaning when the *restore_thread* method is called.
- As per the definition of a stack, it grows continuously in one direction (usually downwards), unlike a heap, which can arbitrarily allocate data within a dataspace. Therefore, a heap dataspace must always be recovered entirely because RTCRv3 cannot know which parts of the dataspace are unused. However, in the case of a stack dataspace, the restorer knows the stack pointer at the time of a checkpoint. This means that the checkpointed stack pointer address is a lower bound where before this point RTCRv3 does not have to recover the stack as it was unused by the checkpointed child process.

This shows that RTCRv3 only needs to restore data within the said limits because in theory, only this area of the stack is relevant for the proper functioning of the child process. Figure 5.7 shows how the parameters of *restore* can be adjusted to partially restore a dataspace from a checkpoint.

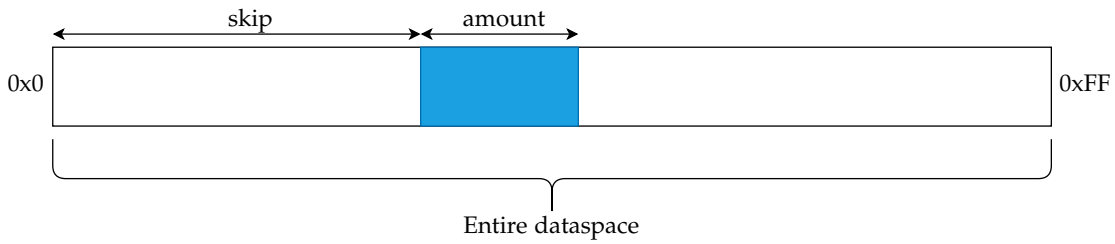

Figure 5.7.: Example illustration of the behavior of *restore* on *Ram_dataspace* and its parameters. The blue area is the data restored from the checkpoint.

```
1  Stack area:
2         Region Map: badge=1252, size=0, ds_badge=1248, sigh_badge=0
3         badge=1560, [0xfc000, 0x100000) exec=no
4         badge=1564, [0x1f0000, 0x200000) exec=no
5         badge=1600, [0x2fc000, 0x300000) exec=no
6  Linker area:
7         Region Map: badge=1260, size=0, ds_badge=1256, sigh_badge=0
8         badge=1508, [0x0, 0x0) exec=yes
9         badge=1644, [0x1000, 0x1000) exec=no
10 Ram dataspaces:
11        badge=1352, size=12288, cached=2
12        badge=1556, size=177452, cached=2
13        badge=1560, size=16384, cached=2
14        badge=1564, size=65536, cached=2
15        badge=1600, size=16384, cached=2
16        badge=1644, size=332, cached=2
17        badge=1648, size=16384, cached=2
18        badge=1652, size=81920, cached=2
```

Figure 5.8.: Default dataspaces of a Genode component without any manual allocations. Highlighted lines show dataspaces used as stacks. Sizes are provided in bytes.

The proposed optimization might particularly benefit RTCR, because around 25.4 % of all dataspaces (excluding manual allocations) in bytes are part of the stack area-region map. This was verified using checkpointed information seen in figure 5.8.

Unfortunately, this approach could not successfully be implemented as no reliable way to retrieve the stack pointer at the time of *restore_thread* was found.

The first attempt was to simply retrieve the thread state after it was paused. However, the stack pointer interestingly always had the same address as the checkpointed thread

```
1  void Component::construct(Genode::Env &env) {
2          if (env.cpu().restore_thread(Genode::Thread::myself()->name())) {
3                  while (true) {
4                          __asm__("NOP");
5                  }
6          }
7          uint64_t magic_val = 0xfc792fbcf18c5ed0;
8          log(&magic_val);
9          uint8_t n = 1;
10         while (true) {
11                 log("Sheep: ", n);
12                 n++;
13         }
14 }
```

Figure 5.9.: Simpified version of the sheep_counter program.

```
1  void Component::construct(Genode::Env &env) {
2          1000000: e92d40f0 push {r4, r5, r6, r7, lr}
3          1000004: e24dd04c sub sp, sp, #76 ; 0x4c
4          ....
```

Figure 5.10.: Excerpt of the disassembled code seen in 5.9. The full code can be found in appendix A.2.

state. Looking at the simplified source code of the sheep_counter in figure 5.9, there are multiple variables declared after the endless loop, in which a restoring thread locks up. Thus, if a checkpoint *c* was initiated at line 10, the stack pointer should theoretically deviate at least by a few bytes due to the additionally allocated variables.

The only way to fully investigate this matter further is by looking at the resulting assembly code. Practically, the Genode build system by default produces two versions of each binary. One contains debugging symbols, while the other is stripped. Both can be found under *$GENODE_DIR/build/armv_v7a/app/sheep_counter/*. Using *objdump*, the binary can be disassembled and inspected in a regular text editor. The parameters used can be found in appendix A.1.

Looking at the ARM assembly code of *Component::construct* revealed that only two instructions affect the stack pointer as seen in figure 5.10. The first one is a *PUSH* of a few registers onto the stack while the second one is a *SUB*-instruction specifically decreasing the stack pointer by an immediate value. Thus, the size of the stack is

```
1  void __attribute((noinline)) helper() {
2          int local{0};
3          Genode::log("Address␣2:␣", &local);
4  }
5
6  void grows_down() {
7          int local{0};
8          Genode::log("Address␣1:␣", &local);
9          helper();
10 }
```

Figure 5.11.: Simple method to verify stack growth direction.

increased. Both instructions are located at the beginning of the function. The ARMv7-A Architecture Reference Manual was used for verification [1, p. A8-717].

Therefore, the stack pointer is an unreliable source to determine the relevant upper bound as the stack pointers of the checkpointed and newly started process will very likely have the exact same value. A proposed solution can be found in subsection 7.4.

Still, the current implementation in RTCRv3 optimizes the process slightly by restoring only data above the checkpointed stack pointer and thus, skipping unused areas of the stack. This thesis assumes that the stack always grows down which might not always be the case depending on the architecture and kernel in use. One simple way to dynamically retrieve the direction of the stack is shown in figure 5.11. If the value at "Address 1:" is higher than at "Address 2:", the stack must grow down. On the other hand, a lower value means growth in the opposite direction. Note that the *helper* function must not be inlined as the order of both local variables could be changed otherwise.

Finally, dataspaces used for stacks are still fully restored during *alloc*, even though this is not necessary as mentioned in the beginning of this section. This optimization was purposely not implemented because RTCRv3 cannot know whether a newly allocated dataspace will be used for a stack since this information is not provided during allocation. Therefore, a heuristic or other architectural change is required to properly detect this during *alloc* which is not in the scope of this bachelor's thesis.

6. Evaluation

The following subsections discuss the method of evaluating RTCRv3 restore performance as well as the test cases considered.

6.1. Genode Profiler

On a regular Linux machine, developers have access to powerful tools such as *perf* for cpu cycles measurements, cache hit rates and more. Unfortunately, this kind of tool is not available in a specialized environment such as Genode.

Thus, Johannes Fischer has developed a simple profiler based around timers and print-statements. The integration into the application is straightforward: A Genode component must establish a connection to a timer (usually provided by the base platform) and associate it with the profiler using *PROFILE_INIT*. A new measurement can be initiated by inserting the *PROFILE_FUNCTION*-macro. It expands to creating an instance of the class *Profiler::Scope*, which stores the starting time in its constructor. Once the function returns, the instance goes out of scope and its destructor retrieves the elapsed time. Both start and stop values are then printed on console formatted in JSON. However, since *log* has some considerable overhead due to propagation via all parent processes, Genode provides a function named *raw*, which directly prints via the kernel debugger. As this is a potential security risk and should not be allowed in production, this feature must be explicitly enabled for Fiasco.OC by setting the flag *map_debug_cap* in *$GENODE_DIR/repos/base-foc/src/core/platform_pd.cc* to true.

Nevertheless, the output must be captured by redirecting stdout to a local file. Using the tools provided by the Genode Profiler, it is possible to extract performance values from the logs and additionally plot them as an SVG.

Apart from measuring functions, the *PROFILE_SCOPE*-macro allows for profiling only certain blocks of code. Hence, it must be provided with a name for the scope, which will be used for evaluation.

6.2. Test Applications and Environment

Multiple test programs have been written to evaluate the performance of the memory restore. They are mainly designed to highlight optimizations applied to RTCRv3 as well as the overhead during *alloc*-calls when compared to allocations without a restore. The source code has been attached to appendix B.

Affected source files:

- run/test_*
- src/app/rtcr_benchmark/main.cc
- src/test/*

- **small_dataspaces**: This test is particularly interesting when compared to the results of *large_dataspaces* as both allocate the same amount of memory. While the first spreads it across 100 dataspaces with a minimum size of 4096 bytes each, the latter allocates the same amount of memory in the form of a single dataspace (4096 * 100 bytes).

- **stack_counter:** This program allocates a *uint16_t* counter variable on the stack, which is incremented in an endless loop. This test is a simplified version of the *sheep_counter* and serves as a minimal verification for a stack restore.

- **large_dataspace:** As described above, this program is designed as a counterpart to *small_dataspaces* and allocates one large dataspace with 409600 bytes instead of many small ones. Comparing measurements of both tests allows to evaluate the effects of RTCRv3 as its overhead becomes more evident when more dataspaces must be restored.

- **large_stack_counter:** This test is similar to the regular *stack_counter* except that its stack is nearly fully utilized. By default, Genode provides a stack of 65536 bytes for the *ep*-thread. Therefore, *large_stack_counter* allocates an array of 31650 *uint16_t*-values to nearly deplete it.

  The goal is to measure the performance benefit of an implemented optimization described in section 5.2.3. Thanks to this, *stack_counter* should require significantly less time for restoration.

All tests are executed by a benchmarking program found in *src/app/rtcr_benchmark/main.cc*. It launches the configured child process and waits a number of iterations before it will be paused, checkpointed and restored in a new instance. The program waits once more before it terminates itself. The number of iterations can be configured dynamically as well as can be seen in one of the run-files (*run/test_restore_stack.run* for example).

Finally, all programs and libraries are compiled with an optimization level of *-O2* (as defined by default in *$GENODE_DIR/repos/base/mk/global.mk*).

6.3. System Environment

The development was mostly done on Ubuntu running in VirtualBox. However, virtual machines add unpredictable overhead which was apparent in initial tests resulting in random and highly fluctuating values. Therefore, the setup was recreated on a bare-metal installation albeit the benchmark was still executed using QEMU, which is used for arm_v7-emulation.

The processor used for measurements was an Intel Core i7-7700HQ (4 Cores with SMT) and DDR4 memory with a configured speed of 2133 MT/s. The cpu frequency was locked to 800 MHz as described later in section 6.4.

6.4. Test Results

While the Genode profiler is capable of drawing graphs for all measures automatically, they were found too large to be included in this chapter and therefore, have been attached to appendix C.

When looking at a graph, the x-axis represents the continuous flow of time, while every line on the y-axis presents a certain function or scope. Each colored block on a line is a single measurement. The numbers after the last block show the total execution time of all measurements of this block/function.

Some lines were given specific names to highlight their meaning, as explained in the following:

- **Dataspace Allocation:** Assesses the execution time of *Rtcr::Pd_session::alloc* **without** restore information. This basically represents the raw allocation performance.
- **Dataspace Allocation and Restore:** Refer to item above. This measures the execution time of *alloc* **with** restore information available. Thus, each assessment is also accompanied by a *Memcpy* measurement. The difference between the results of dataspace allocations with and without a restore allows to evaluate the overhead induced by RTCRv3.
- **Memcpy (alias *Rtcr::Ram_dataspace::restore*):** Shows the duration of a single dataspace restore.

Table 6.1.: Original measurements of *stack_counter* and *large_stack_counter*. Results in ms.

	stack_counter	large_stack_counter
Dataspace Allocation	7	6
Dataspace Allocation and Restore	37	32
Memcpy	13	22
Stack Repair	8	7
Total Restore Duration	242	248
Total Thread Restore Duration	21	58
Bootstrap Duration	n/a	n/a

- **Stack Repair *(alias Rtcr::Pd_session::restore_ds)*:** Measures the time of restoring the stack during a thread restore. Note that the graphs found in appendix C additionally contain the values *Dataspace Region-Map Look-Up* and *Dataspace Checkpoint Look-Up* which are subscopes of *Stack Repair*.
- **Thread Restore:** Measures the time for fixing the stack, restoring remaining dataspaces and resetting the instruction and stack pointer to their checkpointed values. Therefore, each measurement also contains one for a *Stack Repair*.

Furthermore, the blocks in the graphs are color coded. **Red** ones are part of checkpointing-related functions, while **blue** ones represent restore-related methods. **Violet** blocks show pure *Dataspace Allocations* and finally, **green** values are used for thread resume/pause-operations.

The most relevant numbers have been extracted and presented in tables 6.1, 6.2 and 6.3. Note that the *Total Restore*, *Total Thread Restore* and *Bootstrap Duration* values have been calculated manually. The first two can be recomputed by subtracting the end of the *Thread Restore*-measurement from the initial *Dataspace Allocation and Restore/restore_thread*-timestamps respectively. The *Bootstrap Duration* is the difference of these two values.

Result Inaccuracies

When analyzing table 6.1, a few interesting points can be observed. First of all, the *Dataspace Allocation* differs by one ms, although it should approximately be equal for both test programs as they both allocate the exact same amount of dataspaces with the same sizes as well. The same applies for *Dataspace Allocation and Restore*, which deviates by even 5 ms, which is quite a lot when considering the scale of these values.

```
1  cat /sys/devices/system/cpu/cpu0/cpufreq/
       energy_performance_available_preferences
2  # OUTPUT: default performance balance_performance balance_power power
3  # Set performance hint to "power" for all cores
4  sudo x86_energy_perf_policy --all power
5  # Verify
6  cat /sys/devices/system/cpu/cpu0/cpufreq/energy_performance_preference
7  # OUTPUT: power
8  cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_min_freq
9  # OUTPUT: 800000
10 cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq
11 # OUTPUT: 800000
```

Figure 6.1.: Configuring the *intel_pstate*-driver to let the CPU run as "slow" as possible.

The main cause for this difference likely comes from inaccurate results as the Genode Profiler assesses time in milliseconds. However, this unit is too coarse-grained for measuring small scopes, where often many of them take less than 1 ms to execute. Dataspace allocations for instance occur multiple times but due to their short duration, values are often rounded down to either 1 or 0 ms.

One approach attempted at first was to assess applications in microseconds, as Genode already provides an appropriate *elapsed_us*-method. However, the observed results still seemed to only have millisecond-precision.

Another idea was to limit cpu usage to purposely decrease performance and receive durations on a larger scale. Unfortunately, no appropriate option for QEMU was found to achieve this. Instead, the intel_pstate driver of the Linux-kernel, responsible for tuning power/performance metrics of the host cpu, could be configured to manipulate the execution of RTCRv3. The driver provides different performance levels (see line 1 in figure 6.1), where one of them is *power*. According to the output of *scaling_min_freq* and *scaling_max_freq* (see figure 6.1), this locks the cpu to 800 MHz. [4]

Rerunning the tests resulted in higher values as expected, however, they were still deviating too much. This can have multiple causes:

- **Influence of Host-System:** While all measurements were taken with no major applications running, background processes could still lead to slight deviations in the results. In addition, scheduling issues mentioned below apply to the Linux kernel of the host system as well. This point could likely be the main cause for the deviations since interference from other processes in combination with the decreased performance of the host CPU has a higher impact.

- **Overhead of *elapsed_ms*:** The function used to receive the current timestamp is an RPC-call to a timer component which in turn must communicate with the Fiasco.OC-kernel. This overhead involves multiple context switches which decreases the accuracy of the measured value.

  On the Genode mailing list it was suggested to use *Trace::timestamp* provided by Genode. This function returns an architecture dependent value. On arm_v7 it returns the *Performance Monitors Cycle Count Register* (PMCCNTR) which is incremented either every single or every 64 processor cycles [1, p. B6-1894]. Using this value, more accurate results could be achieved [17]. However, due to running inside an emulator, this technique was not considered useful but it definitely should be applied when testing on real hardware.

- **Scheduling and context switching:** The current setup has no influence on the scheduling policies applied by the Fiasco.OC-kernel. Unlucky scheduling by the Fiasco.OC kernel can lead to unpredictable context switches. For instance, all memcpys during a restore of a *stack_counter* could have been interrupted by fewer context switches than its counterpart *large_stack_counter*, resulting in different durations for the same code path.

To work around these problems, 20 runs instead of a single one were conducted for each test application in order to then calculate averages for all scopes. Unfortunately, the Genode profiler is not capable of handling multiple measurements independently. Therefore a custom tool named *profiler-analyzer* was written in Rust. This program receives multiple files as commandline arguments, parses their profiling data, calculates the average for each measurement taken and writes the results into *results.json* of the current working directory. The outputted file shares the same structure as the ones generated by *profiler-filter* (part of the Genode Profiler), which allows plotting the data with the same tools. Note that the order of the measurements in the input files must be the same across all of them. Otherwise, the *profiler-analyzer* will panic accordingly.

Evaluation

When analyzing table 6.2 and 6.3, all test scenarios take equally long for bootstrapping the process (around 815 ms on average). This is expected as this phase is part of Genode itself and therefore the same for all programs. The *Stack Repair*-duration is also quite similar, interestingly, even in the case of *large_stack_counter*. In theory, this test should yield longer durations due to the application having a larger stack. This observation can be analysed further in table 6.4, where the memcpy-duration of the stack repair has been extracted from the averaged profiling data. The table clearly shows that memory pressure does not seem to be an issue as the duration is even shorter than compared to

Table 6.2.: Comparison of *stack_counter* and *large_stack_counter*. Results in ms. Each value is an average of 20 runs.

	stack_counter	large_stack_counter
Dataspace Allocation	39.65	25.85
Dataspace Allocation and Restore	142.65	124.65
Memcpy	60.85	53.00
Stack Repair	54.15	47.00
Total Restore Duration	941.2	1017.45
Total Thread Restore Duration	141.7	141.7
Bootstrap Duration	799.5	875.75

Table 6.3.: Comparison of *small_dataspaces* and *large_dataspaces*. Results in ms. Each value is an average of 20 runs.

	small_dataspaces	large_dataspaces
Dataspace Allocation	324.00	51.25
Dataspace Allocation and Restore	1948.35	191.75
Memcpy	663.00	100.2
Stack Repair	54.75	50.75
Total Restore Duration	3839.55	1001.35
Total Thread Restore Duration	3049.95	214.5
Bootstrap Duration	798.6	786.85

Table 6.4.: Stack analysis of *stack_counter* (sc), *large_stack_counter* (lsc), *small_dataspaces* (sd) and *large_dataspaces* (ld).

	sc	lsc	sd	ld
Stack Repair	54.15 ms	47.00 ms	54.75 ms	50.75 ms
Memcpy (Stack Repair only)	8.2 ms	7.85 ms	7.6 ms	10.3 ms
Stack size	1072 bytes	64376 bytes	1504 bytes	1088 bytes

Table 6.5.: Dataspace allocation analysis of *stack_counter* (sc), *large_stack_counter* (lsc), *small_dataspaces* (sd) and *large_dataspaces* (ld).

	sc	lsc	sd	ld
Dataspace Allocation	39.65	25.85	324 ms	51.25 ms
Dataspace Allocation and Restore	142.65 ms	124.65 ms	1948.35 ms	191.75 ms
Relative overhead	72.2 %	79.3 %	83.3 %	73.3 %
Relative overhead (w/o Memcpy)	51.5 %	63.9 %	74.8 %	44.0 %

those of programs with a considerable smaller stack. Note that this could significantly differ on real hardware as the memory subsystem of an embedded system is usually less potent than on regular personal computers.

Another interesting case is *small_dataspaces*. Table 6.3 clearly shows that a program generally requires more time for restoring many small dataspaces, especially when compared to *large_dataspaces* which allocates the same amount of data in a single dataspace. Furthermore, to better understand the impact of the restore process on the overall allocation, table 6.5 shows the relative overhead induced by RTCRv3. As expected, the overhead gets larger when creating many small dataspaces. On the other side, the table also reveals that most of the overhead does not originate from memory copies but comes from RTCR itself. One notable exception is *large_dataspaces* as it requests only one additional but large dataspace. This overhead by RTCRv3 is particularly irritating because all operations in the restore-procedure of *Pd_session::alloc* are O(1) and no other computational demanding code is executed, especially when compared to memory copies. For confirmability, the full *alloc*-source code has been included in figure 6.2.

Validity

In conclusion, the results do not match the expected outcome, which is likely a cause of inaccuracy rather than faults in the implementation. Thus, all measurements above should not be considered valid without further research. For instance, a program with a larger stack cannot consume less time to restore than one with a smaller stack assuming the implementation is correct. Furthermore, the high overhead of RTCRv3 during allocation cannot be reasonably explained. A suggested way to resolve this issue is by conducting tests on actual arm_v7 hardware, see also section 7.7.

```
1  Genode::Ram_dataspace_capability
2  Pd_session::alloc(Genode::size_t size, Genode::Cache_attribute cached) {
3          const char *scope_name{};
4          const char *color{};
5          if (_restore_info && !_ds_allocation_completed) {
6                  scope_name = "Dataspace␣Allocation␣and␣Restore";
7                  color = "blue";
8          } else {
9                  scope_name = "Dataspace␣Allocation";
10                 color = "violet";
11         }
12         PROFILE_SCOPE(scope_name, color);
13
14         // Create Ram dataspace via parent
15         auto ds = new (_md_alloc) Ram_dataspace(_env, _md_alloc, size, cached);
16         if (_restore_info && !_ds_allocation_completed) {
17                 // SCOPE: "Dataspace Allocation and Restore"
18                 if (!_restore_next_ram) {
19                         _restore_next_ram = _restore_info->pd_session->
                               ram_dataspaces;
20                 }
21
22                 ds->restore(*_restore_next_ram);
23                 _restore_next_ram = _restore_next_ram->next();
24                 if (!_restore_next_ram) {
25                         // All dataspaces have been restored
26                         _ds_allocation_completed = true;
27                 }
28         }
29
30         Genode::Lock::Guard guard(_ram_dataspaces_lock);
31         _ram_dataspaces.insert(ds);
32
33         return ds->cap;
34 }
```

Figure 6.2.: Full implementation of *alloc* in RTCRv3.

7. Limitations and Future Work

This chapter summarizes the unresolved issues that occurred during this thesis as well as other improvements that can be applied to RTCR in the future.

7.1. Decouple Memory Management

For each functional class, a corresponding checkpoint class is allocated on the heap. However, in a realtime scenario, this is undesired as heap allocations do not necessarily have any guarantees regarding their complexity. Instead, the design of the checkpoint storage should be left up to the RTCR implementation.

For instance, all *checkpoint* methods could receive an already allocated info-class by reference. A more advanced possibility would be to offer an interface for a "Data Provider". This would entirely defeat the need for a *Genode::Allocator* inside the RTCR-library because it would request new data structures via the "Data Provider". The concrete allocation strategy used (pre-allocated, on demand, etc.) is decided by the implementor of this interface.

7.2. Improving Deallocation to O(1)

Each time a capability is removed, the corresponding session must lookup the owning functional class in its list, which leads to an upper bound of O(n). This could result in problems in scenarios where capabilities are allocated/freed more frequently in a random fashion by the child. A proposed solution could be the use of a hash table instead of a list, which, depending on the used hash implementation, leads to an upper bound of O(1) for random access. Unfortunately, Genode does not provide an std::map or equivalent data structure in its base library.

7.3. Restoring Transparency

Section 5.2.2 describes the implementation of restoring a child thread. Currently, it is required by the application developer to insert a *restore_thread* call at the beginning of its *Component::construct* method. This breaks one of the main design rules in that

RTCR must work transparently and hence does not require special support by the child process.

This drawback could be resolved by integrating the restore-call into Genode right before calling *Component::construct*. However, this idea lead to the child process consistently locking up.

7.4. Unnecessary Overhead in Stack Restoration

As described in section 5.2.2, stack dataspaces must be restored twice as necessary data could have been overwritten during the Genode bootstrap phase. In section 5.2.3 a performance improvement is proposed by skipping the initial dataspace restore at allocation time and only restore mandatory parts of the stack later when the SP/IP are set to their checkpointed values.

This proposal was not fully implemented due to reasons partially described in section 5.2.3. Notably, the following features are missing:

- Data allocated during bootstrapping are restored from the checkpoint as reliably locating an upper bound for restoring the stack was not possible. One solution could be to modify the signature of *Component::construct*, which additionally accepts an entry stack pointer as an argument. The stack pointer can sort of be retrieved via the method in figure 5.11. Those functions, however, must be called outside of *Component::construct* and thus, Genode must be modified to retrieve the stack pointer before the application-specific code is entered.

  However, it should be noted that during this bachelor's thesis, it was not investigated whether using the newly allocated structures during bootstrap can be reliably used or restoring the checkpointed ones are necessary.

- Stack dataspaces are still fully restored at allocation time because the PD-session does not know what the newly created dataspace is used for. This is also harder to track because dataspaces are attached in another class (*Region_map*).

7.5. Detached Dataspaces Not Restored

A child process can arbitrarily allocate new dataspaces by asking its parent. All dataspaces created after the Genode bootstrap phase are restored by RTCRv3 during a thread restore as described in section 5.2.2. However, RTCRv3 will only reallocate dataspaces which were attached at the moment of a checkpoint.

Figure 7.1 shows an example case where a restore consistently fails. The child process allocates two dataspaces DS1 and DS2, but only the first is also attached. Next, a new

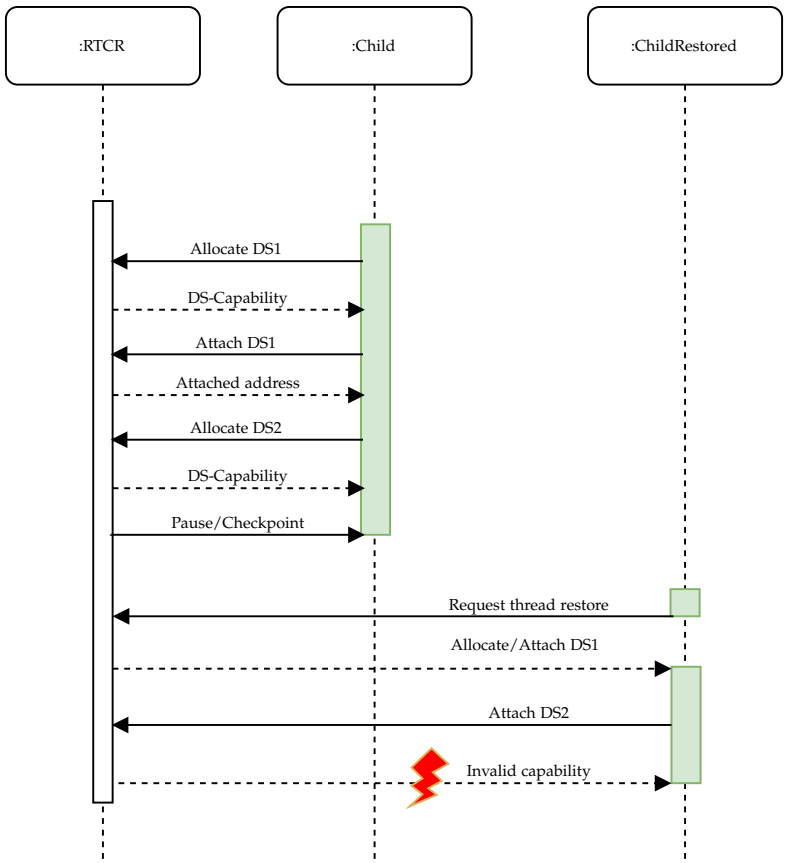

Figure 7.1.: Example case of dataspaces not restored by RTCRv3.

instance of the process shall be restored, but only DS1 is allocated and attached by RTCRv3. However, *ChildRestored* assumes DS2 has been allocated (as it does not know about RTCR) and attempts to attach it. Needless to say, Genode responds with an *Invalid dataspace*-exception.

7.6. Checkpointing Fails during RPC-Call

Before a checkpoint can be initiated all child threads must be paused first to prevent any interference. However, when attempting to pause a thread, which is waiting for an RPC-response, Genode will hang for unknown reasons.

7.7. Real Hardware Testing

Section 6.4 discussed the inaccuracies and validity of the gathered assessments. A better approach for verification is by running all test suits on proper arm_v7-hardware. In addition, a timer was used for measuring elapsed times, which should be avoided as well due to overhead cause by RPC-calls and context switches. In the Genode mailing list it is proposed to use cpu counters, which can be accessed using *Trace::timestamp* [17].

7.8. Testing with Applications in Production

RTCRv3 was mostly tested using the *sheep_counter* introduced in 5.1.1 and specific scenarios applied for evaluation in chapter 6. These tests are sufficient for validating the basic functionality of RTCRv3. However, real-world applications are required for identifying actual bottlenecks, missing functionality and critical bugs. In addition, they are also important to test RTCRv3's long-term stability, considering that it is ultimately designed to be deployed in safety-critical areas.

Testing on applications that are used in production would be optimal, however, developing themselves can be a proper alternative as well.

8. Conclusion

During this thesis restoring all dataspaces and the state of the *ep*-thread has been presented. Furthermore, the architecture has been improved in terms of flexibility to make future adaptations more easily. One notable example is the decoupling of the checkpoint storage from the functional aspect of RTCR. This was also deemed necessary to add support for restore functionality.

Most importantly, RTCRv3 manages to completely recreate a child component based on a checkpoint. All dataspaces are restored during allocation in the *Pd_session* one by one, meaning all checkpointed dataspaces must be in order. As a last step, the thread must be restored to its checkpointed state. Since the selected design lets Genode regularly bootstrap the child component, a barrier is required to notify RTCR that the corresponding *ep*-thread is ready to be reset. This mechanism has been integrated into application-specific code. While the research goals for this thesis include no restrictions in terms of how a thread may be restored, it breaks the transparency principle specified for the design of RTCRv3.

In addition, an optimization to improve the restore duration of stack dataspaces has been integrated as research has shown that a significant amount of unused data can be skipped. However, the implementation of this improvement was not completed. Most notably, stack dataspaces are still fully restored at allocation because RTCRv3 can currently not predict how a newly allocated dataspace will be used. Thus, some heuristic is required which could significantly increase the size of the code base of RTCRv3. As this would not align with keeping the architecture simple, this part has not been implemented.

While the performance of the restore-mechanism is evaluated in this thesis, the results should be verified on actual arm_v7-hardware first because some measurements did not match the expected outcome. This is likely a cause of emulation inaccuracies and influence of the host system. Furthermore, the test cases used have been developed to specifically highlight the overhead of RTCRv3. Future research should focus on evaluating real-world programs in order to identify parts of the restore-mechanism requiring further investigation.

In summary, this bachelor's thesis has shown that a functional and reliable memory and thread restore can be implemented in Genode. Furthermore, it has laid architectural corner stones for future extensions to RTCRv3.

A. Disassembly

A.1. Commands

```
1 /usr/local/genode/tool/19.05/arm-none-eabi/bin/objdump -S --disassemble $
    (GENODE_DIR)/build/arm_v7a/app/sheep_counter/sheep_counter >
    sheep_counter.dump
```

The command above disassembles the given binary (*sheep_counter* in this example) and saves it in *sheep_counter.dump*. Note that the used *objdump*-binary must match the architecture of the compiled program. Replace *arm-none-eabi* accordingly. Short explanation of the given paramters.

- **-S** - Include source code in assembly output
- **–disassemble** - Disassemble binary

A.2. Disassembled Simplified sheep_counter

```
1  void Component::construct(Genode::Env &env) {
2          1000000: e92d40f0  push {r4, r5, r6, r7, lr}
3          1000004: e24dd04c  sub sp, sp, #76 ; 0x4c
4          if (env.cpu().restore_thread(Genode::Thread::myself()->name())) {
5                  1000008: e5903000  ldr r3, [r0]
6                  100000c: e28d4024  add r4, sp, #36 ; 0x24
7                  1000010: e593300c  ldr r3, [r3, #12]
8                  1000014: e12fff33  blx r3
9                  1000018: e5903000  ldr r3, [r0]
10                 100001c: e1a05000  mov r5, r0
11                 1000020: e593602c  ldr r6, [r3, #44] ; 0x2c
12                 1000024: eb000159  bl 1000590 <_ZN6Genode6Thread6myselfEv@plt>
13                 1000028: e1a01000  mov r1, r0
14                 100002c: e1a00004  mov r0, r4
15                 1000030: eb00014d  bl 100056c <_ZNK6Genode6Thread4nameEv@plt>
16                 1000034: e28de030  add lr, sp, #48 ; 0x30
17                 1000038: e1a0c00d  mov ip, sp
18                 100003c: e8be000f  ldm lr!, {r0, r1, r2, r3}
19                 1000040: e8ac000f  stmia ip!, {r0, r1, r2, r3}
20                 1000044: e89e0003  ldm lr, {r0, r1}
21                 1000048: e88c0003  stm ip, {r0, r1}
22                 100004c: e1a00005  mov r0, r5
23                 1000050: e894000e  ldm r4, {r1, r2, r3}
24                 1000054: e12fff36  blx r6
25                 1000058: e2506000  subs r6, r0, #0
26                 100005c: 0a000001  beq 1000068 <_ZN9Component9constructERN6Genode3EnvE+0x68>
27                 while (true) {
28                         __asm__("NOP");
29                         1000060: e320f000  nop {0}
30                         1000064: eafffffd  b 1000060 <_ZN9Component9constructERN6Genode3EnvE+0x60>
```

```
31                  }
32          }
33          uint64_t magic_val = 0xfc792fbcf18c5ed0;
34          1000068: e28f3078  add r3, pc, #120 ; 0x78
35          100006c: e1c320d0  ldrd r2, [r3]
36          log(&magic_val);
37          uint8_t n = 1;
38          1000070: e3a04001  mov r4, #1
39          * Helper for the sequential output of a variable list of arguments
40          */
41          template <typename HEAD, typename... TAIL>
42          static void out_args(Output &output, HEAD && head, TAIL &&... tail)
43          {
44                  print(output, head);
45                  1000074: e59f7074  ldr r7, [pc, #116] ; 10000f0 <_ZN9Component9constructERN6Genode3EnvE+0xf0>
46                  uint64_t magic_val = 0xfc792fbcf18c5ed0;
47                  1000078: e1cd21f8  strd r2, [sp, #24]
48
49                  /**
50                  * Write 'args' as a regular message to the log
51                  */
52                  template <typename... ARGS>
53                  void log(ARGS &&... args) { Log::log().output(Log::LOG, args...); }
54                  100007c: eb00013d  bl 1000578 <_ZN6Genode3Log3logEv@plt>
55                  _acquire(type);
56                  1000080: e1a01006  mov r1, r6
57                  1000084: e08f7007  add r7, pc, r7
58                  void log(ARGS &&... args) { Log::log().output(Log::LOG, args...); }
59                  1000088: e1a05000  mov r5, r0
60                  _acquire(type);
61                  100008c: eb00013c  bl 1000584 <_ZN6Genode3Log8_acquireENS0_4TypeE@plt>
62                  * constant object reference as argument.
63                  */
64                  template <typename T>
65                  static inline void print(Output &output, T *ptr)
66                  {
67                          print(output, (void const *)ptr);
68                          1000090: e5950014  ldr r0, [r5, #20]
69                          1000094: e28d1018  add r1, sp, #24
70                          1000098: eb00013f  bl 100059c <_ZN6Genode5printERNS_6OutputEPKv@plt>
71                          _release();
72                          100009c: e1a00005  mov r0, r5
73                          10000a0: eb000143  bl 10005b4 <_ZN6Genode3Log8_releaseEv@plt>
74                          void log(ARGS &&... args) { Log::log().output(Log::LOG, args...); }
75                          10000a4: eb000133  bl 1000578 <_ZN6Genode3Log3logEv@plt>
76                          _acquire(type);
77                          10000a8: e3a01000  mov r1, #0
78                          void log(ARGS &&... args) { Log::log().output(Log::LOG, args...); }
79                          10000ac: e1a05000  mov r5, r0
80                          _acquire(type);
81                          10000b0: eb000133  bl 1000584 <_ZN6Genode3Log8_acquireENS0_4TypeE@plt>
82                          Output::out_args(_output, args...);
83                          10000b4: e5956014  ldr r6, [r5, #20]
84                          print(output, head);
85                          10000b8: e1a01007  mov r1, r7
86                          10000bc: e1a00006  mov r0, r6
87                          10000c0: eb00013e  bl 10005c0 <_ZN6Genode5printERNS_6OutputEPKc@plt>
88                          void print(Output &output, unsigned long long);
89
90                          /*
91                          * Overloads for unsigned integer types
92                          */
93                          static inline void print(Output &o, unsigned char v) { print(o, (unsigned long)v); }
94                          10000c4: e1a01004  mov r1, r4
95                          10000c8: e1a00006  mov r0, r6
96                          while (true) {
97                                  log("Sheep:␣", n);
98                                  n++;
99                                  10000cc: e2844001  add r4, r4, #1
100                                 10000d0: eb000134  bl 10005a8 <_ZN6Genode5printERNS_6OutputEm@plt>
101                                 _release();
102                                 10000d4: e1a00005  mov r0, r5
103                                 10000d8: eb000135  bl 10005b4 <_ZN6Genode3Log8_releaseEv@plt>
104                                 10000dc: e6ef4074  uxtb r4, r4
105                                 10000e0: eafffffef  b 10000a4 <_ZN9Component9constructERN6Genode3EnvE+0xa4>
```

```
106                     10000e4: e320f000  nop {0}
107                     10000e8: f18c5ed0  .word 0xf18c5ed0
108                     10000ec: fc792fbc  .word 0xfc792fbc
109                     10000f0: 00000540  .word 0x00000540
```

B. Source code of Test Applications

B.1. small_dataspaces

```
#include <base/component.h>
void Component::construct(Genode::Env &env) {
	using namespace Genode;
	// Check whether we are supposed to restore ourselves
	if (env.cpu().restore_thread(Thread::myself()->name())) {
		while (true) {
			__asm__("NOP");
		}
	}

	constexpr auto size = 100;
	uint16_t *data[size];

	// Initialize dataspaces
	for (int i = 0; i < size; i++) {
		auto cap = env.ram().alloc(4096);
		auto ds_addr = env.rm().attach(cap);
		data[i] = (uint16_t *)ds_addr;
	}

	while (true) {
		for (int i = 0; i < size; i++) {
			*data[i] += 1;
		}
		log(*data[0]);
		// Used as delay without requiring a timer
		for (long long i = 0; i < 10000000; i++) {
			__asm__("NOP");
		}
```

```
30         }
31 }
```

B.2. stack_counter

```
1  #include <base/component.h>
2  void Component::construct(Genode::Env &env) {
3          using namespace Genode;
4          // Check whether we are supposed to restore ourselves
5          if (env.cpu().restore_thread(Thread::myself()->name())) {
6                  while (true) {
7                          __asm__("NOP");
8                  }
9          }
10
11         volatile uint16_t counter{1};
12
13         while (true) {
14                 log(counter);
15                 counter++;
16                 // Used as delay without requiring a timer
17                 for (long long i = 0; i < 10000000; i++) {
18                         __asm__("NOP");
19                 }
20         }
21 }
```

B.3. large_dataspaces

```
1  #include <base/component.h>
2  void Component::construct(Genode::Env &env) {
3          using namespace Genode;
4          // Check whether we are supposed to restore ourselves
5          if (env.cpu().restore_thread(Thread::myself()->name())) {
6                  while (true) {
7                          __asm__("NOP");
8                  }
9          }
10
```

```
11          constexpr auto size = 1;
12          uint16_t *data[size];
13
14          // Initialize dataspaces
15          for (int i = 0; i < size; i++) {
16                  auto cap = env.ram().alloc(100 * 4096);
17                  auto ds_addr = env.rm().attach(cap);
18                  data[i] = (uint16_t *)ds_addr;
19          }
20
21          while (true) {
22                  for (int i = 0; i < size; i++) {
23                          *data[i] += 1;
24                  }
25                  log(*data[0]);
26                  // Used as delay without requiring a timer
27                  for (long long i = 0; i < 10000000; i++) {
28                          __asm__("NOP");
29                  }
30          }
31  }
```

B.4. large_stack_counter

```
1   #include <base/component.h>
2   void Component::construct(Genode::Env &env) {
3           using namespace Genode;
4           // Check whether we are supposed to restore ourselves
5           if (env.cpu().restore_thread(Thread::myself()->name())) {
6                   while (true) {
7                           __asm__("NOP");
8                   }
9           }
10
11          constexpr auto size = 31650;
12          volatile uint16_t counters[size];
13
14          while (true) {
15                  for (int i = 0; i < size; i++) {
```

```
16                      counters[i]++;
17              }
18              log(counters[0]);
19              // Used as delay without requiring a timer
20              for (long long i = 0; i < 10000000; i++) {
21                      __asm__("NOP");
22              }
23      }
24  }
```

C. Profile Graphs

C.1. stack_counter

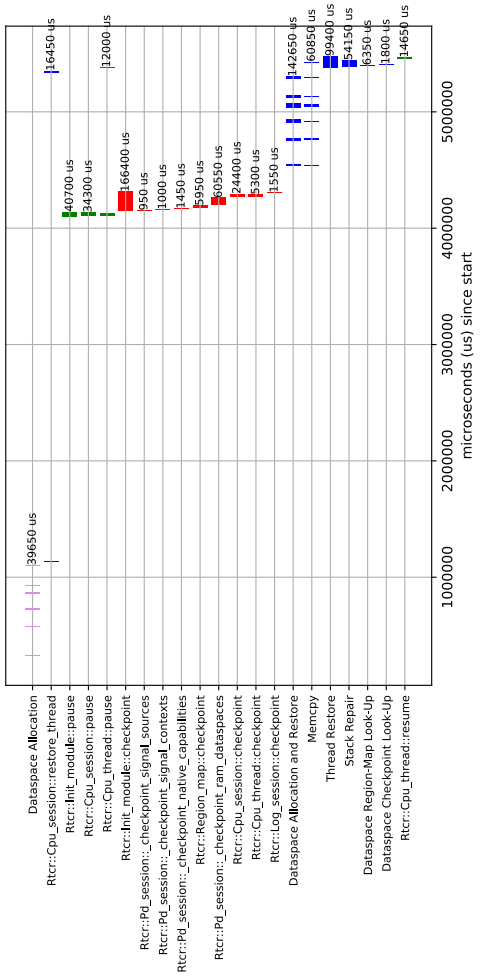

Figure C.1.: Profile of *stack_counter*

C.2. large_stack_counter

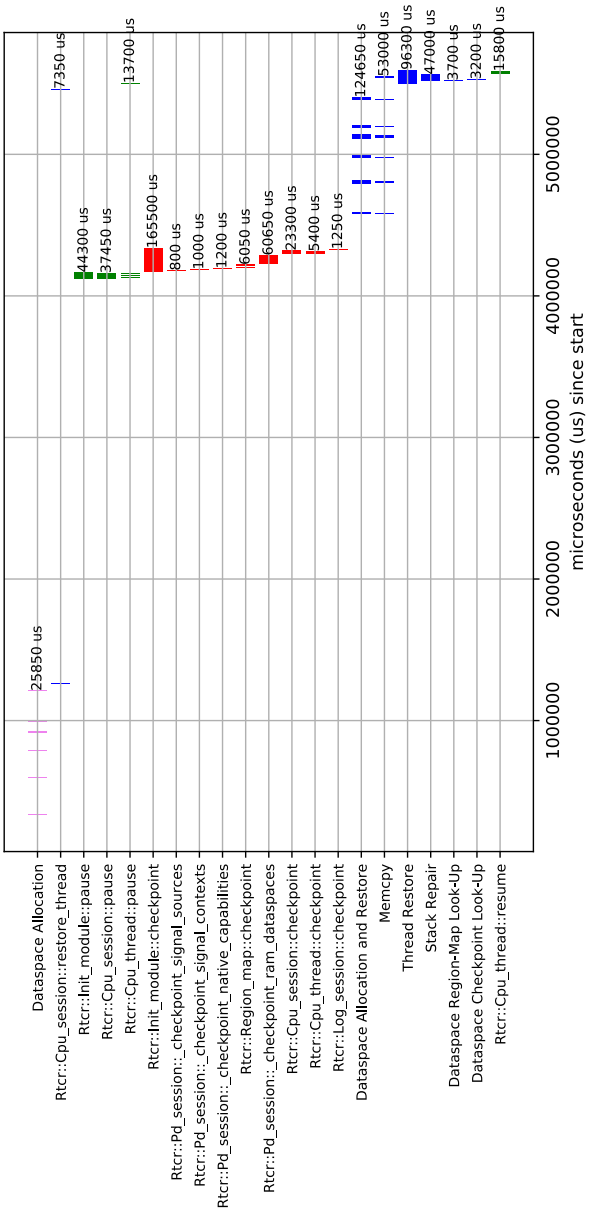

Figure C.2.: Profile of *large_stack_counter*

C.3. small_dataspaces

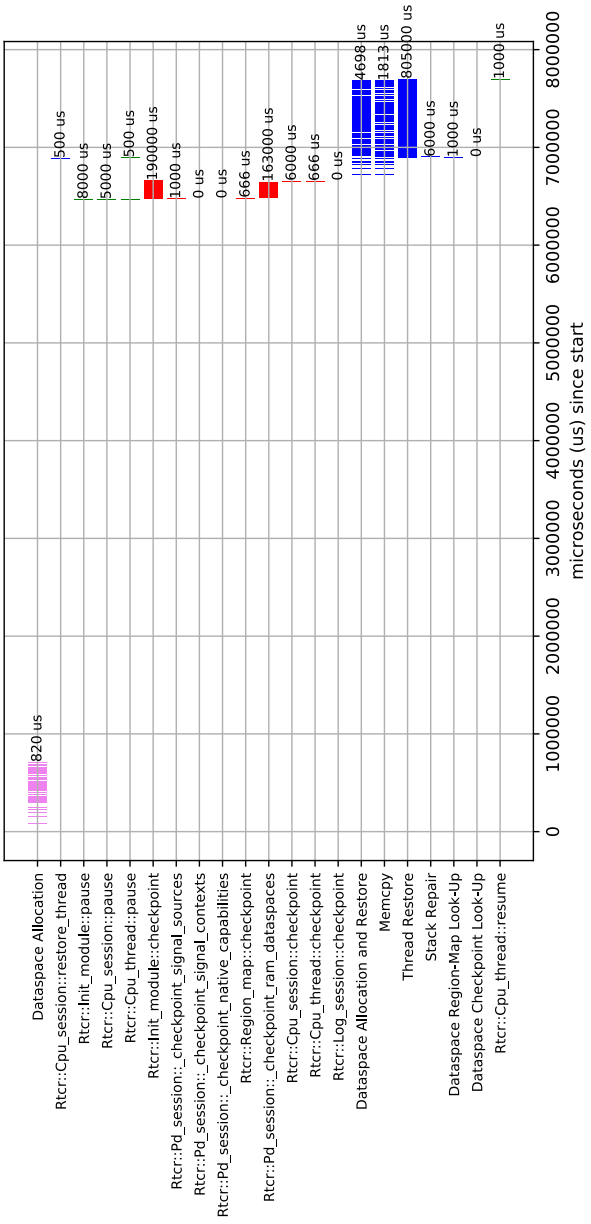

Figure C.3.: Profile of *small_dataspaces*

C.4. large_dataspaces

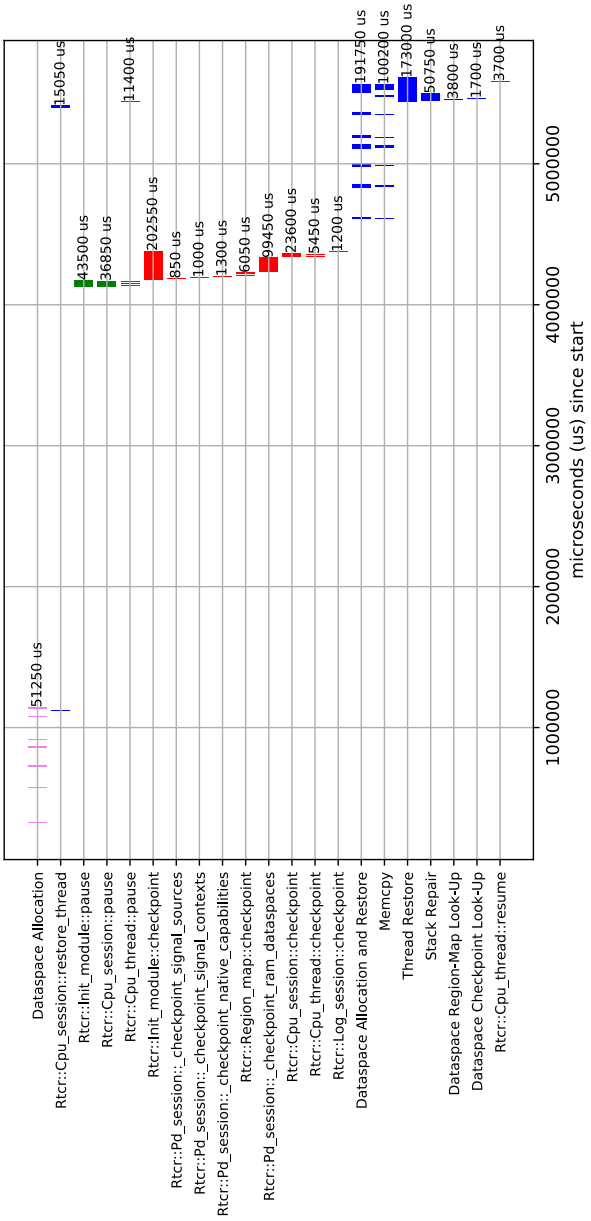

Figure C.4.: Profile of *large_dataspaces*

List of Figures

List of Tables

Bibliography

[1] *ARM Architecture Reference Manual*. ARMv7-A and ARMv7-R edition. ARM Limited, 2018.

[2] S. Bachmaier. *Optimization of a real-time capable Checkpoint/Restore mechanism for L4 Fiasco.OC/Genode by hardware-assisted memory tracing and copying*. Master's thesis. Technical University of Munich, 2019.

[3] *Basic tutorial 11: Debugging tools*. URL: https://gstreamer.freedesktop.org/documentation/tutorials/basic/debugging-tools.html?gi-language=c (visited on 10/05/2020).

[4] L. Brown. *x86_energy_perf_policy - Manage Energy vs. Performance Policy via x86 Model Specific Registers*. Man Page.

[5] *Checkpoint/Restore*. URL: https://www.criu.org/Checkpoint/Restore (visited on 10/04/2020).

[6] N. Feske. *Genode Operating System Framework Foundations*. URL: https://genode.org/documentation/genode-foundations-19-05.pdf (visited on 04/30/2020).

[7] *Fiasco.OC - Introduction*. URL: http://os.inf.tu-dresden.de/L4Re/doc/l4re_intro.html#fiasco_intro (visited on 10/12/2020).

[8] D. Huber. *Design and Development of real-time capable Checkpoint/Restore Mechanisms for L4 Fiasco.OC/Genode*. Master's thesis. Technical University of Munich, 2016.

[9] F. Johannes. *Modularization of a Checkpoint/Restore Mechanism for L4 Fiasco.OC/Genode and Performance Evaluation Considering Existing Hardware/Software-based Optimizations*. Master's thesis. Technical University of Munich, 2020.

[10] L. Joos. *Extending L4 Fiasco.OC/Genode OS by Capability Checkpoint/Restore*. Bachelor's thesis. Technical University of Munich, 2018.

[11] *KIA4SM: Verschiedene Themen für BA/IDP/MA*. URL: https://www.in.tum.de/os/aktuelles/article/kia4sm-verschiedene-themen-fuer-baidpma/ (visited on 10/10/2020).

[12] *L4Re - Overview*. URL: https://l4re.org/ (visited on 10/04/2020).

[13] *Memory Allocation in C Programs*. URL: https://www.gnu.org/software/libc/manual/html_node/Memory-Allocation-and-C.html (visited on 10/12/2020).

[14] *Microkernel*. URL: https://commons.wikimedia.org/wiki/File:OS-structure.svg (visited on 10/04/2020).

[15] *Moe, the Init process*. URL: http://l4re.org/doc/l4re_servers_ned.html (visited on 10/04/2020).

[16] *Moe, the Root-Task*. URL: http://l4re.org/doc/l4re_servers_moe.html (visited on 10/04/2020).

[17] *No output with Genode::raw and Fiasco.OC*. URL: https://lists.genode.org/pipermail/users/2020-October/007252.html (visited on 10/06/2020).

[18] *Programming languages — C++ (ISO/IEC 14882:2017(E))*. Standard. 2017.

[19] A. Rimmel. *Extending L4 Fiasco.OC / Genode OS by Multi-Core Supported Capability Checkpoint / Restore*. Bachelor's thesis. Technical University of Munich, 2019.

[20] *Sigma0, the Root-Pager*. URL: http://l4re.org/doc/l4re_servers_sigma0.html (visited on 10/04/2020).

[21] A. Tanenbaum and T. Austin. *Structured Computer Organization*. Sixth edition. Pearson, 2013. ISBN: 978-0-273-76924-8.

[22] *The Fiasco microkernel - Features*. URL: https://os.inf.tu-dresden.de/fiasco/features.html (visited on 10/04/2020).

[23] D. Werner. *Überführung eines bestehenden Linux-basierten Checkpoint/Restore-Mechanismus (CRIU) auf L4 Fiasco.OC/Genode*. Bachelor's thesis. Technical University of Munich, 2016.

[24] *What is L4*. URL: https://wiki.tudos.org/What_is_L4 (visited on 10/04/2020).

[25] *Where's the Template?* URL: https://gcc.gnu.org/onlinedocs/gcc/Template-Instantiation.html (visited on 10/05/2020).