DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Modularization of a Checkpoint/Restore Mechanism for L4 Fiasco.OC/Genode and Performance Evaluation Considering Existing Hardware/Software-based Optimizations

Johannes Fischer

DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Modularization of a Checkpoint/Restore Mechanism for L4 Fiasco.OC/Genode and Performance Evaluation Considering Existing Hardware/Software-based Optimizations

Modularisierung eines Checkpoint/Restore Mechanismus für L4 Fiasco.OC/Genode und Evaluierung der Performance unter Berücksichtigung von Hardware/Software-basierten Optimierungen

Author:	Johannes Fischer
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisor:	Sebastian Eckl, M.Sc.
Submission Date:	01/15/2020

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

Munich, 01/15/2020 Johannes Fischer

Acknowledgments

I want to thank my advisor Sebastian Eckl for his tireless dedication and help to stay focused and never feeling left alone. I also thank Prof. Dr. Uwe Baumgarten for offering an interesting topic.

I thank Alexander Weidinger from the heart for all the exhausting hours, which he spent in preparing the hardware and software required for this thesis. Lots of thanks are also directed at David Werner, who gave helpful advices and were eager to discuss ideas. Overall the whole chair deserves a thank you, for being an amazing team and the warm welcome at the chair.

I would particularly like to thank my family, my parents and my brothers and sisters, who supported me all the time.

Abstract

The demand for safety-critical applications, that are embedded in the domain of autonomous driving, has been risen in the recent past. A hardware-based redundancy has been ensured the reliable performing of critical software components so far, but has an impact on the production costs and power consumption as well. These factores might be reduced by taking the approach of software migration from the domain of high performance computing in consideration. Hereby, it is required that the migration process fulfills the real-time requirement. The migration software Real-time Checkpoint/Restore (RTCR) has been developed at the chair of operating systems of TU Munich in order to establish a foundation for research on the microkernel-based operating system Genode. In context of contributed works, many optimization approaches regarding performance were explored.

This thesis implements the second version of the RTCR software by refactoring the core mechanism and indroducing a modularisation concept. This results in significantly reduced code complexity, an easier integration of optimization approaches and a support for microkernels and CPU architectures.

The second part of this thesis evaluates the approaches regarding Genode 16.08, Genode 19.08, the microkernels Fiasco.OC and seL4, as well as the ARM architectures ARMv7 and ARMv8. Measurements revealed a superiority of the Fiasco.OC over the seL4 regarding performance. A performance loss by switching from the 32-bit architecture to the 64-bit architecture can be observed in Genode as well.

Contents

1 Introduction

The electronic control unit (ECU) in nowadays cars have made the transition from simple devices for fuel control to complex powerful on-board computers. They have been linked to lidar and radar emitters performing collision warnings, safe distance-keeping and providing automated parking. Due to advancing integration, the focus on fail-safe ECUs is becoming more important in future.

One possible solution is introduced by the research project *Cooperative Integration Architecture for Future Smart Mobility Solutions* (KIA4SM) of the operating system chair at the Technical University of Munich. The presented approach changes from using highly specialized ECUs, each maintaining a pre-defined software to a network of equivalent multi-purpose units. Therefore software is not limited to a single ECU and a hardware failure of one ECU does not necessarily mean the loss of functionality. In order to guarantee a functioning system, the migration of software from one ECU to another must be feasible in real-time.

In fact, this requirement is fulfilled by the library *Libckpt*. But it only runs in the Unix domain and it can not be applied on safty-critical microkernel-based platforms. This gap is filled by the RTCR research project. It bases on the Genode OS Framework, whichs support multiple microkernels, like seL4 and Fiasco.OC, and provides a toolkit for building highly secure and minimal operating systems. The foundation of the RTCR is a mechanism for capturing the state of a software component. Afterwards the state can be transfered to another device, where it is restored to a running component again. Since the foundation of RTCR in 2016, a lot of work has been contributed to the project and several optimization approaches have been developed [2].

1.1 Motivation

All optimization approaches regarding RTCR have in common, that they extend the code base of the initial software. The design of the early RTCR is mainly based on a prototypical implementation to prove that intercepting, checkpointing and restoring of the software components is possible. In fact, there are multiple redundant code bases. These exist not only due to optimization approaches, but also for supporting the microkernels Fiasco.OC and seL4. Another reason is the progress of the Genode Framework with a release cycle of three months.

On the one hand, this complicates the maintenance of the software. This implies that new features and patches need to be implemented in multiple sources. On the other hand, the possiblity to compare the approaches with each other is limited by several aspects. One of these is the logging of messages, which slows down the execution and leads to a bias in the results of the performance measurement. It can only prevented by removing all logging commands from the sources manually.

In order to continue providing a research platform for the RTCR project, the software architecture has to be refactored on the base of the experience, which was gained during the last six years. This also includes a modularisation concept, that encapsulates the source code of the approaches. Thus experimental approaches do not affect the stability and reliability of other implementations.

Some performance measurements of approaches have already been made in previous theses. This thesis will expand the measurements on more approaches and will compare these under following aspects:

- Genode Release Version
- Fiasco.OC and seL4 microkernel
- Hardware- and software-based optimizations
- 32-bit and 64-bit ARM architecture
- ARM core architecture, like Cortex-A7

But the performance measurements are only focused on the checkpointing process of the RTCR. Due to that, the restoring process has not been working yet.

1.2 Outline

The thesis starts with a short introduction to the Genode OS foundations. Followed by a brief overview of the Checkpoint/Restore (C/R) mechanism in chapter 2. In chapter 3 the related work of C/R contributors is presented. In chapter 4 Design the requirements for a modularisation are elaborated and three modular concepts are evaluated. Based on the result, the refactoring of the RTCR is summarized. The chapter 5 Implementation explains the software architecture based on class diagrams. The testbench for the automated execution of test-cases is described in chapter 6. The results of the performance measurement are discussed in chapter 7 Evaluation. Finally, chapter 8 summarizes the limitations and mentions future work regarding C/R.

2 Background

This chapter references to all theoretical concepts of Genode and introduces the C/R software.

2.1 Genode Architecture

The Genode OS Framework is an open source project developed by Genode Labs GmbH in Dresden, Germany. Genode is an operating system with focus on microkernels and security. This is achieved with a hierarchical organization of processes, a management of resource budgets and a capability-based policy.

Recursive System Structure

Genode organizes its processes, which are called components, in a tree structure. This implies an ownership relation between a parent component and its child components. Each component requires physical resources such as memory or CPU time. A child component will get these resource from its parent, if the parent component agrees to share its budget of resources. In fact, the parent component has full control over its children. This includes the possibility to deconstruct a child component at any time in order to regain the shared resources. The component, which is the root of the tree, is named *core*. Further information can be found in the offical Genode OS Framework Foundations book written by Feske [4, Chapter 3.2: Recursive system structure].

Core Services

A child and parent component can establish a communication channel. In general, the parent component provides service interfaces to the child components. But it is also possible that a child component announces a service to the parent component. Each communication channel with a service is handled by a session. The *core* component provides a set of eight default services. The PD (Protection Domain) service provides access to capabilities. It also manages the access to physical memory, which has been provided by the RAM service until Genode 17.05. The ROM service provides access to all files and binaries stored in the ELF image. The RM (Region Map) service handles virtual memory regions. The session of the CPU service is an allocator for processing time. It also provides an interface for the creation, control and destruction of threads. The IO_MEM and IO_PORT services enable the memory-mapped IO for user-land

device drivers. Diagnostic messages are printed with the LOG service. The TRACE service provides a light-weight event tracing facility [4, Chapter 3.4: Core - the root of the component tree].

Physical & Virtual Memory

The access to memory is organized in dataspaces. Each dataspace represents a portion of memory. The RAM dataspaces are backed with physical memory [4, Chapter 3.4.1: Dataspaces].

A region map represents the layout of a virtual address space. The dataspace can be attached to an arbitray location in a region map, which makes the memory content accessible. A region map can also be interpreted as dataspace [4, Chapter 3.4.2: Region maps].

Capability-based Security

Another fundamental mechanism of Genode is the capability-based security. Each resource, like memory, is protected by a capability. This is a token, which allows the owner to access the resource. In comparison to accessing a resource by its name or memory address, a capability cannot be guessed or forged. It can be shared between components. For example, a component requests physical memory from the PD service. The PD service will share a capability of a RAM dataspace. Eventually the component can access the memory content with that capability [4, Chapter 3.1: Capability-based security].

CPU Core Assignment

Genode introduces the concept of the *Affinity Space* in order to assign a set of CPU threads to the number of cores provided by the CPU. An *Affinity Space* is a two-dimensional array that represents the available CPU cores. Every core is mapped to a specfic sub-space of the array. Likewise all components are mapped to sub-spaces of this array. This allows a direct correlation from CPU cores to components. This has the adavantage, that the concept can be applied to the strict hierarchical organisation of components. A parent component can reassign a part of its sub-space to its child components [4, Chapter 6.2.7: Assigning systems to CPUs].

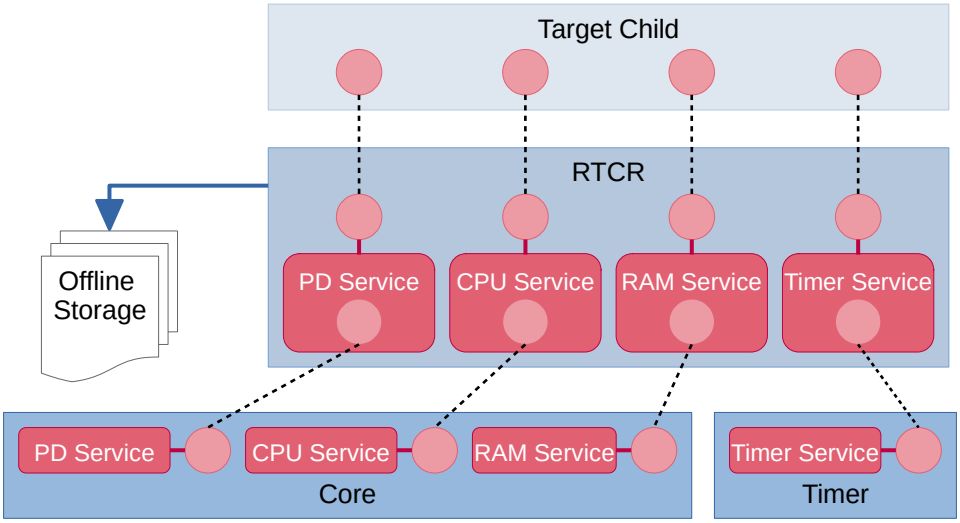

Figure 2.1: Interception of Communiation between Target Child and Core/Timer

2.2 Real-Time Checkpoint/Restore

The migration process of the C/R project is divided in three steps. At first, the execution state of a target component is checkpointed. The saved state has to be serialized, transferred to another machine and finally deserialized. In the last step, the target component is restored from the state on the target machine. The checkpoint- and restore-process are implemented in the component.

As shown in figure 2.1, the RTCR component is emedded between the core and target component. It implements its own set of services and provides these to the target application. Each session-request to the custom services is forwarded to the core services. This MITM-attack[1] allows the RTCR component to intercept all communication between the services of core and the target component. The incoming communication is directly copied into an *Online Storage*. During frequently executed checkpoints, the data is transfered from the *Online Storage* to the *Offline Storage*. The *Offline Storage* is a persistent storage, which is only modified during a checkpoint. Between two checkpoints, the data of one *Offline Storage* can be transfered to an *Offline Storage* on a second machine.

Checkpoint Process

Before the actual checkpointing starts, the threads of the target components are paused. Otherwise, it is not guaranteed that the *Online Storage* is altered during the copying process. The copying process duplicates states of following target-related entities:

[1] Man in the Middle Attack

- Allocated physical memory (Ram Dataspaces)
- Virtual memory regions (Region maps)
- State of threads
- Internal states of the sessions
- Capabilities
- Mapping from capabilities stored in kernel-land to capabilities stored in user-land
- Linker Area (Region map)
- Address Space (Region map)
- Stack Area (Region map)

After the copying, the threads of the target component are restarted and the component continues the execution regularly. Various improvements will extends the checkpoint process. Those are introduced in the next chapter.

3 Related Work

In this chapter, the publications regarding the C/R project are presented. The design decisions in chapter 4 Design are based on these.

Shared-Memory Checkpointing

In the first draft, Werner proved that a migration process can also be implemented for the Genode Framework [23]. He developed a component named *Proxy*, which is based on an extended GNU Debugger (GDB) for Genode. Starting from this approach, Huber developed the initial base for the C/R application. The so called *Shared-Memory* approach represents the most simplest concept regarding memory checkpointing. When a checkpoint is triggered, the content throughout all memory regions is copied. It quickly became apparent that this approach can not fulfill real-time capabilities. Further research is required to improve the performance of the C/R. Due to this fact, Huber also implemented a prototype of the *Incremental* approach, which reduces the overhead of successive checkpoints. However, the restoring process of the C/R is not fully functional [9].

Incremental Checkpointing

Schweigert continued to pursue the *Incremental* approach and integrates a *Copy-on-write* mechanism [18]. It has been realized by registring a page fault handler, which is triggered for accessed memory regions. Thus it is possible to keep track of memory regions, which have been modified since the last checkpoint. By only copying the modified memory regions, the performance of the C/R can be significantly improved. Werner also contributed to the *Incremental* approach during his interdisciplinary project [21].

Redundant Memory

The *Redundant Memory* approach is introduced by Stark and completely separates the memory copying from the checkpointing process. As well as the *Incremental* approach, it bases on a page fault handler. It is triggered on each write access and emulates the operation on a second distinct memory. This way, a write operation is duplicated and a copy of the original memory is created during the execution of the target application. However the emulation of each write call has a strong impact on the execution time of the target application. In fact, the emulation is still limited to a subset of registers and

only supports a single dataspace. Even though Stark has been able to demonstrate the approach, he advises further research in order to improve the reliability of *Redundant Memory* checkpointing [19].

Hardware-assisted Memory Tracing

Bachmeier implemented the *hardware-assisted Memory Tracing* approach. It bases on a FPGA implementation, which copies data from write accesses into a second memory region. In FPGA, each write operation issued by the target application is inserted into a queue. The queue is processed by removing the first element and translating it to a write operation on the second memory region. According to Bachmeier, this technique can replace the *memcpy* function of Genode. However, measurements revealed a 900% performance loss during the copy [1].

Multi-core based Checkpointing

Schön focused on a multi-core support for C/R. At first, he analysed the existing dependencies between program routines within the C/R. In the second step, an analysis of execution times revealed the performance impact of each routine. Based on these results, a control flow graph for the parallelization has been elaborated. It was finally implemented with multiple threads. The implementation was not only tested in Qemu, but also on a Wandboard. In both cases, the multi-core based approach was not faster in comparison to the original implementation. In the course of the evaluation, a general performance overhead through multiple threads has been observed [17].

Parallelized Memory Copying

In Werner's master thesis, an approach for a *Parallelized Memory Copying* has been developed. Each memory region is break down in smaller partitions. The work of copying each partition is taken over by a number of threads, where each thread is executed on a separate CPU core. Thus it can be evaluated, whether the bottleneck of the copying process is caused by a fully occupied I/O bus or by the maximum workload of a CPU core. Originally, the question should be answered with performance measurements on Qemu. Unfortunately the effect of the parallelization could not be evaluated, as Qemu could not be configured for emulating multi-core CPUs. Therefore the evaluation on real hardware is still outstanding and the question has not been answered yet [22].

Porting of Checkpoint/Restore to seL4

The C/R mechanism was originally implemented only for the microkernel Fiasco.OC. Weidinger successfully ported the C/R mechanism to the seL4 microkernel based on Genode 18.02. One major difference is the creation of a kernel capability mapping. This mapping translates user-land capabilities to the corresponding capabilities stored in the kernel. Due to the fact, the capabilities in user-land and kernel-land are the same in seL4, the mapping is not necessary for the seL4 microkernels. This does not only save 3 - 5 milliseconds during a checkpoint, but also makes an inelegant workaround obsolete. Due to the fact, that the porting of C/R to Genode 18.02 was in progress, not every component was fully adapted to the changes between Genode 16.08 and Genode 18.02. Furthermore, Weidinger observed that the *Timer* service of Genode behaves unpredictable under Qemu. Sending a thread to sleep for a few milliseconds can block the application for several seconds. Therefore, an application has to workaround this functionality with a busy-waiting loop. Luckily, this only concerns the development under Qemu and the C/R runs succesfully on real hardware, like the Wandboard Quad [20].

Hardware-based Memory Copying

Fischer extended the C/R with the hardware-based memory copying approach *CDMA*. It is named like that, because the software-based memory copying is replaced by a *Central Direct Memory Access Core* on the FPGA of a Zybo board. Due to this direct access on the memory, the copying process is 70% faster than before. But this approach also has one drawback. In order to guarantee that the copied data is valid, only uncached memory can be copied. Therefore this method can not be applied to every type of memory region. In a second implementation, Fischer also proved that further C/R-related routines, like the capability mapping, can be outsourced to a FPGA [3].

Memory-Copying Acceleration with Co-Processor

Tsareva implemented the hardware-based memory copying approach *Mailbox* on the co-processor Microblaze. This is a RISC soft processor core running on a FPGA. The corresponding Genode driver bases on the work of Bachmeier [7].

Serialization

Reisner evaluated several network data transfer protocols regarding real-time capabilities in his master thesis. As a result, *protobuf* is the protocol of his choice for serializing and deserializing the state of a checkpoint. In combination with the network library *lwip*, Reisner was able to successfully transfer a checkpoint within a network [13].

4 Design

The foundation for the software architecture of the Real-time Checkpoint/Restore Version 1 (RTCRv1) is laid by the thesis of Huber [9]. The design is mainly based on a prototypical implementation in order to sucessfully prove that intercepting, checkpointing and restoring of sessions is possible. Further theses followed, which focused on optimizations regarding memory management. Each approach bases on a fork of the original code base. Therefore RTCRv1 is split into several Git repositories with a lot of redundant code. This makes bug fixing and the implementing further features not feasible. Based on these facts, alternative architectures with focus on modularisation are discussed and a refactored Real-time Checkpoint/Restore Version 2 (RTCRv2) is presented in this chapter.

4.1 Goals & Requirements

Based on the drawbacks of the current architecture, following requirements are derived and taken as targets for the development of the RTCRv2.

Support for Performance Measurements

Performance measurements are required in order to compare implemented optimizations. It would be desirable that RTCRv2 integrates a concept for performance measurements with a minimum of computationally overhead.

Support for Multiple Kernels

Weidinger sucessfully ported RTCRv1 to the microkernel seL4 [20]. Therefore RTCRv2 shall support multiple microkernels by design.

Uniform Configuration Interface

RTCRv2 shall provide an uniform interface for configurations. This will prevent misconfiguration due to undefined values or undocumented default configurations.

	XML-File	kconfig
Syntax Validation	Compiletime	Compiletime
Parsing	Runtime	Compiletime
Format	XML	C Macro Syntax
Configuration	Plain Text	CLI[1], TUI[2], GUI[3]
External Dependencies	No	Yes
Reconfiguration	Yes	No
Implementation	Low	High
Usablity	Low	Medium

Table 4.1: Comparison of *kconfig* and Genode XML Configuration

Modularisation

It shall be possible to include and exclude program code based on a modularisation concept. Experimental program code can be bundled in a module and easily excluded whenever a stable RTCRv2 is required. This way, implementations of future theses will not be based on unreliable program code. Because all modules can be exluded and ported individually, porting to another version of Genode will be simplified.

Multi-core

The support for multi-core in RTCRv1 was investigated by Schön [17] and Werner [22]. A general concept and tool-kit for parallelization of checkpoints and restores shall also be part of the RTCRv2.

4.2 Configuration

Since the beginning of the Genode Framework, the runtime components of the operating system are configured by a single XML file. This file is generated and included during the compilation process of the Genode OS and later accessible while the OS is running. In contrast to that, microkernels are configured in advance. Most microkernel uses an adapted version of the configuration tool *kconfig* from the Linux kernel project.

[1]Command Line Interface
[2]Terminal User Interface
[3]Graphical User Interface

Based on pre-defined rules, the tool displays a list of choices in a Graphical User Interface (GUI). After the configuration, a text-based configuration file is generated. Each line of this file consists of `CONFIG_*=y|n`, which defines whether a choice was selected (`y`) or not (`n`). The precompiler interpretes each line as C macro, which will include or exclude source code of the kernel during compile-time. The advantages and disadvantages of both concepts are compared in table 4.1.

kconfig

The main advantage of using a tool like *kconfig* is the validation of selected options during the compile-time. Invalid combinations are either disabled in the selection menu or identified by the preprocessor. For example, a single-choice list allows to pick one module and exclude the unused program code of all other modules. But *kconfig* is not a standalone tool, needs to be ported to RTCRv2 and depends on libraries like *ncurses*. A new tooling has to be wrapped around the Genode configuration process in order to change library dependencies in Makefiles, define macros and read default configurations from multiple files. A fast User Interface (UI) driven configuration comes at the expense of high implementation cost. Most developer will not be familiar with the *kconfig* tool family and needs to understand the complex underlying process in order to integrate their new written module. Thus the usability of *kconfig* is limited.

Genode Dynamic XML Configuration

In contrast to the choice-based configuration of *kconfig*, the XML configuration accepts any file with valid XML syntax. While the XML syntax is validated before the XML file is packed into the image, the semantic validation has to be done separatly by each component. In case of an invalid option, like a negative integer, the component has to fallback to a default value or terminate the execution. In both cases the misconfiguration has to be reported to the logging system. Nevertheless the run-time validation also comes with the advantage of dynamic reconfigurations. Genode allows components to modify the XML configuration in order to reconfigure a child component. This is helpful if the RTCRv2 must be reconfigured by third-party components at run-time.

It can be concluded that an easier configuration interface comes with a more complex integration in Genode and higher efforts on the side of future developers. Because of that, the RTCRv2 will drop UI configuration and uses the board resource of the Genode Framework.

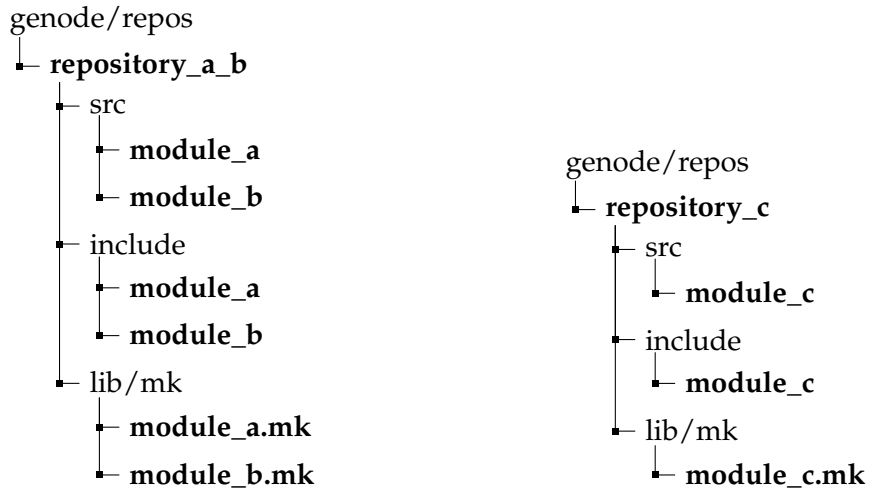

Figure 4.1: Directory Structure of Genode Repositories

4.3 Modularisation

This chapter is focused on required aspects to achieve the modularity. The modularisation of RTCRv2 is based on three pillars. It starts with an extandable **file- and project-structure**. It shall be possible to **exclude program code** which is not used during execution in order to reduce the memory footprint. Finally the choice of the correct **software design** is the most important aspect to reduce the complexity of the program code.

4.3.1 File Structure

Program code and libraries are organized in repositories which are installed to the Genode directory. As shown in figure 4.3.1, a repository has a source directory (*src*), a directory for header files (*include*) and an optional directory for Makefiles (*lib/mk*). RTCRv1 has been developed in a single repository. In contrast to that, each module of RTCRv2 is developed in a single repository. Consequently program code of modules are strictly separated from each other. This has the advantage, that a developer only has to look at a subset of the program code, which can be understand much quicker.

Finally the use-case of providing multiple modules in a single repository is still valid, as illustrated in the first directory structure of figure 4.3.1. Each module is implemented in a subdirectory and the corresponding Makefile, which contains instructions for compiling and linking the module, is stored in *lib/mk*.

4.3.2 Linking

Linking is the process which resolves undefined symbols in a binary. These symbols will only be resolved if the referenced program routines exists. In static linking, missing libraries routines are copied into the binary. In contrast to that, dynamic linking deferring the resolution until a program is executed.

The idea is to use linking in order to exclude modules which are not used. Therefore each module is implemented as library and will be linked to the core implementation of RTCRv2 if necessary. This approach poses the following question. Should modules be linked dynamically or statically?

Dynamic Linking

One benefit of dynamic linking is that often-used libraries need to be kept in memory only once. This reduces the memory footprint of applications. But if a library does not provide required symbols, linking errors will only appear during run-time.

Static Linking

The approach of static linking does not raise linking errors during runtime, because static linking is done during compile-time. But static linking does require more memory space, if multiple binaries use the same program routines of a library. However, this disadvantage does not apply to RTCRv2, as it is intented that only a single component (*parent component*) takes over the task of checkpointing and restoring. Thus static linking is clearly preferred over dynamic linking.

4.3.3 Module Registration

Both approaches raise the same question: How are linked modules registered? In particular, a module must provide an entry point which will allocate required memory and start working threads. This routine must be executed first, if the module is loaded. Furthermore the module a be linked to a binary, that has been already compiled. Consequently this binary needs a reference to the entry point of the module in order to intialize it. This would be fulfilled if all references to existing modules are statically deposited in the binary. But this implies that the RTCRv2 can not be extended by further unknown modules. In order to overcome this disadvantage, each linked module is required to register itself to a globally accessible list when the binary is executed. This leads to the question, how such a registration routine can be implemented for each module.

Like the `main()` function in binaries, also dynamically linked libraries can be specified with a load-time entry point, but this is not the case for statically linked libraries. The absence of such a function has to be solved with a statically initialized class. Listing 4.1 shows such a class (`Module`). It is initialized as static object. Therefore the constructor

```
1  /* list of all registered modules */
2  List<Module> _global_module_list;
3
4  /* static object of class Module*/
5  Module _static_module;
6
7  class Module
8  {
9      Module() {
10         /* register this module */
11         _global_module_list.add(this);
12     }
13 }
```

Listing 4.1: Module Registration Routine

`Module()` is executed during the dynamic initialization phase of the class and before the first program statement of the global `main()` function. In this constructor, the reference of the object is added to a global list of modules.

4.3.4 Software Architecture

Previous sections introduced how modules are organized on file level, loaded and registered. These are just two pillars of the modularisation concept, but the most important one is about the software architecture. In order to understand the necessity of a new software design, the drawbacks of RTCRv1 has to be discussed.

Figure 4.2 shows an abstract overview of the architecture of RTCRv1. Each `Target_child` initiates its own set of intercepting sessions S_*. As described in chapter 2 Background, each interceping session delegates its Inter Process Communication (IPC) to the corresponding session provided by the core component. Therefore the session classes only contain a minimal functionality. The checkpointing logic L_* of each service is implemented in the class `Checkpointer`. This architecture was mainly chosen due to the prototypical implementation which implied a strong cohesion between all checkpointing routines. Although it has one major design flaw. A child component communicates with services based on the *Server-Client* pattern. The child is the client and the service is the server. The implementation of the child component (`Target_child`) should not depend on the actual implementation of a service. As long as the used Application Programming Interface (API) does not change, the implementation will be interchangeable. Thus the child implementation and services can be separated. That also has the advantage, that multiple instances of a child application use the same service. Therefore, independently of the chosen modularisation design, RTCRv2

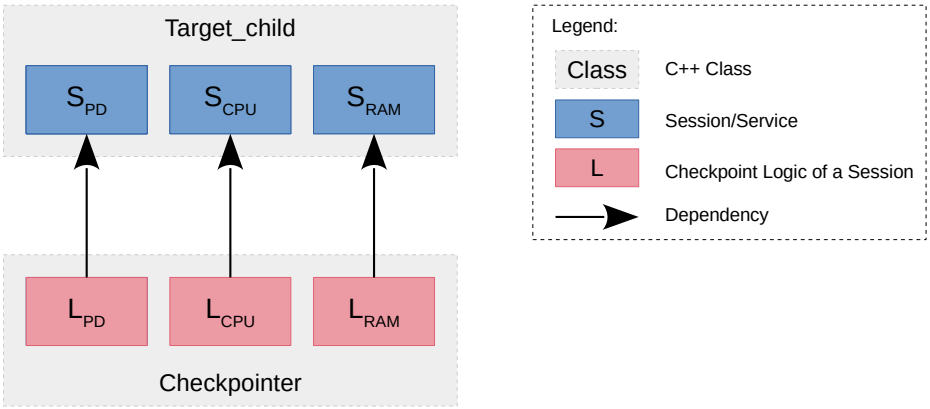

Figure 4.2: Session and Checkpointing Logic of RTCRv1

separates the `Target_child` implementation from all service implementations.

Basically, the modularisation can be achieved by defining borders separating one module from the other. High cohesion[4] within modules and low coupling[5] between modules are often regarded as related to high quality in Object Oriented Programming (OOP) languages. Another trade-off to consider is code duplication versus dependencies. In order to extend a module, the program code of the module could be copied and modified or the code could be inherited and only a small portion of code is replaced. While the maintenance of the program code suffers from the code duplication in the first case, the second approach leads to a strong coupling between modules.

Also, regarding the requirement of supporting multi-core CPUs, several questions arise: Does each module handle the pool of threads itself? Are modules executed in parallel? How to prevent deadlocks between modules?

One Module per Service Unit

In Genode a service provides one specific service function to the client components. As shown in figure 4.3, this is basically adapted to the modularisation model. Each service (S) and the corresponding C/R logic (L) are wrapped in a module. This allows to extend the service landscape by adding further modules. Also a whole module can be replaced by one with a different implementation. But this idea has following problems:

Problem 1: Dependencies Some services have dependencies to other services and use their API. For example, the CPU service requires capabilities for creating new threads. But these are only provided by the PD service. Thus the service S_{CPU} depends on the service S_{PD}. As tracking these capabilities is part of the C/R

[4]Cohesion is the indication of the relationship **within** a module.
[5]Coupling is the indication of the relationships **between** modules.

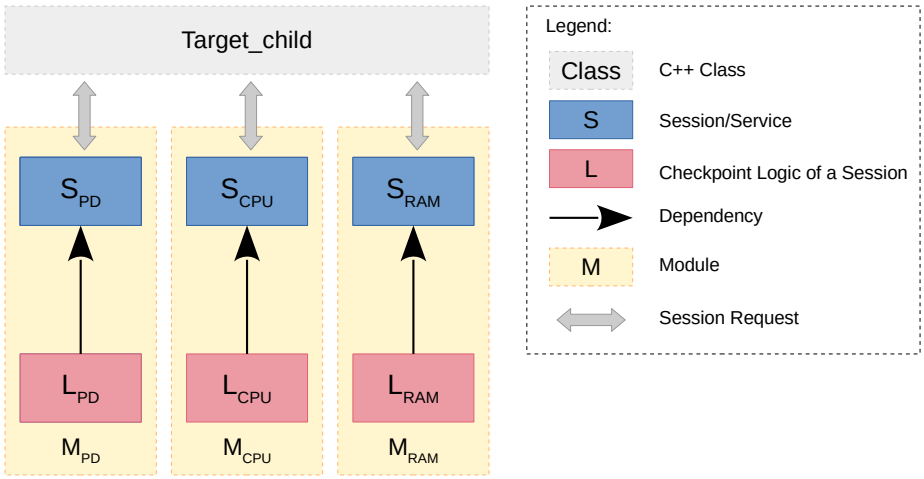

Figure 4.3: One Module per Service Unit

process, such dependencies are also found between L_{CPU} and L_{PD}. Consequently a module requires access to attributes and methods of other modules. This leads to the question, how a module will be informed about the existing of other modules. The most common solution is the *Publish-Subscribe* pattern, where all modules subscribe to a list and the reference to a new module is published to the subscribers. A reference to a module object will only exists, if the constructor to a module object did not fail. Initialization routines, which depend on other modules, can not be executed within the constructor. This has a big impact on the initialization phases of a module:

1. Initialize internal non-depending ressources of modules
2. Sharing references between all modules
3. Initialization of resources which depend on other modules

Not only that this routine implies an overhead during the startup, but also missing modules are only detected after the construction of the module. This allows to initialize modules which finally fails due to the missing dependencies.

Problem 2: Threading Most routines of logic units are executed in threads. Like the dependencies between logic units (L_*), also threads of logic units depend on each other. Deadlocks can only be prevented if the execution order has been defined in advance. Consequently the user, who chooses the modules, also has to define the correct execution order. This is a sign for bad usability. On the one hand, it requires deep understanding of the internal routines in order to use the RTCRv2. On the other hand, writing a new module also requires knowledge of all other modules to prevent deadlocks with these.

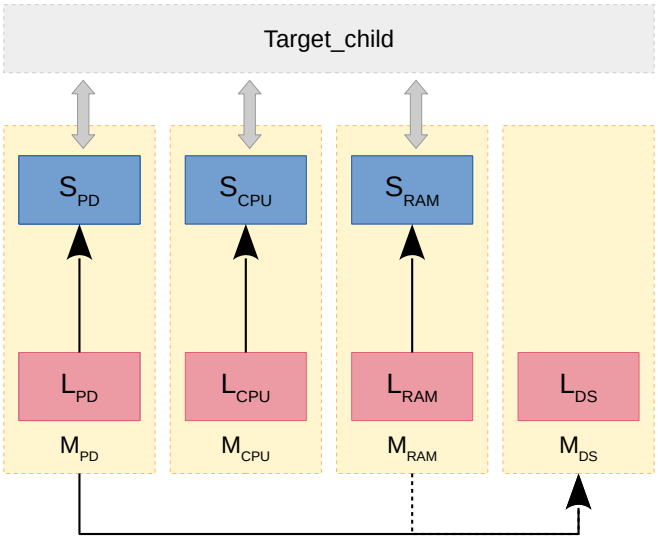

Figure 4.4: One Module per Logic Unit

Problem 3: Combination Diversity The possible combinations of modules are highly limited in this model, because most optimization approaches are based on the PD service. For example, it would be interesting to combine the parallelization and incremental memory copying approaches in order to shorten the checkpoint execution time. But both modules can not be chosen at the same time, as each extends S_{PD}.

In summary, it makes no sense to modularize RTCRv2 based on services.

One Module per Logic Unit

In order to increase the number of combinations, the rule of one service per module is disbanded, as shown in figure 4.4. Consequently the logic for checkpointing of dataspaces L_{DS} can be separated from L_{PD} and packed in the module M_{DS}. Now, the incremental approach is packed as M_{PD} and the parallelized memory copying approach as M_{DS}. But such a fine-grained module landscape also comes with drawbacks.

Due to the splitting of functional relationships, the cohesion, which previously only existed within a module, becomes coupling between modules. This results in more methods provided by modules and a higher complexity.

Also the interfaces of each module, which might be necessary in future would have to be designed in advance, which is not possible due to the research character of RTCRv2.

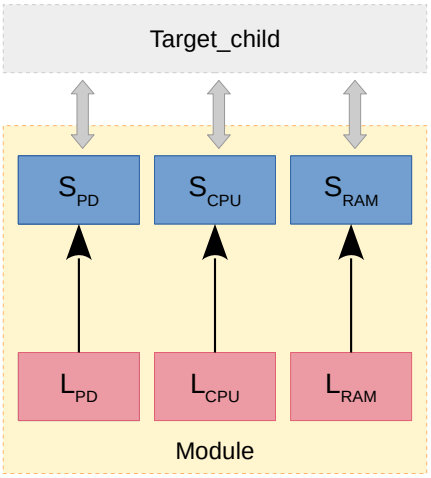

Figure 4.5: One module for all Units

One Module for All Units

The previous approach showed that there is a trade-off between complexity and combination diversity. As a fined-grained approach quickly reaches its limits in practice and increase the complexity, a single-module model is considered in more detail. As shown in figure 4.5, all units are packaged in a single module which is exclusivly loaded during runtime.

The main drawback (Problem 3) of this approach is the fact, that a combination of modules has to be hard-coded implemented by combining program code from multiple modules. This can mostly be reduced by class inheritance in C++.

But on the other hand, this approach solves Problem 1 by reducing the complexity of the module initialization. It is not nececessary to solve dependencies between modules and the *Publish-Subscribe* Pattern does not have to be implemented, too.

Because only a single module is loaded, the set of interacting threads is defined in advance. Therefore Problem 2 can be solved, as deadlocks between depending threads can be programmatically prevented.

4.3.5 Workflow

Figure 4.6 illustrates the workflow of linking, compiling and configurating RTCRv2. All currently implemented modules are listed in table 4.2. The library `rtcr` implements the core routines of C/R. It provides the module `base`, which is similar to the shared memory approach. In order to use the module `cdma` and `inc`, the corresponding libraries must be linked to the core library `rtcr`. This can be done by appending the library names to the `LIBS` variable in the Makefile of the `rtcr` library. For example in listing 4.2, the modules `rtcr_cdma` and `rtcr_inc` are linked.

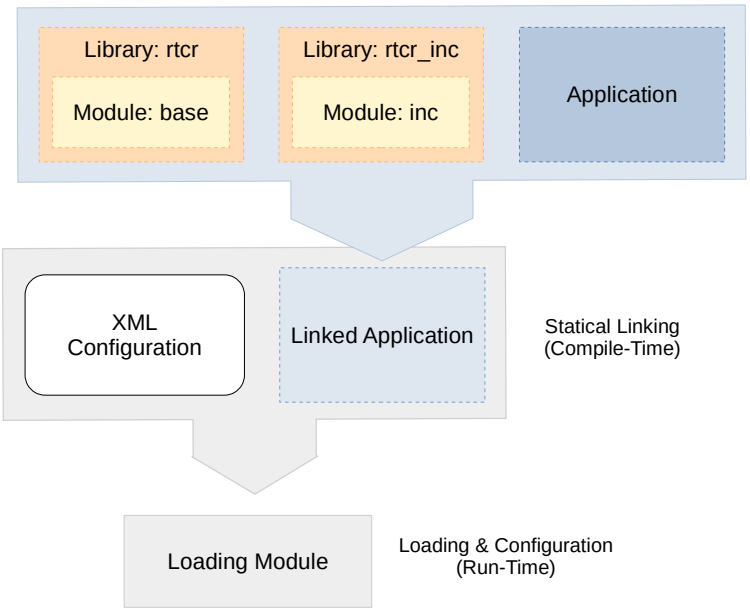

Figure 4.6: Module Loading and Configuration

Module Name	Library	Description
base	rtcr	Core Implementation based on Shared-Memory Approach[9]
inc	rtcr_inc	Incremental Approach [18]
cdma	rtcr_cdma	FPGA assisted Memory Copying [3]
para	rtcr_para	Multi-threaded Memory Copying [22]
mbox	rtcr_mbox	Co-Processor assisted Memory Copying [7]
memtrace	rtcr_memtrace	FPGA assisted Memory Tracing & Copying [1]

Table 4.2: Available Modules for RTCRv2

```
1 ...
2 LIBS += rtcr_cdma rtcr_inc
3 ...
```

Listing 4.2: Append Modules as Library Dependencies (*rtcr/lib/mk/rtcr.mk*)

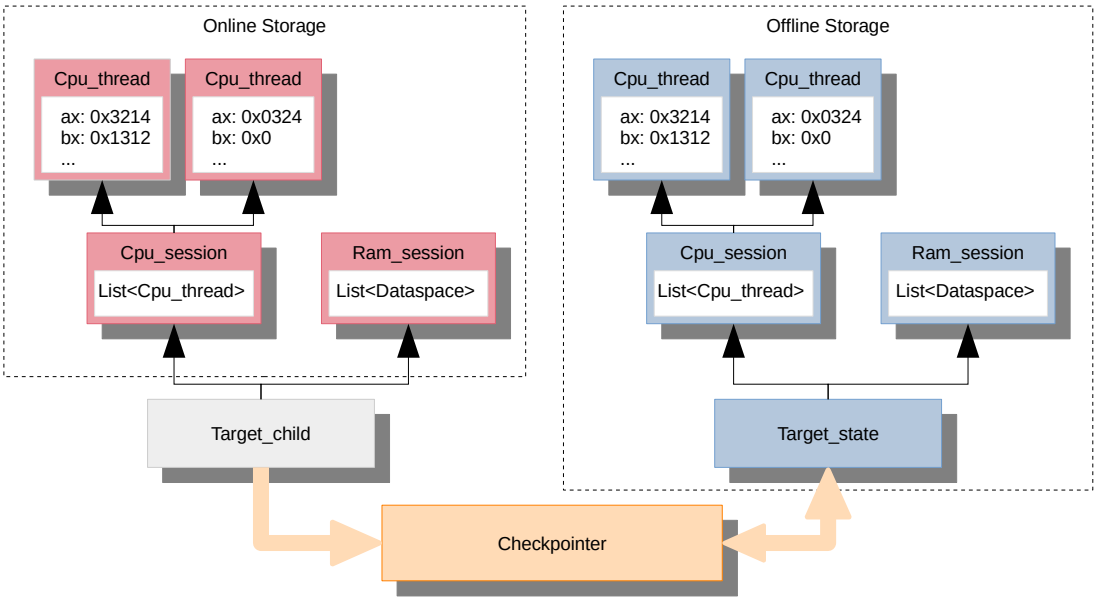

Figure 4.7: Checkpointer copies States from *Online Storage* to *Offline Storage*

In the next step, the `rtcr` library is finally linked to the application, which initializes the module. The configuration for the module is passed as XML file and parsed during runtime. An overview of all existing configuration parameters is attached to the appendix D.1.

4.4 Checkpoint Logic

The modularisation also provides the opportunity to refactor the main logic of the checkpointing routines. The necessity of such changes is discussed now.

In general, the current state of a child component is tracked in a hierchary of tree-like storage classes, which is named *Online Storage*. Such storage classes do not only exist for each session, but also for threads, dataspaces, capabilities and region maps. The previously introduced `Checkpointer` class accesses these states through the `Target_child` class and stores a snapshot in a so-called *Offline Storage*. This storage is represented by a similiar tree-like hierachy of storage classes with the `Target_state` class as root node.

Each *Online Storage* class is considered as a black-box. The current state of the storage is only accessible during a checkpoint. As visualized in figure 4.7, the state of a session consists not only of attributes, but also of lists with references to other objects from the *Online Storage*. For example, the PD session contains a list of all currently running threads, where each thread holds a state of registers. During a checkpoint,

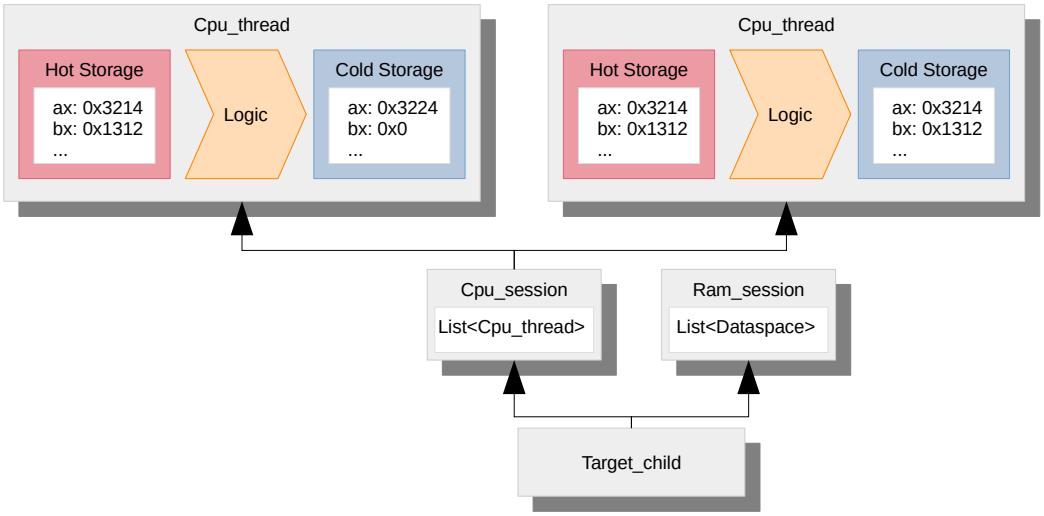

Figure 4.8: Integration of *Hot Storage* and *Cold Storage* in Storage Classes

the list of threads from the `Target_state` is updated to match the list of threads from the `Target_child`. As this checkpointing logic is implemented in the `Checkpointer` class, a strong coupling exists between the classes `Checkpointer`, `Target_child` and `Target_state`. This architecturs implies a lot of *Getter/Setter* methods and code routines for updating these lists.

The initial RTCRv1 was able to store each snapshot in a new `Target_state` object, whereas the later implemented *Incremental Approach* and *Redundant Approach* require the same `Target_state` object for a sequence of checkpoints. In fact, supporting multiple *Offline Storages* substantially increases the complexity and also has no more use.

One aspect of C/R is the transfer of an *Offline Storage* to another computer. On condition that a second checkpoint is in process, while the first is still transfered, multiple *Offline Storages* are required. It can be assumed that the execution frequency of checkpoints is constant. The transfer rate of each checkpoint would be lower than the rate of creating checkpoints. Consequently the memory footprint is increasing and the process runs out of memory. Therefore the frequency of checkpoints has to be lower than the required transfer time for a checkpoint and only one *Offline Storage* is needed.

Due to these facts, the storage concept of RTCRv2 fall back on a much simpler concept with only one backing storage.

4.4.1 Hot & Cold Storage

The storage concept is limited to an active storage (*Hot Storage*) and a backing storage (*Cold Storage*). As shown in figure 4.8, each `Cpu_thread` class integrates both kind of storage.

In particular, the current state of the class is represented by the internal-accessible *Hot Storage* and the public-accessible *Cold Storage* holds the state of the previous checkpoint. The logic can be integrated directly in each `Cpu_thread` class. This design has the following advantages:

Less Complexity The complex logic of `Checkpointer` class is splitted into small pieces.

Less Coupling The high coupling is reduced, because the logic and the storages are united in corresponding classes.

Parallelisation Every storage class is independent of each other. Thus each class can be executed in the context of a thread.

State Tracking The checkpointing logic has access to all *Getter/Setter* methods, because it is integrated in the session. Therefore state changes can be tracked in real-time.

The *Hot & Cold Storage* has one disadavantage compared to the checkpointing concept of RTCRv1. The information, whether a thread has been created or deleted since the last checkpoint, can be extracted by comparing the lists of threads of the *Online Storage* and the *Offline Storage*. If both storages have been moved into a thread class, only one list of thread exists. In order to keep track of new created and deleted threads, a list with checkpointing capabilities has to be implemented.

4.4.2 Checkpointing a List

The implementation of such a list L_c depends on an *Insert List*, a *Remove List* and a *Reference*. The *Reference* points to an element in the *Insert List* and represents the state of the last checkpoint.

Insert Element

In order to add an element to the list L_c, it is inserted at the end of the *Insert List*. This operation is in $O(1)$, because just a reference to the last element has to be updated.

For example in figure 4.9 the *Insert List* has three elements. As the *Reference* (Ptr) points to the first element T_1, the last two elements T_2 and T_3 has beeen added since the last checkpoint.

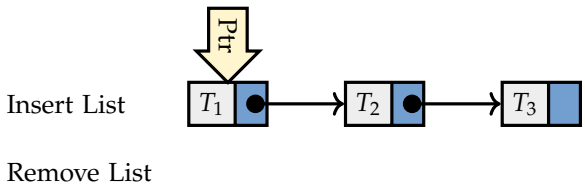

Figure 4.9: *Insert List* with two new Elements since the last Checkpoint

Remove Element

Elements, which should be removed from the list L_c, are added to the *Remove List*. Thus this operation is in $O(1)$, too. Furthermore the element is marked as removed, which is important for the identification of inactive elements when the *Insert List* is iterated.

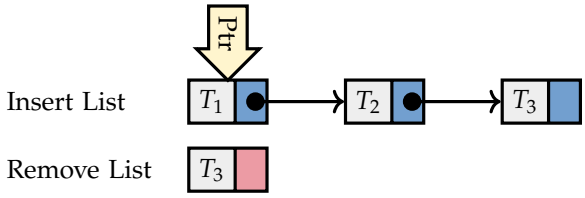

Figure 4.10: Deleted Elements are added to the *Remove List*.

Checkpoint

Checkpointing the list L_c and the corresponding elements is splitted in four steps:

Step 1 Each element in the *Remove List* is processed and taken out of both lists. If no elements has been removed since the last checkpoint, the best case runtime of this step is in $O(1)$. In case that the *Insert List* is implemented as a double-linked list, the worst case is in $O(n)$.

Step 2 (Optional) Some elements, like dataspaces, must be prepared for the first checkpoint. Therefore all elements, starting from the *Reference* to the tail of the *Insert List*, are prepared.

Step 3 (Optional) The method `checkpoint()` of each element in the *Insert List* is called. This way, the checkpoint command is passed to the next level in the hierarchy of storage classes.

Step 4 All elements in the list L_c are checkpointed. Therefore the *Reference* is set to the tail of the *Insert List*. This operation is in $O(1)$.

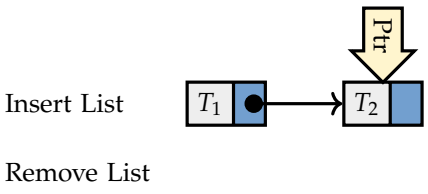

Figure 4.11: *Insert List* and *Remove List* after a Checkpoint

In conclusion, the best case for checkpointing the list L_c is in $O(1)$ and the worst case is in $O(n)$. In contrast to that, the list operations of RTCRv1 are in $O(n^2)$. This is due to the fact that the lists of *Offline Storage* and *Online Storage* are compared in nested loop.

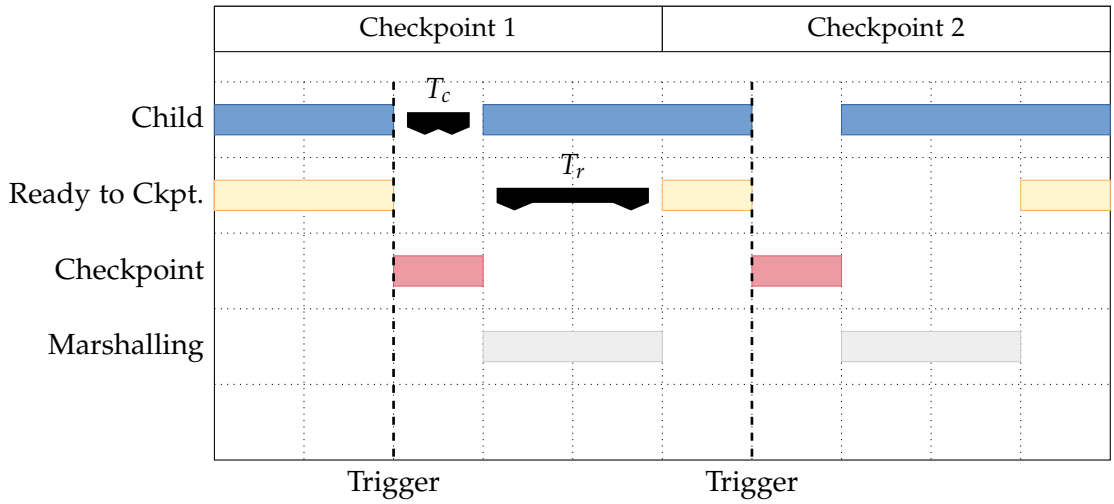

Figure 4.12: Timing constrants regarding Checkpointing and Marshalling

4.4.3 Checkpoint Frequency

The frequency of checkpoints does not only depend on the checkpoint time T_c, but also on the duriation T_r, which is necessary to process the *Cold Storage*. This relation is visualized in figure 4.12. In general the *Cold Storage* is serialized and compressed to a binary format (*Marshalling*). In this time, the *Cold Storage* is accessed, which prevents further checkpoints. Consequently the *Ready to Ckpt.* bar is only visible if no checkpoint is in process and no marshalling is active. This example shows, that not only T_c, but also T_r must be taken in consideration in order to evaluate the efficency of RTCRv2.

4.5 Parallelization

The efficiency of the RTCRv2 is not only gained with a refactored checkpointing logic, but also the parallelized execution of program routines on different cores is a major aspect. The core design of RTCRv1 has been changed due to the modularisation. Therefore the dependency analysis elaborated by Schön can not be applied to RTCRv2 [17]. The following chapter recaps results of his thesis and shows that the design of RTCRv2 rules out previously existing dependencies.

4.5.1 Dependency Analysis

The result of the dependency analysis can be seen in figure 4.13. It shows the key functions within the checkpointing process. Each set of independent functions are grouped in a box with time proceeding downwards. For example, all functions in the first box are executed at first, because the functions in the second box depends on these.

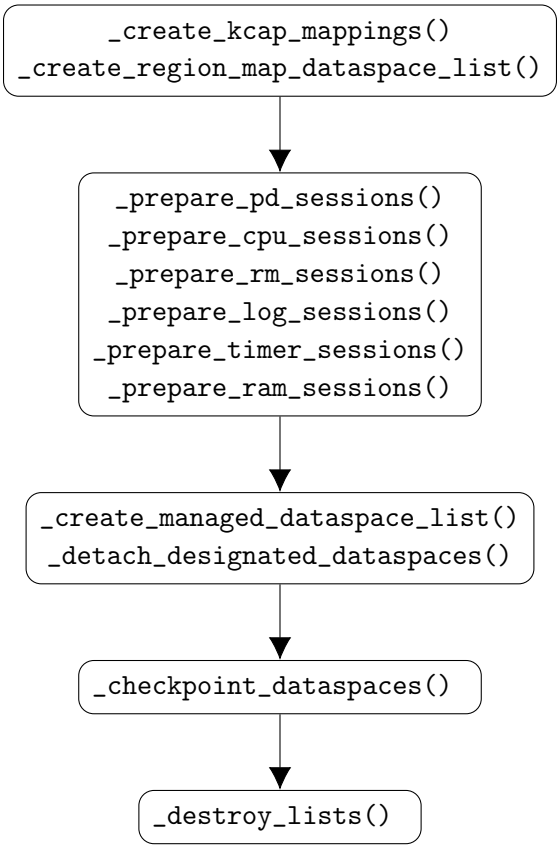

Figure 4.13: Functions of RTCRv1 grouped by Dependencies

Although the order of functions within a box can be be shuffled arbitrarily. In order to understand that these dependencies are avoidable, each function is discussed in the following.

_create_kcap_mappings()

The function `_create_kcap_mappings()` creates a lookup table which maps each kernel capability of Fiasco.OC to a badge. This mapping is necessary to restore the correct kernel capabilities during a restore of a checkpoint. In particular, each `_prepare_*_sessions()` depends on this lookup table. These functions synchronize the state of the *Online Storage* to the *Offline Storage*. Within this step each badge, which is stored in the *Online Storage*, is translated to a kernel capability and is stored to the *Offline Storage*. RTCRv2 outsources this part of the checkpoint to the post processing of the *Cold Storage*. In detail, also the lookup table and each badge are stored to the *Cold Storage* and the actual lookups are part of the marshalling process.

Another dependency arises, because a list is used as data-structure in order to store the capabaility-badge tuples. Not only, that this list is created anew, but also released by the `_destroy_lists()` function for each checkpoint. In fact, the maximum count of kernel capabilities is fixed to 4096, which makes an array much more appropriate in this case. As this array is only initialized once for each child application, no clean up routines are required within a checkpointing phase. Consequently, the dependencies on `_create_kcap_mappings()` are resolved.

_create_region_map_dataspace_list()

The function `_create_region_map_dataspace_list()` produces a list of region maps, which are created for the linker, stack and adress space of the child application. The function `_prepare_rm_sessions()` also attachs region maps, which are explicitly allocated by the child application. A region map represents a virtual address space in which dataspaces can be attached. It can also be interpreted as a dataspace, so combining multiple dataspaces to a single dataspace is possible. Such a region map based dataspace should not be checkpointed, as its attached dataspaces have already been part of a checkpoint. For this reason, the `_checkpoint_dataspaces()` function depends on the list of region maps in order to filter out these dataspaces.

However, this consideration is based on a fallacy. The list of all dataspaces is managed by the RAM session. But this service only serves dataspaces backed by physical memory. Furthermore the creation of a region map does not require an allocation of a physical dataspace, as a region map internally maintains its state with a heap allocator.

Due to these facts, region map based dataspaces are not part of the checkpoint by default. So the function `_create_region_map_dataspace_list()` is unnecessary.

_create_managed_dataspace_list()

The function `_create_managed_dataspace_list()` is part of the *Incremental* approach. In contrast to the *Shared Memory* approach, managed dataspaces are only partially copied. The copying routines in `_checkpoint_dataspaces()` require a list of managed dataspaces in order to distinguish normal dataspaces from managed dataspaces. But if the *Incremental* approach is enabled, all dataspaces are affected and only managed dataspaces exist. Hence the decision, wether all dataspace are copied fully or partically, only depends on a single boolean and the differentiation with an extra list unnecessary.

Finally, the dependency on `_destroy_lists()` is also resolved due to the absence of a managed dataspace list.

_detach_designated_dataspaces()

The function `_detach_designated_dataspaces()` is part of the *Incremental* approach. Managed dataspaces, which were attached to the virtual memory during the runtime of the child application, are detached again by this function. This routine is executed before `_checkpoint_dataspaces()`, because `_checkpoint_dataspaces()` requires that all dataspaces are detached. In fact, `_checkpoint_dataspaces()` attachs every dataspace, copies its memory and finally detachs it again. This is unnecessary in the context of incremental copying, because the attachment of a dataspace is the only indicator that the dataspace was modified. Therefore only attached dataspaces must be copied. So `_detach_designated_dataspaces()` will be not necessary, if managed dataspaces are not attached again by the `_checkpoint_dataspaces()` function.

_checkpoint_dataspaces()

The function `_checkpoint_dataspaces()` copies the memory of dataspaces from the *Online Storage* to the *Offline Storage*. It depends on the following methods:

- `_create_managed_dataspace_list()`
- `_detach_designated_dataspaces()`
- `_create_region_map_dataspace_list()`
- `_prepare_ram_sessions()`

Based on the discussed dependency resolutions, only the dependency on `_prepare_ram_sessions()` still exists. This function copies dataspace attributes, like size, cached and executable flag, from the *Online Storage* to the *Offline Storage*. Because there is only one more dependency, the functionality of `_prepare_ram_sessions()` and `_checkpoint_dataspaces()` can be merged to a single function.

RTCRv1	RTCRv2
`_create_kcap_mappings()`	`Capability_mapping::checkpoint()`
`_prepare_pd_sessions()`	`Pd_session::checkpoint()`
`_prepare_cpu_sessions()`	`Cpu_session::checkpoint()`
`_prepare_rm_sessions()`	`Rm_session::checkpoint()`
`_prepare_log_sessions()`	`Log_session::checkpoint()`
`_prepare_timer_sessions()`	`Timer_session::checkpoint()`
`_prepare_ram_sessions()` `_checkpoint_dataspaces()`	`Ram_session::checkpoint()`
`_create_managed_dataspace_list()` `_detach_designated_dataspaces()`	`Ram_inc_session::checkpoint()`

Figure 4.14: Corresponding Functions in RTCRv1 and RTCRv2 under Genode 16.08

_destroy_lists()

The function `_destroy_lists()` releases lists, which have been created during a checkpoint. It concerns the following functions:

- `_create_kcap_mappings()`
- `_create_region_map_dataspace_list()`
- `_create_managed_dataspace_list()`

These lists are not necessary anymore with the previously discussed dependency resolutions. Therefore this function can be erased from the dependency tree.

The dependency analysis shows that all dependencies can be solved in RTCRv2. All functions listed in the column RTCRv1 of table 4.14 are independent of each other and executable in parallel. The second column shows the corresponding functions in RTCRv2. As discussed, most of the checkpointing logic is refactored into multiple classes, which explains the fact that each class provides a `checkpoint()` function. In order to execute each of these functions in parallel, the class `Checkpointable` is implemented.

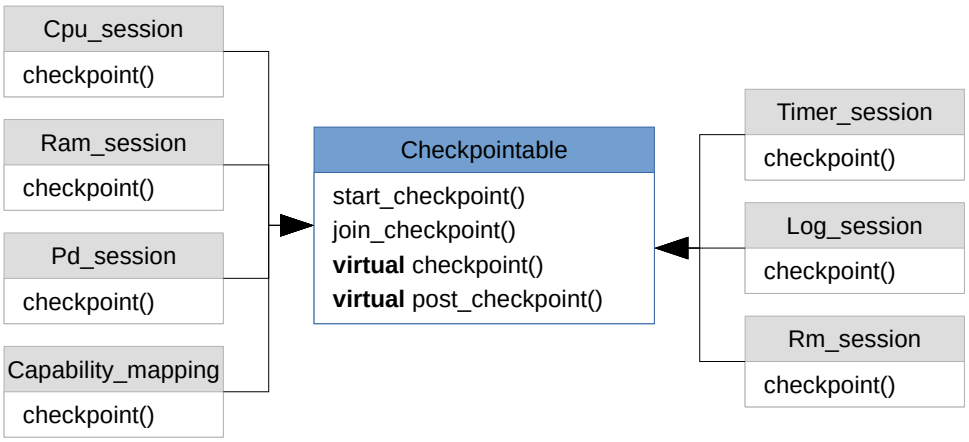

Figure 4.15: Extending Classes with Multi-threading Capabilities through Inheritance of `Checkpointable` class

4.5.2 Checkpointable Class

The class `Checkpointable` extends an arbitrary subclass with multi-threading capabilities. It requires the implmentation of the method `checkpoint()` in the subclass. This method is executed by an isolated thread. As shown in figure 4.15, all checkpointing classes are extendend this way. Each class also inherits the method `post_checkpoint()`. This method can optionally be overriden by a program routine, which should be executed after an actual checkpoint. Thus it is possible to prepare for the next checkpoint, while the child application is running. The method `start_checkpoint()` starts the execution of the corresponding `checkpoint()` method. With `join_checkpoint()`, the calling thread is paused until the method `checkpoint()` finished.

CPU Core Assignment

Each thread is assigned to a CPU core by setting a corresponding affinity location in the XML configuration. A thread is addressed by the unique attribute `name` and is assigned to an affinity location by the XML attributes `xpos` and `ypos`. Listing 4.3 shows a typical configuration for a dual-core platform. Except for the RAM session, all threads are executed on first CPU core. It makes sense to put the RAM session on the second core, it lasts the most in a checkpoint.

```
<config>
        <checkpointable name="cpu_session" xpos="0" />
        <checkpointable name="pd_session" xpos="0" />
        <checkpointable name="ram_session" xpos="1" />
        <checkpointable name="rm_session" xpos="0" />
        <checkpointable name="rom_session" xpos="0" />
        <checkpointable name="log_session" xpos="0" />
        <checkpointable name="timer_session" xpos="0" />
        <checkpointable name="capability_mapping" xpos="0" />
</config>
```

Listing 4.3: Assignment of Checkpointable Threads to CPU Cores

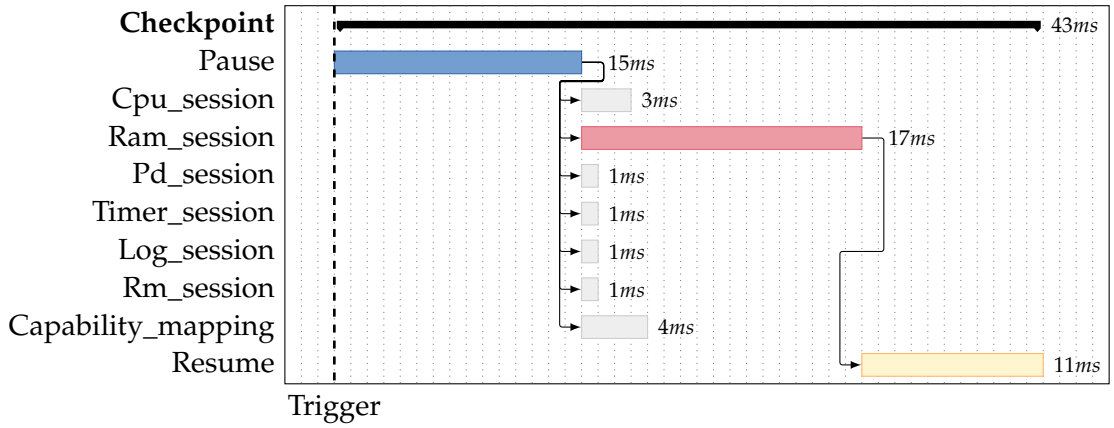

Figure 4.16: First Checkpoint with a 4 KiB Dataspace Allocation on Fiasco.OC & Zybo Board

4.5.3 Checkpoint Stages

All sessions are checkpointed in parallel, so the checkpoint process is divided only in three stages. At first, the child application is paused. In the second stage, all session are checkpointed. In the last stage, the child application is resumed. These stages are also recognizable in figure 4.16. It shows the first checkpoint of a child application, which allocates one 4 KiB dataspace. The measurements are taken from figure 7.14 and base on test-cases executed with Fiasco.OC on a Zybo board. It can be seen, that all sessions are checkpointed in parallel after the child is paused. The resuming depends on the longest task, that is in general the memory copying process implemented in the RAM session. In fact memory copying takes 40% of time of a checkpoint. This confirms the efforts of past works, which have been focused on optimizing the copying process.

4.6 Checkpoint Control Flows

This chapter combines the concepts of section 4.4 Checkpoint Logic and section 4.5 Parallelization by considering the execution flows during a checkpoint. Describing all control flows is beyond the scope of this chapter. Therefore only the CPU session is embedded in the comprehensive checkpoint process of RTCRv2. This is shown in figure 4.17 and the sequence diagrams of all further sessions are listed in appendix A. The control flow graphs of these sessions can be embedded in figure 4.17 as it is done exemplary with the CPU session.

In this figure, the *Actor* describes the application, which initializes the child application and loads the *Base_module*. With the start of the child application, the *Cpu_session* will be created. This provides an API for creating the *Threads* of the child application. The checkpoint is getting started by calling the method `checkpoint()` of the Base_module.

Stage 1: Pausing

At first, the `pause()` method of the CPU session is called. This iterates through the list of created threads and pauses each thread one by one.

Stage 2: Checkpointing

With this stage, the actual checkpointing starts. The `Checkpointable` object is notified with `start_checkpoint()`, which starts the execution of the `Cpu_session::checkpoint()` method in a separated thread. If the `Checkpointable` thread is started, the *Base_module* calls the method `join_checkpoint()`. This pauses the calling thread, which is the main thread of the *Actor*. Meanwhile, the CPU session iterates through the list of threads and calls the `checkpoint()` method of each element. The method checkpoints the state of the thread. If the checkpointing of the CPU session is derminated, the sleeping thread of the *Actor* will be resumed.

Stage 3: Resuming

Now, each thread of the child application is resumed by calling the `resume()` method of the CPU session. But the checkpoint has not been done yet. The `Checkpointable` executes the `post_checkpoint()` method in order to give the CPU session the possiblity to prepare for the next checkpoint.

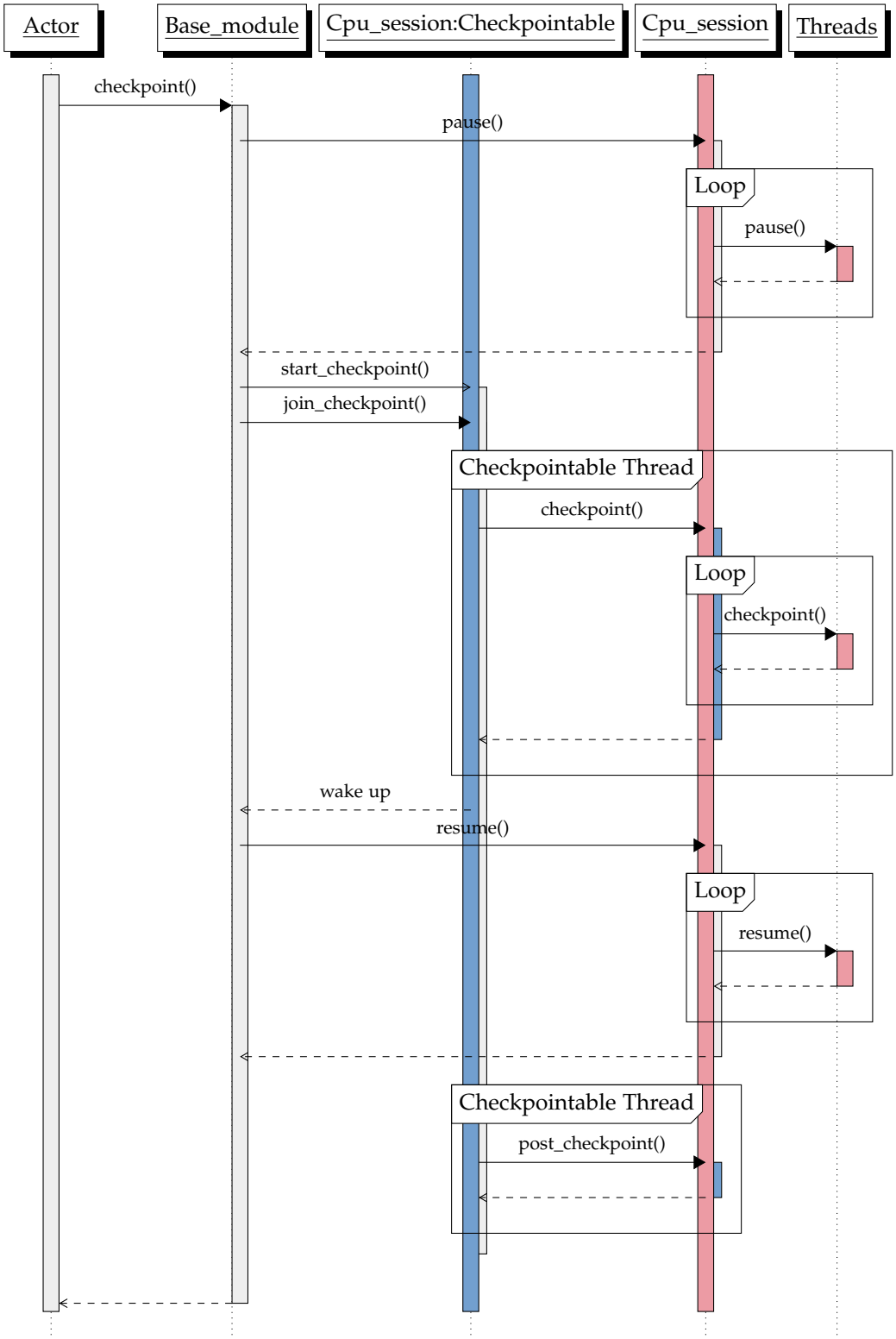

Figure 4.17: Pausing, Checkpointing and Resuming a Child Application in RTCRv2

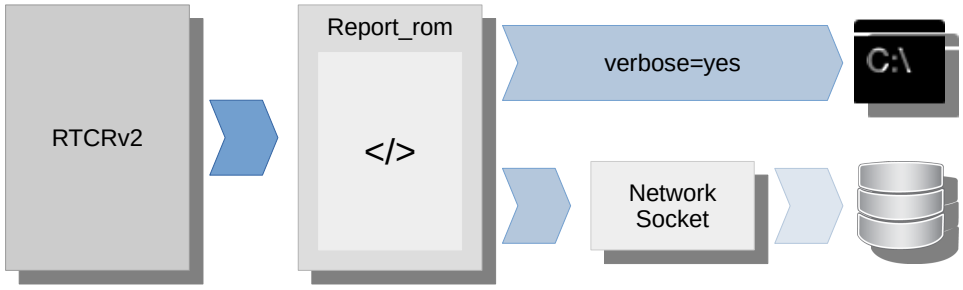

Figure 4.18: Transfer of the Measurement Data from RTCRv2 to a Network Storage

4.7 Performance Measurements

One important key function is the possibility to measure internal program routines of the RTCRv2 with a minimum of overhead. The measurement logic is integrated in the `Checkpointable` class.

Results of measurements in previous theses are based on the profiling library `genode-Profiler` [5]. Unfortunately, each single measurment of a time period has a bias of 3 - 5 ms, because the time for transfering the measurement result via UART to the recording computer is taken in account. Often called functions significantly distort the measurement results. In order to bypass this disadavantage, the performance measurements of threads are recorded during a test on the target platform and subsequently transfered to the recording computer.

For this reason, the Report API of Genode is integrated in RTCRv2. Basically, the *Report_rom* is a service which provides a XML-based storage to other applications. Such a storage is protected by a pre-configured policy, which assigns read and write accesses to selected applications. Whenever a content of a storage is modified, the subscribed applications will be notified. One use-case is shown in figure 4.18. Measurements of the last checkpoint are stored to the *Report_rom*. Another application reads measurement data from the storage and transfers it to a network database. This way, future performance profiling can be automated in large scale and it is less error prone than an UART link. Additionally, if the *Report_rom* is configured with `<config verbose=yes/>`, all measurements will be printed directly on the console.

4.8 Marshalling

Marshalling is the process in which the *Cold Storage* is serialized and compressed to a binary format. On the one hand, this includes the serialization of all objects. On the other hand, all dataspaces of the *Cold Storage* must be compressed and packed, too. Also packing the binary of the application is a requirement, because it might not be available on the receiver side.

A solution for serializing the *Offline Storage* of the RTCRv1 is implemented by Reisner [13]. It bases on the serialization library *protobuf*, that serializes all objects to a string [6]. This string is directly transfered with the network to another machine. Subsequently the content of all dataspaces is written to the open network socket. The transfer is realized with the network library *lwip*. On the receiver side, the *protobuf* buffer and dataspaces are translated to a valid *Offline Storage*. This approach has two disadvantages:

- The serialization process depends on an open connection. Due to the exclusive access to the *Cold Storage*, the marshalling blocks further checkpoints in the meanwhile. An unstable network connection influences the frequency in which checkpoints are taken.
- Each dataspace is directly sent to the receiver. In case of a bad utilization of allocated memory, a lot of unsused memory is transfered. This could be prevented by interposing a compression.

Both problems can be solved by separating the serialization and the network logic. The marshalling of the *Cold Storage* produces a simple dataspace, which contains a compressed snapshot of a *Cold Storage*. Such a dataspace can be processed further, for example by a network component. For the integration of marshalling in RTCRv2, following two approaches are discussed.

Post Checkpoint Marshalling

Like the checkpointing logic, the serialization can be integrated into each session class. The post processing hook `post_checkpoint()` can be used, because each class inherits from the `Checkpointable` class. The integration of the serialization into the `post_checkpoint()` method matches with the design concept of RTCRv2, but it has the following disadavantages:

- The serialization techniques can not be replaced easily, because each session class depends on the *protobuf* library.
- As the C/R, the *protobuf* must be updated to newer Genode releases as well. However this is not trivial, because *protobuf* depends on other libraries, like *zlib*, *pthread* and the C++ Standard Library. Broken library dependencies will increase the risk that the RTCRv2 can not be upgraded to future releases of Genode.

Based on these facts, the following approach is chosen for the implementation.

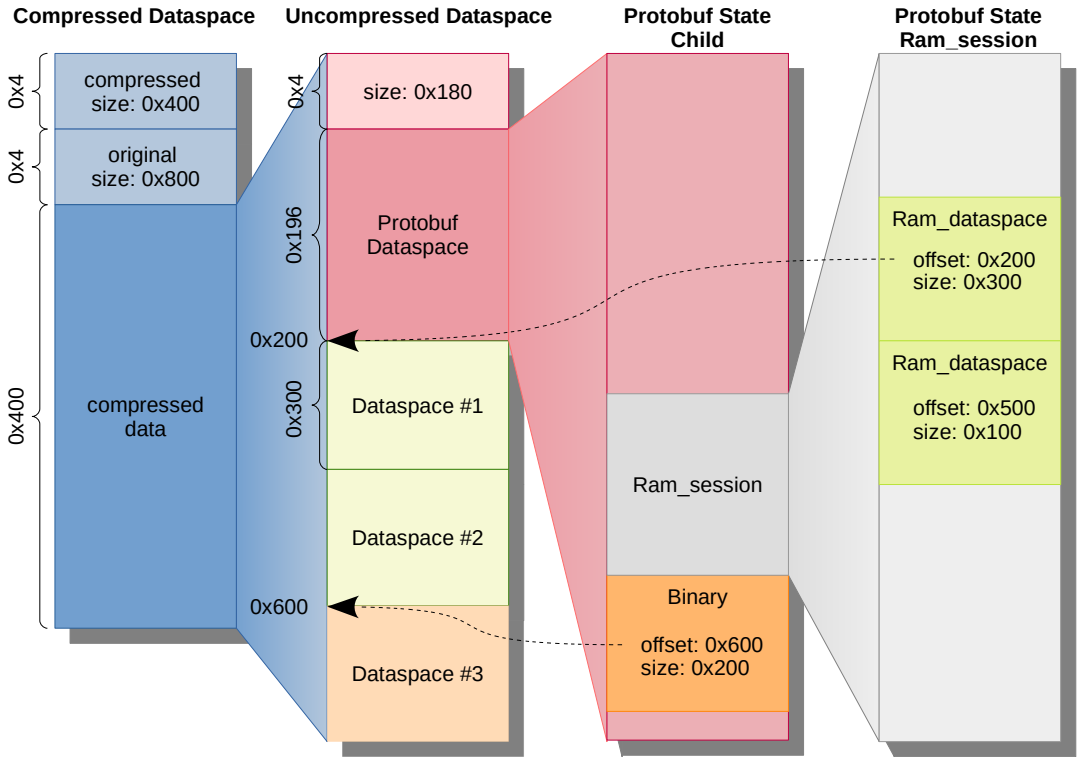

Figure 4.19: Internal Representation of a Serialized and Compressed *Offline Storage*

External Marshalling

The serialization logic is implemented in a new class. This is possible, because all *Cold Storage* objects are public accessible. Thereby various serialization classes supporting different concepts are implementable.

As proof of concept, the library *rtcr_serializer* is implemented. It bases on the work of Reisner, but also integrates a *zlib*-based compression and optionally packs the binary of the child application. In order to support these features, a layer-based binary format is required.

Binary Format

The internal representation of the binary format is shown in figure 4.19. The outermost layer (blue) has a header of 8 bytes in which the sizes of the compressed and uncompressed data are stored. These values are important for uncompressing the data. On the one hand, it is not possible to conclude from the size of the compressed dataspace to the size of data, because dataspace sizes are aligned to the page size. On the other hand, the uncompressed size is required for preparing a dataspace, in which the uncompressed data can be written. The uncompressed dataspace is split in the serialized

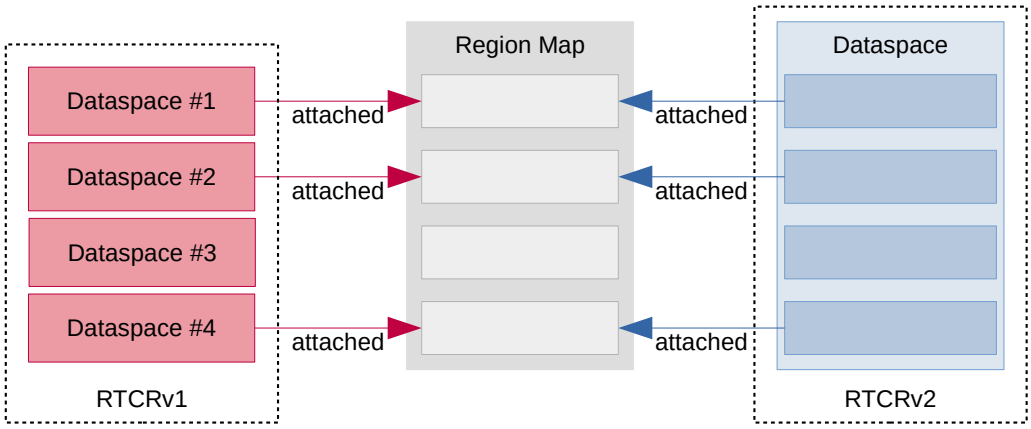

Figure 4.20: Implementation of Incremental Approach in RTCRv1 and RTCRv2

protobuf data (red) and the content of further dataspaces (light green/orange). The size of the protobuf data are stored in the first 4 bytes and the protobuf data starts with the fifth byte. The protobuf data can be deserialized to protobuf states. These are generic container objects which stores the information of each *Cold Storage*. The *Cold Storage* of the *Ram_session* is presented by a state object (gray). This state stores the meta information (green) of each dataspace, which has been allocated by the child application. This also includes an offset attribute, which references to a position in the uncompressed dataspace. At this position, the raw content of the dataspace is deposited. The position of the binary (light orange) is stored within the `Binary` state (orange).

4.9 Optimization of Incremental Approach

The *Incremental* approach copies only memory regions, which have been modified since the last checkpoint. In order to detect write accesses, the incremental checkpoint mechanism utilizes managed dataspaces. In contrast to a regular dataspace, a managed dataspace is not backed by physical addresses but by a virtual memory region. A managed dataspace can be created by using a region map. If a child application allocates memory, a managed dataspace will be handed out. Then the managed dataspace is backed by a multiple of smaller and regular dataspaces. These are detached from the virtual memory at the beginning. Therefore, whenever the child application tries to access a virtual memory, a page fault is thrown. In case of the incremental approach, a page fault handler catches the fault and attachs the correct dataspace to the virtual memory. So only attached dataspaces must be copied during a checkpoint.

This mechanism has one drawback. As smaller the size of the backed datapaces is, as more efficient is the approach. Consequentaly, a lot of small regular dataspaces are

required. But for each dataspace, a capability is nececessary. Finally, the kernel is not able to handle this number of capabilities, which results in an overflow of the capability space.

This problem is fixed with the implementation of the *Incremental* approach in RTCRv2. Figure 4.20 shows the difference of the implementations between RTCRv1 and RTCRv2. In fact, it is possible to attach only a portion of a regular dataspace. The `Region_map` class provides the method `attach_at(ds_cap, local_addr, size, offset)`, which attachs the dataspace `ds_cap` at the virtual address `local_addr`. Instead of the whole dataspace, only the memory from the `offset` to `offset+size` is attached. Therefore all small backing dataspaces can be replaced by one dataspace, which has the same size as the managed dataspace.

5 Implementation

This chapter gives at first an introduction to the modularisation techniques of RTCRv2. An overview of the software architecture follows and each class is described in detail. The consequences of porting RTCRv2 from Genode 16.08 to Genode 19.08 are introduced. The chapter ends with a guideline for implementing a new module.

5.1 Modularisation

A general overview of the file structure within a Genode repository and a module is given in the chapter 4 Design. This file structure is considered in more detail for explaining the pre-compiling mechanisms, which are required for supporting multiple kernels and architectures. The debug mode is also introduced.

Genode Specializations

The build system of Genode depends on several variables. One of these is the `SPECS` variable in which keywords are stored. Based on these keywords, a target platform is characterized by

- a kernel API such Fiasco.OC, Linux, L4.
- a hardware architecture such as X86, ARM, AARCH64.
- a certain hardware facility, for example custom devices.
- other properties such as software license requirements.

If the keyword `sel4` is included in `SPECS`, the portion of code that is required for binding seL4 to the Genode OS, will be included. In particular, a make-rule of the corresponding *REPO/mk/spec-SPECNAME.mk* file is included to the build process. On one hand, the `SPECS` variable is set by the user in the *etc/specs.conf* file of the build directory. On the other hand, the variable can be extended on a repository base. For example, if the sel4 kernel is chosen in the main configuration file *BUILDIR/etc/build.conf*, the *base-sel4/etc/specs.conf* add automatically `sel4` to the `SPECS` variable. This way, the build processes for the RTCRv2 libraries are configured to support the sel4 and Fiasco.OC kernels, as well as the 32-bit and 64-bit architectures.

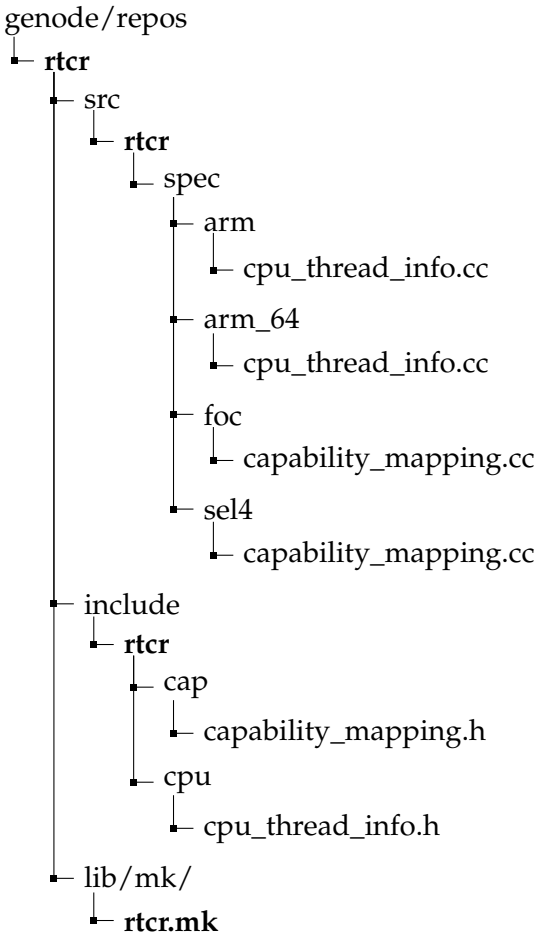

Figure 5.1: Directory Structure for Kernel & CPU Architecture related Source Code of RTCRv2

```
1 SRC_CC += cpu_thread.cc
2
3 ifeq ($(filter-out $(SPECS),arm),)
4 vpath % $(REP_DIR)/src/rtcr/spec/arm
5 endif
6
7 ifeq ($(filter-out $(SPECS),arm_64),)
8 vpath % $(REP_DIR)/src/rtcr/spec/arm_64
9 endif
```

Listing 5.1: Include Implementation File based on SPECS variable

Support of Multiple Kernels & Architectures

The directory tree presented by figure 5.1 shows an extract of the source files. The header files *capability_mapping.h* and *cpu_thread_info.h* declares only the classes, but the actual implementation is in the **.cc* files. Some implementations depends on an architecture or kernel.

For example the code in *cpu_thread_info.cc* accesses the CPU registers. These differ from architecture to architecture. Another example is the checkpointing routine of the kernel capabilities. Therefore, two different implementions exist for the kernels Fiasco.OC and seL4.

The plaform-related implementions are stored in *src/rtcr/spec/SPECNAME*. Which implementation is chosen, bases on the make-rules of *lib/mk/rtcr.mk*. In this file, the whole build process of RTCRv2 is defined. Listing 5.1 shows the make-rules for selecting the *Cpu_thread* implementation. The search path to the source file *cpu_thread.cc* is configured, depending on the variable SEPCS.

Debug Mode

Another aspect of the compile process is the possibility to enable/disable the debugging and profiling mode of RTCRv2. If the debugging mode is enabled, most functions write a short notice to the console. Due to the fact that this significantly slows down the checkpoint process, the logging commands can be excluded from the compile process by the pre-compiler. In order to enable logging, the keyword DEBUG must be added to the variable SPECS in *BUILDDIR/etc/specs.conf*. This causes the build process of Genode to load *rtcr/lib/mk/spec/debug/rtcr.mk* instead of the default makefile. The content of this file is shown in listing 5.2. It includes the default makefile, but extends it with the two macro definitions DEBUG and VERBOSE. These instructs the pre-compiler to include all program routines, which are wrapped in `#if DEBUG ... #endif` or `#if VERBOSE ... #endif`.

```
1 include (REP_DIR)/lib/mk/rtcr.mk
2 CC_OPT += -DVERBOSE
3 CC_OPT += -DDEBUG
```

Listing 5.2: Enable Debugging by setting the Macros `DEBUG` and `VERBOSE`

5.2 Classes

An overview of the most important classes in RTCRv2 is given by figure 5.2. The classes are grouped in logic units.

Session Logic Each session is implemented in a class, which includes the *Hot & Cold Storage* concept and a corresponding checkpointing logic. The class `Cpu_session` is examplarly presented in this chapter.

Parallelization Logic Each session depends on the `Checkpointable` class. It extends the derived class with parallelization capabalities.

Service Logic The `Root_component` template provides a service class for each session. This constructs a session object, if the child application establishs a connection to a service.

Child Logic The `Child` class represents the child application on RTCRv2-side. It responds to service requests from the application.

Module Logic A module inherits the core implementation from `Init_module` and is mainly responsible for initializing a service instance for each session.

Module Registration Logic In case of multiple modules, the `Module_factory` provides an API for dynamically creating modules based on the module name. The `Module_factory_builder` template registers modules.

5.2.1 Module_factory

In the `Module_factory` is the logic for registring a module. As shown in figure 5.3, the class combines the *Registry* and *Factory* pattern. Each instance of the class registers itself to the global module list, but the `create()` method also provides an interface for initializing a module. The template class `Module_factory_builder` even simplifies the registration process of a module, as shown exemplary for the `Base_module` in listing 5.3. The static method `Module_factory::get(MODULE_NAME)` returns the module factory, which corresponds to the `MODULE_NAME`. The factory has the method `create()`, which initializes the module and returns a pointer to the module object. Listing 5.4 shows the creation of the base module.

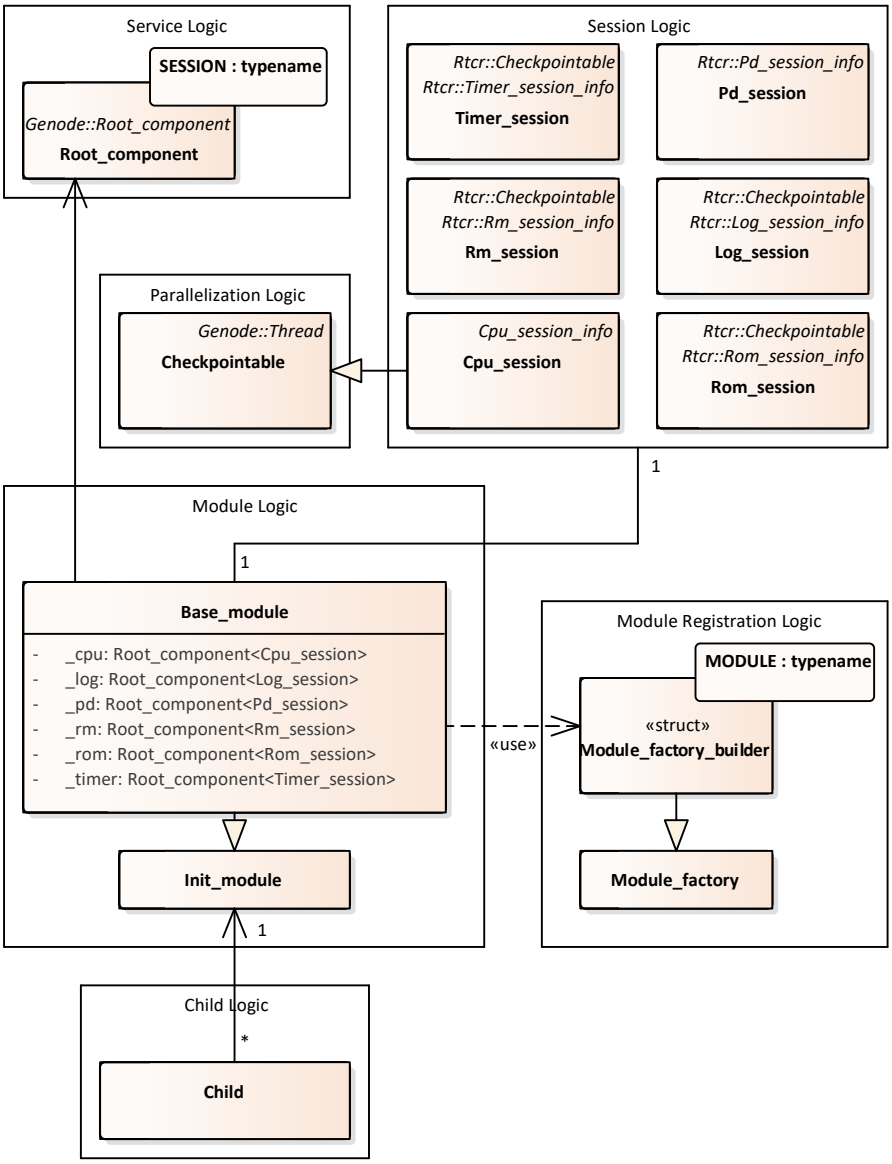

Figure 5.2: Overview of RTCRv2 Classes grouped in Logic Units

```
1 class Base_module {};
2
3 Module_factory_builder<Base_module> _base_module_factory_instance;
```

Listing 5.3: Registring the `Base_module` (*src/rtcr/base_module.cc*)

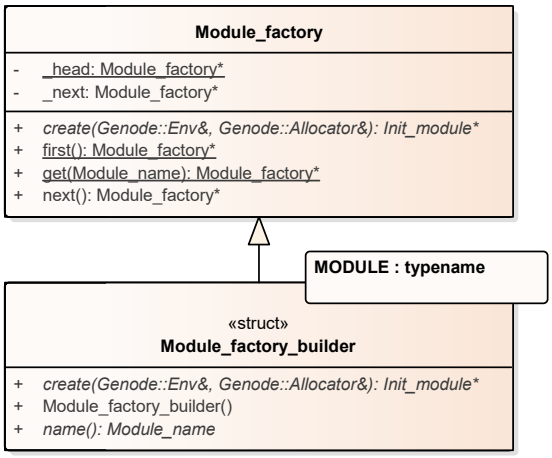

Figure 5.3: `Module_factory` & `Module_factory_builder` Class

```
1 Base_module *module = Module_factory::get("base")->create(env, heap);
```

Listing 5.4: Creating a instance of the `Base_module` by its name (*src/app/rtcr_app.cc*)

5.2.2 Init_module

A module provides an interface for pausing checkpointing and resuming all child applications. In particular, if `checkpoint()` is called, all `Checkpointable` threads will be started. `child_info()` provides an access to the *Cold Storage* of the child applications. A module is also responsible for updating the performance measurement values in the `Report_rom`, when it is actived via `report_enabled(bool)`.
Due to the fact, that the implementation of these methods is similiar for each module, the code is provided in the abstract class `Init_module`. Thus each module, for example

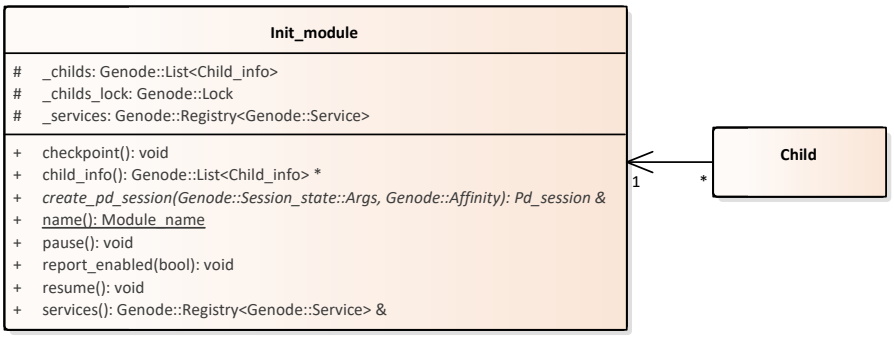

Figure 5.4: The class `Init_module` provides the core functionality of a module

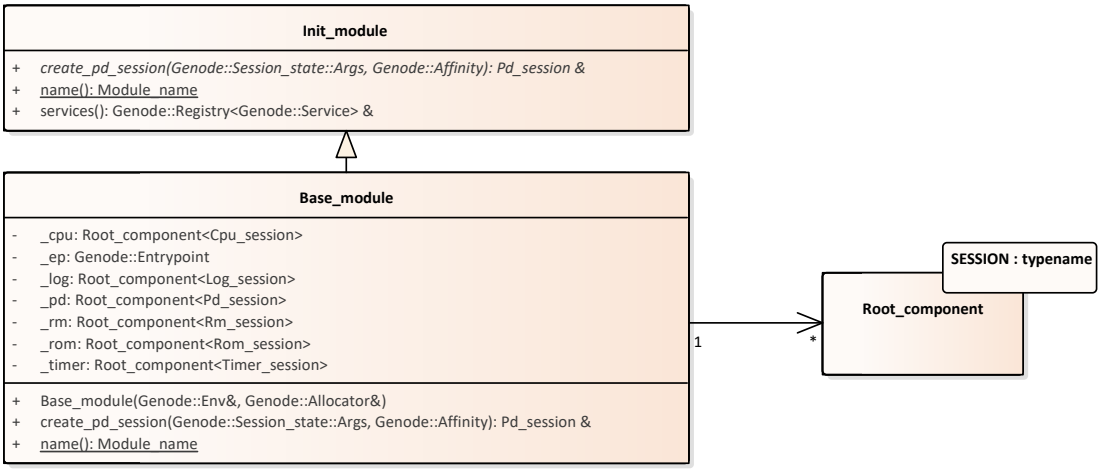

Figure 5.5: Class `Base_module` inherits from `Init_module`

the `Base_module`, primarily inherits from `Init_module`.
The class diagram of `Init_module` shows two more methods in figure 5.4. The virtual method `create_pd_session()` is a stub, which will be implemented by the inheriting class. It provides, independently of the actual module implementation, an interface for creating a PD session. This is important, because the initialization of a child requires such a PD session. A child application also requires access to the services, which are started by a module. Therefore the `Init_module` has a service registry object named `_services`. The `Child` class can access it through the `services()` method. Due to the inheritance, the derived module has a direct access on the object. This is also the case for the list of child applications `_childs`. Due to the fact that `_childs` is accessed in multiple threads, the lock `_childs_lock` is required. It ensures an exclusive access of `_childs`.

5.2.3 Base_module

The general implementation of a module is exemplarly introduced in four steps shown in figure 5.5. At first, a module inherits from `Init_module`, which extends the module class with the required core logic. Second, the static function `name()` is overriden with the name of the module. This helps the `Module_factory` to identify the module in the list of registered modules. Third, the `create_pd_session()` is implemented. Finally the module initializes a set of services in the constructor. A service is implemented with help of the template `Root_component` and provides an interface for creating sessions. For example, the `Root_component<Pd_session>` creates PD sessions. Listing 5.5 shows the constructor of the `Base_module` class. Each service is initialized with a reference to the Genode Environment `env`, a heap allocator `alloc`, a module-related Entrypoint `_ep`, the

```
Base_module(Genode::Env &env, Genode::Allocator &alloc)
    :
    Init_module(env, alloc),
    _ep(env, 16*1024, "resources ep", Genode::Affinity::Location()),
    _pd(env, alloc, _ep, _childs_lock, _childs, _services),
    _cpu(env, alloc, _ep, _childs_lock, _childs, _services),
    _log(env, alloc, _ep, _childs_lock, _childs, _services),
    _timer(env, alloc, _ep, _childs_lock, _childs, _services),
    _rom(env, alloc, _ep, _childs_lock, _childs, _services),
    _rm(env, alloc, _ep, _childs_lock, _childs, _services) {}

Pd_session &create_pd_session(Genode::Session_state::Args args,
                              Genode::Affinity affinity) override {
    return _pd.create(args, affinity);
}

static Module_name name() { return "base"; }
```

Listing 5.5: Initialization of `Base_module` and implementation of `create_pd_session()` and `name()`

list of child applications `_childs` and the service registry `_services`. Consequentally, each service is directly available after the initalization of the module.

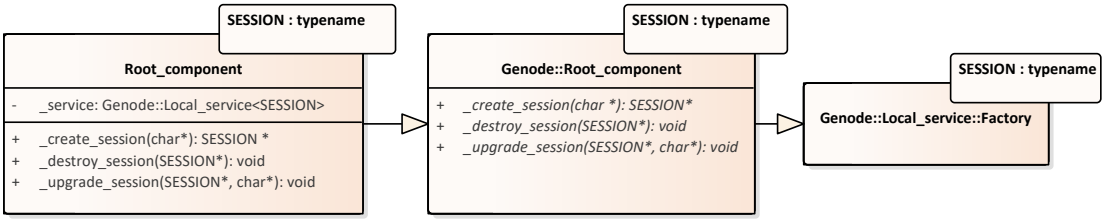

Figure 5.6: Class Template `Root_compoent` provides a Service Interface

5.2.4 Root_component

Figure 5.6 shows the inheritance chain of the `Root_component` template class. It inherits from the class `Genode::Root_component`, which in turn inherits from `Genode::Local_service::Factory`. The `Root_component` implements the methods:

_create_session() Creates a new instance of a session.

_destroy_session() Destroys an instance of a session

_upgrade_session() Upgrades a session with resources like capabilities or memory.

If a child application establishs a connection to a service, `_create_session()` creates a new session object, which actually handles the communication between child and service. The linking between application and service is implemented with the `Genode::Service` class. It is a wrapper around the API of the `Root_component` and provides a similar API. Thus, each service is represented by an object of type `Genode::Service`, which makes the service handling much easier.

Basically, the `Genode::Service` class can not be initialized directly, instead the base-class template `Genode::Local_service` is required. But this template requires a template parameter of type `Genode::Local_service::Factory`. Due to the inheritance chain of the `Root_component`, is possible to create directly a `Genode::Service` object from the `Root_component`. The service object is registered to the service registry, which is handed over in the constructor.

The second important task of the `Root_component` is managing the list of child applications. This list comes from the `Init_module` and is also handed over in the constructor. If a new session is requested by a child application, the routine in listing 5.6 is executed. At first, an existing child object is searched by its name. The name of the child application is part of an argument string which is handed over in the `_create_session()` method. If the corresponding `Child_info` is not found, a new object will be created and inserted in the list of childs (`_childs`). Therefore, a child application is inserted into the list with the first session request. It will be removed from the list, if the PD session and CPU sessions are destroyed again. Without these sessions, a child can not run.

```
1  _childs_lock.lock();
2  Child_info *info = _childs.first();
3  if(info) info = info->find_by_name(name.string());
4
5  /* child_info does not exist, let's create it */
6  if(!info) {
7    info = new(_alloc) Child_info(name.string());
8    _childs.insert(info);
9  }
10 _childs_lock.unlock();
```

Listing 5.6: Registring new `Child_info` objects for newly created child applications in *include/rtcr/root_component.h*

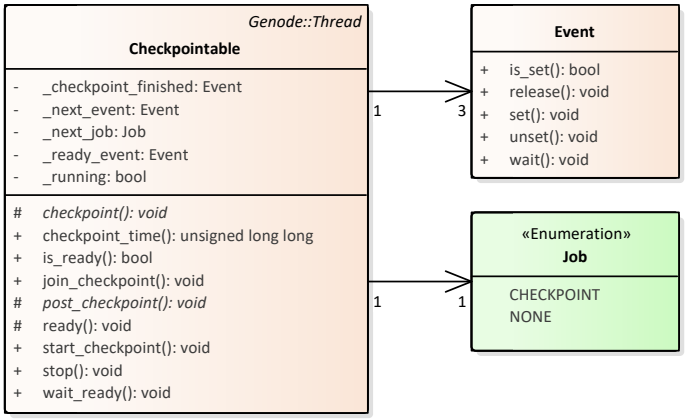

Figure 5.7: Class `Checkpointable`

5.2.5 Checkpointable

The abstract class `Checkpointable` extends an arbitrary derived class with multi-threaded checkpointing capabilities. The method `start_checkpoint()` starts the execution of the virtual method `checkpoint()` in a thread. The method `join_checkpoint()` pauses the calling thread and wakes it up, when the `checkpoint()` method returned. After `checkpoint()`, the method `post_checkpoint()` is executed in a thread. This allows the derived class to process post-checkpointing routines without blocking the execution of the child application. If `wait_ready()` is called, the calling thread will be paused until `post_checkpoint()` is finished. The method `checkpoint_time()` returns the execution time of the last `checkpoint()` call.

The logic of `join_checkpoint()` and `wait_ready()` is actually provided by the `Event` class. It wraps an boolean value, which can be accessed with `is_set()`, `set()` and

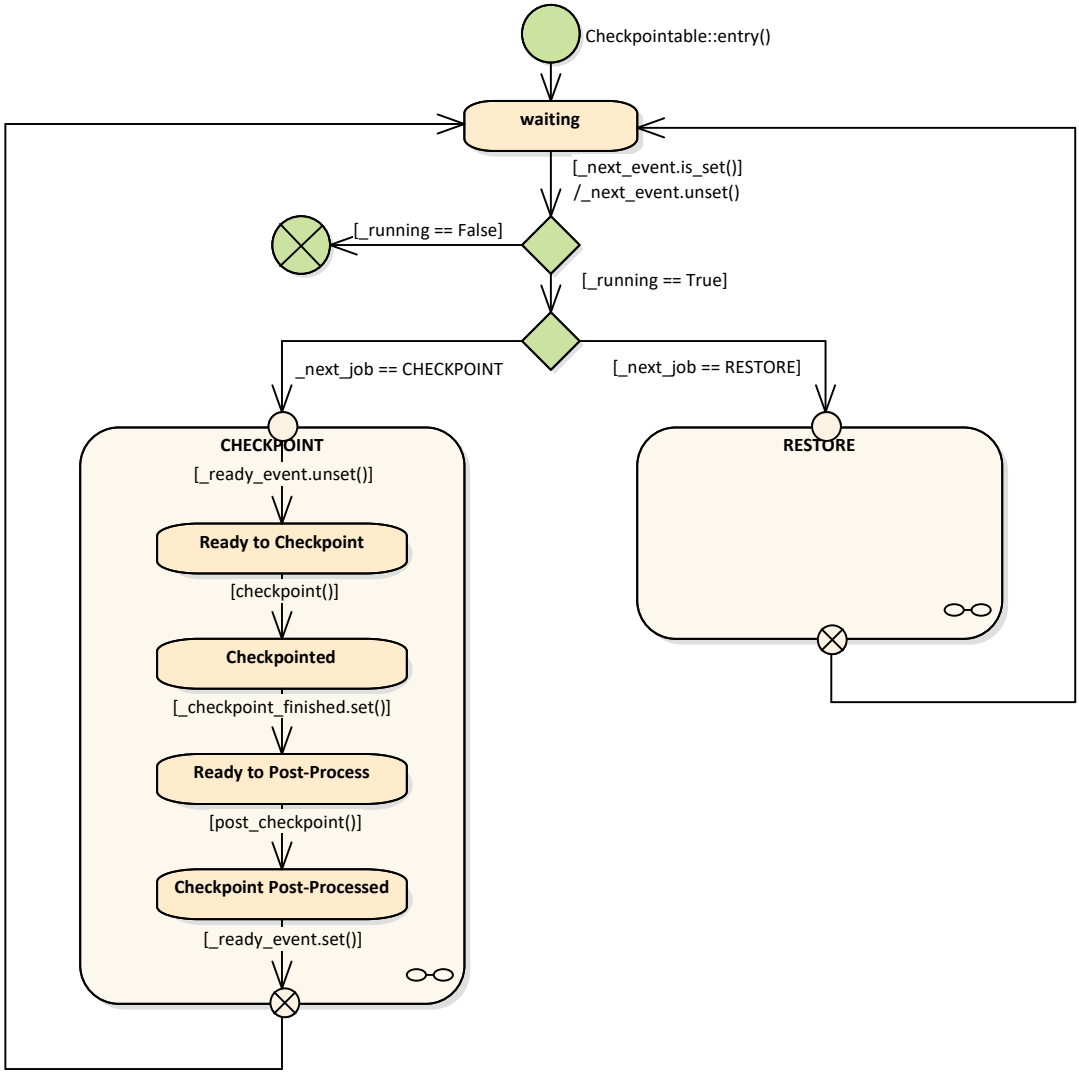

Figure 5.8: State-Machine of the Class `Checkpointable`

unset(). It will block a calling thread, if the method wait() is called. In particular, the calling thread will be added to a list of waiting threads. If set() or release() is called, all threads from the waiting list are waked up again. With this functionality, it is possible to implement the state machine of the Checkpointable class.
The state machine shown in figure 5.8 does not only support a checkpoint, but also the restore process has already been prepared for a future implementation. As long as the variable _running is set, the thread of the Checkpointable class is executed. This boolean will be unset, if the class is destructed. Calling start_checkpoint() will set _next_event and the _next_job = CHECKPOINT. This will cause the state machine to enter the checkpoint routine, where checkpoint() and post_checkpoint() is called. After this routines, the state machine will enters the waiting state and will be waked up again, if _next_event is set.

5.2.6 Cpu_session

Describing all sessions is beyond the scope of this chapter. Therefore, the concept of the *Hot & Cold Storage* is examplarly introduced with the Cpu_session class. Figure 5.9 shows the corresponding classes. If a child application wants to create a thread, it will establish a session with the CPU service and call the method create_thread of the session. Even if the child application only receives the capability to the new created thread, a Cpu_thread object is created on the service side. A reference to this object is also stored in the list _cpu_threads. The methods pause() and resume() pauses/resumes all threads in this list. Due to the fact that a child application is associated with a single CPU session, this will pause/resume all threads of the application.

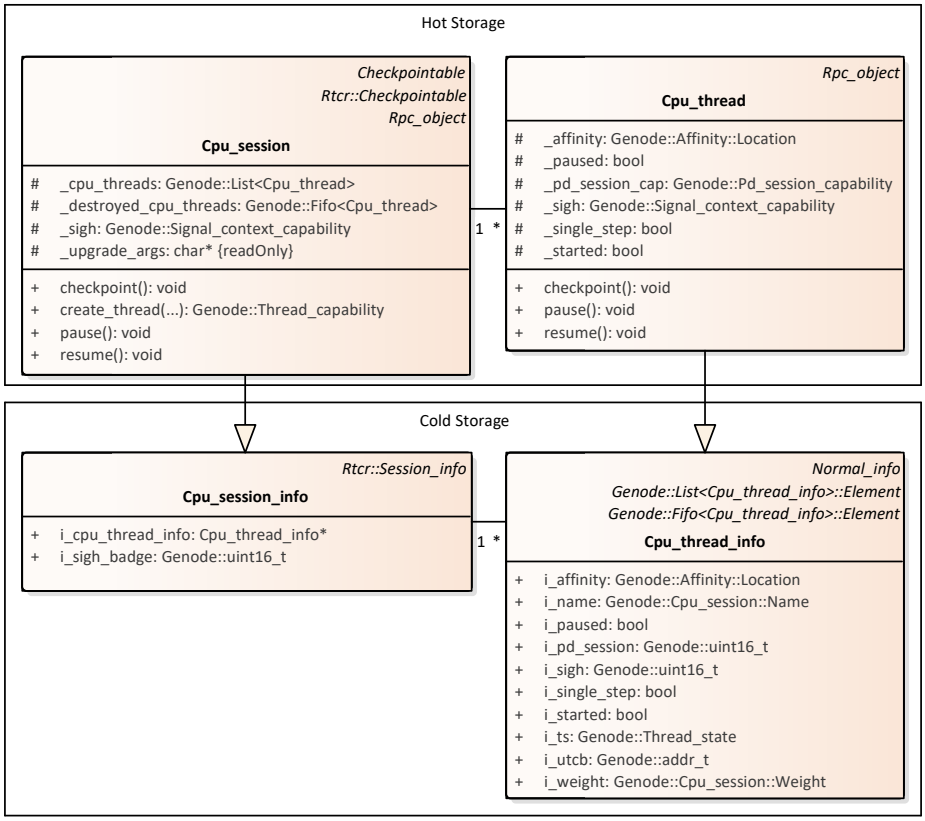

Figure 5.9: Classes `Cpu_session` and `Cpu_thread` with the corresponding *Cold Storage* classes

Hot Storage

The *Hot Storage* is represented by the class attributes which holds the current state of the `Cpu_session` and `Cpu_thread`:

- Cpu_session
  - **_sigh** A signal context capability for handling exceptions.
  - **_upgrade_args** An argument string, which stores the values of the last ressorce upgrade.
- Cpu_thread
  - **_started** Indicator if the thread is started.
  - **_paused** Indicator if the thread is paused.
  - **_single_step** Indicator if the thread is single stepped. This is a feature for the debugging tool GDB in order to execute instructions step by step.
  - **_affinity** The affinity position of this thread.
  - **_sigh** A signal context capability for handling exceptions.
  - **_pd_session_cap** The capability of the PD session which is associated with this CPU thread.

Cold Storage

Contrary to the *Hot Storage*, the *Cold Storage* of each class is implemented in a corresponding class with the name suffix `*_info`. This has the advantages, that if the implementation of the session changes, the classes of the *Cold Storage* are not affected. Therefore, the serializer and other components with access to the *Cold Storage*, do not depend on all modules, but only on the small interface of the *Cold Storage* classes. In order to make the attributes of the *Cold Storage* as accessible as it is the case for the *Hot Storage*, the `Cpu_session` and `Cpu_thread` classes inherit their *Cold Storage* classes. In general, an direct mapping from attributes of the *Hot Storage* to the attributes of the *Cold Storage* exist. For example, the `_sigh` capability also exist as `i_sigh` attribute in the *Cold Storage*. The *Cold Storage* attributes are distinguished by the prefix `i_*`. The *Cold Storage* has some more attributes. For example, attributes which are not changed during runtime and only need to be stored once.

```
1  void Cpu_thread::checkpoint()
2  {
3      i_started = _started;
4      i_paused = _paused;
5      i_single_step = _single_step;
6      i_affinity = _affinity;
7      i_sigh = _sigh;
8      i_pd_session_cap = _pd_session_cap;
9
10     /* get register state */
11     i_ts = _parent_cpu_thread.state();
12 }
```

Listing 5.7: Checkpointing `Cpu_thread` (*src/rtcr/cpu_thread.cc*)

These attributes are:

- Cpu_session_info

  i_cpu_thread_info A reference to a `Cpu_thread_info` object. This object is part of the `Cpu_thread` list and represents the head of the list since the last checkpoint.

- Cpu_thread_info

  i_weight Weight which influences the scheduling of the thread.

  i_utcb Dataspace capability of the User-level thread control block.

  i_name Name of the thread.

  i_ts Object of type `Genode::Thread_state`, which stores the state of the registers.

Checkpointing Cpu_thread

The `checkpoint()` method is shown in listing 5.7. In case of the `Cpu_thread`, the method is simple. Each attribute state of the *Hot Storage* is copied to the corresponding attribute in the *Cold Storage*. Only the registers are an exception. Those are obtained with the `state()` method of the parent thread.

Checkpointing Cpu_session

The checkpointing routine of the `Cpu_session` class is shown in listing 5.8. The first two lines checkpoints the attributes of the `Cpu_session`. In the following lines, the list of threads is checkpointed. This is identical to the concept described in chapter 4.4.2 Checkpointing a List.

```
/* checkpoint Hot Storage of Cpu_session */
i_upgrade_args = _upgrade_args;
i_sigh = _sigh;

/* Checkpoint list of CPU threads */
/* Step 1: Remove destroyed threads */
_destroyed_cpu_threads.dequeue_all([&] (Cpu_thread_info &cpu_thread) {
    _cpu_threads.remove(&cpu_thread);
    Genode::destroy(_md_alloc, &cpu_thread);
});

/* Step 3: checkpoint all threads */
Cpu_thread_info *cpu_thread = _cpu_threads.first();
while(cpu_thread) {
    static_cast<Cpu_thread*>(cpu_thread)->checkpoint();
    cpu_thread = cpu_thread->next();
}

/* Step 4: Update Reference */
i_cpu_thread_info = _cpu_threads.first();
```

Listing 5.8: Checkpointing `Cpu_session` (*src/rtcr/cpu_thread.h*)

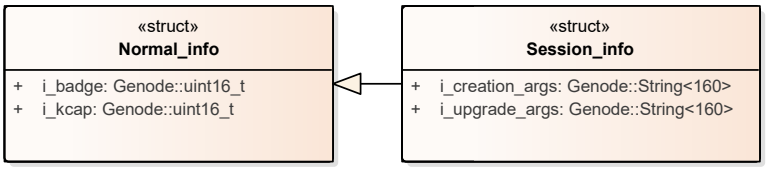

Figure 5.10: Class `Normal_info` and Class `Session_info`

At first, the elements from `_destroyed_cpu_threads` are dequeued and removed from the list of threads. The second step is absent as new created threads do not need to be prepared. In step three, each `Cpu_thread` in the list is checkpointed. In the last step, the reference is updated to the head of the list.

5.2.7 Cold Storage Classes

The previous section has already introduced the integration of the *Cold Storage* within the `Cpu_session` class. Two classes were hidden from the class diagram for reasons of representability. These are shown in figure 5.10.

In fact, almost each class of the *Cold Storage* inherits from one of these classes. This is due to the fact that most of the classes can be referenced by a capability. For example, the `Cpu_thread_info` inherits from `Normal_info`, because the `Cpu_thread` is referenced by capability. This way, the attributes `i_badge` and `i_kcap` are inherited. The `i_badge` value references to the capability of the class in user-land and `i_kcap` is the corresponding kernel capability. Session classes additionaly inherit from from `Session_info`. This class provides attributes for storing the creation- and upgrade-argument of a session. All `*_info` classes of the *Cold Storage* are shown in the appendix B.1.

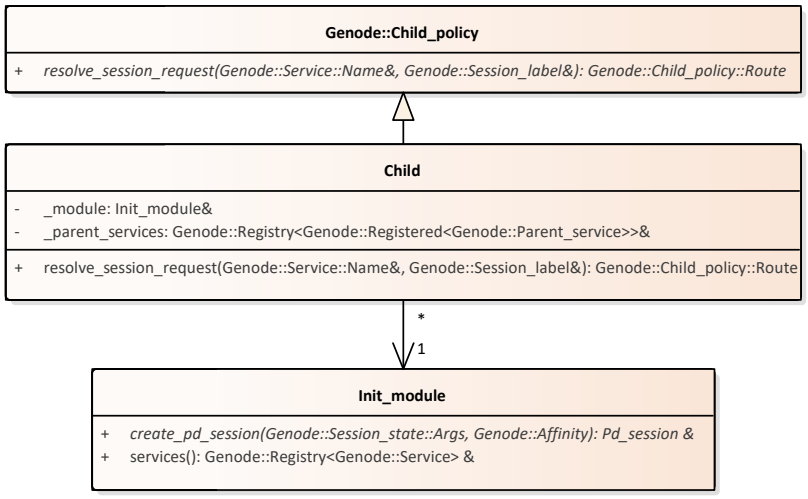

Figure 5.11: Class `Child`

5.2.8 Child

The class `Child` implements the RTCRv2-side of a child application. It creates a PD session for the child application, replies service requests and transfers requested RAM and capability quota. As shown by figure 5.11, the class depends on the `Init_module`. In particular, the `create_pd_session()` delivers a new created PD session and the `services()` method provides access to the service registry. With this access, it is possible to resolve the session requests of the child application and hand out the correct services. This routine is shown in listing 5.9. In fact, this is the logic, that hand out the custom intercepting services instead of the core services. Which service is requested, can be determinated by the argument `name`, which is passed by the function call.

1. In the first step, it will be checked, if the child requests a PD service. In this case, the previously created PD session is chosen. But as an object of type `Service` is espected, the PD session must be wrapped by a `Genode::Local_service::Single_session_factory` service. This is a special service class, which always returns a reference to the same session, instead of creating new session objects for each request.

2. In case, that the child application requests another service, like the CPU service, the matching service is searched in the service registry of the module.

3. If the requested service can not be served by the module, the request will be replied with a service provided by the parent component. These services are handed over in the constructor of the `Child` class and accessible with the `_parent_services` variable.

```
Genode::Child_policy::Route Child::resolve_session_request(
      Genode::Service::Name const &name,
      Genode::Session_label const &label)
{
        Genode::Service *service = 0;

        /* PD session is created in advance, so just hand over the
         * corresponding service */
        if (name == "PD") {
                service = &_pd_service;
        }

        /* service is provided locally */
        if (name != "ROM") {
                _module.services().for_each([&] (Genode::Service &s) {
                  if (service || s.name() != name) return;
                        service = &s;
                });
        }

        /* service is provided by parent */
        if(!service){
                _parent_services.for_each([&] (Genode::Registered<Genode::
                    Parent_service> &s) {
                  if (service || s.name() != name) return;
                  service = &s;
                });
        }

        if(!service) {
                Genode::error("Unknown service: ", name);
        }

        return Route { *service, label, Genode::Session::Diag{false} };
}
```

Listing 5.9: Resolving session requests of the child application (*src/rtcr/child.cc*)

5.3 Porting to Genode 19.08

The Genode API changed from release 16.08 to 19.08. This mostly considers the PD and RAM service. As described in the release notes of Genode 17.05, the API of the RAM service is integrated in the PD service. The separation of the memory management (RAM service) from the protection domain (PD service) based on an academic point of view, but did not yet provide any benefit. On the one hand, only one-to-one relationships between PD sessions and RAM sessions exist. On the other hand, the PD service has already served the role of an accountant for physical resources by managing the capability quota. Due to these reasons, the API of the RAM service is merged into the PD service and the RAM service becomes depracated in Genode 17.05 [14].

So this change concerns RTCRv2, as most checkpointing logic is implemented in session classes. In particular, the source code regarding RAM session is copied one-to-one into the PD session. The `Pd_session` is extended with two internal `Checkpointable` classes. The class `Pd_checkpointable` executes PD-related checkpointing routines henceforth. The `Ram_checkpointable` checkpoints the ram dataspaces. This fact is also visualized in the sequence diagram of the PD session in figure A.2 of the appendix.

Starting with Genode release 18.02, the child application did not started sucessfully on Fiasco.OC anymore. Weidinger identified the unimplemented methods `native_cap()` and `thread_state()` of the `Cpu_session` as cause of the problem. His patch setups a `Native_cpu_component`, those capability is requested by the `native_cap()` method. It has not been evaluated yet, whether this native cpu component must be checkpointed, too. Therefore, it is not part of a checkpoint.

5.4 Custom Module Creation

This chapter gives a guideline for creating a module. This is also documented in `rtcr/doc/module.md` of the `rtcr` repository. The required steps are examplary discussed based on the *cdma* module. This module implements a hardware-based acceleration of the memory copying. It can be found in the Git repository `rtcr_cdma`.

5.4.1 Creating a Repository and Preparing the Directory Structure

```
1 cd genode/repos
2 mkdir -p rtcr_cdma/{doc,include/rtcr_cdma,lib/mk,src/rtcr_cdma}
```

Listing 5.10: Preparing directory structure of the *cdma* module

It is assumed, that the main directory of Genode is prepared. The module is implemented in a new Genode repository. The repository name consists of the prefix

rtcr_ and the actual module name cdma. The commands in listing 5.10 create the required directories for a new module.

5.4.2 Creating a PD Session

The CDMA driver Cdma::Connection provides the copying method memcpy(). This method works like the software-based alternative Genode::memcpy(dst,src,size), but requires physical memory addresses for the parameters dst and src. Due to the similiarities of both copying approaches, only the Pd_session::_copy_dataspace() need to be replaced. Therefore, a custom PD session class is implemented first, as shown in listing 5.11.

In the beginning, the new PD session class Pd_cdma_session inherits from the original Pd_session of the *base* module.

In the next step, the _copy_dataspace() method is overriden. As shown in line 16 of listing 5.11, the CDMA copying approach will only work, if the dataspace is uncached. If this is not the case, the copying routine will fall back to the software-based copying by calling to the original Pd_session::_copy_dataspace() method. Otherwise, the physical addresses of the source dataspace (*Hot Storage*) and destination dataspace (*Cold Storage*) are determinated and the CDMA driver is instructed with the memory copying.

In the last step, the constructor of the derived class is implemented. It is important that the constructor of the base class is called. This happens in line 34. In the next line, the connection to the CDMA driver is established.

The implementation of the custom PD session is done. In the next step, the module class is implemented in order to serve the new PD service.

5.4.3 Creating a Module Class

Finally, the module class is implemented. As shown in listing 5.12, the module inherits from Init_module. Therefore, the abstract methods name() and create_pd_session() must be implemented. The lines 47-52 declare a service for each session. In line 47, the service for the previously implemented PD session is declared. These are initialized in the constructor (line 24 ff.). In order to register the module in the global list of modules, an instance of the template class Module_factory_builder is created and initialized in line 42. This makes the module accessible with Module_factory::get("cdma").

The implementation of the *cdma* module is finished. In a last step, the module is packed as library.

```
#include <rtcr/pd/pd_session.h>
#include <cdma_session/connection.h>

class Rtcr::Pd_cdma_session : public Rtcr::Pd_session
{
private:
    /* connection to CDMA driver */
    Cdma::Connection _cdma_drv;

protected:
    /* replace memory copying method */
    void _copy_dataspace(Ram_dataspace *ds) override
    {
        /* only copy a dataspace with hardware-acceleration, if it is
         * uncached */
        if(!ds->i_cached) {
            Genode::Dataspace_client dst_client(ds->i_dst_cap);
            Genode::Dataspace_client src_client(ds->i_src_cap);

            _cdma_drv.memcpy(dst_client.phys_addr(),
                             src_client.phys_addr(),
                             ds->i_size);
        } else {
            /* otherwise fallback to origin software-based copying */
            Pd_session::_copy_dataspace(ds);
        }
    }

public:
    /* constructor */
    Pd_cdma_session(Genode::Env &env, Genode::Allocator &md_alloc,
                    Genode::Entrypoint &ep, const char *creation_args,
                    Child_info *child_info)
    : Pd_session(env, md_alloc, ep, creation_args, child_info),
      _cdma_drv(env) {}
};
```

Listing 5.11: Implementation of a CDMA-driven PD session for the module *rtcr_cdma*

```
#include <rtcr_cdma/pd_session.h>
#include <rtcr/init_module.h>
#include <rtcr/root_component.h>

class Rtcr::Cdma_module : public virtual Init_module
{
private:
    Genode::Entrypoint _ep;

    /* create a service for each session class */
    Root_component<Pd_cdma_session> _pd; */ new PD session */
    Root_component<Cpu_session> _cpu;
    Root_component<Log_session> _log;
    Root_component<Timer_session> _timer;
    Root_component<Rom_session> _rom;
    Root_component<Rm_session> _rm;

public
    /* constructor */
    Cdma_module(Genode::Env &env, Genode::Allocator &alloc)
      : Init_module(env, alloc),
        _ep(env, 16*1024, "resources ep", Genode::Affinity::Location()),
        /* initialize one service instance for each session */
        _pd(env, alloc, _ep, _childs_lock, _childs, _services),
        _cpu(env, alloc, _ep, _childs_lock, _childs, _services),
        _log(env, alloc, _ep, _childs_lock, _childs, _services),
        _timer(env, alloc, _ep, _childs_lock, _childs, _services),
        _rom(env, alloc, _ep, _childs_lock, _childs, _services),
        _rm(env, alloc, _ep, _childs_lock, _childs, _services) {}

    /* name of the new module */
    static Module_name name() { return "cdma"; }

    /* create a new PD session */
    Pd_session &create_pd_session(Genode::Session_state::Args args,
                                  Genode::Affinity affinity) override {
        return _pd.create(args, affinity);
    }
};

/* Register module with help of the template Module_module_factory */
Module_factory_builder<Cdma_module> _cdma_module_factory_instance;
```

Listing 5.12: Implementation of the module class `Cdma_module`

```
1 SRC_CC = pd_session.cc cdma_module.cc
2 vpath % $(REP_DIR)/src/rtcr_cdma
```

Listing 5.13: Library Makefile of module *cdma* (*rtcr_cdma/lib/mk/rtcr_cdma.mk*)

5.4.4 Creating a Makefile

In order to provide the module as static library, the makefile in listing 5.13 is necessary. The variable `SRC_CC` lists all source files. The variable `vpath` specifies a list of directories that the make-tool should search. Now, the module is available as the static library `rtcr_cdma` and it can be linked with the core library `rtcr`.

6 Testbench

The evaluation of RTCRv2 bases on more than 10.000 measurements, targeting different architectures, platforms, kernels and modules. In order to generate this amount of test cases, an automated testbench is required. This chapter covers the necessary hardware components for the tests and the generation and execution of test cases. Finally, the data aggregation and evaluation is indroduced with a short example of Python code. The source code and documentation for the testbench are in the git repository for quality assurance `rtcr_qa`.

6.1 Hardware Components

The networking of all hardware components is shown in figure 6.1. The most important device is the Power Over Ethernet (POE) driven network switch (**gray**). It provides the required power for the Devices under Test (DUTs) (**blue**) by using the ethernet links. In combination with a POE splitting adapter, which is connected with each DUT, a DUT can be powered on and off by the switch. This can be controlled by a command that is sent to the switch.

If a DUT is switched on, it will connect to the DHCP server (**orange**) at first. This assigns an IP address and transmits the network location of the image[1] to the DUT. In this case, the image is provided by the Control Server with the TFTP protocol. This server is also responsible for the execution of test cases and the recording of measured results.

The measurement results are generated by the DUTs and printed out to the UART links. Each UART link is connected to a Raspberry Pi (**red**) with an UART-to-USB adapter. This way, the Control Server establishs a SSH session to the Raspberry Pi and accesses the output of the connected UART ports. The application `tio`[2] proved to be the most stable solution for this use case.

6.2 Test-case Generation

Each test-case is represented by an **.run* TCL Script. This is the common way for configurating, building and running an instance of the Genode OS in the Genode

[1] Compiled uImage & image.elf, which can be executed by a DUT.
[2] TTY terminal I/O Application

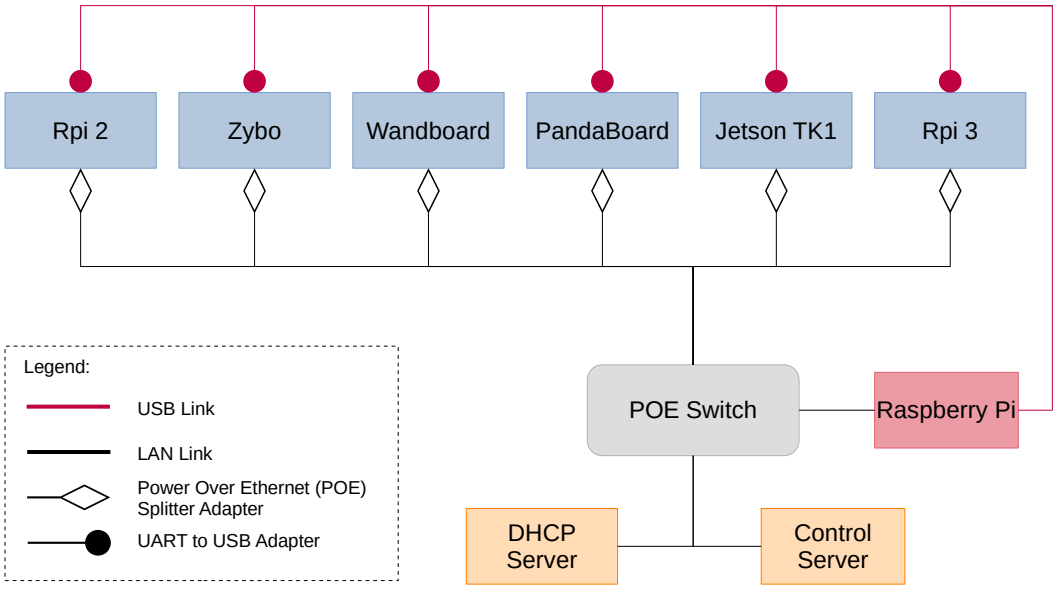

Figure 6.1: Hardware Components of Testbench

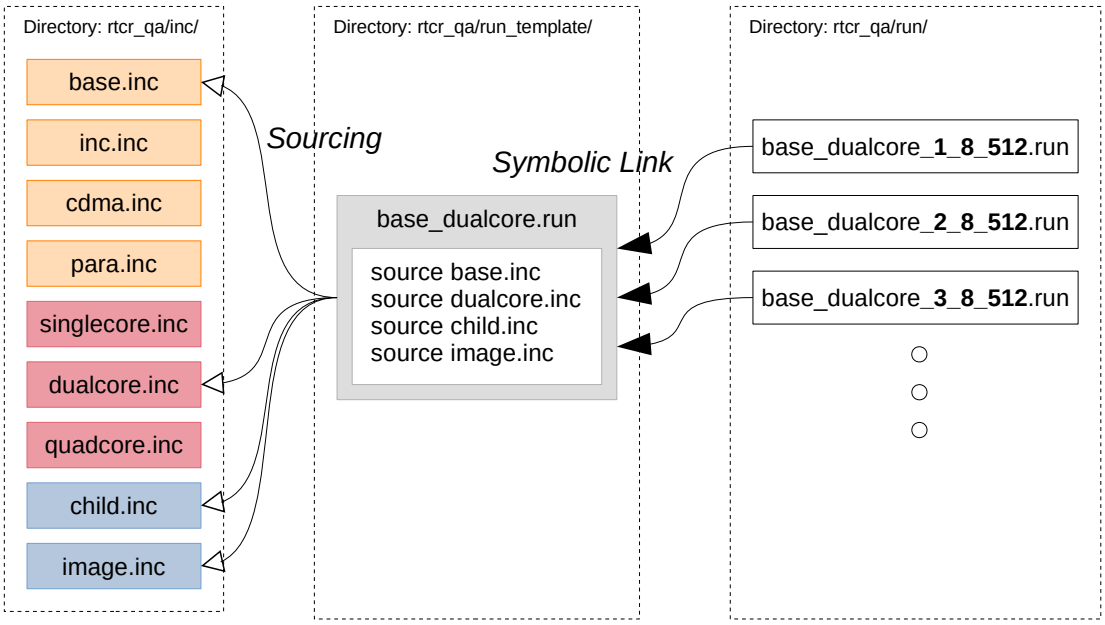

Figure 6.2: Generation and Configuration of Run-Script based Test-Cases

```
cd genode/
tool/autopilot -k sel4 -p zybo -d ./results \
  -r base_dualcore_1_8_5288.run \
  -r base_dualcore_2_8_5288.run \
  -r base_dualcore_3_8_5288.run
```

Listing 6.1: Executing Multiple Test-Cases on the Zybo Platform with `autopilot`

Framework. The scripting logic is principally the same in every test-case. Due to the amount of test-cases, the creation of test-cases bases on symbolic file links and the TCL script command `source`. This is visualized in figure 6.2. Each test-case has a module configuration (**orange**), a configuration based on CPU cores (**red**) and a configuration for the run-time components (**blue**). In order to simplify the building of test-cases, each part of the configuration is stored as a building block in *rtcr_qa/inc/*.

In the next step, a template of a test-case is generated. This means, that building blocks are chosen and merged to a complete TCL script. These files are stored in *rtcr_qa/run_template/* and follow th specific file naming guideline: <*MODULE*>_<*CORE*>*.run*.

In the last step, the actual test-case scripts are generated by combining a template and further configuration parameters. In particular, a symbolic link is created within the directory *rtcr_qa/run* and points to a template file. The name of the test-case is inherited from the template file, but is extended with the actual values of each configuration parameter. This means, that each parameter has a specific position within the name of the test-case file. The file name is handed over during the execution of a TCL script by default. The values are parsed from this string and used to parameterize the Genode OS.

This procedure has following advantages. First, each test-case is cleary identifiable by its name. This implies that each test-case is unique. Second, each test-case shares the same code base. Changes, made in a **.inc* file, affect all test-cases. Third, the generation takes only a few milliseconds, as symbolic links are created and no file content is copied.

6.3 Test-case Execution

The execution of test-cases bases on the testing environment of the Genode Framework. The script *tool/autopilot* automates the execution of multiple **.run* scripts and assigns these to pre-configured target platforms (DUTs). For example in listing 6.1, the autpilot is configured to run three test-cases in combination with the microkernel seL4 on the Zybo board. All logging results are saved to the directory *./results*. Each test-case passes following steps.

1. Compiling & Linking all applications.
2. Packing image of Genode OS.
3. Copying image to the directory of the TFTP server.
4. Open connection to UART port.
5. Powering on target platform. The target acquires the image from the TFTP server and loads it.
6. Waiting for the successful boot message by the kernel. If this fails, the target platform will be reset. The test-case will fail after three boot attempts. This can have several reasons. The bootloader might be misconfigured or a wrong image is not compiled for the target architecture. Also, a wrong configuration of the kernel could lead to this result.
7. Waiting for the message `QA test finished`. This indicates that the test-case successfully passed. After a timeout of 240 seconds, the test-case is also marked as failed. If a message contains the regular expression `[eE]rror`, the test-case will immediately be terminated.
8. Powering off the target platform.

In order to integrate the present hardware in the testing environment, the build configuration of Genode is extended with the following TCL scripts.

tool/run/load/scp Copies the image to an arbitrary host in the network

tool/run/log/ssh Opens a connection to an arbitrary host and interpretes incoming messages as logging from the target machine.

tool/run/power_on/microtik Enables the POE at a specific port of a microtik switch.

tool/run/power_off/microtik Disables the POE at a specific port of a microtik switch.

Finally, the file *tool/builddir/build.conf/run_remote* configures the build target for the corresponding target machine. An example configuration for the board Zybo is shown by listing 6.2. The `tftp_1`, `switch_1` and `pi_1` are aliases of host configurations. Therefore, the actual IP addresses and SSH keys of these hosts are configured in *~/.ssh/config*.

```
# general configuration for all target platforms
RUN_OPT += --include image/uboot \
   --include load/scp --load-scp-machine "tftp_1" \
   --include power_on/microtik --power-on-microtik-machine switch_1 \
   --include power_off/microtik --power-off-microtik-machine switch_1

# copy image to local path
BOARD_RUN_OPT(zybo) += --load-scp-absolute /var/lib/tftpboot/zybo \

# power on/off port with alias 'zybo'
BOARD_RUN_OPT(zybo) += --power-on-microtik-device zybo \
                       --power-off-microtik-device zybo

# connect to UART at /dev/ttyUSB1
BOARD_RUN_OPT(zybo) += --include log/ssh \
                       --log-ssh-machine "pi_1 -t tio /dev/ttyUSB1"
```

Listing 6.2: Configuration of a the Zybo Platform in the Genode Framework

```
# change directory and start python
cd a_measurements && python3

# load all data as panda dataframe
>>> import csv2panda
>>> df = csv2panda.load()

>>> df.columns.values
array(['cpu_session', 'ram_dataspaces', 'ds_count', 'ckpt',
       'serialize_used_caps', 'pause', 'serialized_size', 'board',
       'checkpoint_used_caps', 'bpw', 'rom_session', 'checkpoint',
       'resume', 'parse_used_caps', 'timer_session', 'pd_session',
       'serialize_used_ram', 'worker', 'checkpoint_used_ram', 'kernel',
       'logfile', 'rm_session', 'log_session', 'parse','status','cores',
       'module', 'ds_size', 'granularity', 'parse_used_ram','serialize',
       'genode'], dtype=object)
```

Listing 6.3: List available Columns in Panda Database

```
>>> df[df.module == "base"] # Base module
      [df.kernel == "sel4"] # seL4 microkernel
      [df.board == "zybo"] # Zybo Board
      [df.cores == "singlecore"] # Singlecore
      [df.ds_count == 1][df.ds_size == 1024] # 1 Dataspace with 1 MiB
      [df.status == "success"] # only passed test-cases
      [df.ckpt == 1].serialize # serialization time of 1. Ckpt.
311.0
```

Listing 6.4: Required Time for Marshalling a Checkpoint, if the Child allocates a 1 MiB Dataspace.

6.4 Data Aggregation & Access

The script *rtcr_qa/csv-exporter.py* processes the directory *./results* and extracts all measurement results to a CSV files. These files are accessible in the repository `qa_measurments`. It also includes all scripts for plotting the measurements with the data analysis frameworks *pandas* and *matplotlib* [12] [11]. The script *csv2panda.py* is the start point for loading all results in a *pandas* database. This step is also examplary shown in listing 6.3. The code lists all available columns of the database. Based on this information, further queries are possible. In listing 6.4, the required time for marshalling a checkpoint with a 1 MiB dataspace is queried. In case of a Zybo Board with the sel4 microkernel, it takes 311 milliseconds.

7 Evaluation

7.1 Parameter Set

The results of the evaluation depends on the set of the chosen parameters. These are discussed first.

Modules

The evaluation focuses on the four modules, which have been implemented at the time of the evaluation.

base The base module does not have any extra parameters.

inc The inc module requires that the parameter *granularity* is set. It influences the number of pages, in which each dataspace is split. The size of a single page is calculated by $granularity * 4096$. Thus, a smaller granularity leads to more fine-graned pages and less memory to copy. In this evalation, a *granularity* from 1 to 128 (4 KiB - 512 KiB) is chosen.

para This module copies the memory with parallelized threads. Therefore, the module has a number of worker threads. Each thread is assigned to a CPU core. In order to make single-core and multi-core platforms comparable, the number of worker threads is set to four.

cdma The hardware-based memory copying approach is configured with the *scather-gather* mode. This is slightly faster for large dataspaces.

Child Application

count This parameter defines the number of child applications. The evaluation focuses on checkpointing one application. In general, it is possible to checkpoint multiple child application.

ds_count The child application is configured to allocate a number of dataspaces between 1 and 100.

ds_size For each dataspace, a size of 4 KiB to 10 MiB is chosen.

	Single-Core	Dual-Core		Quad-Core			
Core	1	1	2	1	2	3	4
Child	✓	✓		✓			
Log_session	✓	✓		✓			
Timer_session	✓	✓		✓			
Rm_session	✓	✓		✓			
Cpu_session	✓	✓			✓		
Pd_session	✓	✓				✓	
Memory Copying Threads of Modules							
base	✓		✓				✓
inc	✓		✓				✓
cdma	✓		✓				✓
para[1]	4	2	2	1	1	1	1

Table 7.1: Assignment of Threads to CPU Cores in Test-Cases

bytes_per_write The total size of memory, which is copied, is calculated by $ds\_size * ds\_count$. In order to comapre the parameters *ds_count* and *ds_size* comparable, the number of write accesses on dataspaces depend on *bytes_per_write*. Unless otherwise stated, this parameter is set to 512 KiB. This means, for every 512 KiB of the dataspace, a 32-bit integer is modified randomly. Each modification is distributed randomly on the whole memory of the dataspace.

CPU Core Assignment

The assignment of threads to CPU cores is another aspect of the evaluation. The table 7.1 lists each assignment. The goal of C/R, that the child application is not paused during a checkpoint, has not been reached yet. Whenever, checkpointing threads are executed, the child application is paused by default. Thus, it is valid to assign the child application and checkpointing threads to a single core.

The checkpointing threads for *Log_session*, *Timer_session* and *Rm_session* are sharing the same core, because their execution time are minimal. Previous evaluations showed, that the *Cpu_session* and *Pd_session* takes longer than any other session.

[1] Number of Copying Threads

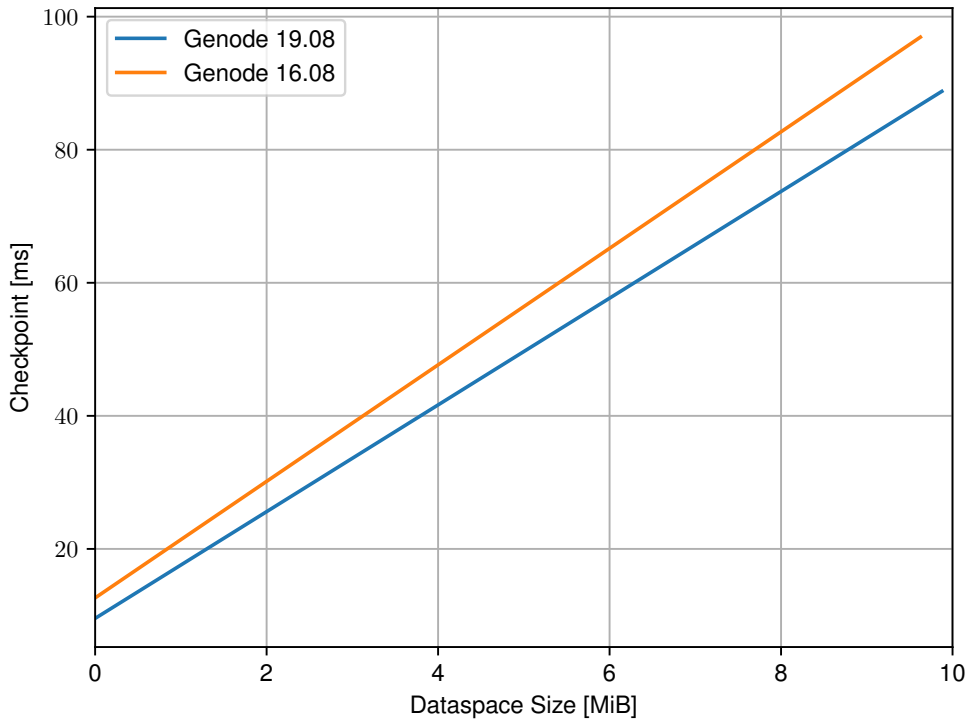

Figure 7.1: Comparison of Genode 16.08 and Genode 19.08 on Fiasco.OC and Zybo

Therefore, each of these is assigned to one core on quad-core platforms. The longest process in RTCRv2 is the memory copying. Due to this priority, the copying threads are assigned to as many resources as possible. In particular, the thread of the modules *base*, *inc* and *cdma* is assigned to an own core on dual- and quad-core platforms. The working threads of the parallelized copying approach (*para*) are distributed equally on all available cores of a platform.

7.2 Genode 16.08 versus Genode 19.08

The improvement from Genode 16.08 to Genode 19.08 regarding checkpoint time is shown in figure 7.1. The measurement bases on the module *base*, Fiasco.OC and the Zybo board. The graphic shows that Genode 19.08 is faster than Genode 16.08. The proportinal share of improvement is shown in figure 7.2. Half of the improvement is caused by a faster memory copying in Genode 19.08. Processes regarding threads are improved as well. Checkpointing the CPU session is 3% faster, which is an indicator for an faster copying of thread registers. Even if pausing a thread has not been improved,

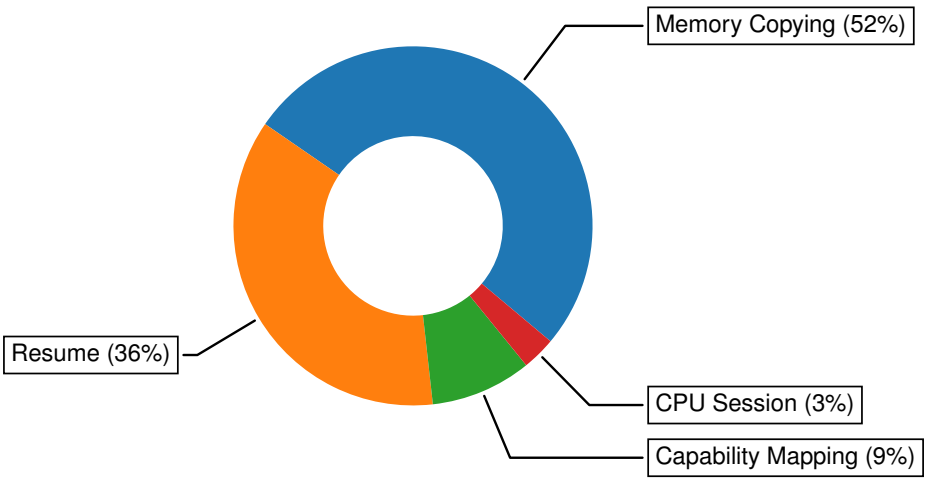

Figure 7.2: Improvements of Checkpointing Routines from Genode 16.08 to Genode 19.08

resuming the paused threads works 36% faster in Genode 19.08. Creating the capability mapping takes around 3 - 5 ms in Genode 16.08. Due to the fact, that this has not been implemented for Genode 19.08 yet, 9% of a checkpoint is capability checkpointing in Genode 16.08. In fact Genode 19.08 is also shipped with a newer Fiasco.OC kernel. Therefore the performance gain could not only be attributed to Genode, but also to the newer kernel.

7.3 Boards

The checkpoint performance of the *base* module is evaluated with six boards on Genode 19.08 and Fiasco.OC regarding increasing dataspace sizes. The results are presented in figure 7.3. The specification of the boards are listed in table C.1 of the appendix. In fact, the slowest board Zybo with a processor frequency of 650 MHz is the slowest in computing a checkpoint as well. The boards Wandboard and Panda base on the same Cortex-A9 core with a processor frequency of 1 GHz, but the Panda board is 25% faster than the Wandboard.

Another unexpected difference can be observed in the architectures ARMv8 (64-bit) and ARMv7 (32-bit). The Raspberry Pi 2 with an ARMv7-A architecture is faster than the Raspberry Pi 3 with an ARMv8. In fact, the 64-bit support has been new introduced since Genode 19.05 [15]. The Jetson TK1 by Nvidia is the fastest board in the set of boards.

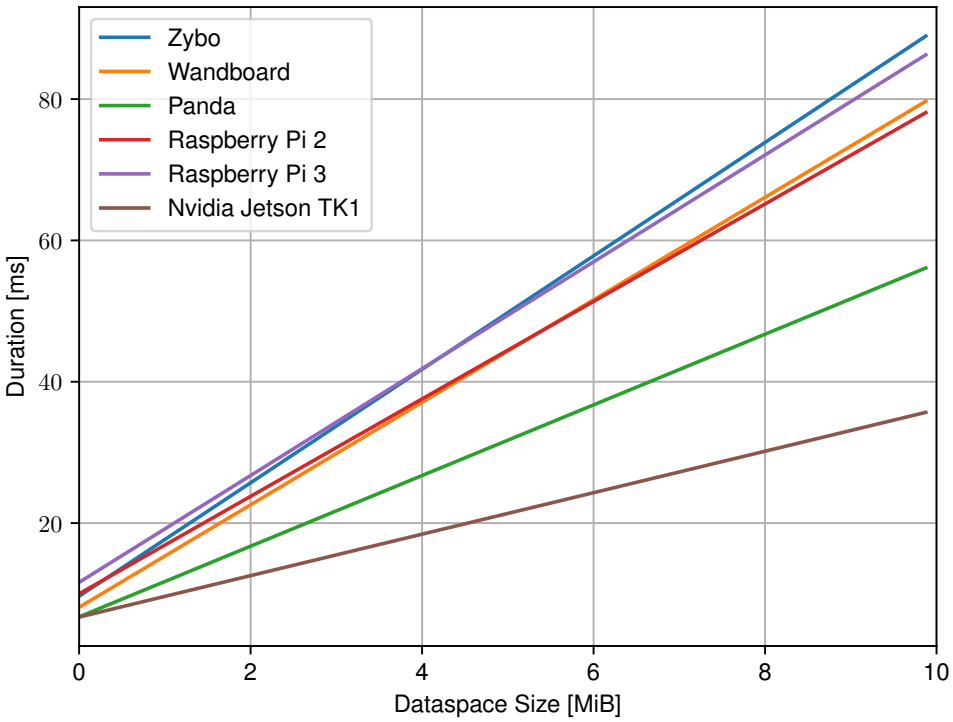

Figure 7.3: Comparison of Boards (Single-Core Configuration) under Fiasco.OC and Genode 19.08

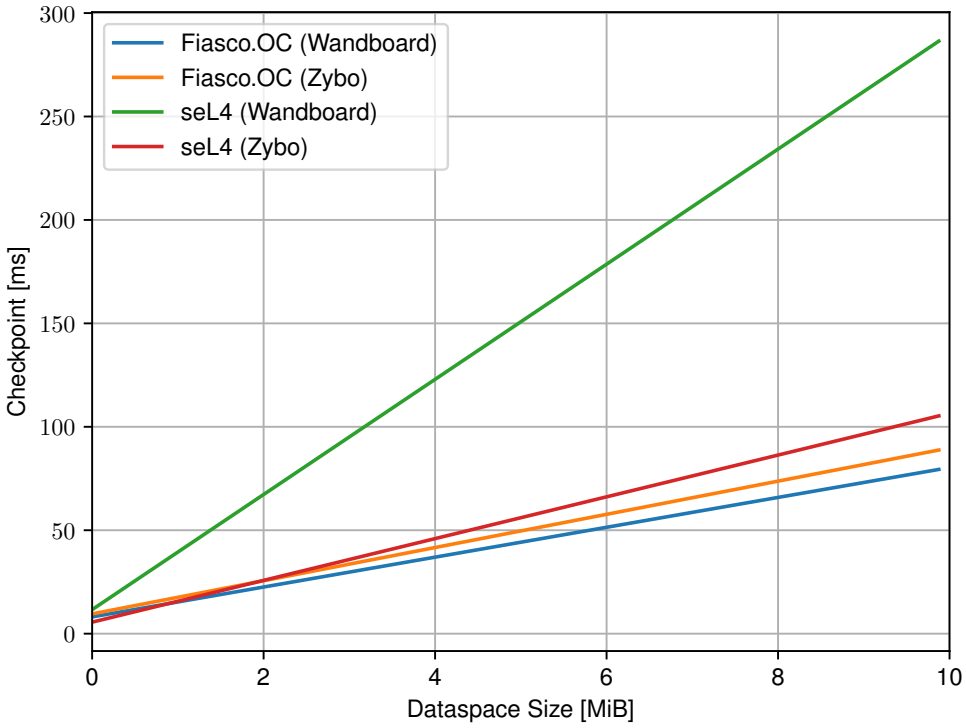

Figure 7.4: Comparison of Microkernel seL4 and Fiasco.OC on Genode 19.08

7.4 Kernel seL4 versus Fiasco.OC

In figure 7.4, the microkernels Fiasco.OC and seL4 are compared on the platforms Wandboard and Zybo. The checkpoint time of the combination Zybo board and seL4 kernel has a constant overhead of 500 ms. This bias is eliminated in the plot. It is assumed, that the clock driver of the platform is initialized at the first measurement, which could lead to this bias. Based on this assumption, the Fiasco.OC kernel is faster than the seL4. The Fiasco.OC is faster on the Wandboard, but the seL4 kernel is faster on the Zybo platform. This indicates that the platform-dependent implementation in the kernels has a big impact on the actual performance of RTCRv2.

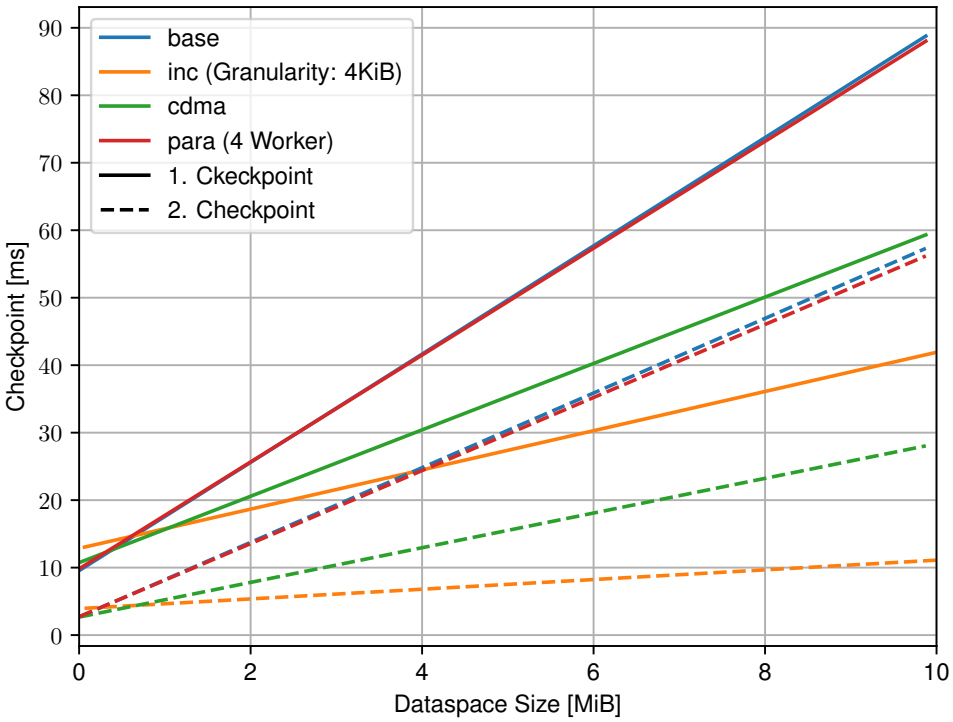

Figure 7.5: Modules in Comparison regarding rising Dataspace Sizes on Zybo & Fiasco.OC (Single-Core Configuration)

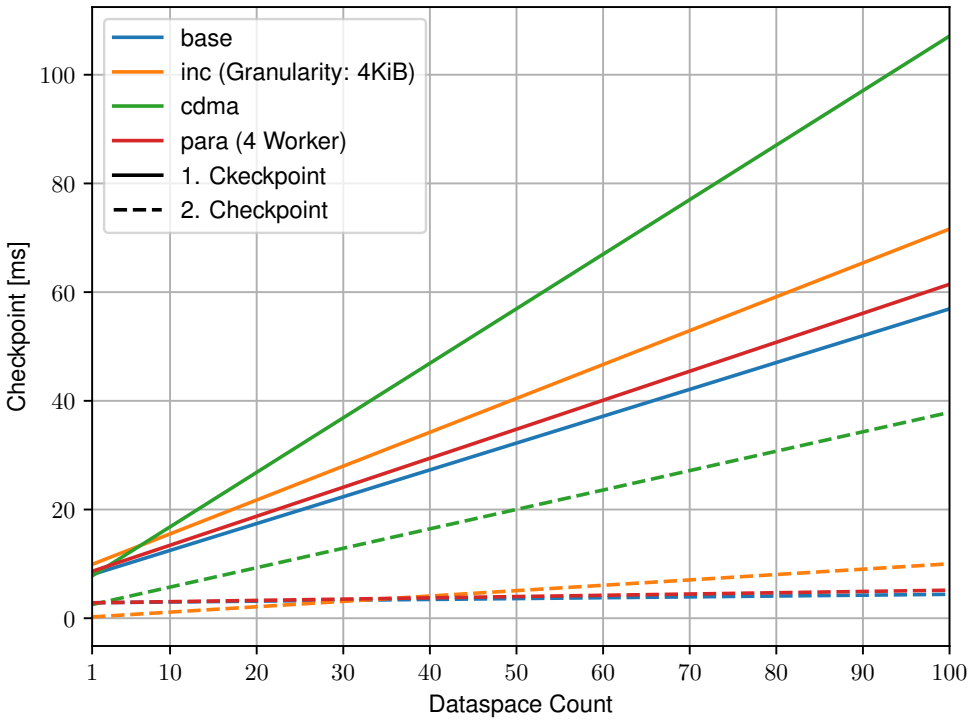

Figure 7.6: Modules in Comparison regarding rising Count of 4 KiB Dataspaces on Zybo & Fiasco.OC (Single-Core Configuration)

7.5 Modules

7.5.1 Dataspace Size

The checkpoint performance of the modules regarding a rising dataspace size is shown in figure 7.5. In general, the first checkpoint lasts longer, because the dataspace of the *Cold Storage* is allocated in the first checkpoint. For example, the first checkpoint of the *base* module lasts 1,5-times longer than the second checkpoint.

The first checkpoint of the *inc* module is faster than the first checkpoint of the *base* module. This can be explained with the fact, that the *inc* module must allocate the dataspaces of the *Cold Storage* simultaneously with the *Hot Storage*, while the child application is running. Therefore, no time for the memory allocations is required during the checkpoint. The performance of the *Incremental* approach is the best, because it only copies modified memory regions.

The plot of the *cdma* module confirms the measurements results of Fischer's work once again [3]. The performance gain will be around 30%, if the copying is outsourced to the FPGA of the Zybo board.

The *para* module is configured with four copying threads. Due to the Single-Core configuration of the test-cases, all threads are executed on the first core of the Zybo board. Therefore, it is interesting that the *para* module slightly performs better than the *base* module in both checkpoints. It would mean that the memory copying does not only depend on the core usage or the throughput of the IO bus, but also on some other facts. Finally the deviation is only one millisecond for a memory range of 1 MiB, which makes a measurement inaccuracy more likely.

7.5.2 Dataspace Count

The checkpoint time does not only depend on the size of a dataspace, but also on the number of allocated dataspaces. Therefore, figure 7.6 shows the checkpoint time of each module in relation to the number of 4 KiB dataspaces. The first and second checkpoint of the modules *base*, *inc* and *para* performs similar under a rising count of dataspaces. Allocating a lot of small dataspace needs more time than allocating the same amount of memory with a single dataspace. For example, the *base* module only requires 10 milliseconds for the first checkpoint with a 400 KiB dataspace. But the first checkpoint with 100 dataspaces à 4 KiB lasts six times longer. After the first checkpoint, the modules perform like the corresponding measurements with only a single dataspace.

The module *cdma* is an exception. It requires much more time in both checkpoints in order to copy the content of the dataspaces. This is not unusual, because the CDMA core of the FPGA is programmed for each dataspace again by the driver. This programming leads to the overhead. Therefore, the *cdma* module has the worst performance of all modules regarding a large number of small dataspaces.

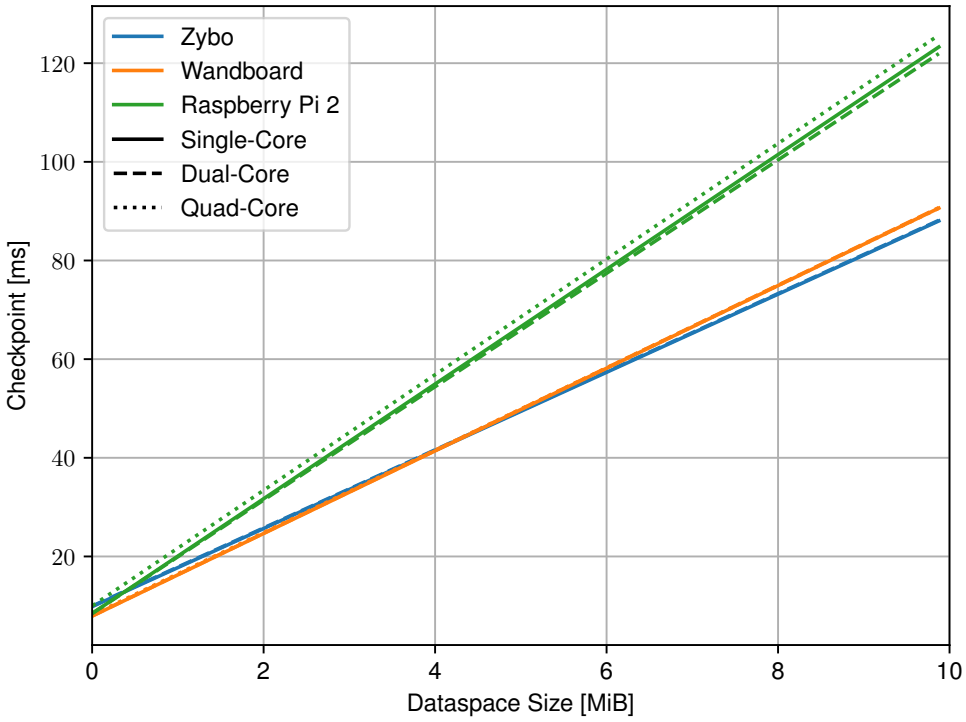

Figure 7.7: Parallelization of Memory Copying with four Threads on Fiasco.OC

7.6 Parallelization of Memory Copying

Regarding the parallelized memory copying approach, there is one more question. It has no been evaluated yet, whether the bottleneck of the copying process depends on a fully occupied I/O bus or is rather limited by the maximum workload of a CPU core. In order to answer the question, the *para* module is configured with four copying threads and evaluated on the Zybo board, Wandboard and Raspbery Pi 2. Figure 7.7 shows the required time for a checkpoint regarding a rising dataspace size on Fiasco.OC. In case of the Zybo and Wandboard, the plots for the single-, dual- and quad-core configuration are overlapping. No significant improvement regarding checkpoint time can be found, which implies that the I/O connection is the bottleneck.

The plots of the Raspberry Pi 2 differ from this observation. The dual-core configuration is slightly faster, which is an indicator that the I/O has not been fully occupied yet. However, the quad-core configuration is slightly slower than the single-core setting.

Based on the assumption that this is not a measuring inaccuracy, running the same amount of threads on two cores is faster than the execution on four cores.

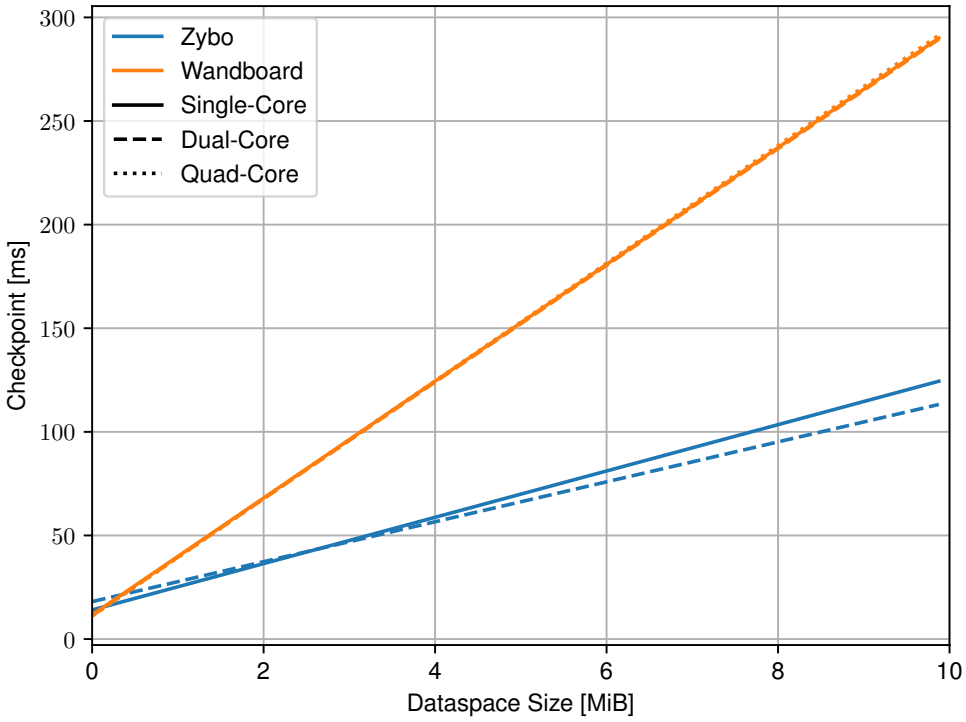

Figure 7.8: Parallelization of Memory Copying with four Threads on seL4

The same test-cases have been executed with the seL4 kernel. The results are shown in figure 7.8. The performance of the seL4 kernel is 30% slower in comparison with the Fiasco.OC on the Zybo board. For dataspace sizes larger than 3 MiB, the dual-core configuration performs better than the single-core configuration. The seL4 kernel performs three times slower on the Wandboard. A similar situation has already been observed with the *base* module in figure 7.4.

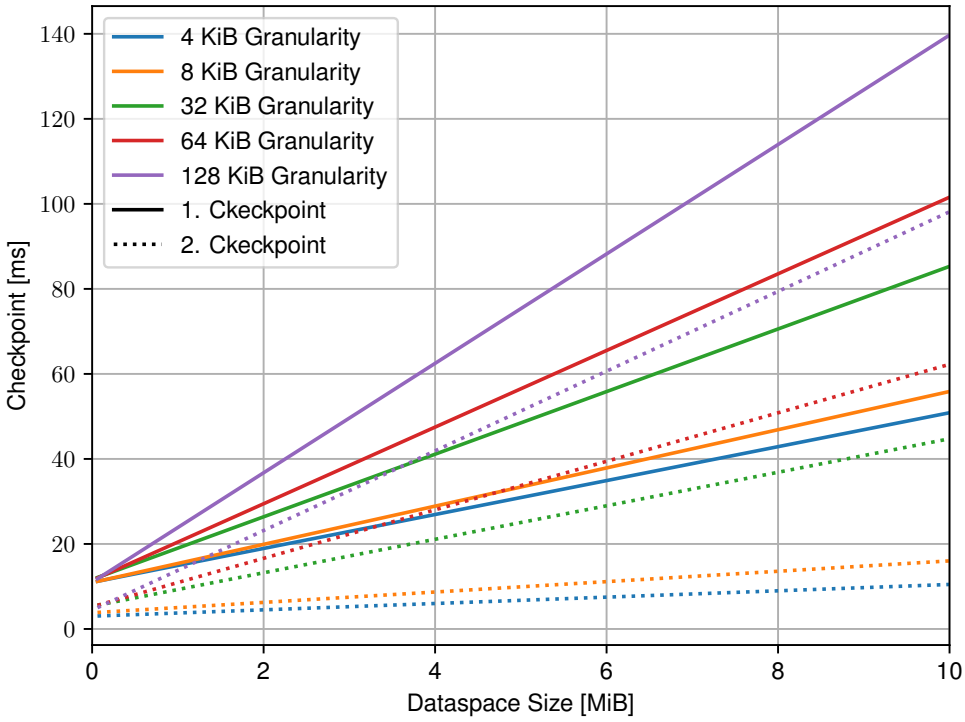

Figure 7.9: Incremental Checkpoints with Variable Page Sizes (Granularity) on Fiasco.OC and Wandboard.

7.7 Incremental Checkpoint

The incremental checkpoint is implemented as the module *inc* and depends on the attribute *granularity*. The *granularity* defines the number of pages, in which a dataspace is split. Each page is checkpointed independently. Thus, a smaller granularity means less unmodified memory, which is copied during a checkpoint. This fact is evaluated in figure 7.9, where the *bytes_per_write* is fixed and one dataspace is used. It can be seen, that the performance of the first and second checkpoint improves with smaller pages. Consequently, the smallest possible granularity of 4 KiB pages is the optimum.

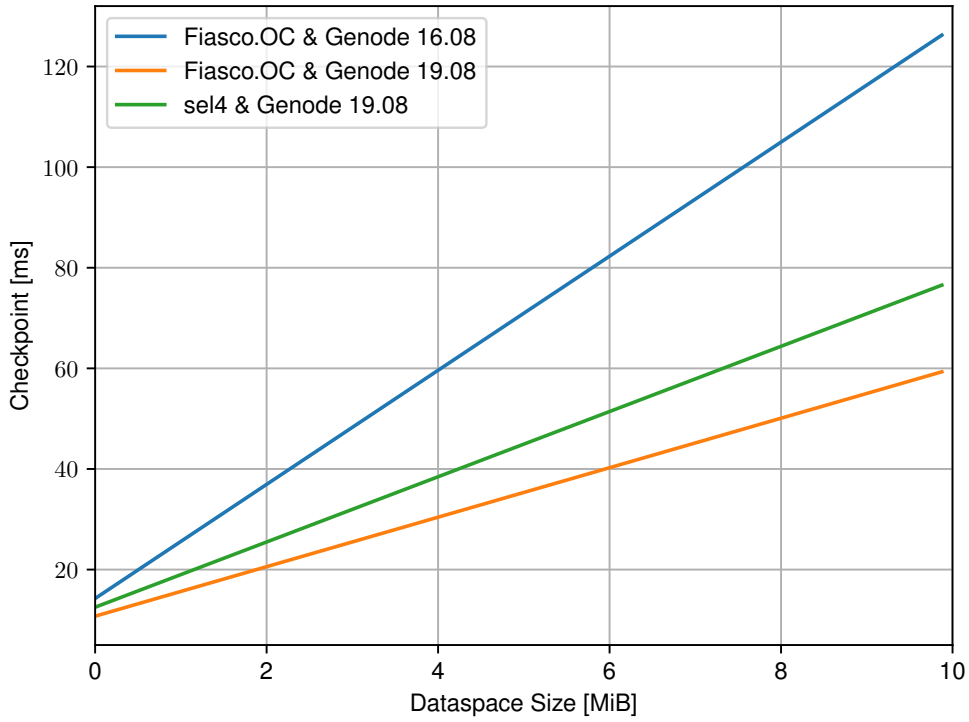

Figure 7.10: Comparison of seL4 and Fiasco.OC in Genode 19.08 regarding Hardware-based Acceleration

7.8 Hardware-based Acceleration

In figure 7.10, the hardware-based memory copying acceleration with the module *cdma* is plotted. The figure compares the required checkpointing time in relation to a increasing dataspace size. Due to the fact, that the copying is outsourced to a FPGA, it is assumed that the OS-based overhead is insignificant. But this is not the case for the microkernels Fiasco.OC and seL4. The Fiasco.OC is constant faster than seL4. In spite of that fact, it can not be analyzed, whether the overhead results from the driver implementation or the memory-mapped based communication between driver and FPGA.

By comparing the performance of Genode 16.08 and Genode 19.08, it is shown that Genode 19.08 is more than twice as fast as Genode 16.08. On the one hand, this might results from the newer version of Fiasco.OC, which is used by Genode 19.08. On the other hand, optimizations within the Genode Framework might contribute to this improvement.

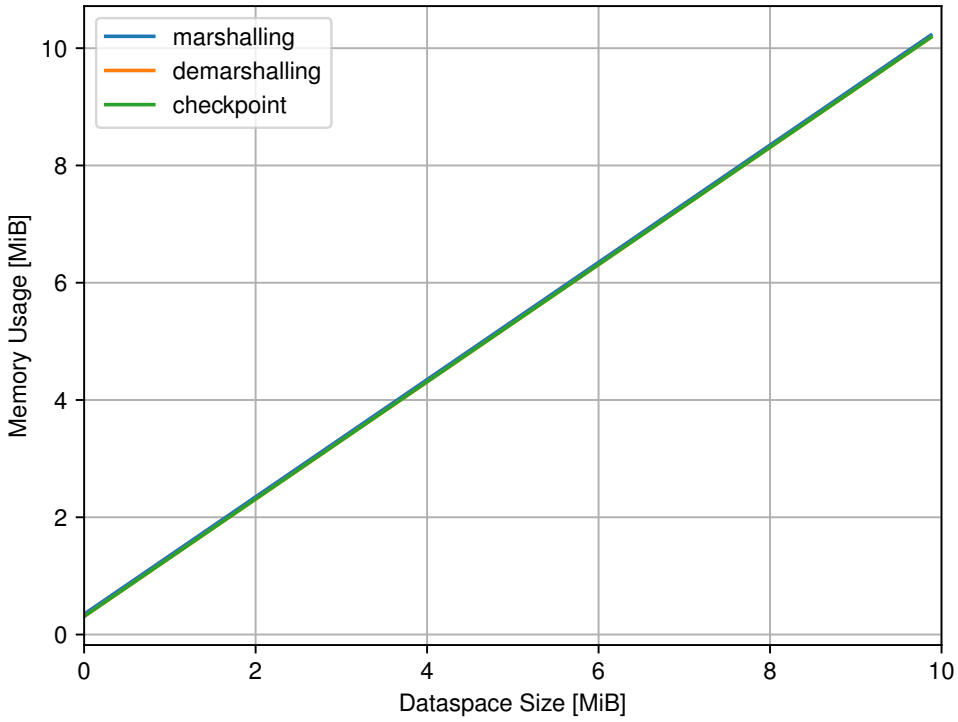

Figure 7.11: Memory Usage of (De-)Marshalling a Checkpoint

7.9 Checkpoint depending Resources & Sizes

7.9.1 Memory

Another important aspect of the RTCRv2 is the required memory for checkpointing a child application. Not only, that the child consumes memory (*Hot Storage*), but also the *Cold Storage* and serialized dataspaces exhaust the resources of the memory. Figure 7.11 puts the size of an allocated dataspace by the child application in relation to the allocated memory for checkpointing, marshalling and demarshalling. It can be stated, that each allocation is directly proportional to the memory allocation of the child application. Therefore a checkpointer application has to reserve the double amount of memory of the child application in order to sucessfully process a checkpoint. A half of the reserved memory for the *Cold Storage* and the other half for the temporary marshalled/demarshalled dataspace.

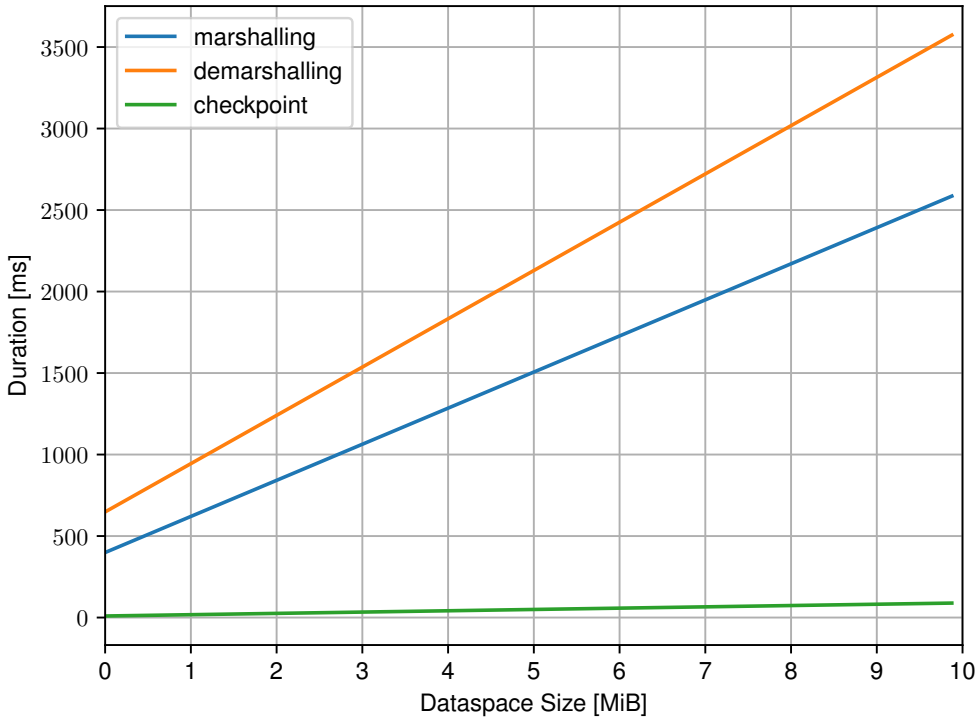

Figure 7.12: Duration of (De-)Marshalling a Checkpoint

7.9.2 Checkpoint Frequency

The dependency between marshalling time and checkpoint frequency is introduced with the *Hot & Cold Storage*. This relation is analyzed in subsection 4.4.3 Checkpoint Frequency. The figure 7.12 plots the time consumption of checkpointing (T_c), marshalling (T_r) and demarshalling in relation to the allocated dataspace size of a child application. It can be seen that the marshalling process lasts up to 10 times longer than the checkpointing. The overhead between marshalling and demarshalling can be explained with the fact, that the dataspaces of the *Cold Storage* are re-allocated during the demarshalling. The minimal time between two checkpoints can be calculated by adding T_c and T_r. For example, a child application with a dataspace size of 5 MiB can be checkpointed every $T_r + T_c = 1500ms + 50ms = 1,55$ second.

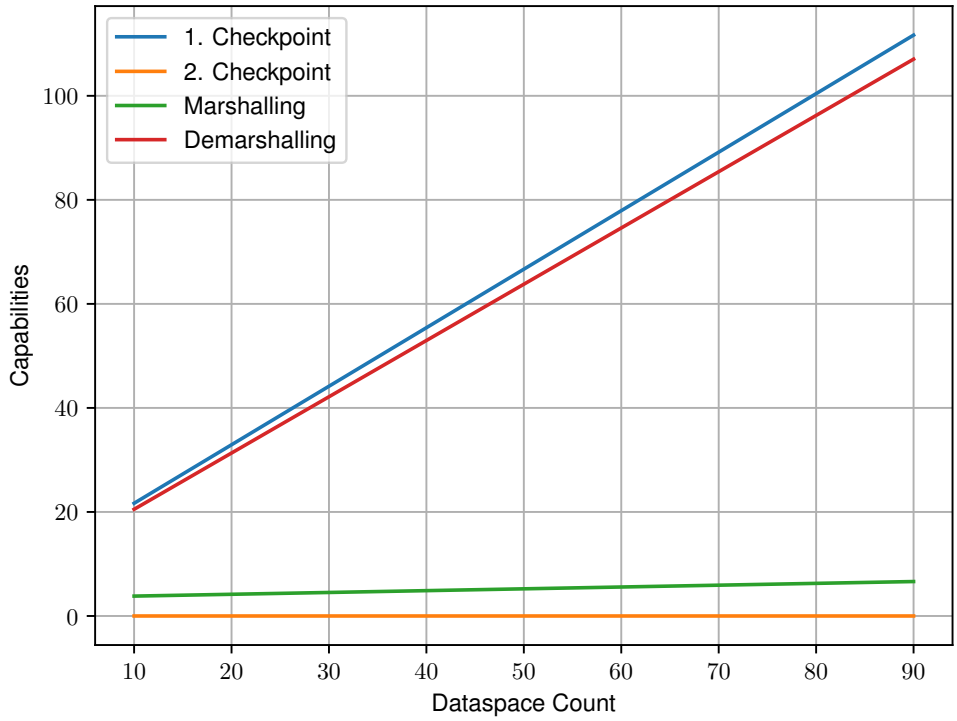

Figure 7.13: Capability Usage of (De-) Marshalling & Checkpoints with *base* Module

7.9.3 Capabilities

The number of required capabilities mostly depends on created components, threads, signal contexts, sessions and dataspaces. This evaluation focuses on capabilities, which are necessary to checkpoint a child application with multiple dataspaces. Figure 7.13 shows the required capabilities in relation to the count of dataspaces allocated by the child application. The relation is similar for all modules, so the module *base* is exemplary chosen.

During the first checkpoint, the dataspaces of the *Cold Storage* are initialized. Thus the *base* module requires one capability for each checkpointed dataspace. In the second checkpoint, all dataspaces are allocated and no further capabilities are requested. Marshalling the checkpoint costs eight capabilities. But in contrast to that, demarshalling the *Cold Storage* costs as much capabilities as dataspaces are in the *Cold Storage*. It can be stated that each restored dataspace costs one capability.

In conclusion it should be mentioned, that a restored *Cold Storage* will not require further capabilities during the first checkpoint, because all dataspaces within the *Cold Storage* have already been allocated. The total count of capabilities, which are available

to the checkpointer component, is limited by the factor a half. For each dataspace capability of the *Hot Storage*, a corresponding capability is required for the *Cold Storage*.

7.10 Time Consumption of Checkpointing Threads

So far only the time consumptions of a whole checkpoint has been evaluated. This section presents the time measurements of the most significant program routines within a checkpoint. Figure 7.14 shows the worst-case execution time of the functions `pause`, `resume` and the threads, proportionally to the total duration of the first checkpoint. In this case, the *base* module checkpoints a child application, which allocates one 4 KiB dataspace. Due to the small dataspace size, the checkpointing time of the memory copying thread is relativly short.

In contrast to that, pausing and resuming the child application occupy the checkpointing time for more than half of the time. The *cpu_session* and *pd_session* threads show some load during the first checkpoint. In this time, the *Offline Storage* for the child-related threads is initialized.
A slightly difference will be recognizable, if the first checkpoint is compared to the second checkpoint (figure 7.15). On the one hand, the copying thread requires less time due to the already allocated dataspaces. On the other hand, the *cpu_session* and *pd_session* threads take less time. Since the previous checkpoint, no further threads are created and no *Offline Storage* must be prepared.

7.11 Failure Analysis

Not only the performance, but also the number of system failures, which raise during a checkpoint, is important in order to evaluate the stability of an approach. The full list of passed and failed test-cases is in appendix E.1. All errors, which occured due to an instable UART link, are excluded from this list.

If an *Invalid Signal Context* is raised, a signal can not be handled and the execution of the test-case will be terminated. If the system does not respond, a *Timeout* will stop the execution after four minutes. Both are critical failures and should not occure during the normal run-time of the Genode OS. Luckily most test-cases have been successfully passed and the failure rate is lower than 1% for the most platforms. A few combinations of board, kernel and module seem to be more error-prone. These are visualized in figure 7.16.

The *inc* module has the most errors with 2% of invalid signal contexts and 3% of timeouts on the Fiasco.OC & Zybo board. But also all other modules failed more often with the Fiasco.OC than with the seL4 kernel.

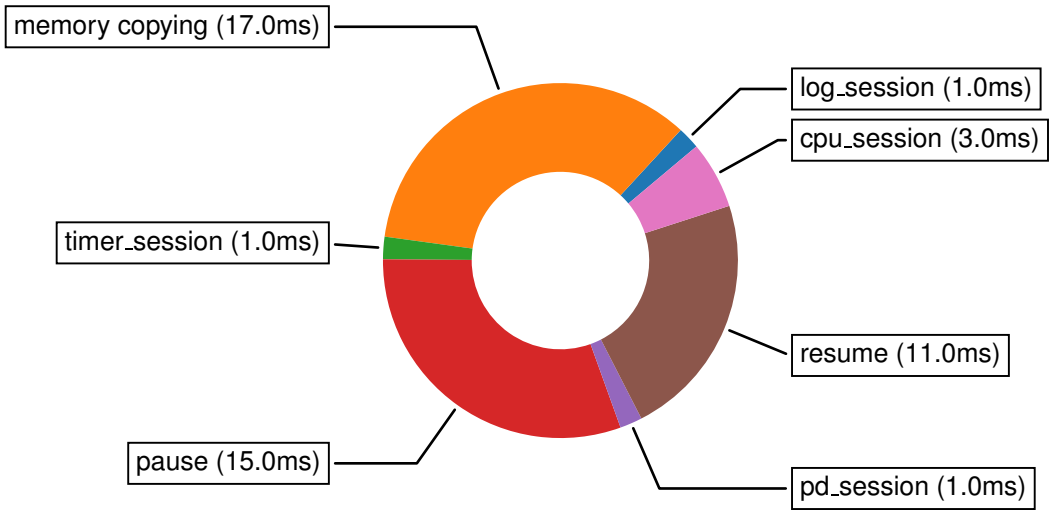

Figure 7.14: Worst-Case Time Consumption of Checkpointing Threads during the first Checkpoint with one 4 KiB Dataspace

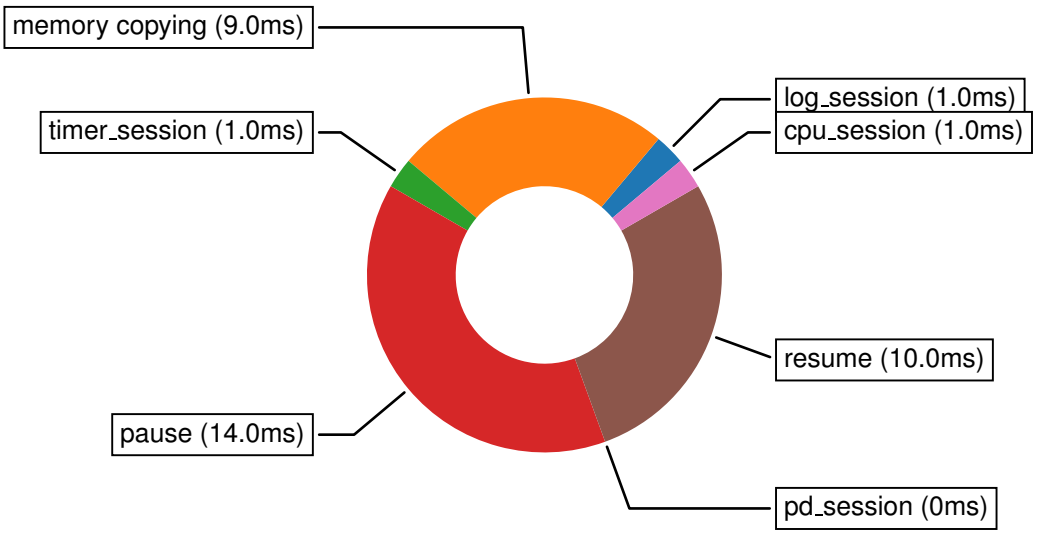

Figure 7.15: Worst-Case Time Consumption of Checkpointing Threads during the second Checkpoint with one 4 KiB Dataspace

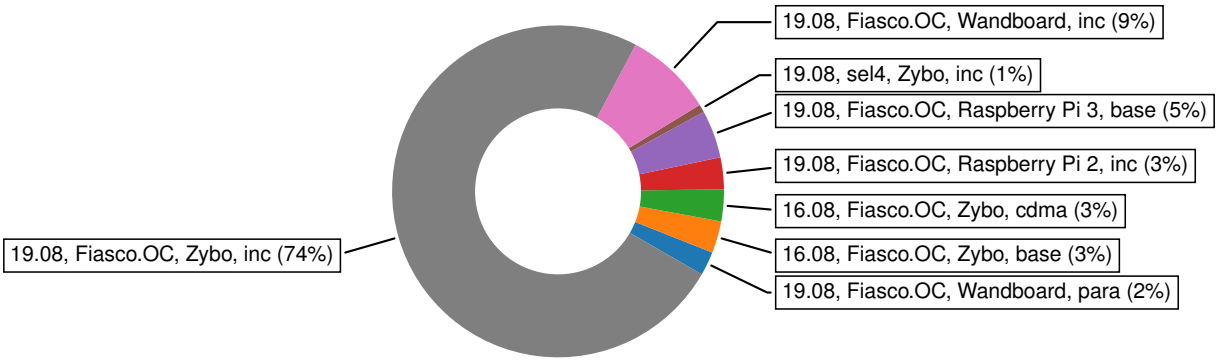

Figure 7.16: Proportional Failures per Combination-Set of Genode Version, Kernel, Board & Module

8 Limitations & Future Work

Based on the results of the evaluation, this chapter describes the limitations of the RTCRv2. Being aware of what is still not ported to RTCRv2 and Genode 19.08, a roadmap for future work is elaborated.

8.1 Checkpointing Process

Contrary to the integration of the seL4 kernel in RTCRv2, the integration of Fiasco.OC requires to checkpoint the list of kernel capabilities. This part of the checkpoint depends on modification within the PD ssession of Genode. That is so far only implemented for Genode 16.08. Due to this required modification, the aim was to drop the support for Fiasco.OC in favor of seL4. But the evaluation revealed that the Fiasco.OC is superior regarding performance. Thus, it is worth considering an ongoing support for Fiasco.OC and an integration of the missing `Capability_mapping` implementation in Genode 19.08.

Starting with Genode 18.02, the child application did not sucessfully start on Fiasco.OC anymore. Weidinger identified the problem and provided a patch. This also includes a new `Native_cpu_component`, which did not exist in RTCRv1. It is not evaluated yet, whether this native cpu component must be checkpointed, too. The current checkpoint does not include its state.

8.2 Restoring Process

The restoring process of RTCRv1 is not fully working yet. According to the thesis of Joos, not all kernel capabilities are completly restored [10]. It leaves the child in a non respondable state after a restore happened. His work focuses on the Fiasco.OC kernel, but Weidinger also suggests a closer look to the seL4 microkernel in this regard [20]. He also noticed an inconsistency between bootstrapper code and child restoring concerning attached region maps and the Rom session. Rimmel worked out several logic flaws and problems within the restoring process [16].

Due to this reasons, the restoring process is prepared but is not yet implemented in RTCRv2. It is suggested to implement a minimal working prototype for the restoring process at first and then integrate the solution in RTCRv2.

8.3 Evaluation & Integration of Further Modules

Redundant Memory Approach

The redundant memory approach implemented by Stark is limited to a subset of registers. In general, the child application is massivly slowed done by the software-based emulation of instructions. Another performance loss arises through the limited possibilities to distinguish between read- and write- accesses. According to Stark, there is the chance that this feature is implemented in Genode 19.08. The evaluation and re-implementation of this approach goes beyond the scope of this thesis, but worth a look in future [19].

Hardware-based Approaches

Tsareva implemented a co-processor based memory copying approach similar to the module *cdma*. It is successfully integrated as module *mbox*, but not yet evaluated in the context of RTCRv2 [7].

Meanwhile Tsareva ported the hardware-based memory tracing approach by Bachmaier as the module *memtrace*. This is not yet evaluated, too. Especially the functionality of *memtrace* is interesting for a mixed software-hardware based implementation of the redundant memory approach [1] [8].

Copy-On-Write Approach

Werner's work extends the incremental approach with a copying process, which is active while the child is running. Due to an issue with the threading of the fault handler, the approach could not be fully evaluated. With RTCRv2, the newer Genode 19.08 and a second microkernel is supported. Therefore it would be the next step to investigate, whether the problem is resolved with Genode 19.08 or seL4. Based on that, the copy-on-write mechanism can be integrated in RTCRv2 and evaluated [22].

8.4 Marshalling

The marshalling/demarshalling implementation proves, that the new introduced *Hot & Cold Storage* can be accessed and processed similar to the previous *Online & Offline Storage* concept. In this context, two further improvements are conceivable. First, the evaluation showed that the compression process lasts up to 1 second for a dataspace size of 1 MiB. This work can easily be assigned to multiple cores or even outsourced to a hardware-based implementation. Also the point on which compressing and transfering the data lasts longer than the transfer of the uncompressed data is of interest. Finally, the idea of the incremental approach can be applied to the marshalling.

Instead of the whole dataspaces, only the delta of modified memory is serialized. This would decrease the size of the transfered data and improves the performance of the marshalling process.

8.5 CPU Core-based Pausing & Resuming

The evaluation showed that pausing and resuming the threads of the child application are a major part of the checkpoint regarding time consumption. The order, in which the threads are paused and resumed, is not investigated yet. What happens to the information of an IPC call, which can not be delivered to an paused thread? On the way to the long term goal, which is a checkpoint without pausing a child application, the next step is to improve this part. It is possible to execute the child application on a separate CPU core in RTCRv2. If the execution pipeline of the core is paused, it would not be necessary to interrupt each single thread of the child application. That also has the advantage, that the threads are not paused/resumed in sequence. Possible inconsistencies in the communication between the threads would be prevented.

8.6 Testbench & Hardware-based Measurements

For this thesis, a testbench for the evaluation of RTCRv2 is introduced. It proved to work reliable with multiple devices, architectures, kernels and modules. In order to produce results with this quality in future theses, a similar testbench concept would be necessary. However, this is usually out of scope in a thesis. Therefore, it would be reasonable to establish a testbench for a long-term use and provide the capabilities to future theses. Additionaly, the stability of recording the results can be improved by extending the testbench with a network database, as it is suggested in section 4.7 Performance Measurements. Finally, the evaluation indicates that the software-based timer components sometimes lead to biases in the measurements. Also most timers reach their limits with a measuring resolution in milliseconds. Due to these limitations, hardware-based measurement methods should be part of future investigations.

9 Conlusion

This section concludes the thesis by giving a summary of the work and by highlighting its contribution to the C/R project.

The main goal of this thesis is the evaluation and implementation of a modularisation concept in the existing C/R project. This has been achieved by analysing the software architecture and the run-time dependencies between program routines. Based on these findings, most run-time dependencies could be resolved and the previous *Offline & Online Storage* has been replaced with a simpler concept, called *Hot & Cold Storage*. The implementation of a serialization library showd, that the newly introduced storage can also be serialized and compressed sucessfully.

Finally, the modularisation has facilitated the summarizing of all existing C/R implementations to one code base. This can be extended to support multiple target platforms. Thus, it is possible to compile the project not only for the microkernels Fiasco.OC and seL4, but also for the 32-bit and 64-bit architectures.

This thesis accomplished the porting to the newly released Genode 19.08. Therefore the project can now benefit from the support of the Genode community again.

The second part of this thesis is an extensive evaluation regarding performance on multiple platforms, Genode versions, microkernels and C/R-related optimizations. In the course of this, a significant performance loss switching from 32-bit to 64-bit architecture could be observed within Genode.

Moreover, the superiority of the Fiasco.OC over the seL4 microkernel regarding performance could be revealed. Based on the evaluation, two more task areas beside the restoring process have been opened up. On the one hand, the marshalling process can be optimized with only compressing deltas of dataspaces as well as with hardware-based compression. On the other hand, the thread-based pausing and resuming of the target application should be optimized.

In conclusion, the work of this thesis brought the C/R project a major step towards an efficient migration software within the real-time domain.

List of Figures

List of Tables

Bibliography

[1] Sebastian Bachmaier. "Optimization of a real-time capable Checkpoint/Restore mechanism for L4 Fiasco.OC/Genode by hardware-assisted memory tracing and copying." MA thesis. Technical University of Munich, Jan. 2019.

[2] Sebastian Eckl, Daniel Krefft, and Uwe Baumgarten. *KIA4SM - Cooperative Integration Architecture for Future Smart Mobility Solutions*. Apr. 2015.

[3] Johannes Fischer. "Optimization of a real-time capable Checkpoint/Restore Mechanism by Hardware Assistance." Interdisciplinary Project. Technical University of Munich, July 2019.

[4] *Genode Operating System Framework Foundation*. URL: https://genode.org/documentation/genode-foundations-19-05.pdf (visited on 01/13/2020).

[5] *Github: Function Profiler based on Serial Log Output for Genode*. URL: https://github.com/pecheur/genode-Profiler (visited on 01/01/2020).

[6] *Github: Protocol Buffers - Google's data interchange format*. URL: https://github.com/protocolbuffers/protobuf (visited on 01/01/2020).

[7] *Gitlab rtcr$_w$orkspace/rtcr$_m$box*. URL: https://gitlab.lrz.de/rtcr_workspace/rtcr_mbox (visited on 01/12/2020).

[8] *Gitlab rtcr$_w$orkspace/rtcr$_m$emtrace*. URL: https://gitlab.lrz.de/rtcr_workspace/rtcr_red-hw (visited on 01/12/2020).

[9] Denis Huber. "Design and Development of real-time capable Checkpoint/Restore Mechanisms for L4 Fiasco.OC/Genode." MA thesis. Technical University of Munich, Dec. 2016.

[10] Lukas Joos. "Extending L4 Fiasco.OC/Genode OS by Capability Checkpoint/Restore." BA thesis. Technical University of Munich, July 2018.

[11] *Matplotlib: Python plotting*. URL: https://matplotlib.org/ (visited on 01/02/2020).

[12] *pandas - Python Data Analysis Library*. URL: https://pandas.pydata.org/ (visited on 01/02/2020).

[13] Alexander Reisner. "Finalization of an Existing Migration Execution Process and Corresponding Performance Evaluation with Regard to Real-Time Behaviour." MA thesis. Technical University of Munich, Feb. 2018.

[14] *Release notes for the Genode OS Framework 17.05*. URL: https://genode.org/documentation/release-notes/17.05 (visited on 01/03/2020).

[15] *Release notes for the Genode OS Framework 19.05*. URL: https://genode.org/documentation/release-notes/19.05 (visited on 01/03/2020).

[16] Andreas Rimmel. "Extending L4 Fiasco.OC/Genode OS by Multi-Core Supported Capability Checkpoint / Restore." BA thesis. Technical University of Munich, Aug. 2019.

[17] Andreas Schön. "Extension of L4 Fiasco.OC / Genode by a Multicore-based Checkpoint / Restore Mechanism." BA thesis. Technical University of Munich, Apr. 2019.

[18] Quirin Schweigert. "Development of a Multicore-Supported Incremental Checkpoint/Restore Mechanism for L4 Fiasco.OC/Genode." BA thesis. Technical University of Munich, Sept. 2017.

[19] Josef Stark. "Development of a Redundant Memory Based Checkpoint/Restore Mechanism for L4 Fiasco.OC/Genode." MA thesis. Technical University of Munich, Apr. 2018.

[20] Alexander Weidinger. "Evaluation of the Microkernels L4 Fiasco.OC and seL4 using the Example of a Checkpoint/Restore Component to be ported." MA thesis. Technical University of Munich, May 2019.

[21] David Werner. "Development and Comparison of two different Checkpoint/Restore Mechanisms for L4 Fiasco.OC/Genode." Interdisciplinary Project. Technical University of Munich, Mar. 2019.

[22] David Werner. "Performance Optimization of Shared Memory and Redundant Memory based Checkpoint/Restore Mechanisms for L4 Fiasco.OC/Genode." MA thesis. Technical University of Munich, Apr. 2019.

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

A Sequence Diagrams of Checkpoint

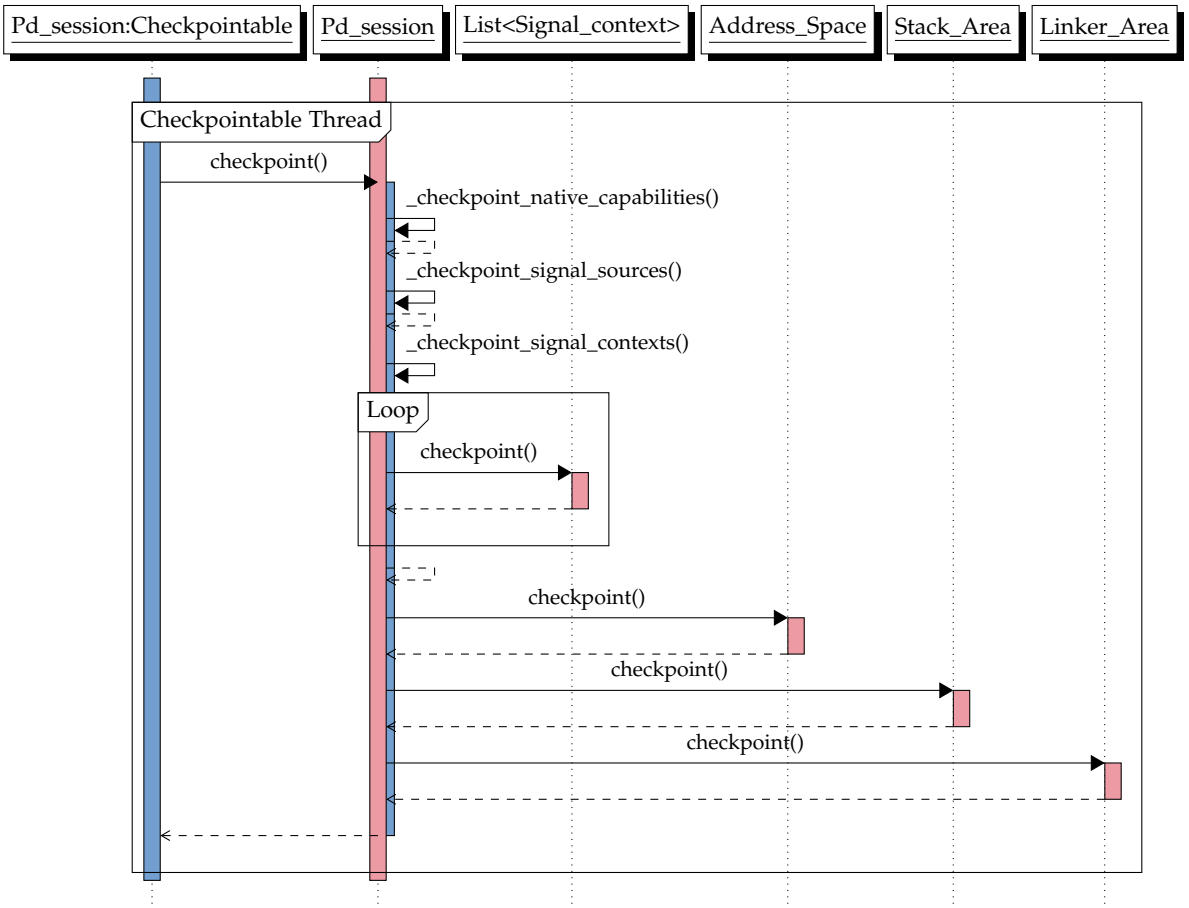

Figure A.1: Checkpointing PD Session in Genode 16.08

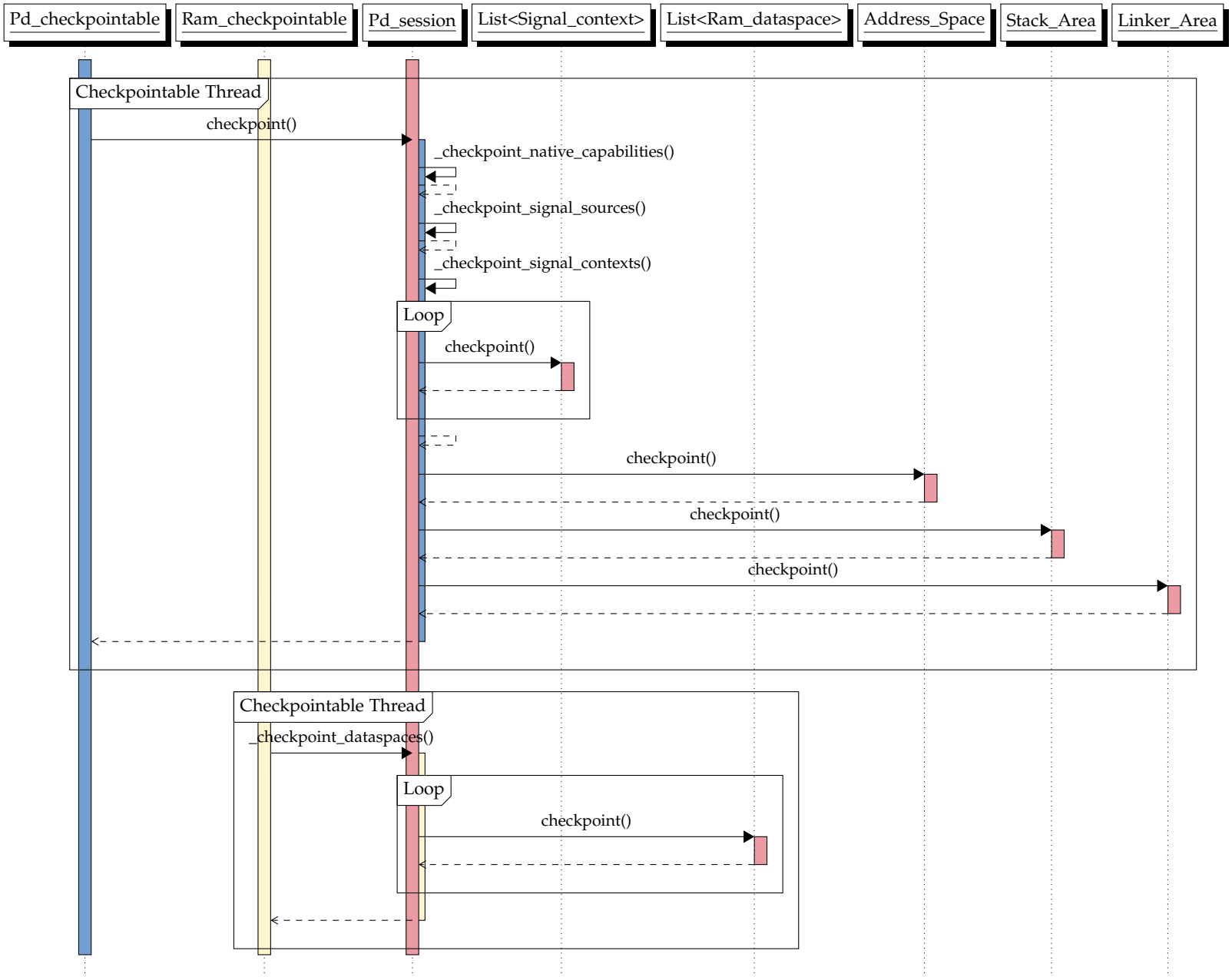

Figure A.2: Checkpointing PD Session in Genode 19.08

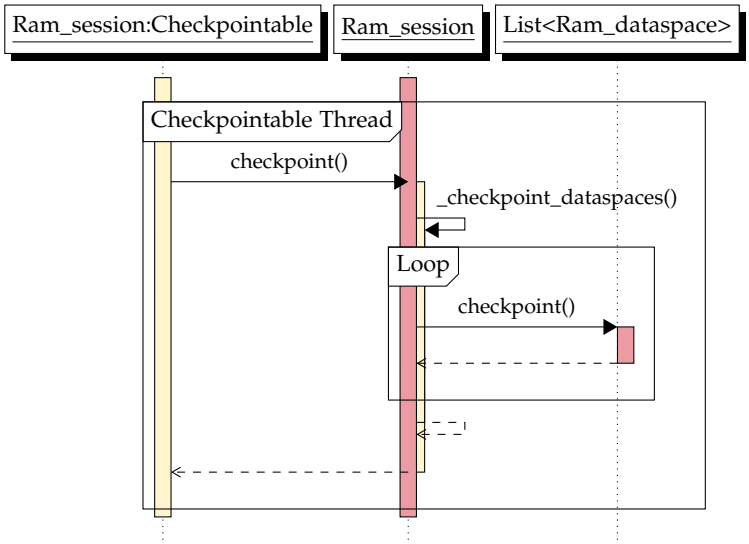

Figure A.3: Checkpointing RAM Session in Genode 16.08

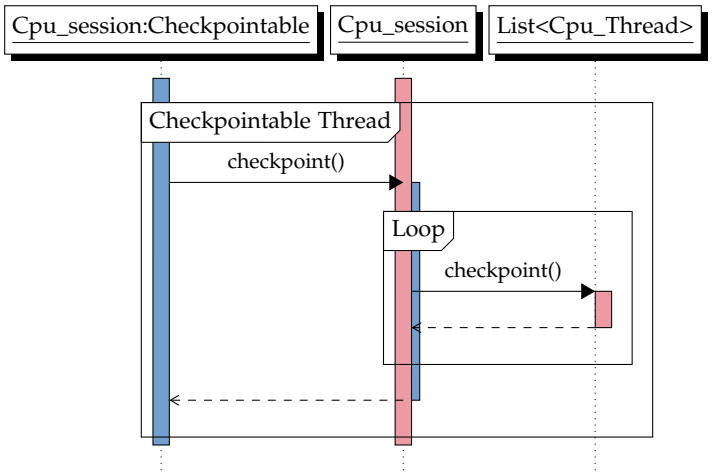

Figure A.4: Checkpointing CPU Session in Genode 16.08 & 19.08

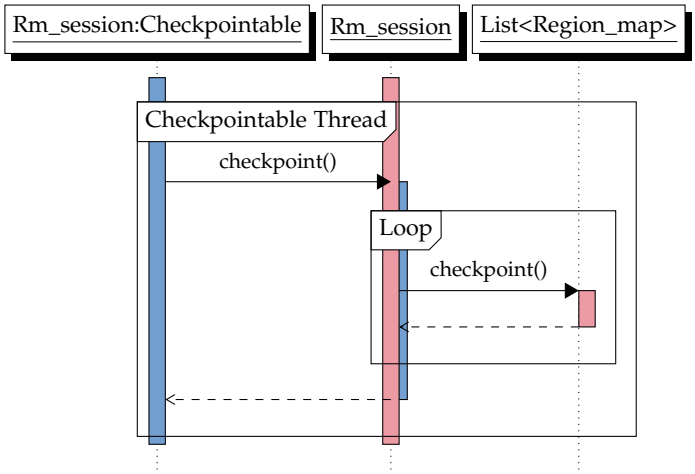

Figure A.5: Checkpointing RM Session in Genode 16.08 & 19.08

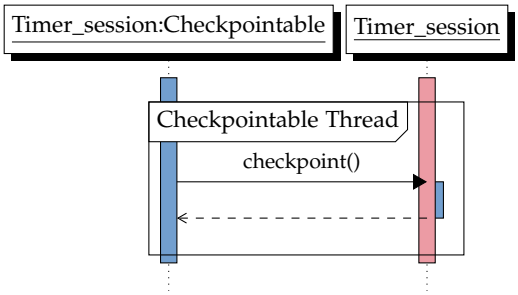

Figure A.6: Checkpointing Timer Session in Genode 16.08 & 19.08

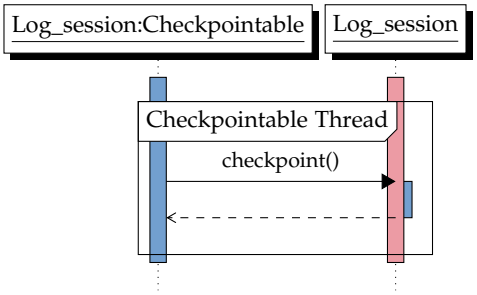

Figure A.7: Checkpointing Log Session in Genode 16.08 & 19.08

B Cold Storage Classes

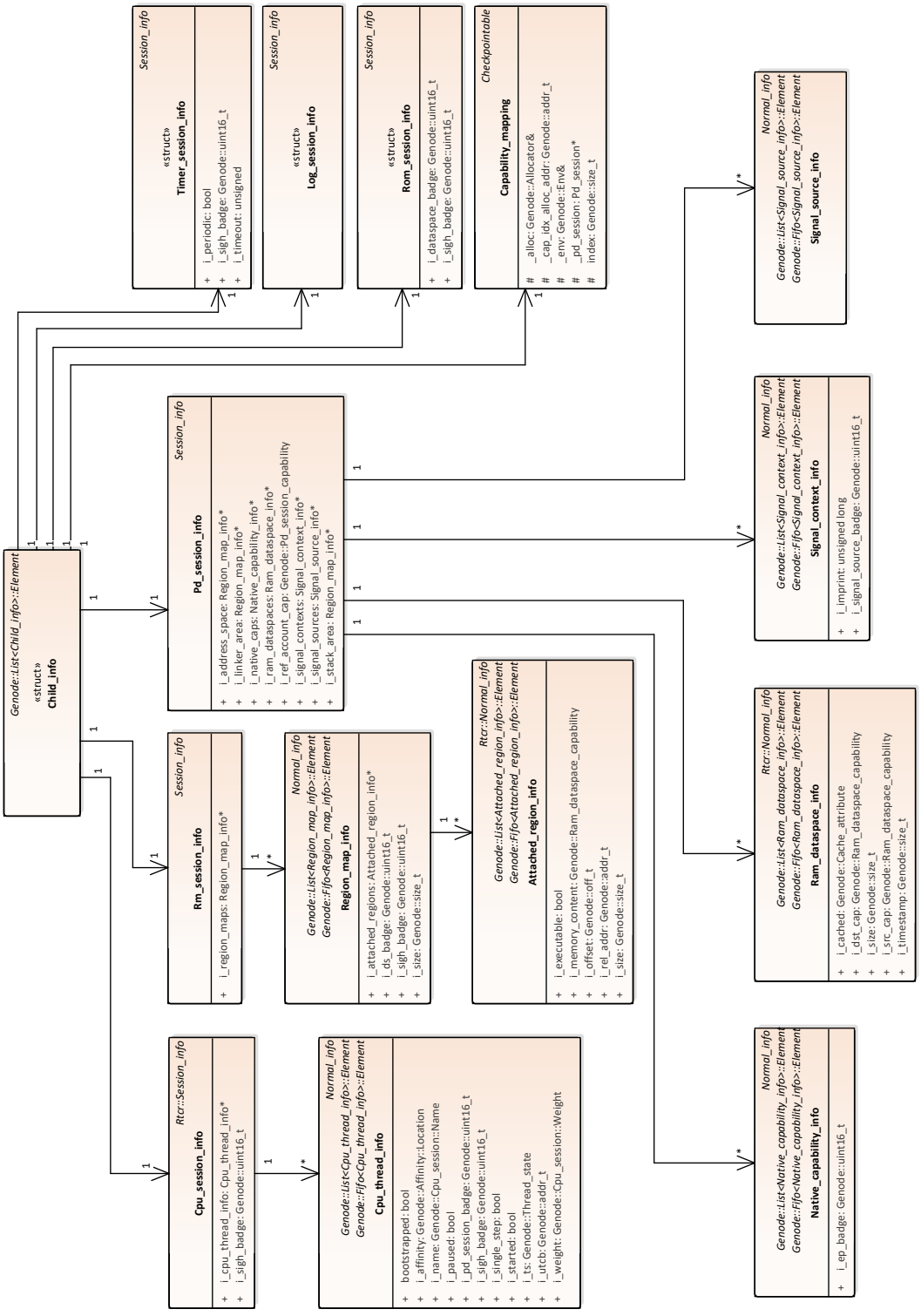

Figure B.1: Class Diagram of all `*_info` Classes which stores Information of the *Cold Storage*

C Board Specifications

Board	Processor	Processor Frequence	Architecture	Fiasco.OC	seL4	Memory
Zybo	2x Cortex-A9	650 MHz	ARMv7-A	x	x	512 MB DDR3
Wandboard Quad	4x Cortex-A9	1 GHz	ARMv7-A	x	x	2 GB DDR3
PandaBoard	2x Cortex-A9	1 GHz	ARMv7-A	x		1 GB DDR2
Raspberry Pi 2 V1.1	4x Cortex-A7	900 MHz	ARMv7-A	x		1 GB DDR2 (450 MHz)
Raspberry Pi 3	4x Cortex-A53	1.2 GHz	ARMv8	x		1 GB DDR2 (900 MHz)
Nvidia Jetson TK1	4x Cortex-A15	2.32 GHz	ARMv7?	x		2 GB DDR3L (933MHz)

Table C.1: Specification of boards used in the Evaluation

D Configuration of RTCRv2

XML node	**Description**			
module	Sets the module which is loaded. This XML node is only valid in combination with the `app/rtcr_app` application.			
	Attributes	**Type**	**Default**	**Description**
	name	string	not optional	Name of the module.
	report	boolean	False	Write results of measurements to the Report_rom.
child	Configures the attributes of a child application. This node can be defined multiple times in order to configure more than one child application.			
	Attributes	**Type**	**Default**	**Description**
	name	string	not optional	Name of the binary.
	quota	integer	no	Available Memory passed to the child application
	xpos	integer	0	Affinity location of the child process
	ypos	integer	0	Affinity location of the child process
	caps	integer	not optional	Available capabilities passed to the child application

checkpoint	Configures the general multi-threading behaviour			
	Attributes	**Type**	**Default**	**Description**
	parallel	boolean	True	If set to False, all threads are executed in serial.
checkpointable	Configures the behaviour of a thread			
	Attributes	**Type**	**Default**	**Description**
	name	string	not optional	Name of the Thread.
	xpos	integer	0	Affinity location of the thread
	ypos	integer	0	Affinity location of the thread
granularity	Sets the granularity of the `inc` module			
	Attributes	**Type**	**Default**	**Description**
	value	integer	not optional	The size of a page is calculated by `granularity*4096`
worker	Configures a worker thread of the `para` module. This node can be defined multiple times. For each node, a worker thread is created.			
	Attributes	**Type**	**Default**	**Description**
	xpos	integer	0	Affinity location of the worker thread
	ypos	integer	0	Affinity location of the worker thread

Table D.1: Overview of available Configuration Parameters of RTCRv2

E Passed & Failed Test-Cases

Genode	Kernel	Board	Module	Invalid Signal Context	Timeout	Passed
19.08	Fiasco.OC	Zybo	inc	36 (2%)	60 (3%)	1972 (95%)
19.08	Fiasco.OC	Wandboard	inc	0 (0%)	11 (1%)	1144 (99%)
19.08	Fiasco.OC	Raspberry Pi 3	base	6 (1%)	0 (0%)	820 (99%)
16.08	Fiasco.OC	Zybo	base	0 (0%)	4 (1%)	635 (99%)
16.08	Fiasco.OC	Zybo	cdma	0 (0%)	4 (1%)	732 (99%)
19.08	Fiasco.OC	Raspberry Pi 2	inc	0 (0%)	4 (0%)	996 (100%)
19.08	Fiasco.OC	Wandboard	para	3 (0%)	0 (0%)	1796 (100%)
19.08	sel4	Zybo	inc	1 (0%)	0 (0%)	1912 (100%)
19.08	Fiasco.OC	Pandaboard	base	0 (0%)	0 (0%)	754 (100%)
19.08	Fiasco.OC	Pandaboard	para	0 (0%)	0 (0%)	488 (100%)
19.08	Fiasco.OC	Raspberry Pi 2	base	0 (0%)	0 (0%)	782 (100%)
19.08	Fiasco.OC	Raspberry Pi 2	para	0 (0%)	0 (0%)	1890 (100%)
19.08	Fiasco.OC	Raspberry Pi 3	para	0 (0%)	0 (0%)	532 (100%)
19.08	Fiasco.OC	Nvidia Jetson TK1	base	0 (0%)	0 (0%)	808 (100%)
19.08	Fiasco.OC	Zybo	base	0 (0%)	0 (0%)	754 (100%)
19.08	Fiasco.OC	Zybo	cdma	0 (0%)	0 (0%)	1292 (100%)
19.08	Fiasco.OC	Zybo	para	0 (0%)	0 (0%)	784 (100%)
19.08	sel4	Wandboard	base	0 (0%)	0 (0%)	832 (100%)
19.08	sel4	Wandboard	para	0 (0%)	0 (0%)	1942 (100%)
19.08	sel4	Zybo	cdma	0 (0%)	0 (0%)	824 (100%)

19.08	sel4	Zybo	para	0 (0%)	0 (0%)	776 (100%)

Table E.1: Passed & Failed Test-Cases