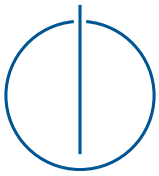
DEPARTMENT OF INFORMATICS

TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Port and Extension of a Toolchain Regarding Machine Learning Supported Schedulability Analysis in Distributed Embedded Real-Time Systems

Georg Guba



DEPARTMENT OF INFORMATICS

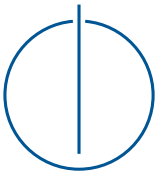
TECHNISCHE UNIVERSITÄT MÜNCHEN

Master's Thesis in Informatics

Port and Extension of a Toolchain Regarding Machine Learning Supported Schedulability Analysis in Distributed Embedded Real-Time Systems

Portierung und Erweiterung einer Toolchain zur Machine Learning gestützten Schedulability Analyse in verteilten eingebetteten Echtzeitsystemen

Author:	Georg Guba
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisors:	Sebastian Eckl, M.Sc.; Daniel Krefft, M.Sc.
Submission Date:	15.04.2016



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

Munich, 15.04.2016 Georg Guba

Acknowledgments

This thesis would not exist if not for a few people who helped me one way or another to develop and write this work. First and foremost, I would like to thank my two advisors Sebastian Eckl and Daniel Krefft who accepted my request for a master's thesis without hesitation and provided me with amicable and helpful support throughout.

Naturally, thanks also go to Prof. Dr. Baumgarten and the Technische Universität München for offering me an enjoyable environment and opportunity to finalize my studies in this thesis.

Thanks to my fellow inmates at the Chair of Operating Systems, Michael Kubitza, Paul Nieleck, Bernhard Blieninger, and Alexander Weidinger, who are finishing their theses and are working as research assistants as I write and helped me with their experience on the project.

Lots of love and thanks go out to Saskia Wassermann, a brilliant scientist, competent doctor, and unique and loving woman. You make life so much more fun.

Thank you to my family; my parents, my brother, and my sister, for raising me with the possibilities, skills, and freedom to live the life I want.

Last but definitely not least thanks to Michael Och. Even though his direct contribution to this thesis was little, his contribution to my life choices has been great. I genuinely believe him to be one of the greatest minds of security engineering of our time and he deserves every bit of success that is bound to come to him.

Abstract

Machine learning enables computer scientists to shift parts of the responsibility of manually analyzing and recognizing data patterns toward an automated process, avoiding the need for explicit search for causations and correlations. However, the notion of applying machine learning concepts to schedulability analysis is still pioneering work and mostly unproven in practice.

Especially in embedded automotive systems, profiling data volumes arising from novel smart mobility solutions and increasing quantities of electric control units are on the verge of exceeding what is feasible to process and interpret manually but still require tests to be run on finalized hardware. Machine learning in combination with data mining may be a promising approach to manage this kind of data.

Therefore, this thesis aims to provide a toolchain designed for the Genode Operating System Framework and the Fiasco.OC microkernel, inspired by previous prototypical work on the L4Re framework that intends to facilitate remote interaction of future machine learning implementations with the analyzed embedded system on two ends. Firstly, client applications are able to transmit generated task descriptions and binaries of arbitrary load profiles and functional requirements to the embedded system where they can be further controlled remotely. Secondly, monitoring data is logged by the task management system and can be requested by the client at any time. Client-side, this data is further refined and fed into an SQL database to simplify structured access to the transmitted information.

Tests of the final system indicate promising potential for future applications implementing an automated machine-learning-based feedback loop of constant task management, monitoring, data mining, and readjustment. The system, as a proof of concept implementation, is also designed to be easily programmable and extensible, allowing straightforward addition of further functionality that may be required for specialized applications.

Contents

1 Introduction

The concept of dynamic task generation, management, and monitoring implemented in this thesis is to be integrated in an established environment of software frameworks and hardware specifications. Specifically, the presented implementation is deployed on embedded devices targeted at the long-term goal of utilization in automotive systems.

Historically, the name of the developed task management and monitoring server *dom0* stems from its precursor implementations on the L4Re framework which in turn had its name lent to it from the Xen hypervisor. In the Xen Project, dom0 — or domain zero — is the first domain started by the Xen hypervisor and manages the allocation and dispatch of new domains using special privileges.

However, in the Genode implementation of dom0 presented here, it is merely a user space process on top of the Genode boot system with no dedicated permissions other than the complete control over lifetime and resources of its children, i.e., tasks. Furthermore, dom0 is charged with additional duties over the original Xen implementation, namely network communication and process monitoring.

This thesis will present design concepts and implementation details of the task management and monitoring toolchain including and surrounding dom0, ultimately aimed for the use in machine learning applications concerning schedulability analysis.

1.1 KIA4SM project overview

The dom0 toolchain is intended as an addition to the KIA4SM (Cooperative Integration Architecture for Future Smart Mobility Solutions) project developed at the Chair of Operating Systems at the Technical University of Munich. KIA4SM focuses on developing systems for the interaction and coordination between computer-assisted vehicles, be it partially or fully autonomously functioning actors.

Contrary to the usual approach of designing mostly isolated Intelligent Transportation Systems (ITS), KIA4SM aims to improve on the ad-hoc networking between vehicles as illustrated in Figure 1.1. This car-to-car coordination is a prime example of distributed embedded systems that particularly lend themselves to the application of *Organic Computing*. Organic computing describes the life-like, learning-based approach to solving problems that require adequate reaction to a multitude of unpredictable situations which would be infeasible to be handled all by pre-defined behavioral specifications.

Organic Computing works in tandem with the MAPE (monitor, analyze, plan, execute) design concept. An observer collects data of the system and its surroundings and processes this information into a global state of indicators (monitor and analyze). A

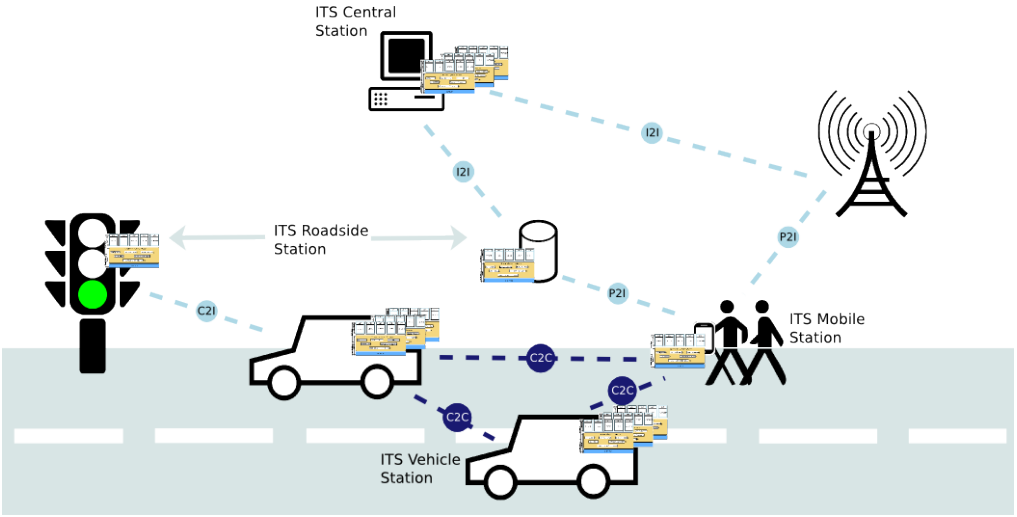


Figure 1.1: The vision of KIA4SM [EKB15].

controller then decides whether and how to react to these indicators and subsequently formulates actions to be executed by the system (plan and execute).

This notion also influences the design and architecture of the implemented monitoring and task management toolchain, as will be elaborated in the next section.

1.2 Motivation

Static schedulability analysis exceeds its capability when confronted with dynamic and unpredictable tasks. As described in the previous section, many monitoring and evaluation tasks within the KIA4SM project are expected to result from organic reactions to unforeseeable environmental factors, leading to highly intermittent task loads of varying dimensions. Additionally, contingencies have to be taken into account, like for example failing computing elements. This in turn calls for the employment of redundant hardware, e.g., processing units.

Combined, these aspects necessitate the design of a central software instance, dynamically managing migration and prioritization of tasks within the system, as static analysis would fail to cover every possible use case.

The intended approach for schedulability analysis and process migration policies of the KIA4SM project fits nicely in the context of organic computing and the MAPE paradigm: the central server process dom0 is employed to both monitor and autonomously manage tasks that are queued for execution, while at the same time it accepts external control instructions. These enable a remote client application to repeatedly benchmark the embedded system with different sets of load tasks and collect behavioral data on them, storing it in a database as illustrated in Figure 1.2. Eventually, the long-term goal aims to allow the system to learn from this data and use analysis thereof for improved task scheduling and migration behavior.

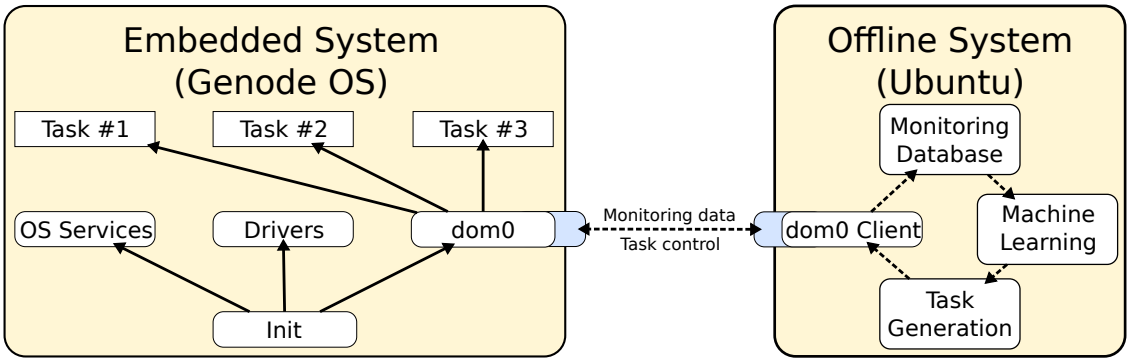


Figure 1.2: Simplified toolchain overview.

To that effect, a functional dom0 server needs to be designed for the Genode Operating System. This includes the management and supervision of tasks on the one hand and the interaction with outside clients via a fixed protocol on the other. Together with offline task description generation and monitoring data management, the presented work is supposed to provide a framework for future schedulability and migration analysis involving machine learning.

1.3 Thesis Structure

This thesis aims to describe the implemented toolchain using a top-down approach by first describing the scientific and programmatic environment before delving into the toolchain design on an abstract level. The main section of this work then goes into implementation details and hurdles encountered along the way. Finally, the thesis concludes with an outlook on possible future work, particularly extensions of the presented system with regards to machine learning applications.

Specifically, chapter 2 highlights related work that has been taken into account during the implementation of this toolchain. Starting with previous work on the topic of dynamic task management in embedded systems, this chapter also discusses similar projects realized on the Genode Operating System Framework and scientific papers on the topic of schedulability analysis and machine learning, and combinations thereof.

Chapter 3 gives an abstract overview of both the Genode Operating System and the alternative framework L4Re before elaborating on important Genode features that were essential for the realization of this toolchain, including the selection of microkernels, the overall organizational structure of Genode, and the testing and debugging framework used during development.

Chapter 4 finally goes into more detail about the toolchain design concept that has been described earlier. All components of the toolchain are illustrated here and put into context within the Genode OS.

In the central chapter 5 the thesis expands upon the toolchain design by providing more information on implementation details. This chapter will make clear how ev-

ery single part of the previously presented toolchain design is realized in the given environment and made to interact with each other for a functional design.

The following chapter 6 will further expand upon the implementation details by presenting design choices made and limitations encountered during the development of this toolchain. Workarounds and alternatives are considered in this section.

Finally, chapter 7 gives a step-by-step description of a toolchain showcase. It features both the building process and usage instructions required for execution of and interaction with the toolchain.

Chapter 8 concludes the thesis, illustrating potential future work that might benefit from the realization of this project, including both extensions to the implemented system as well as utilization of its provided features. Ultimately, this chapter summarizes the results obtained from this thesis.

2 Related Work

While the concept of using machine learning for schedulability analysis is relatively new and unprecedented, this thesis still relies on previous work, specifically system modules of similar functionality implemented in different frameworks. This chapter aims to give a quick overview of related theses and papers that have been helpful during conception and development of the dom0 toolchain for Genode.

2.1 Related Work on Dynamic Task Management

The concept of starting and managing tasks dynamically over the network was integrated by Josef Stark in his bachelor's thesis *"Dynamic task management between interconnected L4 microkernel instances"* for the L4 Runtime Environment into the KIA4SM project before its transition to the Genode Operating Systems Framework in 2015 [Sta14]. In his work, Stark implemented the central network communications task *dom0* which enabled client applications to execute Lua code and precompiled binaries on the embedded system using the extended L4Re init process *Ned*. Additionally, Stark implemented a protocol to migrate tasks between multiple connected L4Re instances.

Avinash Kundaliya extended Josef Stark's work in his master's thesis *"Design and prototypical implementation of a toolchain for offline task-to-machine mapping in distributed embedded systems"* [Kun15]. In his thesis, Kundaliya incorporated the task description tool tms-sim into a toolchain built around the dom0 and Ned task manager modules. As part of this toolchain, dom0 and Ned were extended by functionality to read and interpret said task descriptions, and a prototypical implementation of a monitoring module.

Porting and extending functionality of these two theses to the Genode Operating System Framework with the added focus on machine learning applications is the main goal of the toolchain presented in this thesis.

Extending the concept of optimal task scheduling to multiple cores was part of Stefan Groesbrink's dissertation *"Adaptive Virtual Machine Scheduling and Migration for Embedded Real-Time Systems"* [Gro15]. In this dissertation, Groesbrink presented and implemented various approaches to schedule tasks on multiple homogeneous processor cores by partitioning and mapping virtual machines wrapping these tasks to cores, including the real-time scheduling and migration of said partitions. While migration and actual scheduling is not yet part of this thesis, future work building on top of the dom0 toolchain is expected to implement many of these concepts.

2.2 Related Work on Genode

The main source of development information about the Genode system is the *"Genode Operating System Framework Foundations"* book by Norman Feske [Fes15]. This foundations book features detailed explanations of Genode system components and a wide variety of useful sample applications showcasing every single feature. The Genode system will be explained in more detail in chapter 3.

Since the transition of KIA4SM from L4Re to Genode is relatively recent, there is few project-related work implemented on Genode at this moment. The most relevant work is Alexander Reisner's bachelor's thesis *"Extension of the Genode OS Framework by a Component for Runtime-Monitoring of a Real-Time Operating System"*. In this work, Reisner ported the *Ferret* monitoring library from L4Re to Genode, implementing a producer and consumer system in shared memory, similar to the trace buffer functionality presented in chapter 3.

2.3 Related Work on Schedulability Analysis and Machine Learning

Ahmad et al. presented a novel approach to complex schedulability analysis using game theory in their work *"Using Game Theory for Scheduling Tasks on Multi-Core Processors for Simultaneous Optimization of Performance and Energy"* [ARK08]. Their main focus lied on energy-aware scheduling of tasks in multicore processor environments. Eventually, Ahmad et al. were able to implement algorithms based on multi-objective optimization of a cooperate game that performed significantly better than established heuristics.

In Basu et al.'s paper *"An Optimal Scheme for Multiprocessor Task Scheduling: a Machine Learning Approach"*, machine learning was used to develop scheduling rules of mostly periodic tasks that minimized the number of preemptions and overheads required in a multiprocessor context [BF09]. Methods included the extension of established deadline scheduling algorithms by machine learning techniques in order to incorporate inductive and rule-based concept learning.

Generally speaking, there is very few work on the topic of machine learning associated with schedulability analysis or the automatic development of scheduling rules. Incorporating machine learning into real-time scheduling of tasks in distributed systems is a mostly pioneering concept and still largely experimental.

3 Genode Operating System Framework

The Genode Operating System Framework is the main framework of choice for the KIA4SM project. The following chapter features the most important data points, design concepts, and structural aspects of the Genode Operating System and its extended framework.

3.1 Overview

The Genode OS Framework is an open source project for x86 and ARM processors, mainly supported by Genode Labs GmbH based in Dresden, Germany. Genode is intended for creating both special-purpose operating systems, including minimalistic embedded systems, and medium-scale dynamic general-purpose systems, partly able to replace and emulate existing desktop operating systems. In combination with its focus on high security, Genode is an ideal choice for embedded automotive systems as required by the KIA4SM project.

For one, security and safety are both major concerns in Intelligent Transport Systems. Security may easily be compromised in distributed ad-hoc networks such as the one proposed by KIA4SM. Employed systems need to ensure that participants, both benevolent and malevolent, have no chance of deliberately or accidentally interfering with crucial vehicle functionality, like for example braking systems or sensor equipment. With almost all life-saving subsystems relying on electronics and software, the need for a secure architecture is paramount.

Genode offers security through its recursive organizational structure in that each program only has access to its very own sandbox with explicit permissions to specific services and access rights granted by its parent process. This compartmentalization is an ideal environment for security and safety requirements, as the implementation of such need, for the most part, only be applied to isolated sectors due to the reduced possibility of error and malware propagation to other parts of the system.

At the same time, Genode offers the potential to create applications of minimal size, both regarding memory consumption and processing power, emphasized further by the wide variety of supported microkernels. In addition, Genode offers vast extensibility through optional libraries and apps including support for C and C++ standard and lwIP networking libraries. Considering the architecture KIA4SM aims to deploy, including numerous preferably inexpensive microprocessors connected in a distributed manner, Genode is a great candidate for this project.

3.2 Related Framework L4Re

Previously, KIA4SM relied on the L4Re (L4 Runtime Environment) framework developed by the Dresden University of Technology in Germany [L4Re]. Projects such as the one presented by Avinash Kundaliya or Josef Stark, both of which helped build the foundation for this toolchain, all implemented their applications on L4Re [Sta14][Kun15].

L4Re is a runtime environment built around the Fiasco.OC microkernel and also developed at the Dresden University of Technology. Similar to Genode, L4Re supports x86 and ARM platforms, C and C++ standard libraries and networking features. However, development has largely stalled and documentation support is sparse. It was thus decided to drop L4Re support and switch to Genode for better usability and a larger spectrum of features, such as additional libraries and ports of third party software.

3.3 Evaluation of Microkernels

Genode supports a wide variety of microkernels, including a custom kernel for ARM-based platforms named *base-hw*. All of these kernels have their pros and cons, although only the most important ones are discussed here.

3.3.1 Fiasco.OC

Fiasco.OC is the kernel originally used in the KIA4SM project when it was developed for the L4 Runtime Environment. Fiasco.OC – or *base-foc* as it is named in the Genode repositories – was initially chosen for its vast platform support, including the previously used BeagleBone and current PandaBoards, offering more flexibility for design changes throughout the development of KIA4SM.

The dom0 toolchain uses the Fiasco.OC kernel for development, specifically built for the PBX-A9 board. Even though PBX-A9 boards will most likely not be used in the final implementation, this platform supports QEMU, Genode, and Fiasco.OC, making it an ideal testing target.

3.3.2 NOVA

NOVA (NOVA OS Virtualization Architecture) is a microhypervisor based on the Fiasco microkernel and offers a secure virtualization environment built around a minimal trusted computing base [NOVA]. NOVA (*base-nova*) is currently the main development platform for Genode Labs GmbH, therefore offering the best support and broadest feature band of all microkernels presently available on Genode. For example, Genode developers are able to run a native Genode OS based on the NOVA hypervisor on their laptops, featuring a large subset of what established desktop operating systems offer, and emulating what it does not.

Most relevantly for the dom0 toolchain, NOVA is currently the only microkernel to offer scheduler execution time readouts. However, since this was easily implemented in

base-foc too, and because NOVA does not fully qualify as a microkernel but rather a virtualization hypervisor, the dom0 toolchain supports Fiasco.OC instead.

3.3.3 base-hw

Finally, Genode's custom microkernel called *base-hw* is another viable candidate and may in fact be a future option for the KIA4SM project intended to let Genode run directly on hardware. Since Genode Labs are developing this kernel themselves, it is specifically tailored to the needs of the Genode OS Framework.

On the other hand, base-hw, being a custom implementation, does not contain the full feature set that other kernels offer, as features are only implemented as needed by the developers. One such example is the aforementioned possibility to query scheduler execution time of specific process. This functionality is also lacking in the base-hw kernel and is non-trivial to implement.

Ultimately, this led to Fiasco.OC as the kernel of choice, even though base-hw might be a good alternative in the future.

3.4 Organizational Structure

Genode features a heavily recursive structure, meaning that processes inherit permissions, access rights, resources, and services by their parent processes. This design is realized by a variety of features implemented in the Genode framework that are described in this chapter.

3.4.1 Capabilities

Capabilities in the Genode system are like tickets bound to a specific resource. Resources in Genode like dataspaces or CPU sessions are represented by *RPC objects* (remote-procedure call objects). RPC objects provide a clearly defined interface through which processes may leave their own protection domain and interact with the remote object.

Whenever a process attempts to call methods provided by an RPC object, it requires a capability that is bound to the object's identity. When accessing the RPC object with a capability, the kernel tests access rights using the capability space of the calling process as seen in Figure 3.1. Only through the object identity known to the kernel can the process access resources of other protection domains. Artificially created capabilities, i.e., capabilities not created by the associated object, fail lookup in the kernel's capability space and are denied access to the called object.

Since Genode supports a variety of kernels as discussed previously, Genode translates kernel-abstract capability usage into their native types using kernel-specific implementations. This provides a unified interface to capabilities without the need for addressing individual kernels explicitly.

Figure 3.1 also illustrates the delegation of capabilities. Capabilities are initially only handed out by the associated object itself. Access to a resource must thus directly be

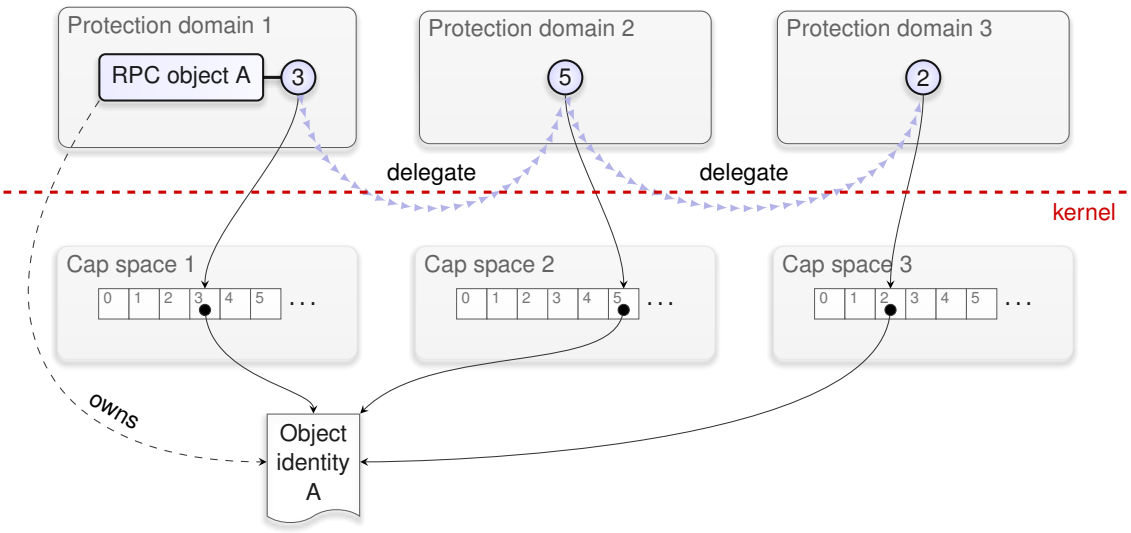


Figure 3.1: Capability domains and delegation [Fes15].

requested from the resource or passed on by a process that already possesses a matching capability. The capability object itself represents access rights to a resource and passing it on to other processes will also pass on associated access rights.

Ultimately, this defines a clear trusted computing base for any process within the Genode system: child processes may only possess capabilities passed on to it by its parent and any additional, directly issued capabilities by the requested resource itself.

3.4.2 Process Creation

Processes – or *components* as they are called in Genode – are created and started by instantiating the `Child` class. This child object represents the running process, including all its threads. The creation of a child is tightly tied to its parent component, in that it inherits resources and services from the creating process.

In order to properly initialize a child several key sessions, i.e. connections, to core services are required.

RAM Session Each component has its own memory address space as part of its sandbox environment. However, since RAM may not be allocated at will, existing RAM sessions have to be used and reused as dataspaces. Every RAM session keeps track of a *quota* indicating how much memory is associated with the respective session. In order for a child to gain access to a proper RAM session, the parent must create a new connection to the RAM service for the child and transfer some of its own quota to it, enforcing the recursive Genode design in memory management. Finally, the parent delegates the capability associated with this connection to the child which it may use in turn, for example,

to create additional children.

ROM Session — The child's executable binary is usually stored within a ROM session, i.e., a shared dataspace loaded at boot time and expected to be constant through runtime. Genode uses the capability attached to this ROM session to initiate the child's code.
Connections to the ROM service may be created at will by requesting a binary name from the ROM registry. Per default, this ROM registry only loads ELF binaries specified in the Genode run script at boot time. For the dom0 task manager, however, it was necessary to be able to start binaries from RAM.

CPU Session — A connection to the CPU service associates the component with a kernel process to be scheduled by the scheduler. Preceeding child creation, the parent is required to create a CPU connection for the child, specifying session name and its scheduling priority, as will be described later.

RM Session — The region manager is responsible for mapping global and shared dataspaces to the local address space of a component. This ensures that children have access to and only to their own sandbox memory, including dataspaces associated with capabilities explicitly delegated to the component.

PD Session — The protection domain session created by the parent represents the isolated sandbox of the child. It comprises the capability space of a component and also identifies and supervises hardware utilization like memory and CPU access.

Finally, the child is associated with a so-called *entrypoint*. A Genode entrypoint is simply a thread that starts the execution of a program. When binding an entrypoint to a child, it can either use this thread for its fixed one-time execution, or provide other processes access to the entrypoint via capabilities and only run when called, i.e. acting as an RPC server.

Genode applications are initially started by the Init module which in turn is started by Core. Core provides all the services listed above, whereas Init executes the run script provided to Genode. The run script specifies services, drivers, and components to be launched on startup, including applications like dom0 that in turn are able to launch other components.

3.4.3 Scheduling

Scheduling in Genode is for the most part handled by the chosen kernel. Genode merely abstracts the common scheduler parameters, namely priority and CPU core affinity and makes these values available to the framework. However, how these parameters affect performance greatly depends on the chosen kernel.

Priority values in Genode are passed to the child's CPU session on creation. Values range from 0 to a fixed limit of `PRIORITY_LIMIT` ($2^{16} - 1$ per default), with 0 represent-

ing highest priority. On Fiasco.OC this value is inverted and scaled by Genode to a value between 0 and 127. Additionally, priority level resolution can be defined in a program's run script configuration by specifying the `prio_levels` attribute. The available range of priorities is then subdivided into sets of equivalent priorities. For example, specifying a `prio_levels` value of 2 would subdivide priorities into the equivalence ranges 0 to $2^{15} - 1$ and 2^{15} to $2^{16} - 1$. The complete process of priority translation between Genode and kernel for two discrete priority levels is illustrated in Figure 3.2. Possible loss of priority information due to a low number of priority levels is communicated via a warning on the console.

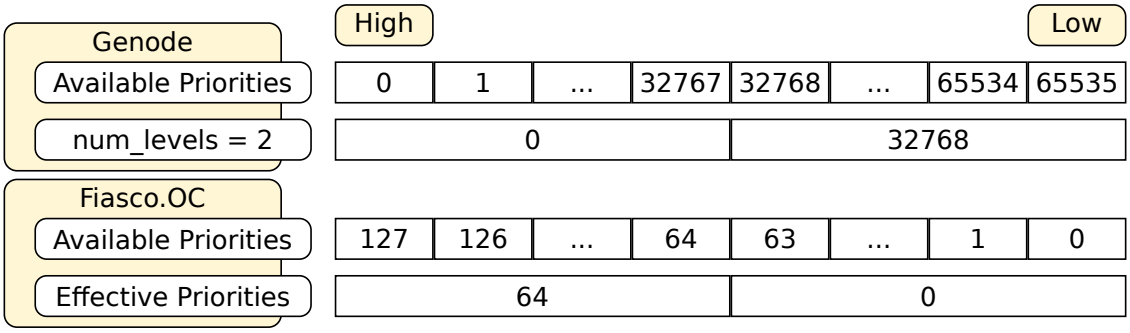


Figure 3.2: Genode and Fiasco.OC priority values.

In traditional L4 microkernels such as Fiasco.OC and NOVA, scheduling uses static priorities: scheduler time is divided into time slices and for a given slice the scheduler always selects the thread with the highest priority [Döb]. Subsequently, this thread runs until its time slice runs out, it blocks, or a higher-priority thread becomes available. If multiple threads share the same priority, their execution time is allocated by a round-robin policy.

Genode's custom base-hw kernel implements a more sophisticated scheduling policy on top of static priorities. First, it divides threads into *claims* and *fills*. Claims are intended for jobs with low or medium throughput that require fast reaction times like GUI tasks or driver software. Fills on the other hand are reserved for computational heavy tasks that do not require low latencies, for example compiling or rendering tasks. Within a given time period called a *super period*, e.g. one tenth of a second, the base-hw scheduler always schedules the claims first until they have filled their previously assigned fixed share of the super period. Once all claims have received their share, the remaining time of the super period is dedicated entirely to fills. This approach is intended to prevent high-priority jobs from starving lower-priority ones while at the same time still providing fast scheduling without adding too much complexity to the policy.

Accordingly, the choice of microkernel and subsequently scheduling policy is expected to affect performance. Monitoring data stemming from one kernel will not be applicable to scheduling analysis of other kernels.

3.4.4 Inter-Process Communication

Processes that need to communicate with other components in the Genode system need to do so via connections to their services. A component may provide services by registering their server entrypoint under a given name with a service registry. Typically, this is the parent's registry and on registration the parent also announces that a new service is available, waking up any processes waiting for this service.

All components are also responsible for routing service requests to their corresponding services. Specifically, every component needs to keep track of services directly or indirectly available through its parent and children, unless services are to be explicitly denied to certain sections of the component tree. Once a service has been requested and found, the client process is able to directly communicate with the server via a capability.

In the example service requests illustrated in Figure 3.3, App #1 requests two different services. Both services are found in App #1's parent service registry, so the request is rerouted to Init. Init then finds the requested services in its child and parent registry respectively, rerouting them to their target destinations.

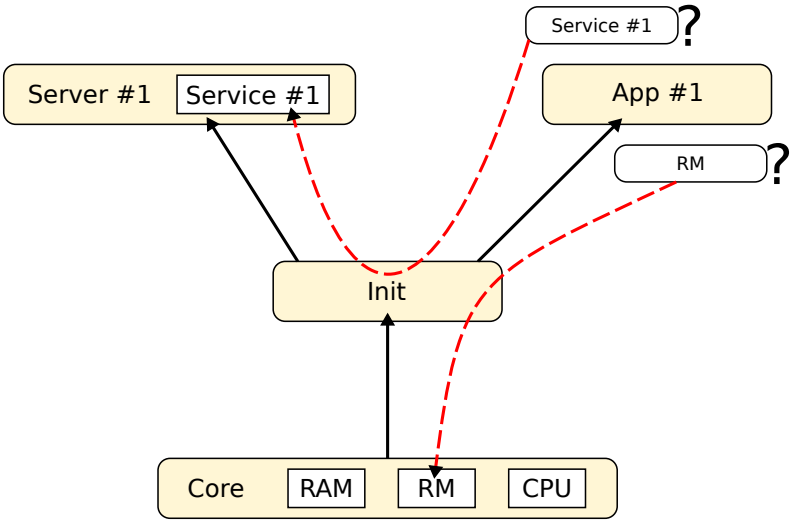


Figure 3.3: Service routing.

Calls to RPC server objects enable processes to leave their own protection domain and enter external ones through their entrypoints. Similar to a C function call where arguments are pushed to the stack only to be read later by the called procedure, Genode RPC calls place arguments in shared memory, enabling triggered services to read these arguments, and optionally even to return values. The major difference, however, is that target entrypoints run on – or rather are – a different thread, so the calling process needs to block for the duration of the RPC function call in order to enable synchronous operation.

Inter-process communication in Genode requires a standardized setup of RPC client and server objects as illustrated in Figure 3.4. More specifically, these objects have to be

mirrored by C++ classes and interfaces.

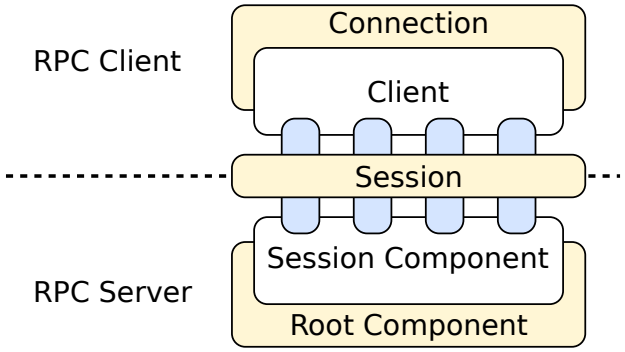


Figure 3.4: Genode inter-process communication.

Session	The Session class is the central interface between client and server. This class merely declares functions that are to be implemented by the server and can be requested by the client. However, it does not define any of them. In C++ this class is implemented using only purely virtual member functions which is often called an interface class.
Client	The Client class holds the capability to use an RPC Session interface. Accordingly, it must translate offline client-side calls to RPC methods into online capability invocations to the server. Usually, this is a simple one-to-one translation of C++ methods to capability invocations of similar names, including the forwarding of arguments and return values.
Connection	Interacting directly with the Client class is still rather complex, as the Client needs to be initialized using session-construction arguments, connected, and closed again. In order to simplify this process, the Connection class encapsulates a Client and provides a single interface for a component to use.
Session Component	The Session Component class is the actual RPC object that is to be called. It features implementations of the functions defined in the Session interface. Functions executed in this class usually run on a different thread than invocations of the Connection and Client classes, despite calls still being managed synchronously.
Root Component	The Root Component class, similar to the Connection class on the client side, encapsulates the usage of a Session Component. It announces the service by name and with it the capability to create RPC sessions to the parent.

3.4.5 Signalling

In contrast to calls to RPC objects, Genode signals provide the ability to trigger events asynchronously. Signals in Genode are triggered by *signal dispatchers*. Signal dispatchers can for example be registered with timers, so that when the timer expires, the timer issues a signal using the dispatcher to a receiver.

The signal dispatcher sends the signal to a receiver by invoking a context capability associated with said receiver. Signal contexts offer the possibility for receivers to distinguish between signals, as a single receiver might be used to handle different types of signals.

Signal receivers may be polled manually for any signals that may have arrived since the last poll. Special care needs to be taken when dealing with these signals, as they may arrive from different threads. Alternatively, using a receiver, it is possible to block the current thread until a signal is received.

Server entrypoints may also be used as signal receivers. In this case, signal handling is executed synchronously on the same thread as the associated server. This can be very helpful, as will be made clear in the implementation description of the dom0 task manager server later in this thesis.

Genode provides the helper class `Signal_rpc_member` that uses an entrypoint as receiver and uses this entrypoint together with a signal context capability to call a specific class member function when signals are received. This is ideal for registering callbacks with timer timeouts.

3.4.6 Process Tracing

Out of the need for process performance monitoring, Genode recently introduced the ability to trace processes. Tracing enables a monitor process to provide some of its own RAM quota to log information about other processes. The dedicated dataspace provided by the monitor process is called *trace buffer* and can be filled with arbitrary content.

Traced processes can request a capability to this trace buffer and log whatever they might find worthy of note and store it in the buffer. This could, for example, be used to regularly and asynchronously log progress of a high-throughput operation like image processing. With the monitor process providing the trace buffer, traced processes need not sacrifice any of their own quota for performance analysis.

The main focus of process tracing in Genode, however, lies on event-driven tracing that requires no cooperation of the traced process. This is realized by using trace policies. Trace policies are implementations of the trace policy interface that can be redefined at runtime. In order to trace a process using a policy, the information to be logged needs to be defined. This is done for example in the sample policy `rpc_name` which writes the name of every RPC call to the trace buffer.

Trace buffer access and event tracing is managed by the Core service *TRACE* as illustrated in Figure 3.5. Furthermore, reading from and writing to trace buffers is accomplished without any system calls as this would defeat the purpose of lightweight

and nonintrusive event tracing.

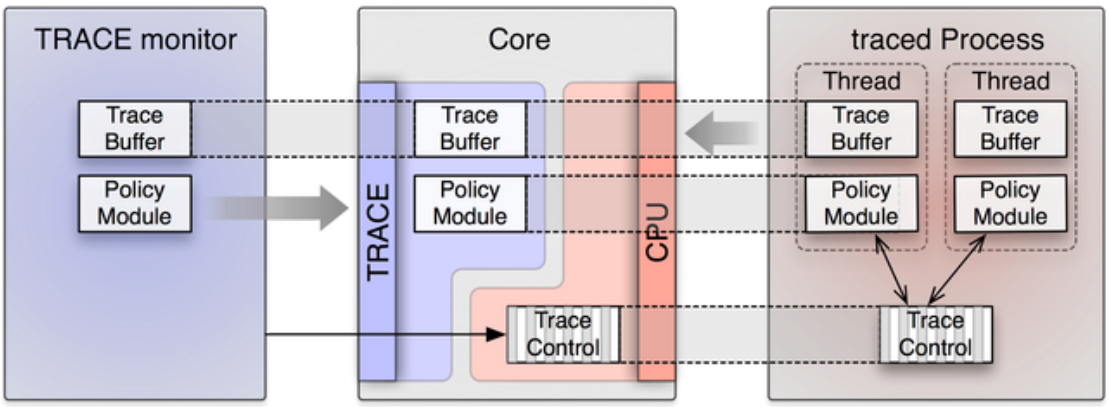


Figure 3.5: Genode event tracing [Gena].

Events that are currently registered by trace policies include RPC calls, returns, dispatches and replies, and signal submission and reception. All of these events can be hooked by defining the function bodies of a trace policy and registering the corresponding trace policy ROM dataspace at runtime.

However, the TRACE service not only provides information about explicitly traced processes but rather any process currently running on top of Core. This information includes session and thread name, tracing state, trace policy, thread CPU affinity, and most importantly thread execution time. This last data point, however, is currently only implemented on the NOVA hypervisor. For other microkernels, this value is always 0 and requires manual implementation.

3.4.7 Repository Structure

The Genode repository is clearly structured [Genb]. All source code lies in the `repos` subdirectory. As the Genode framework is built in multiple levels on top of the kernel, the source code structure tries to closely mirror this pattern.

Genode's Core component is divided into its platform-specific and platform-independent subcomponents. The `base-<platform>` subdirectories each define the abstraction layer of a specific microkernel to Genode Core services. Kernel-specific functionality is translated here to fit the general interface of Genode Core, such as thread management, scheduler readouts, etc.

The platform-independent Core components are placed in `base`, further separating public Core interfaces from its implementation. The `base/include` subdirectory contains all services that are available to user-space applications, like ROM and RAM session interfaces, whereas `base/src` holds implementation details and functionality deliberately hidden from applications to enforce Genode's recursive and encapsulated structure.

On top of the base layer lies the OS layer, including for example the aforementioned Init process or device drivers. Code of this layer can be found in the `os` subdirectory, mirroring base's structure in that `os/include` holds user-space library and service code while `os/src` houses implementation details inaccessible to applications.

More source code featuring example usage of important Genode features can be found in the subfolders `demos` and `gems`. Most OS and base functionalities also provide tests in respective `test` subfolders, giving great guidelines on how to use specific features and services. Additionally, ports of popular applications and libraries including `libstdc++` are located in the subdirectories `ports` and `libports` respectively.

Another noteworthy directory for this project is the `tool` directory on the same level as `repos`. This folder contains tools and scripts required for building Genode and its toolchain.

While there is a lot more to Genode's repository structure, these are the most important pointers.

3.5 Testing Framework

With Genode being an entire operating system framework built on top of any of a selection of microkernels, standard debugging and testing tools are unsuitable. The following section presents two tools used for testing and debugging the developed toolchain on Linux systems.

3.5.1 QEMU

QEMU (Quick Emulator) is an open source hardware emulator for a variety of platforms, including the PBX-A9 computer of the ARM Cortex-A9 family. A single-board computer of the same family, namely PandaBoard, is used in hardware in the KIA4SM project, so QEMU is a decent choice for emulating the target hardware. QEMU, however, offers no direct support for PandaBoard computers themselves.

QEMU features legacy VDE support, thus offering the ability to test applications and services including Ethernet communication without the need of deploying code on actual hardware, greatly speeding up the development process.

3.5.2 VDE

Since the presented dom0 toolchain incorporates an active TCP/IP connection, an Ethernet configuration is required that enables communication between a Linux client and a QEMU server instance. VDE (Virtual Distributed Ethernet) provides exactly this. VDE offers the ability to create additional virtual network kernel devices, called TAP interfaces and offers to connect these interfaces to a virtual switch. On the other end, QEMU can connect to this switch due to its built-in VDE support.

Once routing has been set up properly to forward client requests through the virtual TAP device, clients can connect to servers that are hosted on the previously connected QEMU instance. Servers can be addressed by their static IP, provided that they are hosted without DHCP support. Should DHCP be desired, servers need to listen to the `INADDR_ANY` address and a NAT router able to manage virtual switches needs to be employed, like for example slirpvde.

The dom0 toolchain presented here supports both static server IPs and dynamic addresses using slirpvde.

4 Toolchain Design

The final goal of the toolchain presented in this thesis is the novel attempt to produce schedulability data of organic task sets that can eventually be used in the online system to dynamically allocate and schedule unpredictable task loads.

With the Machine Learning module of the system still requiring conception and implementation, the remaining toolchain developed here provides the foundation for designs involving organic schedulability analysis.

4.1 Overview

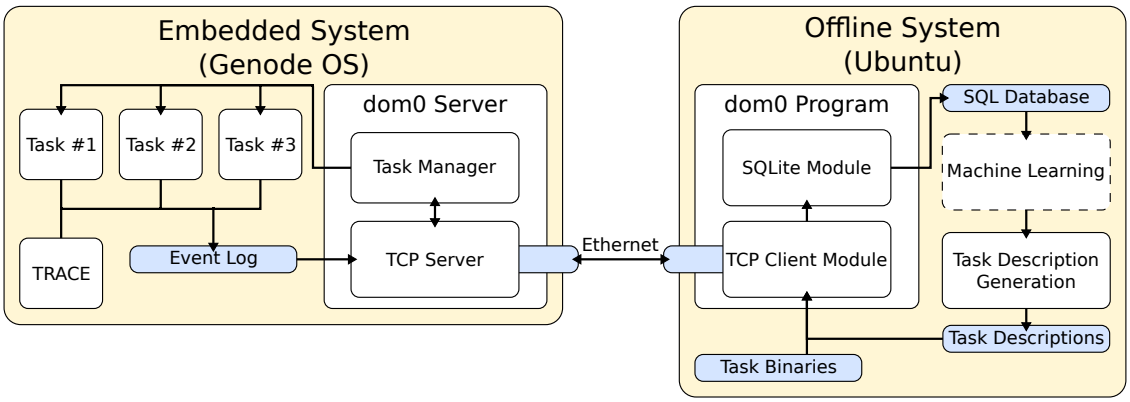


Figure 4.1: dom0 Toolchain Design.

The toolchain developed around dom0 is a relatively complex design, as illustrated in Figure 4.1. Generally speaking, the toolchain is divided into two larger parts. First, the actual embedded system running on Genode provides a realistic framework to schedule and monitor task loads. As this system is able to run on final hardware, it is also called online.

The offline system on the other hand merely communicates with the embedded part via Ethernet and beyond that is able to run on arbitrary hard- and even software. For ease of use during development Ubuntu 14.04 was employed here. With this concept in mind, most analytical tasks are outsourced to the offline system, leaving the embedded part of the framework with mostly nonintrusive monitoring and logging tasks, reducing interference with the actual testing environment to a minimum.

In order to facilitate the organic design targeted at schedulability analysis, the offline system represents a feedback loop: task descriptions are generated with input from

previous testing cycles. Task descriptions and their corresponding binaries are subsequently fed into the dom0 client program to start a new benchmarking run on the online system by forwarding task data to the server and controlling their activity remotely according to a freely programmable command sequence. Profiling data collected from this test run is then received in a raw format and refined by the client program's SQL module in order to be fed into an SQLite database. This database ultimately acts as input for future machine learning applications which in turn are intended to generate new task descriptions in order to both analyze and actively optimize monitoring data. Conclusions from this feedback loop are eventually anticipated to provide scheduling information that can be used on an autonomous online system for better task allocation and scheduling.

Whereas the offline system is designed as a continuous feedback loop, the online embedded system acts as a server that only reacts to requests from the dom0 client program. Tasks are allocated, run, and monitored throughout as dictated by the Ethernet connection, creating an event log in shared memory that can be requested by the offline system at any time.

4.2 Task Description Generation Using tms-sim

Task description generation is the process of defining properties of existing binaries that are to be run on the server. Specifically, these task descriptions define behavioral information of tasks, for example the average execution time, arrival period, binary name, or a task's critical time, and how best to manage them for optimal schedulability.

Based on previous work by Avinash Kundaliya, task description generation is largely unchanged from his precursor implementation of the dom0 toolchain on L4Re [Kun15]. Accordingly, not all task description parameters provided by the established system are actively used in the current Genode implementation of the dom0 server and even more parameters may be valuable additions in future works.

Task description generation is realized using the existing tms-sim framework, extended by Avinash Kundaliya. This open-source C++ tool is able to generate XML files describing execution parameters of a task. Additionally, tms-sim features the ability to simulate real-time scheduling of tasks based on these descriptions which may be useful for further offline schedulability analysis.

For the purposes of the Genode dom0 server and toolchain, this task description generator had to be subjected to minor changes and extensions. Chapter 5 goes into more detail about task description generation and its parameters.

4.3 dom0 TCP/IP Server Module

The dom0 TCP/IP server is the central network communication module of the dom0 Genode server. As its purpose is mainly to serve requests by the client, it lies idle most

of the time. The TCP server mediates between the requests of the offline client and the running task manager on the embedded system.

Contrary to its L4Re counterpart, the dom0 TCP server does not accept Lua code. Rather, received binaries have to be built beforehand using the Genode toolchain. These binaries are stored in dataspaces allocated by the task manager, leaving the TCP server as a very lightweight component. Control command issues directed at the task manager, like starting and stopping tasks, or profiling data requests, are simply forwarded to the task manager directly.

4.4 dom0 Task Manager Module

The dom0 task manager is the centerpiece of the dom0 Genode server. As previously stated, it provides dataspaces for task binaries and controls task lifetimes. Additionally, it manages all meta data and resources shared by the tasks, like event log dataspaces and task entrypoints.

Furthermore, the task manager supervises the execution of tasks according to their descriptions, taking care of critical times, forced terminations, periodic tasks, and external logging events. It is also responsible for invoking calls to the TRACE service in order to log task activity, execution time, and tracing information.

Additionally, the task manager keeps track of information that is not usually available to user-space Genode applications monitoring arbitrary processes. This information includes a task iteration counter for periodic tasks and RAM usage monitoring.

Generally, it is the task manager's objective to ensure proper task event logging by registering event trigger callbacks with its children and constantly monitoring their state. Since the task manager also holds all binaries and logging dataspaces, it is a memory-intensive process and should be granted a sufficient amount RAM quota when configuring the Genode run script.

4.5 dom0 TCP Client Module

The dom0 client module is a library provided to the dom0 client program. It simplifies interaction with the server to a few intuitive function calls, enabling easy reprogramming and live scripting of a dom0 session.

As the client counterpart to the TCP server module, it mirrors its functionality by exposing a simple interface featuring commands directed mostly at the task manager. Furthermore, a special profiling data command permits the client to trigger an external event in the task manager's event logger, allowing users of this library to create system performance snapshots at any time.

Its main objective, however, is to transmit task information, including reading and sending generated task descriptions and binaries. The protocol also permits the client to start and stop tasks at any time.

4.6 dom0 SQLite Module

Finally, the SQLite module, as the last module in the feedback loop, receives raw profiling data from the TCP client module via the dom0 program and processes it. Since the profiling data at this point is a pure event log of performance tracing snapshots, refining of the data is adamant for further analysis.

As mentioned earlier, this refinement is implemented on the offline system to avoid additional load-heavy data processing on the relatively slow embedded system and to prevent monitoring operations from compromising realistic performance data.

SQLite is a C library that implements most of the SQL standard but lacks a proper client-server database engine [SQL]. Instead its focus lies on lightweight implementations and embedded databases, meaning that it is ideal for local storage of smaller databases without the need for a dedicated SQL server setup. Also, Python natively supports SQLite databases via the module `sqlite3`. For the purposes of this toolchain SQLite is plenty sufficient although future implementations may want to transition to a more scalable database with a proper client-server structure. Even then, the transition from SQLite to, for example, MySQL is easily achieved by automated converter tools [SQLc].

5 Implementation Details

The dom0 toolchain was mainly implemented in the C++ language, since the majority of it uses the Genode Operating System Framework. Task description generation using tms-sim was also implemented in C++. Most of the offline system's toolchain has been written in Python for ease of use and scripting support, enabling good maintainability, extensibility, and reusability.

All of the code written as part of this thesis is available on the 702nADOS GitHub [Genb]. Some pieces of this code were adapted or modified from previous implementations presented in Avinash Kundaliya's and Josef Stark's works, mostly regarding the dom0 TCP server implementation and modifications of the tms-sim task description generator.

This chapter elaborates on the actual implementation of each individual design component presented in chapter 4. This also includes utilization of the Genode features discussed in chapter 3.

5.1 Task Description Generation

The generation of task descriptions for use in the dom0 task manager and ultimately the SQL database is implemented as an extension to the tms-sim task description generation and schedulability analysis tool [tms-sim].

Specifically, task description generation of periodic load tasks as designed by Avinash Kundaliya has been extended to provide additional features needed for integration with the dom0 toolchain on Genode [Kun15].

Kundaliya defined the `PeriodicLoadTask` class within the tms-sim project. This new class describes a task with periodic system arrival and its corresponding scheduling values. Concretely, the class comprises the following parameters:

ID	The task ID to be used as identification by dom0.
Execution Time	Worst-case execution time. Currently this is still a random value with no significance to the dom0 toolchain, as actual WCET analysis is not yet part of this project.
Critical Time	The maximum time a task is permitted to run before being exited forcefully by the dom0 task manager.
Priority	The task's priority in the scheduler. Genode priorities range from 0 to $2^{16} - 1$. Priority resolution depends on the `prio_levels` argument in the Genode run script.

Period — The task's periodic arrival time. This period is expected to be bigger than both the critical and worst case execution time to ensure even theoretical schedulability.

Offset — The task's initial time offset before periodically arriving at the fixed period above. Since initial task arrival is currently controlled by the dom0 client, this value is ignored by the dom0 task manager itself.

Quota — Genode RAM quota transferred from the task manager to this task. A minimum value of 512 KiB is required to hold Genode child meta data.

Binary — The name of the task's associated binary file. This will later be incorporated into the task's process name.

Config — Config node to be passed to the Genode child component. This corresponds to a component's config node in the Genode run script and can be accessed by the task itself via the Genode environment. Currently, this holds a prototypical command line argument to be passed to the tasks, although not all tasks use its value.

All time values currently generated in task descriptions are given in milliseconds and are randomized at this time, as precise values would require previous testing runs of the complete toolchain, including the machine learning feedback loop. However, critical time and arrival period are already taken into consideration when managing corresponding tasks and may lead to forceful task exiting in case of exceeded deadlines.

Priorities, too, are for the most part randomized at the moment, with the exception of the `idle` task which is always assigned the lowest priority and is the first task in every task set. Subsequent task binaries are randomized at equal probabilities to one of the following: `hey`, `namaste`, or `tumatmul`, with `tumatmul` being the only load-producing task. Generally speaking, task description generation is currently mainly employed as a randomized testing environment for the developed toolchain.

The modified tms-sim generator is built by running the `build-local.sh` script and executing make, e.g., as done by the central dom0 Makefile's `tms` target:

```
$ mkdir -p tms-sim-2014-12/build
$ cd tms-sim-2014-12/build
$ ../build-local.sh
$ make
$ make install
```

The built generator executable may subsequently be used for description generation of periodic load tasks. The `-n` argument indicates how many tasks are to be generated, with the first task always being the idle task, and the `-o` argument specifies the output file for the generated XML task description:

```
$ tms-sim-2014-12/build/bin/generator -o dom0_client/tasks.xml -n 2
```

An example output of this generator run with 2 generated task descriptions can be seen in Figure 5.1. Once task descriptions are generated, the tms-sim tool takes no further part in the dom0 toolchain until new task descriptions are needed.

In the listed XML file there are two additional values listed that have not been mentioned previously. `ucfirmrt` and `uawmean` are internal values used by the tms-sim simulation engine to calculate scheduler utilization. Currently, they hold no value for the dom0 toolchain and are silently ignored.

5.2 TCP/IP Server

The dom0 TCP/IP server is the central communication module on the online embedded Genode system. As previously seen in Figure 4.1, it provides an interface to external client applications that are not part of the Genode system.

5.2.1 Implementation

The TCP/IP server follows the simple loop design shown in Figure 5.2. As long as there is no client connection, the server waits for incoming connections. Once a connection has been established, the server simply serves the client by answering its requests and forwarding them to the task manager via RPC calls.

The server uses lwIP sockets which feature an API very similar to standard Unix Berkeley sockets. Using lwIP, the server creates a listening socket, binds it to its network address and port, and finally listens to that socket. Subsequently, the server blocks until an incoming connection arrives, causing the server to accept the connection and assigning it to a new socket.

Now, the server enters the main client-specific serving loop in which the server lies mostly idle until a command is received. The first 4 bytes of this command indicate its type. Depending on the command defined in the protocol, more data may be expected of varying size.

Invalid command tags are simply ignored and the serving loop continues with a warning printed to the console. Should the connection be interrupted at any time, macros detect the session failure and force the server to break out of the inner client loop, ultimately waiting for new connections to arrive.

The dom0 TCP/IP server is started using a low to moderate RAM quota after Genode startup according to the dom0 run script. The task manager and TCP/IP server are the only custom Genode services started at boot time, albeit Ethernet device drivers and a few other services including timers and tracing functionality need to be explicitly started as well.

5.2.2 Protocol

The protocol implemented on the dom0 server and client is rather simple. As mentioned previously, the first four bytes of each command message indicate its type. These

```
<?xml version="1.0" encoding="UTF-8"?>
<!--Hier kommen die Tasks-->
<taskset xmlns="http://www.tmsxmlns.com" xmlns:xsi="http://www.w3.org/2001/
    ↪ XMLSchema-instance" xsi:schemaLocation="http://www.tmsxmlns.com␣taskset.
    ↪ xsd">

  <periodictask>
    <id>1</id>
    <executiontime>0</executiontime>
    <criticaltime>0</criticaltime>
    <ucfirmrt/>
    <uawmean>
      <size>10</size>
    </uawmean>
    <priority>4</priority>
    <period>0</period>
    <offset>0</offset>
    <quota>512K</quota>
    <pkg>idle</pkg>
    <config>
      <arg1>102165</arg1>
    </config>
  </periodictask>

  <periodictask>
    <id>2</id>
    <executiontime>5155</executiontime>
    <criticaltime>1155</criticaltime>
    <ucfirmrt/>
    <uawmean>
      <size>10</size>
    </uawmean>
    <priority>2</priority>
    <period>6341</period>
    <offset>0</offset>
    <quota>8M</quota>
    <pkg>tumatmul</pkg>
    <config>
      <arg1>127352</arg1>
    </config>
  </periodictask>

</taskset>
```

Figure 5.1: An example task description set generated by tms-sim.

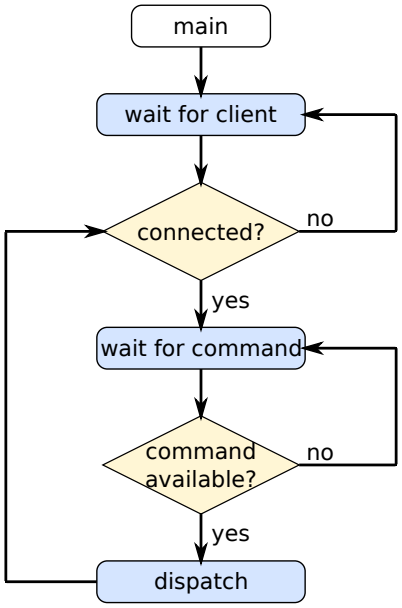


Figure 5.2: dom0 Server Loop.

four bytes are random numbers bound to C++ macros and defined in the header file `communication_magic_numbers.h`. All of the protocol message tags are briefly described here.

`SEND_DESCS` This tag indicates that XML task descriptions generated by the tms-sim were sent by the client and are now available at the socket. The next four bytes in the packet indicate the size of the XML payload in bytes, including the terminating null character. Accordingly, after reading tag and size of this message, the server reserves a shared Genode dataspace and fills it with the content of the XML file available at the socket. This dataspace is only temporary and is destroyed once the task manager has allocated corresponding tasks.

`CLEAR` When receiving this tag, the server issues a command to the task manager to stop and clear all tasks available to it. This also removes previously received task descriptions, since they are stored directly in the task objects managed by the task manager. Essentially, the task manager returns to its initial state.

`SEND_BINARIES` This tag signifies that the client intends to send binaries to the server. The first four bytes following the tag indicate the total number of binaries to be sent. Following a `GO_SEND` command issued by the server, each binary is then received individually in two parts: the first four bytes declare the binary size in bytes, similar to the `SEND_DESCS` protocol. The second part contains the actual binary and is expected to be exactly of the size that was just communicated.

Before reading the actual binary from the client socket, the server requests a dataspace from the task manager in which to store the binary. This way, the server's RAM usage is reduced to a minimum and most of the RAM-heavy storage is allocated by the task manager. Subsequently, the server directly reads the binary data from the socket into the task manager dataspace.
The `GO_SEND` tag is introduced as a safety measure since binary sizes are expected to be reasonably big and transmitting them over the network may take some time. Depending on TCP socket buffer size, sending multiple binaries at once might exceed the socket's capabilities.

`START` The `START` tag is a signal to the server to issue the task manager to start all remaining tasks that have not been started yet or have been paused explicitly. The task manager subsequently ignores any running tasks and only starts tasks corresponding to descriptions that are not running yet. Tasks with a period larger than 0 will run periodically at the given periodic and interrupted at their critical times.

`STOP` This tag forces the task manager to stop all running tasks and pause their periodic timers. Errors and exceptions are expected on the console whenever tasks are forcefully interrupted externally.

`GET_PROFILE` By issuing this command tag, the client requests all profiling data available since the last request or the start of the task manager respectively. The server handles the command by requesting the entire monitoring event log as an XML file from the task manager and forwards it to the client. Again, the server first sends 4 bytes indicating the size of the XML file, followed by the file itself.

5.2.3 Configuration

The TCP/IP server can be configured via the corresponding config node of the dom0 service within the Genode run script. While most configuration values have functional default values, some need to be explicitly stated. The `server` XML node within dom0's `config` node provides the attributes listed in Table 5.1.

5.3 TCP/IP Client

On the other end of the dom0 toolchain, on the offline system's side, the client mirrors the communication protocol defined by the server. For ease of use, the client software has been written in Python and is provided as a module to be imported in scripts and source code.

The `dom0_client` module provides the `Dom0_session` class which encapsulates

Table 5.1: dom0 server configuration.

Attribute	Description
`dhcp`	`"yes"` enables support for dynamic IPs, requiring a NAT router within the network to assign an IP address to the server. The default `"no"` requires a static IP address to be specified as the `listen-address` attribute.
`listen-address`	If DHCP support is disabled, a valid IPv4 address is required for this attribute. This defaults to `0.0.0.0` and must therefore be specified explicitly. The attribute is ignored if DHCP is enabled.
`port`	This specifies the port on which the server listens for new clients. By default this port is 3001.
`network-mask`	The network mask for the current subnetwork. Most of the time this will be `255.255.255.0`, which is also the default value for this attribute.
`network-gateway`	The default gateway for the local network. This defaults to `192.168.0.254`.

TCP/IP communication with the Genode dom0 server. Its usage is simple and designed for both scripting and automated software use. The `help()` function displays available member calls and their default values.

On creation, the `Dom0_session` instantly tries to connect to the dom0 server via the given address and port or defaults to `192.168.0.14:3001` if none are provided. Additionally, the object constructor reads task descriptions from an XML file that defaults to `tasks.xml`. Python sockets make connection handling very intuitive and for the most part self-explanatory, so this part of the implementation will not be discussed here any further.

When reading task descriptions, the Python module parses the `pkg` node of all tasks using regular expressions and stores their values for later transmission. After the XML has been read and the client has connected to the server, information may be transmitted.

The `Dom0_session` class provides functions for transmitting both task descriptions and binaries to the server. In both cases, the client simply follows the protocol defined earlier, preceding all data with a 4 byte size value and waiting for preparation confirmations for each binary.

In addition, the session interface provides access to the simple protocol commands `CLEAR`, `START`, and `STOP` by exposing similarly named functions.

Finally, the TCP/IP client module offers the ability to request profiling data in the form of the task manager's XML event log and store it in a file. This will later be used for further refinement by the Python dom0 SQLite module.

5.4 Task Manager

The dom0 task manager is the central task allocation and control instance within the dom0 toolchain. It is easily the most complex and extensive part developed in this thesis, making use of all Genode concepts presented in chapter 3. Its function is to manage binaries and task descriptions, allocate tasks as children in the Genode system, supervise their lifetime including deadline enforcement, and monitor them in the form of an event log.

5.4.1 Genode RPC Server

As per design, the task manager is intended as a low-throughput process that executes rarely and only acts on events and requests. This makes it ideal for the implementation as a Genode RPC server as described in chapter 3.

As a Genode RPC server, the task manager runs relatively infrequently. Its entrypoint is triggered by two distinct classes of events.

First, direct requests by the RPC client, i.e., the TCP/IP server, let it enter the task manager's protection domain in order to pass and receive arguments. In this case the task manager is a straightforward RPC server, providing the fixed interface defined by its `Session` as shown in Figure 5.3. With the exception of the `binary_ds()` function, this session interface closely mimics the TCP/IP server protocol, as most calls are simply request forwards from the client to the task manager via the TCP/IP server.

```
struct Task_manager_session : Genode::Session
{
    static const char *service_name() { return "task-manager"; }

    virtual void add_tasks(Genode::Ram_dataspace_capability xml_ds_cap) = 0;
    virtual void clear_tasks() = 0;
    virtual Genode::Ram_dataspace_capability binary_ds(
        ↪ Genode::Ram_dataspace_capability name_ds_cap, size_t size) = 0;
    virtual void start() = 0;
    virtual void stop() = 0;
    virtual Genode::Ram_dataspace_capability profile_data() = 0;

    [...]
};
```

Figure 5.3: The task manager session interface.

Secondly, the task manager's entrypoint is also triggered by its own timers managing periodic task arrival and critical time interruption handling. However, since an entrypoint represents a single thread, both types of triggers are processed synchronously, eliminating the need for locks and semaphores with insignificantly slower reaction times as a drawback.

In order for the task manager to act as an RPC server, it requires implementations of the interfaces and classes presented in chapter 3. Accordingly, the task manager provides a `Session`, `Client`, and `Connection` for the client side, and a `Session_component`, and corresponding `Root_component` on the server side.

The `Session` class, as mentioned above, simply mimics the TCP/IP server's requests, providing access to several task management control commands including adding, clearing, starting and stopping tasks, exposing monitoring data, and providing dataspaces for received binaries.

As described in chapter 3, the `Client` class simply forwards client-side function calls to their equivalent RPC capability invocations of the Session interface. One such example is shown in Figure 5.4. The task manager's `Connection` class then simply wraps the `Client` class by inheriting Genode's default `Connection` class with the task manager's `Client` as a template argument.

```
void add_tasks(Genode::Ram_dataspace_capability xml_ds_cap)
{
    call<Rpc_add_tasks>(xml_ds_cap);
}
```

Figure 5.4: Client RPC forwarding.

Server-side, the task manager's `Session_component` defines all of its functional behavior and data required for proper task management. Most importantly, this comprises a struct of shared data that is common to all tasks, including a list of all tasks themselves as seen in Figure 5.5. Task instances are represented by the `Task` class. The other members of the `Shared_data` struct are discussed later in this thesis.

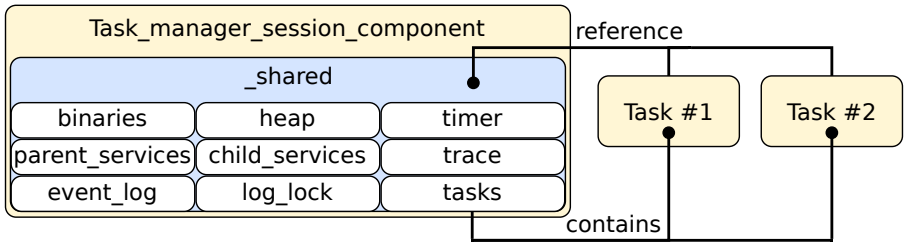


Figure 5.5: Task manager object ownership.

The task manager's `Session_component` also manages config readouts and the conversion of profiling data to XML to be passed to the TCP/IP server. Most of the task management logic, however, is implemented in the `Task` class itself.

Finally, the `Root_component` simply encapsulates and initializes the task manager's `Session_component` for ease of use. Initialization of the root component, in turn, is managed by Genode's `server` library. This library defines the main entrypoint to the RPC server, including its `main()` function, and processes incoming signals received at its entrypoint.

As the task manager is expected to be online and available throughout the Genode system's lifetime, the task manager RPC server is started together with the TCP/IP server on startup, as defined in Genode's dom0 run script. Since the task manager holds task descriptions, binaries, and monitoring data, and also needs to transfer RAM quota to these tasks, the RAM quota for the task manager should be defined sufficiently large. Currently, this quota is defined to be 40MiB.

5.4.2 Shared Dataspaces

As seen previously in Figure 5.3, the task manager session interface defines several functions that make use of dataspace capabilities. These dataspaces are used for communicating data of variable sizes through the RPC channel. Specifically, dataspaces are employed by the task manager session interface to send character strings including XML files, and task binaries.

In the case of the two relevant XML files, namely task descriptions and monitoring information, the dataspaces contain plain character strings. The size of these strings may vary greatly, depending on the size of the task descriptions and the monitoring event log respectively. The binary spaces required for task binaries are purely raw byte information and may also be of varying sizes, depending on each individual task.

In order to create a shared dataspace, it must first be allocated using the local RAM session. Using the component's region manager session, this dataspace must subsequently be mapped to a locally available address by attaching the dataspace. Preferably, this attachment should be made undone on dataspace destruction.

Fortunately, since this task is frequently required in Genode components, Genode provides the `Attached_ram_dataspace` class that encapsulates the process of allocating and mapping a dataspace to a local address, greatly simplifying its usage. From this attached dataspace, capabilities can easily be requested and passed via RPC calls to other components as either arguments or return values.

Once dataspace capabilities are received on either the client or the server of the RPC call, the corresponding dataspaces need to be mapped to local addresses. Again, this is achieved using the region manager which returns a local address. Casting this address to the appropriate type makes it ready for conventional use again.

5.4.3 Task Allocation

Tasks are encapsulated and managed by the `Task` class in a one-to-one relationship. On object construction, XML task descriptions received by the TCP/IP server and segmented by the task manager are parsed and stored as native types. Additionally, the task's binary name and ID are combined into the task name which will later be used by Genode as the corresponding component's session label. Example task names are `01.idle` or `03.tumatmul`.

Task object construction also initializes the local signal dispatchers used for periodic timer callbacks, as described in chapter 3. Specifically, there are three dispatchers: start,

kill, and idle. The start dispatcher is associated with a `Task` member function that initializes and runs a child component, leading to the actual execution of a task. The kill dispatcher is registered to a function that force-exits the running child component or ignores it in case of it already being idle. Lastly, the idle dispatcher is used as a dummy dispatcher since running timers cannot be stopped. Instead, their registered dispatchers are changed to one that has no effect on the execution of any of the tasks. All three dispatchers are associated with the task manager's entrypoint, leading to synchronous management of both RPC server calls and timer callbacks.

After construction, the task lies idle and no executions are triggered, waiting for the `run()` command issued by the task manager which in turn originates from the TCP/IP server by forwarding the `START` message tag.

Once the command to start a task is received, the task is either started right away if its period is defined to be 0, or triggered via a periodic timer. Timers are registered to their corresponding signal dispatchers: the start timer is registered to the start dispatcher, and the critical timer is registered to the kill dispatcher. Periodic timers in Genode always trigger instantly when started initially, so that the task will be started as soon as the task manager's entrypoint is free for signal handling. An example illustrating a possible synchronous execution order of two periodic tasks on the task manager's entrypoint is shown in Figure 5.6.

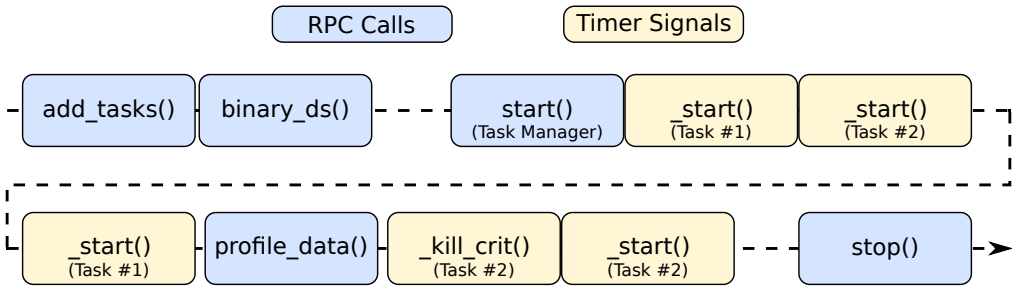


Figure 5.6: Task manager's synchronous timer signal and RPC handling.

Eventually, the task is started. This is achieved in the `_start()` function by creating a Genode child component and subsequently activating its entrypoint. Additionally, the kill timer is started with the critical time specified in the task description as deadline. In case the task does not exit in time, the kill timer triggers the `_kill_crit()` function, force-exiting the child.

Creating the Genode child component requires some setup, as discussed in chapter 3. In order to easily manage the lifetime of a child iteration, all objects and sessions resulting from the setup of a child are stored in the task's meta data structure `Meta_ex`. This structure holds and initializes all sessions required for creating a Genode child component, namely RAM, ROM, region manager, and protection domain sessions. Additionally, it holds a `Server` object that is used to announce and register services potentially provided by the child. The task meta data also contains the actual Genode `Child` class and its corresponding policy that defines component behavior with respect

to service handling, process labelling, child configuration, and the actual task binary.

Once setup is complete and a task's meta data has been allocated on the heap, the child is executed by activating its entrypoint. The Genode system now automatically handles the newly created child.

5.4.4 Task Policies

Genode offers the possibility to define so called *child policies* when creating new children. These child policies implement interfaces inherited from the Genode `Child_policy` parent class. Specifically, a child policy provides callback prototypes for service requests and announcements, and the exit routine of a child. This way, a parent component can decide whether and how to route service requests through the system and what to do once a child component exits.

In the case of the presented dom0 task manager, the task's child policies are as open as possible. The task manager keeps registries of both child and parent services. As dom0 task policies are designed to keep a pointer to their corresponding `Task` object, they also have access to the tasks' shared data, including service registries, as previously illustrated in Figure 5.5.

Genode service requests may also include queries to a config or binary file. Both of these queries require a specialized ROM subpolicy that answers the request with a valid service. Fortunately, these ROM policies do not necessarily require a ROM dataspace on construction. Instead, they simply take a general dataspace capability that is associated with a config or binary file respectively. Generally speaking, Genode does not always require ROMs to be loaded at boot time but rather expects them to be mostly constant and read-only throughout their lifetime. The ROM service on the other hand only registers binary ROMs that are available at boot time.

To that effect, task policies may simply be constructed on top of dynamic dataspaces, such as the attached RAM dataspaces created by the task manager when receiving task binaries and descriptions containing their config. Accordingly, a task's policy and its ROM subpolicies are initialized with capabilities of their corresponding binary and XML config dataspaces respectively.

Eventually, task service requests are resolved in an if-then-else construct akin to a switch statement, as illustrated in figure Figure 5.7. If the request is a config or binary query, the respective ROM policy service is returned. Otherwise, the service is looked up in the parent and child service registries. In case the requested service cannot be found at all, the request blocks until it becomes available in the child registry.

Child services are registered with the shared child service registry whenever they become available. Similarly, they are removed from the registry when they are no longer provided, i.e., when the child is destructed. Parent services, however, are registered in a constant registry that provides all core services.

Child policy functions are usually called from the child's thread, making interaction with data from the task manager's entrypoint difficult. Fortunately, Genode service

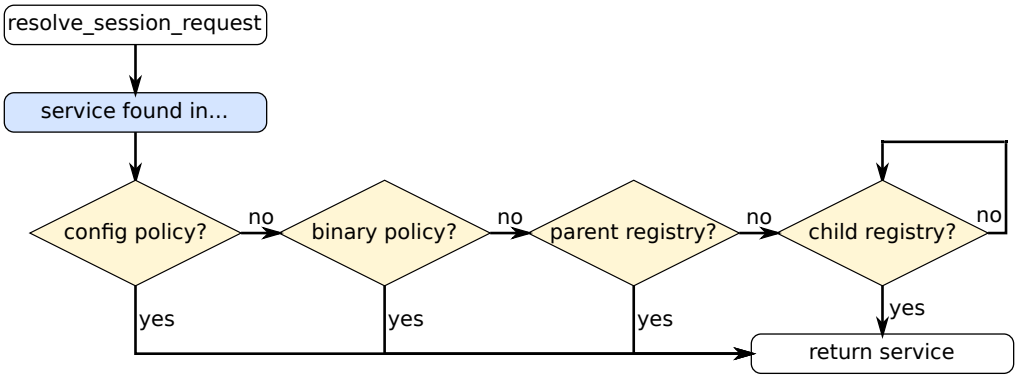


Figure 5.7: Task policy request resolving.

registries are designed to be thread safe and thus require no special handling. Force-exiting by and callbacks to the task manager's thread, however, need special attention.

5.4.5 Multi-Threaded Task Lifetime Management

Genode child components are not automatically destroyed once they exit, leaving them in a zombie state. Accordingly, RAM quotas and sessions still persist after their expiration. Over time, especially with periodic load tasks, this creates unnecessary resource wastage in the form of memory leaks. In order to avoid this, child components have to be monitored for termination and their meta data has to be deleted explicitly.

Child policies offer a way to be notified when a task exits. The policy's `exit()` function is called with its corresponding return code whenever a child component's `main()` function returns. However, destructing the child from within this function would lead to unpredictable results, as it is called from the child's thread. Deleting a thread from within the thread itself is undefined behavior, except for the case where the deletion of the thread coincides with the very last operation of said thread. There is no way to guarantee that the `exit()` function is the last operation in that thread, though. Accordingly, another approach has been implemented here.

Once a task exits, the dom0 child policy submits the child for destruction on a separate thread of the task manager process. This `Child_destructor_thread` is global to the task manager and manages a list of child components that are queued for destruction. The list itself is accompanied by a mutex lock, ensuring that no race conditions occur during the submission or deletion of children, allowing thread safe management of zombie tasks. Meanwhile, the child destructor thread runs in an endless loop, repeatedly activating the list lock and destroying queued child components, eventually releasing the lock again. In order to avoid heavy CPU utilization, the child destructor thread loops only once every 10 milliseconds.

This multi-threaded approach to zombie task purging adds the additional benefit of being able to kill running tasks from a different thread. This happens at two different occasions.

First, when a task's critical time is reached, the kill timer triggers a signal that aims to stop the running task immediately. Accordingly, the kill dispatcher simply calls the task policy's `exit()` function from the task manager's thread with a dedicated return value to indicate a critical exit event. Another policy-wide mutex lock ensures that this does not coincide with a different exit event.

Second, the dom0 client has the ability to stop all running tasks remotely. This is a hard stop and forces all tasks to exit immediately and all timers to be paused – or rather be registered to the idle dispatcher. This *external* trigger passes a different code to the task policy's exit function in order to distinguish it from regular, erroneous or critical exits.

In combination, these features enable dynamic and robust management of task lifetimes.

5.4.6 Configuration

The task manager, similar to the TCP/IP server, features several configuration options. Apart from the standard Genode RAM quota specification in the dom0 run script, the task manager provides configuration entries for some of its sessions and dataspaces, as listed in Table 5.2.

Table 5.2: dom0 task manager configuration.

Node / Attribute	Description
`trace quota`	The RAM quota reserved for the TRACE service connection. This influences how many processes can be traced actively or passively at a given time.
`trace buf-size`	The RAM quota reserved for the trace buffer. Currently, the trace buffer is not used, as RPC calls and signals do not need to be traced. Accordingly, this size may be relatively small.
`profile ds-size`	The maximum size of the XML event log. To avoid constant reallocation of the event log buffer, a static buffer of a fixed size is used. Depending on the number and duration of processes being logged, this size should be defined moderately high. A smaller event log buffer may be used if the client requests it more often, as the event log is flushed on every request.

5.5 Collection of Profiling Data

The dom0 task manager presented so far merely manages the allocation, execution, termination, and destruction of tasks. However, in order to provide data that can be used for schedulability analysis, the managed tasks also need to be monitored. Several parameters have been considered with respect to their merit to future analysis and the chosen approach to gather them is presented in this section.

5.5.1 Trace Subject Information

Most information about Genode processes can be collected using Core's TRACE service. As discussed in chapter 3, Genode supports active and passive tracing of processes. While there was no need for RPC or signalling hooks using a tracing policy, or active tracing involving explicit logging to a trace buffer, TRACE's ability to provide information about even untraced processes was very helpful.

TRACE offers an interface to enumerate every single thread of a running Genode instance. These threads are called *trace subjects* in the context of a TRACE connection. For each trace subject, its position in the Genode component tree is defined by its session label and thread name.

Even untraced subjects provide potentially interesting data for runtime performance analysis. With the exception of CPU affinity, all passive tracing information as mentioned in chapter 3 is logged: session and thread label, tracing state, and scheduler execution time. CPU affinity is only important on multicore processors which were not part of the testing environment, although support for the logging of CPU affinity is easily added to the existing task manager. Each trace subject is also accompanied by a trace subject ID. However, this ID is only used to identify a trace subject within one TRACE query and holds no significance over multiple requests.

The monitoring event log naturally consists of a list of events, with each event comprising a complete snapshot of the trace subject information of every single Genode process. Components that are managed by the task manager furthermore provide additional information as will be described later in this section.

Events are triggered by multiple sources and flagged accordingly. All currently implemented event types are listed in table Table 5.3

Since events may be triggered both by the task manager's and by any of its children's threads, event logging is guarded by a mutex lock, ensuring absolute thread safety. Event logging can be triggered at any time by calling the static `log_profile_data()` member function with an event type enum, a triggering task ID, and a pointer to the task manager's shared data structure. This allows for easy extension of events that are yet to be implemented.

Table 5.3: Monitoring event types.

Event Type	Description
`START`	The `START` event is triggered each time a task is actually started by activating its entrypoint for the first time. The event's task ID corresponds to the starting task. Depending on scheduling, the started task may or may not already appear in the tracing subject list.
`EXIT`	Whenever a task exits regularly with exit code `0`, its child policy causes the `EXIT` event to be logged. Again, the accompanying event task ID belongs to the exiting task. Also, this event's trace subject list always includes the terminating task one last time.
`EXIT_CRITICAL`	When a task's critical time has been reached but the task has not yet terminated regularly, its kill timer forces a critical exit. This event indicates that the associated task did not finish in time and was forcefully exited by the task manager.
`EXIT_ERROR`	`EXIT_ERROR` indicates that a task terminated with an error code other than `0`. Further debugging or error information must be read from the Genode console.
`EXIT_EXTERNAL`	This event is logged for each running task whenever the task manager is requested to stop. All tasks that have not terminated yet are forcefully stopped and trigger individual `EXIT_EXTERNAL` events.
`EXTERNAL`	The `EXTERNAL` event indicates an external trigger. Currently, this occurs only when the existing event log is requested and is triggered right before converting the event log to an XML file, allowing for arbitrarily timed monitoring snapshots of the entire system. This is the only event type that does not hold a valid task ID, as it is not triggered by a task. Instead its task ID is set to `-1`.

5.5.2 Scheduler Idle Time

Currently, Genode offers no platform-independent way of querying scheduler idle time. Accurate data would require manual implementation of kernel-specific routines to request idle time directly from the scheduler, integration thereof into Genode's Core component, and the exposure of said functionality to user-space monitoring processes, for example by extending the TRACE session interface.

The implemented dom0 toolchain provides a different method of scheduler idle time monitoring. With no special implementation required in the dom0 task manager, the task description generator is simply extended by a non-periodic idle task. This task is assigned the lowest priority possible, granting all other Genode processes with a custom priority precedence over it. Most processes started by the Genode system are assigned a default priority which corresponds to the platform-specific maximum priority, resulting in the idle task being scheduled only when there is actual idle time available on the scheduler.

Additionally, the idle task is never actually idle. It finds itself in a constant loop, taking up as much CPU time as it can get. This way, the idle task's execution time is an adequate, albeit not entirely accurate, representation of the idle time available on the scheduler. It may not be completely accurate, as scheduling an additional dummy task causes context switching that would otherwise not impact the system. Furthermore, depending on the scheduling algorithm, the idle task may take up more time than what would be idle time without it. Fair schedulers like the one implemented in Genode's base-hw platform might assign low priority tasks time slices that would otherwise be given to higher priority tasks only. Nonetheless, this method still provides highly indicative data on CPU time utilization.

5.5.3 Managed Task Data

Tasks that are managed by the task manager, i.e. started and controlled, are labelled as *managed*. The task manager provides additional information on managed tasks, as their Core service sessions are available to the dom0 system and their lifetimes are actively monitored.

Accordingly, monitoring events are extended by further information on managed tasks. When enumerating trace subjects, the event logger looks up each subject in the task manager's list of managed tasks. In case a corresponding subject matches the name of a task started by the manager, the trace subject is flagged as managed. An additional check is implemented to avoid the logging of zombie tasks that are scheduled for destruction, as their final monitoring values are already logged by their `EXIT` events.

The task manager component itself is also flagged as managed, as the task manager naturally has access to its own Core service sessions as well.

Ultimately, managed task monitoring snapshots are extended by their current RAM usage and capacity. Genode's RAM connections provide the ability to query these values, justifying the previous design decision to store child component's RAM sessions in their

meta data structure.

Additionally, managed tasks have their iterations counted. In case of the task manager, this value is always 0. Nonperiodic tasks merely count their manual executions triggered by the task manager, starting at 1. For example, starting, stopping, and starting the nonperiodic idle process again will result in an iteration counter of 2. Periodic tasks increment their iteration counter each time they are executed by the periodic timer, making it possible to uniquely identify periodic task cycles.

5.5.4 Communication of Collected Data

The monitoring event log is initially stored as a list of `Event` structures, allocated on the task manager's heap. These structures store an event type as listed in Table 5.3, a triggering task ID, a time stamp queried from the task manager's shared system timer, and a list of `Task_info` structures. A `Task_info`, in turn, holds all its passive tracing information as listed previously, extended by a boolean `managed` flag. In case this managed flag is set to true, according to the criteria mentioned in the previous section, the `Managed_info` substructure is filled with valid data, i.e., task description ID, RAM usage and capacity, and the aforementioned iteration counter.

Once the TCP/IP server receives a request for this event log, the task manager converts it into a human-readable XML file using Genode's recursive XML generator. This converted event log is a direct representation of the list of `Event` structures, converting member variables to attributes and substructures to nodes, with the `managed-task` node only added to managed tasks. An example event in its XML representation can be seen in Figure 5.8, although not all system components are listed.

The XML event log is prefaced by a list of currently managed tasks, mapping managed task IDs from the event log to the most important attributes of their task descriptions. An example list can be seen in Figure 5.9.

Ultimately, the dom0 TCP/IP server forwards this XML document as is to the TCP/IP client where it can be refined further.

5.5.5 Conversion to SQLite Database

While the aforementioned XML document contains all information needed for further analysis in a human-readable format, it is relatively difficult to programmatically extract and process data contained in the document. For this reason, the `dom0_sql.py` Python module exposes a function that converts this XML document into an offline SQLite database, stored in an arbitrary target file.

The SQLite Python module creates three separate tables from the provided data: the first table corresponds to the XML preface data, filling the table with task descriptions using their ID as the primary key. Additionally, Genode process names are reconstructed here using task ID and binary name. A resulting example table can be seen in Figure 5.10.

The second table is a list of all events, identified and keyed using their position in the event log which is expected to be sorted by time stamp. As a foreign key, these events

```
<event type="START" task-id="3" time-stamp="47411">
    <task id="28" session="init␣->␣task-manager␣->␣03.tumatmul" thread="03.
        ↪ tumatmul" state="UNTRACED" managed="yes" execution-time="0">
        <managed-task id="3" quota="8388608" used="167936" iteration="1"/>
    </task>
    [...]
    <task id="26" session="init" thread="init" state="UNTRACED" managed="no"
        ↪ execution-time="100987"/>
    [...]
    <task id="9" session="init␣->␣task-manager" thread="task-manager" state="
        ↪ UNTRACED" managed="yes" execution-time="101490">
        <managed-task id="0" quota="22650880" used="5148672" iteration="0"/>
    </task>
    [...]
    <task id="0" session="init␣->␣task-manager␣->␣01.idle" thread="01.idle"
        ↪ state="UNTRACED" managed="yes" execution-time="0">
        <managed-task id="1" quota="8388608" used="167936" iteration="1"/>
    </task>
</event>
```

Figure 5.8: Excerpt from a monitoring event in its XML representation.

```
<task-descriptions>
    <task id="0" execution-time="0" critical-time="0" priority="0" period="0"
        ↪ offset="0" quota="40521728" binary="task-manager"/>
    <task id="1" execution-time="0" critical-time="0" priority="57344" period="0
        ↪ " offset="0" quota="8388608" binary="idle"/>
    <task id="2" execution-time="3706" critical-time="3706" priority="24576"
        ↪ period="4682" offset="0" quota="524288" binary="hey"/>
    <task id="3" execution-time="5929" critical-time="5929" priority="8192"
        ↪ period="4418" offset="0" quota="8388608" binary="tumatmul"/>
</task-descriptions>
```

Figure 5.9: List of managed tasks in their XML representation.

	id	name	critical_time	priority	period	quota	binary
1	0	task-manager	0	0	0	40521728	task-manager
2	1	01.idle	0	57344	0	8388608	idle
3	2	02.tumatmul	859	32768	3470	8388608	tumatmul
4	3	03.namaste	3207	8192	3740	524288	namaste

Figure 5.10: Tasks SQLite table.

optionally feature the triggering task's ID in the task table, as seen in Figure 5.11.

	id	time_stamp	type	task_id
1	0	38158	START	1
2	1	38218	START	2
3	2	38269	START	3
4	3	38382	EXIT	3
5	4	39040	EXIT_CRITICA	2
6	5	41686	START	2
7	6	41958	START	3
8	7	42070	EXIT	3
9	8	42506	EXIT_CRITICA	2
10	9	45155	START	2
11	10	45698	START	3
12	11	45809	EXIT	3
13	12	45974	EXIT_CRITICA	2
14	13	48091	EXIT_EXTERN/	1
15	14	49087	EXTERNAL	

Figure 5.11: Events SQLite table.

Lastly, the snapshot table contains a list of all task snapshots, i.e., their tracing and managed task data, referencing both the monitored task and the corresponding event as foreign keys. An example table is shown in Figure 5.12.

	task_id	event_id	execution_tir	quota	used	iteration
1	1	0	0	8388608	167936	1
2	0	0	263949	31907840	5271552	0
3	2	1	0	8388608	167936	1
4	1	1	0	8388608	167936	1
5	0	1	274204	23347200	5275648	0
6	3	2	0	524288	167936	1
7	2	2	0	8388608	167936	1
8	1	2	0	8388608	167936	1
9	0	2	284387	22650880	5279744	0
10	3	3	23865	380928	356352	1
11	2	3	0	8388608	167936	1
12	1	3	0	8388608	167936	1
13	0	3	285588	22654976	5279744	0
14	2	4	386031	8224768	3543040	1
15	1	4	0	8388608	167936	1
16	0	4	290457	23355392	5275648	0
17	2	5	0	8388608	167936	2

Figure 5.12: Excerpt of a Snapshots SQLite table.

This database structure allows for a variety of useful SQL queries that provide data on the system's performance. Examples of such queries are discussed in chapter 7.

5.6 Sample Tasks

In order to test the system during development, four simple tasks have been developed that are used to feed the task manager. Task names and function are relics of Avinash Kundaliya's master's thesis on the L4 runtime environment.

The tasks `hey` and `namaste` are minimalistic programs that simply announce their own name to the command line before exiting with a return value of 0. They feature no load-heavy tasks and thus rarely miss their critical deadlines in the testing environment.

The `idle` task, as described previously, is a simple `while (true)` loop that never exits. Depending on the compiler flags, special care must be taken to avoid excessive optimization that might reduce CPU load. Since this task's priority is set to be as low as possible, it is expected to only execute when there is CPU time to spare, giving information about scheduler idle time through its execution time.

The `tumatmul` task is the only task producing actual CPU load. Its objective is to calculate all prime numbers up to a certain maximum and store them in a list on the heap. The maximum number is determined by the `arg1` argument in the `config` node of its task description. The employed algorithm is the inefficient *Sieve of Erathosthenes* of complexity $O(n \log \log n)$ and takes about three seconds to calculate all primes up to numbers of the order of magnitude of about 10^5. When it is done, it announces the last five primes of its list to the console.

All of these tasks are linked dynamically by adding `libc` or a different dynamic library to their dependencies. This ensures that binary sizes are reasonably small by avoiding redundancy involved in the repeated linking of static libraries. The dynamic linker is loaded statically by the task manager on creation using a ROM connection.

6 Limitations and Design Decisions

Throughout the conception and implementation of the dom0 toolchain several hurdles were encountered, stemming both from design decisions and framework limitations. This chapter aims to elaborate on a few of the major decisions made in this project.

6.1 Child Execution

As Genode aims to provide support for full desktop operating systems built on top of its framework, it naturally features components that facilitate the allocation and execution of new tasks. Namely, there are three stock applications that provide the ability to start tasks in a simpler fashion than presented in this paper.

First, the `Launchpad` class used as part of the `demo` application provides features for easy child component allocation. Similar to the custom dom0 task manager implementation, it handles child destruction using a separate thread, creates children by transferring quota and managing resource service sessions, and forwards services using a parent and a child service registry. `Launchpad`, however, can only start binaries that are loaded at boot time and stored in the global ROM registry. Still, this could be remedied by extending `Launchpad` to accept an optional binary dataspace capability on child creation but this would require the modification of Genode code.

`Launchpad` also does not keep track of Core service sessions created for the child; it merely stores their capabilities. The child's RAM session is important for the dom0 task manager as it is used to query RAM usage and capacity and the actual session interface cannot easily be reconstructed from its capability alone. Accordingly, the decision was made to keep the original service connection in the task manager instead of its capability.

The dom0 task manager also employs a custom child policy in order to route services and trigger events on exit. Realizing this on the `Launchpad` would require further modification and circular dependencies between `Launchpad` and the dom0 task manager. Therefore, a custom implementation was chosen over `Launchpad`.

Second and third, the `Launcher` and `Loader` services provided by Genode's `gems` and `os` repository respectively feature interfaces to allocate new processes dynamically. However, both of these are intended for graphical applications on a GUI operating system, making use of `Nitpicker`. `Nitpicker` is a GUI manager service that provides a framebuffer and input handling, both of which would be unnecessary bloat for the dom0 task manager design. The `Launcher` application is mainly intended as a start menu incorporating an information panel, akin to the Unity, XFCE, or Windows panel.

Ultimately, the decision was made to equip the dom0 task manager with a custom implementation of task allocation and lifetime management to provide the maximal possible amount of flexibility and extensibility while at the same time keeping the feature set minimalistic.

6.2 Process Execution Time

Genode's TRACE service supports the querying of execution time information of any task, whether actively traced or not. However, the platform-specific basis for these queries is only implemented on the NOVA platform by default. Other platforms, like base-hw or the featured base-foc, simply return `0` instead.

Accordingly, a workaround or fix had to be found for the Fiasco.OC target platform. Fortunately, implementing process execution time queries on Fiasco.OC was rather simple, as the kernel directly supports these requests. Ultimately, Genode's native `platform_thread` code for the Fiasco.OC kernel had to be extended by the snippet listed in Figure 6.1.

```
unsigned long long Platform_thread::execution_time() const
{
    unsigned long long time = 0;

    if (_utcb) {
        l4_thread_stats_time(_thread.local.dst());
        time = *(l4_kernel_clock_t*)&l4_utcb_mr()->mr[0];
    }

    return time;
}
```

Figure 6.1: Custom execution time implementation for Fiasco.OC.

By default, this function returns time values in milliseconds, indicating the time a given process has spent in the scheduler. In order to increase the time resolution from milliseconds to microseconds, the Fiasco.OC kernel needs to be rebuilt with the `CONFIG_FINE_GRAINED_CPUTIME` flag set in its `globalconfig.out`.

For other platforms, most notably base-hw, execution time queries have yet to be implemented.

6.3 Scheduler Idle Time

Scheduler idle time is a sound indicator for system performance. However, Genode does not support queries for CPU idle times and implementations thereof would require platform-dependent code in the Core component and propagating its information all the way up to a user-space level, possibly via the TRACE service.

An attempt was made to provide such functionality for the Fiasco.OC microkernel by using the `l4_scheduler_idle_time()` function in the context of Genode's target-specific `platform` environment. Return values, however, were persistently erroneous despite supposedly correct usage. The code listed in Figure 6.2 failed to execute successfully, propagating an `L4_EINVAL (22)` error code.

```
l4_sched_cpu_set_t cpus = l4_sched_cpu_set(0, 0, 1);
l4_msgtag_t tag = l4_scheduler_idle_time(L4_BASE_SCHEDULER_CAP, &cpus);
```

Figure 6.2: Attempt at querying Fiasco.OC scheduler idle time.

Instead, the platform-independent approach of using a low-priority, always-busy idle task was implemented. The execution time of this idle task is not as exact as a kernel-specific implementation would be but it is still a great indicator for system performance.

6.4 Unmapping Managed Dataspaces

The unmapping of managed dataspace, i.e. datspaces that are managed by the region manager, is not supported on all platforms. Specifically, the unmapping of managed dataspaces is only supported on base-okl4, base-nova, and base-hw but not on the tested base-foc platform. This leads to major problems when destroying child components, as they each manage their own process dataspace using a region manager.

As previously discussed in chapter 5, the dom0 task manager uses a dedicated child destructor thread in order to properly clean up zombie tasks, i.e., tasks that have finished execution but are still registered in the Genode component tree. Accordingly, whenever child components are destroyed, not all their memory can be freed on Fiasco.OC, leading to memory leaks of the task manager's RAM quota. This is accompanied by the fact that finished and allegedly destructed tasks are still listed in the TRACE subject list, albeit with their state set to `DEAD`. The Python SQLite module purges corresponding processes during the conversion. These leaks naturally also occur on Genode's own `Launchpad` app, as it uses the same underlying features to create and destroy child components.

Fixing this issue would require manual implementation of managed dataspaces on Fiasco.OC or switching development platform to Genode's own base-hw kernel. The latter in turn would necessitate an implementation of the aforementioned `execution_time()` function.

6.5 RAM Usage

Monitoring RAM usage of child components in Genode is nontrivial. The intuitive approach would be to simply query a Core service that provides information about

RAM usage of Genode processes, similar to how the TRACE service provides data on execution time of arbitrary processes.

However, Genode does not support an external RAM monitoring concept. Modifications of Core Genode code could enable this feature by propagating RAM usage via their corresponding RAM service connections to the user space. This in turn would require intrusive alterations of the Genode system and introduce security risks that would be deemed appropriate if there was no other way to monitor RAM behavior of supervised tasks.

Instead, since all tasks that require monitoring are started by the dom0 system itself, RAM usage logging is limited to managed tasks by storing their RAM sessions after creation, as described in chapter 5, avoiding the need for Core-level extensions of the Genode system.

7 Showcase

This showcase has been tested on Ubuntu 14.04 LTS (Trusty Tahr) and demonstrates a sample use case scenario of the implemented toolchain.

7.1 Building the System

The system's build process has been fully automated for the most part. The central Makefile is the only control element required to build, modify, and maintain the toolchain, apart from the dependencies.

The framework's code is hosted on GitHub as a branch of the 702nADOS Genode repository:

```
$ git clone -b dom0 https://github.com/702nADOS/genode.git
```

This will clone the approximately 22MiB repository to a local directory.

7.1.1 Dependencies

Ubuntu Packages

For most dependencies these packages will be required:

```
$ sudo apt-get install make git subversion
```

The following packages are required for building and running Genode:

```
$ sudo apt-get install libncurses5-dev libSDL-dev tclsh expect byacc
    ↪ genisoimage autoconf2.64 autogen bison flex g++ git gperf libxml2-
    ↪ utils xsltproc
```

Additional packages may be required depending on the Genode version and platform. Those should be installed as requested by the build process.

VDE

For network communication, the framework Virtual Distributed Ethernet (VDE) is required. The revision r587 from the VDE SourceForge SVN was used in testing this toolchain [VDE].

In order to build VDE, the instructions should be followed as listed on their website:

```
$ svn co https://vde.svn.sourceforge.net/svnroot/vde/trunk/vde-2 vde_svn
$ cd vde_svn
$ autoreconf -fi
$ ./configure --enable-experimental
$ make
$ sudo make install
```

The commands `vde_switch`, `vde_tunctl`, and `vde_plug2tap` should now be globally available on the system.

QEMU

QEMU is required for virtualizing the target hardware; in this case a PBX-A9 board. A standard build of QEMU is provided by the Ubuntu repositories. This build, however, comes without VDE support. Accordingly, a custom build of QEMU is required. This toolchain has been tested with commit `4b6eda626fdb8bf90472c6868d502a2ac09abeeb` of QEMU's git repository, but any current version will work, too:

```
$ git clone git://git.qemu-project.org/qemu.git
$ cd qemu
$ ./configure
```

In the output of `configure` VDE support should be listed with `yes`. Finally, `make install` will build and install QEMU globally.

7.1.2 Genode, tms-sim, and dom0

The entire toolchain can be built by issuing a single `make` command [dom0]. This make target will build most of the targets listed in Table 7.1 in order. The only targets skipped are `run` which starts the showcase system, and the targets for the setup of an optional DHCP server `dhcp-stop` and `dhcp`, as the static IP `192.168.0.14` is used by default.

After running `make` once, `make run` will start the dom0 TCP/IP server and task manager in a QEMU instance.

7.2 dom0 Task Management

The first part of the showcase demonstrates the dom0 TCP/IP server accepting binaries over the network. The command `make run` issues the startup of a Genode instance within QEMU. After booting successfully, the dom0 server is available to external clients.

The sample client `dom0_program.py` is a Python script that executes a fixed sequence of dom0 requests using the `dom0_client.py` module.

First, the client establishes a connection to the TCP/IP Genode dom0 server. After a successful connection, task descriptions are sent as an XML file. The server then prints this file to the standard output.

Table 7.1: Targets provided by the central Makefile.

Make target	Description
`tms`	Build tms-sim task description generator tool.
`descs`	Generate task descriptions using tms-sim.
`toolchain`	Build and install the Genode toolchain for ARM processors.
`ports`	Download external libraries and Fiasco.OC kernel.
`genode_build_dir`	Create build directory for the PBX-A9 target.
`tasks`	Build dynamic task binaries (`hey`, `namaste`, `tumatmul`, `idle`)
`dom0`	Build dom0 server and task manager.
`vde-stop`	Reset VDE configuration.
`vde`	Setup VDE configuration.
`run`	Build dom0 server and task manager and start the run script.
`dhcp-stop`	Kill the Slirp NAT router.
`dhcp`	Start the virtual Slirp NAT router managing the VDE network.

In a relatively long-lasting operation, the client now sends the binaries featured in the task descriptions. Finally, the tasks are started and repeated periodically by the task manager and their output can be observed on the QEMU console.

After a short amount of time, the client issues a stop command to the server, killing all running tasks and preventing further iterations. The server goes back to an idle mode, waiting for further commands to be issued by the client.

At this point, manual control is transferred to the user through an interactive Python shell. Available commands are printed and can be manually requested with the `help()` function. Any further commands use the existing dom0 session, allowing for example for the restart of previously sent tasks.

7.3 Profiling Data Collection

During the entire runtime of the above sample session between client and server, the dom0 task manager is collecting profiling information on all processes running on Genode. At the end of the session, the Python client requests this information, causing the task manager to register one final external event before the dom0 server converts all logged profiling data to XML and sends it back to the client.

The resulting XML file contains a list of tasks that have been managed by the task manager. Additionally, it contains a series of task events, each including a complete snapshot of all running Genode processes at the time of the event. These snapshots feature session and thread name, tracing status, kernel execution time, and, for managed

tasks, additional RAM usage information and an iteration counter. Events logged include the periodic start of managed tasks, both the regular and premature exit of tasks, and external events like the request of profiling data by the client.

Finally, these raw snapshots and task descriptions are slightly refined and fed into an SQLite database using functions provided by the Python module `dom0_sql.py` which can then be inspected for evaluation, e.g., by the tool `sqlitebrowser`.

7.4 Sample SQL Requests

The SQL database enables a wide variety of requests that offer valuable information about a given test cycle. The following query, for example, displays all tasks associated with the binary `namaste` and lists their final execution times for each iteration and instance by extracting their `execution_time` values at snapshots corresponding to regular `EXIT` events of associated task instances.

```
SELECT name, execution_time, iteration
FROM tasks, events, snapshots
WHERE tasks.binary = 'namaste'
AND events.type = 'EXIT'
AND tasks.id = events.task_id
AND tasks.id = snapshots.task_id
AND events.id = snapshots.event_id
```

In the next query, timestamps of all deadline misses of the task `tumatmul` are listed by searching all events of type `EXIT_CRITICAL` triggered by the `tumatmul` task. This can then be used for further analysis. For example, idle time analysis as presented in the final sample query might provide more information as to why deadlines have been missed.

```
SELECT time_stamp
FROM tasks, events
WHERE tasks.name = '02.tumatmul'
AND events.type = 'EXIT_CRITICAL'
AND tasks.id = events.task_id
```

Finally, this last example lists the accumulated idle time at each event using the dedicated `idle` task, giving more information on CPU load. Tests show that a high amount of deadline misses correlates with sectionally constant idle time values, indicating that the idle process is not scheduled at all in periods of high CPU load.

```
SELECT time_stamp, execution_time, events.id, type, events.task_id
FROM tasks, snapshots, events
WHERE tasks.name = '01.idle'
AND tasks.id = snapshots.task_id
AND snapshots.event_id = events.id
```

Naturally, the number of possible SQL queries to this database is virtually endless and their actual formulations greatly depend on the concrete use case. The basic samples listed here are just intended as an impression of what is possible.

8 Further Work and Conclusion

The project presented in this thesis is merely the groundwork for actual schedulability analysis tools making use of the dom0 toolchain. Several concepts and applications are yet to be designed that could greatly benefit from the developed system. Accordingly, this last chapter gives a short overview of what may still be required or instrumental in future analysis systems incorporating dom0 and its associated tools, subsequently ending with a summary of the achieved results.

8.1 Task Migration

While the dom0 task manager and TCP/IP server provide the ability to dynamically receive, allocate, and stop tasks, there is currently no support for migrating running tasks between processor cores or even processing units over the network.

As presented in Stefan Groesbrink's dissertation and partially implemented in Josef Stark's bachelor thesis, virtualization and migration of tasks may greatly improve system performance and reliability [Gro15][Sta14].

Accordingly, a system porting such functionality to Genode would most likely make use of TRACE's CPU core affinity information in addition to declaring an explicit affinity on CPU connection creation for child components. The dom0 task manager would furthermore need to be extended by a hypervisor that can interrupt tasks and transfer contexts to different cores and systems, involving custom Core components.

8.2 Improved Integration of tms-tim

Currently, tms-sim merely functions as an extended task description generator as implemented by Avinash Kundaliya [Kun15]. However, as discussed in chapter 4, tms-sim also provides functionality to emulate scheduling of task descriptions on an offline virtual scheduler by following various algorithms, like for example fixed-priority scheduling as employed by the Fiasco.OC kernel. In case of the base-hw kernel, its custom scheduling policies involving claims and fills would require special implementation in the tms-sim virtual scheduler.

In combination with tms-sim's emulated scheduler, schedulability analysis could be transformed into a two-pass process, tms-sim virtual scheduling representing the first pass of schedulability analysis and the dom0 tool chain including machine learning providing a second pass that subsequently returns more accurate data, such as execution times tested on final hardware in a realistic environment.

8.3 Support for Different Microkernels

While at present, Fiasco.OC is the microkernel of choice for the KIA4SM project, a wider band of supported microkernels may improve some specific features of components and toolchains implemented using Genode. For example, CPU idle time determination is currently realized using a workaround involving an idle task. Additionally, the unmapping of managed dataspaces is not supported on Fiasco.OC, as discussed in chapter 6.

Accordingly, complexity and reward of supporting different microkernels versus extending base-foc with further functionality will have to be evaluated. The most likely candidate for extended support would be Genode's custom base-hw kernel.

8.4 Machine Learning Approach

The actual machine learning approach to schedulability analysis and the development of organically compatible scheduling rules has yet to be implemented in this toolchain. Future work expanding upon this thesis is expected to integrate the dom0 toolchain into a feedback loop of constant task generation, monitoring, and data mining with the goal of programmatically developing organic scheduling rules for the use in real-time systems.

Two possible design concepts are introduced in this section.

8.4.1 Idle Time Analysis

An intuitive approach to developing scheduling rules is to benchmark the system automatically with random sets of tasks, monitoring CPU idle time in the process. Naturally, depending on the tasks and their descriptions, idle times will vary, indicating CPU load and schedulability. In combination with dynamic task migration features integrated into the dom0 task manager, it is possible to form an input vector of various control parameters, including CPU affinity, task priorities, RAM quotas, etc.

Correlations of this input vector with its output value, namely CPU idle time, could then be further processed and analyzed using machine learning.

By further emulating the benchmarked system on powerful hardware, large amounts of data can be generated in faster-than-real-time and statistically evaluated to be later tested on final hardware.

8.4.2 Incremental Load Testing

In contrast to random task generation and load testing, a more systematic approach involving continuously increasing task loads may be helpful as well. A machine learning application would actively increase the load on the embedded system's CPU and RAM by incrementally demanding additional tasks to be processed in real-time. Throughout this process, system performance would be monitored until the system eventually fails.

Multiple attempts of scheduling the same task sets with different priorities, affinities, and scheduling policies will result in varying system performance and points of failure. By programmatically benchmarking these input vectors with regards to their resource utilization and time to failure, rules may be deducted that enable optimal performance during runtime.

8.5 Summary

The dom0 toolchain consisting of network server and client, task manager, monitoring, and database management provides a wide band of possibilities to dynamically allocate and control tasks on an embedded Genode system. In combination with its monitoring capabilities and easy-to-use client modules, the software project developed as part of this thesis promises a vast variety of applications with respect to task management and schedulability analysis.

Tests of the system show that any pre-programmed – but to the active process management system unknown – sequence of task loads is allocated and managed reliably, while at the same time providing extensive monitoring data indicative of system performance.

Altogether, the dom0 Genode server and accessory components form the foundation for future benchmarking and monitoring tools, including MAPE-inspired feedback loops, implemented as part of the KIA4SM project.

Acronyms

ARM Advanced RISC Machine.

CPU Central Processing Unit.

DHCP Dynamic Host Configuration Protocol.

dom0 Domain Zero.

GmbH Gesellschaft mit beschränkter Haftung.

GUI Graphical User Interface.

IP Internet Protocol.

IPC Inter-Process Communication.

ITS Intelligent Transportation System.

KIA4SM Cooperative Integration Architecture for Future Smart Mobility Solutions.

L4Re L4 Runtime Environment.

lwIP Lightweight Internet Protocol.

MAPE Monitor, Analyze, Plan, Execute.

NAT Network Address Translation.

NOVA NOVA OS Virtualization Architecture.

OC Object-Capability System.

OC Organic Computing.

OS Operating System.

PD Protection Domain.

QEMU Quick Emulator.

RAM Random Access Memory.

RISC Reduced Instruction Set Computing.

RM Region Manager.

ROM Read-Only Memory.

RPC Remote Procedure Call.

SQL Structured Query Language.

TCP Transmission Control Protocol.

tms-sim Timing Models Scheduling Simulator.

VDE Virtual Distributed Ethernet.

WCET Worst-Case Execution Time.

List of Figures

List of Tables

Bibliography

[ARK08] I. Ahmad, S. Ranka, and S. U. Khan. "Using game theory for scheduling tasks on multi-core processors for simultaneous optimization of performance and energy." In: *Parallel and Distributed Processing, 2008. IPDPS 2008. IEEE International Symposium on*. 00073. 2008, pp. 1–6.

[BF09] A. Basu and S. Funk. "An Optimal Scheme for Multiprocessor Task Scheduling: a Machine Learning Approach." In: *Work-In-Progress Proceedings* (2009), p. 29.

[Döb] B. Döbel. *Introduction to Microkernel-Based Operating Systems*. `http://www.cs.hs-rm.de/~kaiser/1314_aos/01-Intro.pdf`. Accessed: 2016-03-30.

[dom0] *702nADOS Genode Fork, dom0 Branch*. `https://github.com/702nADOS/genode/tree/dom0/`. Accessed: 2016-03-21.

[EKB15] S. Eckl, D. Krefft, and U. Baumgarten. "COFAT 2015 - KIA4SM - Cooperative Integration Architecture for Future Smart Mobility Solutions." In: *Conference on Future Automotive Technology*. 2015.

[Fes15] N. Feske. *Genode Operating System Framework 15.05*. CC-BY-SA, 2015.

[Gena] Genode Labs. *Genode 13.08 Release Notes*. `https://genode.org/documentation/release-notes/13.08`. Accessed: 2016-03-31.

[Genb] Genode Labs. *Genode Operating System Framework Source Code*. `https://github.com/genodelabs/genode`. Accessed: 2016-03-30.

[Gro15] S. Groesbrink. "Adaptive Virtual Machine Scheduling and Migration for Embedded Real-Time Systems." PhD thesis. University of Paderborn, 2015.

[Kun15] A. Kundaliya. "Design and prototypical implementation of a toolchain for offline task-to-machine mapping in distributed embedded systems." MA thesis. Technische Universität München, 2015.

[L4Re] *L4 Runtime Environment*. `https://l4re.org/`. Accessed: 2016-03-29.

[NOVA] *NOVA Microhypervisor*. `http://hypervisor.org/`. Accessed: 2016-03-29.

[SQL] *SQLite*. `http://www.sqlite.org/`. Accessed: 2016-04-04.

[SQLc] *SQLite Conversion Tools*. `http://www.sqlite.org/cvstrac/wiki?p=ConverterTools`. Accessed: 2016-04-04.

[Sta14] J. Stark. "Dynamic task management between interconnected L4 microkernel instances." MA thesis. Technische Universität München, 2014.

[tms-sim] *tms-sim*. `http://myweb.rz.uni-augsburg.de/~klugeflo/tms-sim/`. Accessed: 2016-04-04.

[VDE] *Virtual Distributed Ethernet*. `http://vde.sourceforge.net/`. Accessed: 2016-03-21.