TU WIEN Informatics

Echtzeit Container Virtualisierung

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Technische Informatik

eingereicht von

Stefan Walser, BSc
Matrikelnummer 11809605

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Privatdoz. Dipl.-Ing. Dr.techn. Wilfried Steiner
Mitwirkung: Dr. Silviu S. Craciunas
Jan Ruh, MSc.

Wien, 19. Jänner 2024

[redacted]

Stefan Walser

[redacted]

Wilfried Steiner

TU WIEN Informatics

Time-aware Container-based Virtualization

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Computer Engineering

by

Stefan Walser, BSc
Registration Number 11809605

to the Faculty of Informatics

at the TU Wien

Advisor: Privatdoz. Dipl.-Ing. Dr.techn. Wilfried Steiner
Assistance: Dr. Silviu S. Craciunas
Jan Ruh, MSc.

Vienna, 19th January, 2024

Stefan Walser

Wilfried Steiner

Erklärung zur Verfassung der Arbeit

Stefan Walser, BSc

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Wien, 19. Jänner 2024

Stefan Walser

Danksagung

Ich möchte mich bei TTTech Computertechnik AG und Privatdoz. Dipl.-Ing. Dr.techn. Wilfried Steiner bedanken, mir diese interessante Arbeit ermöglicht zu haben.

Desweitern möchte ich mich bei meinen Betreuern bedanken. Bei Jan für die unermüdliche Hilfe in praktischen Angelegenheiten und bei Silviu und Jan für die theoretischen Diskussionen, die mir neue Lösungsansätze oder andere Perspektiven aufgezeigt haben. Vielen Dank auch an Silviu, Wilfried und Jan für das wertvolle Feedback zur schriftlichen Arbeit.
Ich danke euch für die Zeit, die ihr euch genommen habt, um meine Fragen zu beantworten. Eure Verfügbarkeit war für mich von unschätzbarem Wert.

Ich möchte mich auch bei meiner Familie bedanken. Eure Unterstützung hat mich nicht nur während der vergangenen Monate, sondern das gesamte Studium über motiviert und mir durch stressige Zeiten geholfen.

Acknowledgements

I want to express my gratitude to TTTech Computertechnik AG and Privatdoz. Dipl.-Ing. Dr.techn. Wilfried Steiner for enabling me to work on this interesting project.

Furthermore, I would like to thank my supervisors. Jan, for his tireless assistance with practical matters, Silviu and Jan for the theoretical discussions that provided me with new approaches and alternative perspectives, and Silviu, Wilfried and Jan for the valuable feedback on the written work.
I highly appreciate the time you took to answer my questions. Your availability has been invaluable to me throughout this process.

I also want to thank my family. Your support has not only motivated me during the past months but has also been a constant source of motivation throughout my entire studies, helping me during stressful times.

Kurzfassung

In den vergangenen Jahren haben sich industrielle Prozesse zunehmend in Richtung hochgradige Automatisierung entwickelt, um menschliche Interaktionen so weit wie möglich zu minimieren. Das Konzept des Industrial Internet of Things (IIoT) sieht dabei den Einsatz zahlreicher Sensoren und Aktuatoren vor. Die von den Sensoren erzeugte neue Datenmenge muss jedoch verarbeitet und in Steuerbefehle für die Aktuatoren umgewandelt werden. Da Cloud Computing (CC) aufgrund unvorhersehbarer Netzwerklatenzen nicht die richtige Wahl ist, wird Edge Computing (EC) eingesetzt, um Steueranwendungen auszuführen, die in der Regel Echtzeit-Anforderungen haben. Um den Einfluss der Anwendungen aufeinander zu minimieren, wird auf Edge-Servern oft Virtualisierung verwendet. Insbesondere die Container Virtualisierung mit ihren Vorteilen von kleinen Images und schnellen Startzeiten ist in industriellen Umgebungen hilfreich. Zur Erfüllung der zeitlichen Anforderungen einer Anwendung auf einem Edge-Server gibt es mehrere Möglichkeiten.

Diese Arbeit verwendet den Hierarchical-Constant-Bandwidth-Server (HCBS)-Patch für den Linux-Kernel. Dieser Patch ermöglicht das Scheduling von Containern nach einem Earliest-Deadline-First (EDF)-Schema und Prozesse innerhalb von Containern nach einem Fixed-Priority (FP)-Schema.

Der in der Container Virtualiserung übliche Orchestrator reduziert nicht nur den Energie- und Ressourcenverbrauch, indem nicht mehr benötigte Container gestoppt werden, sondern verbessert auch das Echtzeitverhalten des gesamten Systems. Denn der Orchestrator kann Container, deren Anwendungen Deadlines verpassen, auf einen anderen Server im Cluster mit mehr verfügbarer CPU-Zeit bewegen. Es stellen sich jedoch zwei Fragen, wenn Container Virtualisierung im Kontext von Echtzeitsystemen verwendet wird:

1.) Welcher Container muss auf welchem Server gestartet werden, um alle Anforderungen des Containers zu erfüllen, und wie viel CPU-Zeit benötigt er?

2.) Wie kann das Echtzeitverhalten zur Laufzeit aufrechterhalten werden, wenn Störungen durch andere Anwendungen dazu kommen?

Für die Beantwortung von Frage Nummer 1 stellen wir mehrere Solver bereit, die wir hinsichtlich Leistung und Optimalität vergleichen. Bezüglich Frage Nummer 2 implementieren wir einen Mechanismus, der das Verpassen von Deadlines mithilfe von Time-Utility Funktionen reduziert, indem er die CPU-Zeit eines Containers erhöht oder diesen auf einen anderen Server verschiebt. Schließlich evaluieren wir unsere Controller anhand synthetischer Testfälle.

Abstract

During the past years, industrial processes have developed towards a highly automated nature to remove human input wherever possible. This is enabled by many sensors and actuators, following the idea of the Industrial Internet of Things (IIoT). However, this new amount of data created by the sensors needs to be processed and converted to control commands for the actuators. As Cloud Computing (CC) is not sufficient, due to the unpredictable network latencies, Edge Computing (EC) is employed to run control applications, which typically have real-time (RT) requirements. To keep the influence of the applications on each other to a minimum and increase security at the same time, virtualization is often used on edge servers. Especially container-based virtualization with its benefits of small image size and rapid deployment is useful in an industrial setting.
To fulfill the temporal requirements of an application on an edge server multiple adaptations are possible. We use the hierarchical-constant-bandwidth-server (HCBS) patch for the Linux kernel. This patch allows scheduling containers following an earliest-deadline-first (EDF) scheme and tasks inside containers following a fixed priority (FP) scheme.
An orchestration layer that manages the containers and decides which container runs on which server is typically used together with container-based virtualization. This layer not only reduces the energy and resource usage by stopping no longer required containers but also helps to improve the RT behavior of the whole system. If an application inside a container misses its deadlines due to, for example, overutilization of the server, the orchestrator can relocate this container to a different node in the cluster. However, two critical questions need to be answered when using container-based virtualization in the context of RT systems:
1.) Which container has to be started on which server to fulfill all the container's needs, and how much CPU time does it need?
2.) How can the RT performance be maintained at runtime when interferences from other applications come into play?
We answer question number 1 by providing multiple solvers, which we evaluate and compare in terms of performance and optimality. Regarding question number 2 we implement an online adaptation mechanism that reduces deadline misses of the tasks based on time-utility functions. We either increase the CPU time of a container or relocate it to a different node. Finally, we evaluate our hierarchical controllers with a real-world test bed.

Contents

CHAPTER 1

Introduction

Industrial processes have developed rapidly during the past years to automate as much as possible and remove human input from production wherever possible. This automation is enabled by the Internet of Things (IoT), which has been adapted to the Industrial Internet of Things (IIoT), meaning many sensors and actuators (*things*) are deployed around the factories.
However, these sensors produce a great amount of data that needs to be processed and converted to control commands for the actuators. An initial approach could be to first send the data from the sensors to a data center, often referred to as *cloud*. Consequently, the resulting commands for the actuators could then be sent back after the computation. However, the data transfer via the network between a geographically distant data center and the factory plant requires great network bandwidth and introduces delay. Hence, a significant amount of time passes between the reading of a sensor value and the corresponding action of the actuator. Additionally, and even more importantly, this delay can vary over time and is not predictable. This variance is called jitter and poses a problem for factory plants, as they are often intertwined with other plants. If one does not receive its commands on time, the whole production line can halt or faulty parts are produced.

To compensate for the jitter, a concept called *fog computing (FC)* was first mentioned in [Bon11, BMZA12] and introduced to the IIoT in [AZH18]. A core concept of FC is the so-called Edge servers. They are located close to the sensors and actuators integrated into the machinery and production facilities they communicate with. Besides the short distance, edge servers provide significantly more computational power than embedded devices, which could also be located near the things. Together with Time Sensitive Networking (TSN) [Fin18], this computational power close to the edge is fundamental in decreasing jitter.

However, besides the network there is an additional link in the chain that plays a role when it comes to generating timely responses for time-critical functionalities; The computing

device itself. Companies prefer to use commercial-off-the-shelf (COTS) devices due to reduced cost [JLF11]. However, COTS computing devices offer computational power without any temporal guarantees. Suppose applications that require a certain degree of timeliness are executed on such systems where resources like CPU, memory, and cache are shared. In that case, it is necessary to modify the system somehow. The systems must be transformed into so-called real-time (RT) systems. RT systems are defined by their obligation not only to generate correct results but also to produce them at the right moment. Typically, deadlines are associated with each computation, and failing to meet these deadlines can have varying consequences [KS22]:

Soft RT Systems Deadline misses have no severe consequences, and the results still hold some value after the deadline. For instance, reading data from several sensors at regular intervals in a weather station is crucial. Even if one sensor reads the data later than others, the data still retains some value.

Firm RT Systems Deadline misses also bear no severe consequences, but the results become worthless after the deadline. In a video conferencing system, for example, processing and transmitting the video stream within deadlines is essential. Frames not processed due to missed deadlines are useless, as showing older frames to call participants makes little sense.

Hard RT Systems Deadline misses in these systems can have catastrophic consequences. If a brake controller in a car, for example, reacts too late to a brake pedal push, it can lead to life-threatening situations.

As edge servers typically run many applications with many thousand lines of code, they can not be used for hard RT systems, because it is not possible to prove that they always meet the deadline.
To reduce the influence the applications on the server have on each other, which can also contribute to deadline misses, an additional concept called virtualization is employed.

In computing, virtualization means creating a virtual version of computing hardware, called a virtual machine (VM). An application inside a virtual machine only sees the hardware assigned to the VM. However, the execution takes place directly on the hardware itself, contrary to emulation, where the instructions need to be translated before the hardware can execute them. In RT systems it is common to statically partition a physical machine and assign each partition to a different VM [LXRD19, MTS+20]. The isolation aspect not only improves the temporal behavior of the applications but also aids security, as an application within a VM is invisible to other applications in different VMs. Resource utilization is improved as well because different applications with different requirements and dependencies on other software can be run on the same machine. However, the utilization could be improved further, if the resources were shared between VMs, as it is possible that some VMs do not fully utilize their assigned resources. Furthermore, static partitioning is unflexible by design. If an additional VM needs to be started on

a machine that has all its resources already assigned to VMs, it would be necessary to change all configurations. For soft RT systems, it is often not necessary to have strict isolation between the applications, which brings us to containerization.
Containerization, a specific form of operating system (OS) virtualization, is particularly useful in the context of server virtualization. In containerization, each virtualized instance is referred to as a container and uses the OS of the server. The container's image only stores the application itself and the libraries it depends on. This has the advantage of rapid container deployment. If an RT application can not handle the current load or crashes for some reason, it is possible to start a new instance of the same application on the same or a different server within seconds. Subsequently, this technology also saves resources and money, because if not all instances of a control application are required to handle the current load, some of them can be stopped [HSI+19].

In addition to the benefits mentioned above, container-based virtualization allows for the specification of CPU time, also called CPU quota, allocated to an application over a certain period. This concept was introduced in [AB04] and is critical in meeting RT constraints because a certain CPU time can be guaranteed for each application. However, as already examined by Struhar et al. in [SCA+23], two questions need to be answered before containers can be used in such a setting:

1. Which container has to be started on which server to fulfill all the container's needs, and how much CPU quota does it need?
2. How can the RT performance be maintained at runtime when interferences from other applications, e.g. via the cache, and variations in the workload come into play?

Question number 1 can be answered during the so-called *offline phase* which takes place before any applications are running. In this phase, the requirements of the tasks and containers are analyzed. This analysis yields a mapping of a container to a server, also called a node, and a pair of two values, which represent the CPU quota and period of the container.
RT applications are often periodic, meaning they execute one or multiple tasks once every x milliseconds. To compute the quota and period, the analysis uses a value called *worst-case execution time (WCET)*. This value specifies how much time the application needs at most every period to finish its tasks. However, it is a complex task to find exact WCETs and, therefore, only estimates or bounds can be derived. Estimates do not guarantee anything and can therefore not be used in safety-critical systems. Bounds guarantee that the WCET is not above a certain value [WEE+08].

Question number 2 can be answered during the *online-phase*, which is when the applications are executed. As the WCETs are sometimes only estimated, it is possible that the CPU quota assigned to a task is not sufficient to meet its deadlines. Additionally, as mentioned previously, in soft RT systems the computing resources are often not strictly partitioned between the applications to improve utilization and reduce system

costs [AB04]. However, this sharing introduces unpredictable interferences between the applications, e.g., via the cache. Together with additional interferences with, for example, system tasks or CPU overload they can lead to deadline misses of the RT tasks. In such a case, the system must react to mitigate this problem and minimize future deadline misses. There are two possibilities on how to improve on such a situation:

1. Increase the CPU quota of the container.
2. If an increased CPU quota would lead to a CPU overload, search for a different node in the system to which the container can be relocated.

If none of these options is possible, it is not possible to prevent deadline misses.

1.1 Thesis contribution

This thesis addresses both the offline phase and the online phase.

- Regarding the offline phase, we developed three solvers that act as alternatives to the SMT solver-based method from [SCA+23] and can compute solutions for industrial-sized systems efficiently. Each one is based on one of the following three methods:
  - Simulated Annealing.
  - A custom greedy algorithm.
  - An SMT solver, but pursues a different approach than the one in [SCA+23].

  We then compare these solvers in terms of runtime and quality of the results. A custom cost function measures the latter.
- Regarding the online phase, we achieved two improvements compared to [SCA+23].
  - We redesigned the architecture of the controller system by collapsing one level of the hierarchy. We then reimplemented the orchestration framework in a way that allows an easy addition of features.
  - We also enhanced the metrics by which the quota of a container is adapted. Instead of the temporal difference between the expected and actual finishing time of a task, we employ time-utility functions, which assign a utility value to each finishing time of a task. Based on these values, we can then adapt the CPU quotas of those containers specifically, which suffer the most under their deadline miss.

1.2 Structure of the Thesis

This thesis is structured in the following way: Section 2 describes the concepts of Edge and Fog computing and hypervisor and container-based virtualization. Furthermore, it goes into detail on RT-scheduling and explains the theory on which this thesis is founded. Section 3 contains the description of the heuristic solvers we developed and implemented to compute solutions for the allocation problem of the offline phase. Section 4 describes our architecture and controllers used during the online phase to adapt and relocate running containers. The evaluation of our solvers is presented in Section 5.1 and the evaluation of our online control system is in Section 5.2. Section 6 contains the conclusion and Section 7 possible extensions and improvements for our work that are beyond the scope of this thesis.

CHAPTER 2

State of the Art

This thesis revolves around three main topics: fog and edge computing, virtualization, and real-time (RT) scheduling. The following three sections will break down the key concepts in each area, give an overview of the current state of the art, and provide detailed background information on [SCA+23], the basis for this thesis.

2.1 Fog and Edge Computing

In 2017 the OpenFog consortium, now part of the *Industry IoT Consortium (IIT)*, released a reference architecture for fog nodes and networks [ope] in which they define Fog Computing as

> *A horizontal, system-level architecture that distributes computing, storage, control, and networking functions closer to the users along a cloud-to-thing continuum.*

There exist numerous terms and concepts in the area of Fog Computing [YFN+19]. Figure 2.1 puts them into relation and the following paragraphs explain their meanings.

Cloud Computing (CC) The National Institute of Standards and Technologies (NIST) defines CC as a model that enables ubiquitous, convenient, and on-demand network access to a shared pool of configurable computing resources. These resources can be servers, networks, storage, services, or applications [MG+11]. The physical machines of the cloud providers, e.g., Google, Microsoft, Amazon, are usually split up into virtual machines which can be individually configured and scaled by the customers, which can pay exactly for the amount of resources they need. [VRMCL08]

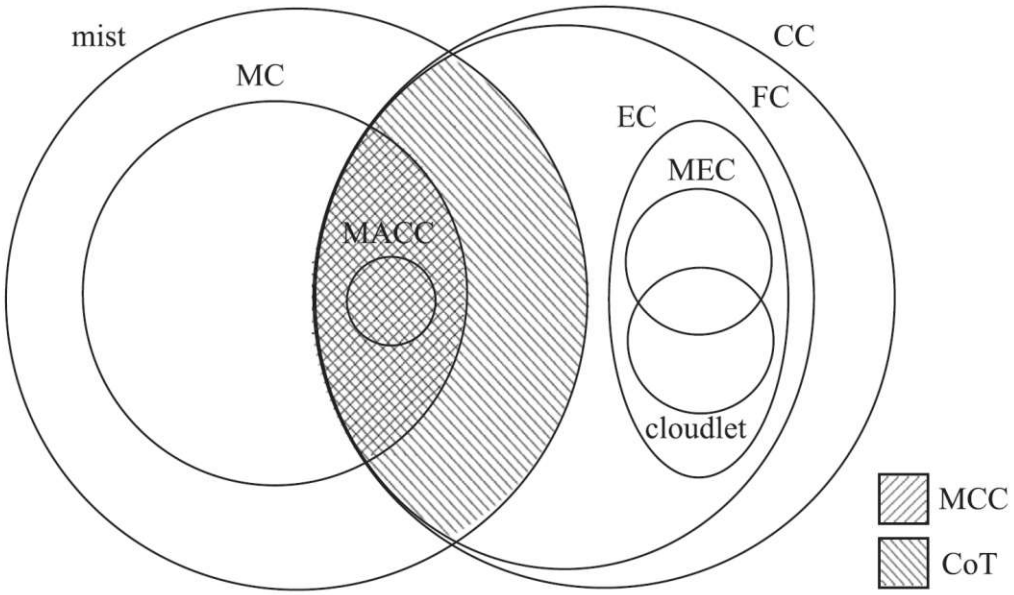


Figure 2.1: Fog-computing overview by [YFN+19]

Fog Computing (FC) The fog is the bridge between cloud and IoT devices, the so-called *things*. This contains the storage of data, networking functions, and computing [YFN+19]. FC was introduced to improve the following three points mainly [LZCC17]:

- Scalability: With the ever-growing number of smart devices in today's IoT networks, a centralized management and monitoring approach cannot be successful. Fog computing decentralizes networks and allows greater scalability.
- RT data analytics: Applications sensitive to jitter and/or latencies experience a degraded quality of service if all data is transferred to and from a cloud server. Fog-computing allows processing the data closely to the things, e.g., on edge servers.
- Saving network resources: By processing the data locally, the network traffic is also substantially reduced.

The main differences compared to CC are the computing power, the latency of messages coming from the things, and the power consumption. All these factors are lower in FC [YFN+19].

Mobile Computing (MC) This is an umbrella term for computing that occurs on mobile devices, such as smartphones, laptops, or tablets, and has been the basis for the developments in FC and CC. The only requirement for mobile devices to participate in mobile computing is the support of a cellular protocol, such as WiFi or Bluetooth.

They can be used in applications where the computation depends on the location of the device [YFN+19]. The core strength of MC is its decentralized and distributed architecture. However, compared to CC and FC, the resources and the communication latency between the devices are constraining factors. [Sat96]

Mobile Cloud Computing (MCC) This concept extends mobile computing by moving computing and storage to the cloud. This then allows the inclusion of devices in mobile computing applications that would previously not have had enough resources, such as IoT devices [DLNW13]. Applications in MCC typically offload bigger tasks to the cloud and compute smaller tasks locally. The downsides are the same as in MC. The benefit of being able to compute greater tasks comes with the downside of needing an active internet connection at all times and latency when offloading the tasks [YFN+19].

Mobile ad hoc Cloud Computing (MACC) In environments with a lack of infrastructure, mobile ad hoc networks (MANETs) can be used for networking. Mobile devices form a network, manage the routing of network traffic, and provide transport protocols. They must be able to handle the highly dynamic nature of the network with devices joining and leaving at any time [HGLBV01]. MACCs extend the concept of MANETs by adding computing and storage functionality to the network. As the computation also has to take place on-site, the difference compared to CC is the lack of a connection to a cloud in a geographically different location. The use cases for MACC can be, for example, disaster relief operations or car-based networks [YFN+19].

Edge Computing (EC) Edge Computing has its name from the place it is located, i.e., close to the things, at the edge of the network. It filters, pre-processes, and aggregates the IoT data before it is transferred to the cloud for further computation. Those computations that can be completed at the edge, typically provide a lower latency compared to a computation in the cloud, given the computational resources at the edge are sufficient. Additionally, EC helps to improve privacy and also reduce the load on the network [YFN+19].
FC and EC follow the same approach, which is moving the computation and storage closer to the things. However, as pointed out in [CHR+17] FC contains EC among other concepts such as, for example, the cloud and the things. As mentioned in the paragraph about FC, it also provides means for network communication and storage while EC provides means for computation mainly [ope].

Multi Access Edge Computing (MEC) Multi-Access Edge Computing is an extension of mobile computing through edge-devices. To include not only mobile device-specific tasks, the concept was renamed from Mobile EC to Multi Access EC. The benefits are again the possibility to offload computationally complex tasks to a data center. However, compared to MCC, the latency is lower and the *things* do not need an internet connection at all times. However, the computing power of the EC data centers is moderate compared to the ones in CC. Compared to EC the communication is done via wireless methods

only, due to the mobility of the devices. Applications for MEC could be video analytics, connected vehicles, health monitoring, or augmented reality. 5G is supposed to play a big role in MEC, due to its low latency and high bandwidth [YFN+19].

Cloudlet Computing (cloudlet) Cloudlets are a concept proposed in [SBCD09]. Essentially, the concept is very similar to MEC whose pros and cons have already been listed in its paragraph. The only key difference is that Satyanarayanan et al. define a strong internet connection for every cloudlet at all times. In EC this is not mandatory. The internet connection is only necessary if a task is too complex for computing resources at the edge [YFN+19].

Mist Computing (mist) Alex Davies of Cisco introduced in [Dav] the so-called Mist Computing. In this extreme version of EC, the *things* themselves provide enough computing resources to process the data they produce. These can be IoT or mobile devices such as smart watches, smart fridges, or smartphones. This concept can, therefore, also be called *IoT computing* [YFN+19].
In [SRS+17] Silva et al. experimented with a soccer game where a group of people was able to watch replays on their smartphones. They replay videos were not always fetched from a central server, but primarily from nearby smartphones, if the video was already available there. Using this method they were able to significantly reduce the load on the WiFi access points at the venue.

Cloud of Things (CoT) Cloud of Things is a concept similar to Mist Computing. In Mist Computing the devices always do their computations locally. In CoT, the IoT devices form a cloud and extend the existing system with their resources for storage, networking, and computation. Edge Nodes discover the IoT devices, virtualize their resources, and set up the cloud [YFN+19]. This cloud can then also be geographically distributed as Abdelwahab et al. showed in [AHGZ16] where they implemented a system providing something they called sensing-as-a-service. The IoT devices provide their sensors to the network and users of the cloud can access the already processed and aggregated data of the system. The key is that the sensors of the devices may not have been implemented for the purpose requested by the user, but the data can be requested nevertheless. For example, temperature and CO_2 data from sensors in vehicles all over the world can be requested to compare the pollution levels.

2.2 Virtualization

The term *virtualization* in the context of computing systems came up already in the 1950s and 1960s [CJ]. At that time scientists used mainframe computers and submitted their jobs in batches one after another [Org72]. As a consequence, hardware virtualization techniques were developed to allow the execution of multiple jobs in parallel [Koz15]. With the rise of personal computers in the 1980s and 1990s, however, the demand for hardware-level virtual machines declined [CJ]. In the 2000s the term virtualization became popular again, due to the rise of CC. The two main benefits of virtualization, namely abstraction and multiplexing, are of great benefit in this area. Different applications can be hosted on the same server, isolated from each other and the machines can be better utilized. With only one application on a physical machine, the hardware may be idling most of the time. With virtual resources (virtual CPUs, virtual memory, containers), however, it is possible to add applications to the server until the utilization is at a desired rate.
In FC, virtualization techniques can help to reconfigure the behavior of devices in the network and deploy services depending on the capabilities of the nodes [MFIT17].
The following two subsections explain the two main approaches to virtualization relevant to FC.

2.2.1 Hypervisor-based Virtualization

A hypervisor is a control program that executes the VMs. According to Popek and Goldberg in [PG74] a hypervisor has three main characteristics:

Equivalence Any program that runs on top of a hypervisor must exhibit the same effects as if it would run on the machine directly. The only differences allowed are those introduced by resource availability and timing dependencies. Due to the existence of multiple virtual machines on the same hardware, there may be influences that lead to different temporal behavior of the programs.

Efficiency Programs that run in this environment show only a minor decrease in speed. Therefore, a "statistically dominant subset" of the virtual machine instructions must be executed on the processor directly according to [PG74] and emulators can not be considered a hypervisor.

Resource Control The Hypervisor controls all the system resources. The virtual machines can only access and use the system resources assigned to them. Furthermore, the hypervisor can also reclaim the resources from them if needed.

In [PG74] Popek and Goldberg showed that a hypervisor that fulfills these three characteristics does exist if the sensitive instructions are a subset of the privileged instructions. To implement a hypervisor three features need to be supported by the system: A single bit that indicates whether the CPU is running in kernel- or user-mode, virtual memory,

and instruction traps. When an instruction traps control is handed over to the hypervisor and therefore a transition from user mode the kernel mode is executed. The instructions that trap are also called privileged instructions.
The sensitive instructions mentioned previously manipulate the system state by, for example, allocating more memory. As the guest OS does not have a holistic view of the physical memory, the hypervisor needs to intervene and emulate the instructions issued by the guest OS. After the allocation is completed, the hypervisor returns from the trap, the CPU is set to user-mode and the upcoming instructions of the virtual machine are again directly executed.
We differentiate type-1 and type-2 hypervisors [G+73]. Type-1 hypervisors, also called *bare-metal* hypervisors, run directly on the hardware. Type-2 hypervisors on the other hand need an operating system beneath them.

2.2.2 Container-based Virtualization

Container-based virtualization follows an alternative approach. Instead of virtualizing the hardware, the OS is virtualized. To achieve this, a virtualized application, called a container, uses the kernel of the OS that is running on the hardware and the kernel also manages the virtualization. In Linux, containers are realized by two kernel features called namespaces and control groups (cgroups). Namespaces are used to isolate kernel resources from each other. Processes running in a certain namespace can not see resources or processes running in a different namespace. Control groups are used to monitor and limit the processes' access to resources such as CPU, memory, or IO [ABC19].
This approach results in smaller VM image sizes because the OS does not need to be part of it and also results in faster startup times [XFJ16] because the OS does not have to be started previous to the application. However, this also means that with the use of containers, it is not possible to run a guest OS different than the host OS without a hypervisor below the containers. [Ede16]. The quick startup times are very useful in CC. Depending on the demand of an application, further instances can be started and moved quickly between server instances if necessary. When the demand decreases, unneeded instances can be stopped to save resources [TRA15].
When executing RT applications in containers on Linux, there are no guarantees that the deadlines of the tasks will be met. There are mechanisms in the generic Linux kernel to support RT applications, but the latencies are not bounded and the deadlines of hard RT tasks may be missed. To reduce these latencies and to, furthermore, be able to run RT applications in containers research has already been conducted. Some approaches modify various parts throughout the kernel and others only modify the scheduler.
The following three approaches are mainly used throughout literature [SBAP20]:

PREEMPT_RT Patch The real-time patch for the Linux kernel increases its preemptible code. This includes for example critical sections and interrupt-handlers [LWT].

Real-time Co-Kernel The additional real-time co-kernel, that runs parallel to the Linux kernel, takes over all the time-critical tasks. This includes interrupt handling

and scheduling RT threads. Whenever this kernel is idle, the standard Linux kernel is executed. This solution has the drawback, compared to the PREEMPT_RT patch, that additional APIs, tools, and libraries are necessary to develop applications [TMV18].

Hierarchical Scheduling This approach enhances the already existing *rt_group_sched* feature of the Linux kernel. The group scheduling allows the user to set a period and a quota of CPU time which is then reserved for RT tasks as introduced by Abeni and Buttazzo in [AB04]. The Earliest Deadline First (EDF) scheduler decides which tasks turn it is. The overall quota and period are monitored by cgroups.
In [ABC19] Abeni et al. extended this in such a way that the EDF scheduler does not schedule tasks directly, but containers. When a container is selected for execution, a fixed priority scheduler selects one of the tasks inside the container for execution. Because of the resulting two-leveled scheduler, this variant is called hierarchical scheduling.
Struhár et al. used this approach in [SCA+21, SCA+23] and therefore we also use it in this thesis.

However, the enhancements to the kernel introduced in the previous subsection are not enough to fully utilize the potential of containerization, but an orchestration tool is necessary. It decides which containers run on which node and starts additional instances of an application if the currently running can not handle the load. If the load decreases the orchestrator stops instances not required. If a node crashes or fails to satisfy the computational requirement of an application, the orchestrator relocates the container to a different node in the cluster.

Fiori et al. presented in [FAC22] an adaption of the orchestration framework Kubernetes that can orchestrate containers in such a way that timeliness is guaranteed. On every node on the system, they used the Hierarchical Constant Bandwidth Server scheduler ([ABC19]). A “manifest” file in YAML format can then be used to specify the Pods. A Pod is a concept of Kubernetes and consists of multiple containers (in RT-Kubernetes only one), network connections, and storage. For Kubernetes to be able to schedule the Pod to a node that is capable of executing the contained application, the manifest file must contain additional RT information. This is, apart from the number of CPU cores (rt_cpu) the container should run on, the runtime Q, and the period P of the container. The scheduler then chooses one of the nodes to schedule the Pod on where it is guaranteed that the container can run for a time Q every P execution time units on rt_cpu cores. This so-called admission test is realized via a utilization-based calculation and can also guarantee hard RT behavior.

2.3 Real Time Scheduling

Before going into the theoretical details of RT scheduling, we introduce a few terms and concepts. In RT systems, we often deal with periodic or sporadic tasks. Periodic tasks are those where the arrival times take place in known time intervals. Sporadic tasks do not have these fixed intervals. Instead, there is only a minimum interarrival time. In other words, the maximum rate at which these tasks need to be executed is known. Additionally, there are aperiodic tasks that can occur at any random point in time.
We denote the i-th task of a system by τ_i. $\Gamma_n = \{\tau_1, \dots, \tau_n\}$ is the set of all tasks. A task consists of a stream of jobs τ_{ik} $(k = 1, 2, \dots)$. T_i is the period of the task and denotes after how many time units another job is requested to be executed. These request or release times are denoted by r_{ik}. The first job of every task is released at $r_{i1} = \Phi_i$ and this initial delay is also called the phase of the task. Every job has a deadline D_i relative to its start time. There exist three different types of relative deadlines:

Implicit Deadline: The relative deadline D_i is equal to the period T_i. This is often the case.

Constrained Deadline: The relative deadline D_i is less than the period T_i.

Arbitrary Deadline: The relative deadline D_i can also be larger than the period T_i.

The absolute deadline for a job is $d_{ik} = r_{ik} + D_i$. f_{ik} is the absolute time at which the job k of τ_i ends. If $t = f_{ik} > d_{ik}$ a job misses a deadline at time instant t and we say an overflow happened at time t. C_i denotes the worst-case execution time of every job of task i.[LL73, BB02]
Priorities are assigned to tasks to create a hierarchy among them. When multiple tasks request a job for execution, the job with the highest priority is executed. The schemes after which priorities are assigned are explained in the following subsections. In a preemptive scheduling scheme, tasks get interrupted if a higher-priority task requests a job during its execution. In a non-preemptive scheme, this is, of course, not the case. Priorities are usually represented with numbers, and the lower the number, the higher the priority. Γ_i is the set of tasks with a priority of i or higher.
The response time of a request for a task is the time between the request and the end of the execution of the job. This now allows us to define the important term of a critical instant. This is an instant where the response time of a request is at its maximum. Liu and Layland prove in [LL73] that a critical instant occurs when a job of a task gets requested at the same time as jobs of all higher priority tasks.
The following subsections go into detail on fixed and dynamic priority scheduling, aperiodic servers, and resource partitioning.

2.3.1 Fixed Priority Scheduling

In fixed-priority scheduling, all tasks have the same priority throughout the execution. The priority level can, for example, depend on their period or their deadline. In this

thesis, we will focus on the Rate Monotonic scheduling algorithm, because this is also the one that is used in the framework developed by Struhár et al. in [SCA+23].

Introduction of Rate Monotonic Scheduling by Liu and Layland [LL73]

Liu and Layland introduced the Rate Monotonic (RM) scheduling algorithm, and the Earliest Deadline First (EDF) algorithm described in Section 2.3.2 and analyzed them under the previously described task model. Additionally, they also defined several constraints that have to hold for their analysis of the algorithm to be true.

- The tasks for which hard deadlines exist must have periodic requests.
- Each task must be completed before the next request occurs.
- All tasks are independent and do not depend on the request or completion of other tasks.
- The run times of all jobs of a task are the same.
- Any non-periodic tasks in the system are either initialization or failure-recovery routines, have no hard deadlines and only displace periodic tasks.
- Switching between tasks does not cause any overhead.

Firstly, they showed that the worst-case scenario for a periodic task set is if all tasks are released at the same time ($\Phi_i = 0$). Therefore, if a set of tasks is schedulable under this condition, it is also schedulable for an arbitrary Φ_i. Second, they showed that given these constraints, RM scheduling is an optimal fixed-priority scheduling algorithm. Optimality in this case means that if any other fixed priority scheduling algorithm finds a feasible scheduling, RM scheduling also finds one, even though it may not be the same schedule. Furthermore, they analyzed their proposed algorithm in terms of schedulability. They showed that in a system with two tasks, a utilization factor of less than $2(2^{\frac{1}{2}} - 1) \approx 83\%$ guarantees schedulability. The processor utilization factor is the fraction of processor time spent in execution. The fraction C_i/T_i represents the utilization factor for task τ_i. The utilization factor of the system is then

$$U = \sum_{i=1}^{n} \frac{C_i}{T_i}$$

$U \leq 1$ is, therefore, a necessary schedulability condition.
Furthermore, they generalized the utilization bound for n tasks and showed that, in general, a system is schedulable if $U \leq n(2^{1/n} - 1)$. Therefore, for $\lim_{n\to\infty} U \approx \ln 2 \approx 69\%$. However, this bound is not necessary, but only a sufficient condition. This means if the aforementioned equation does not hold, it may still be possible to schedule the tasks. For tasks with harmonic periods for example, Han and Tyan showed in [HT97] that the utilization bound is 1.

Exact analysis and schedulability test by Lehoczky et al. [LSD89]

If systems were designed following the aforementioned bound, they would often waste resources. As the example before shows, the relationship between the tasks' periods influences the utilization factor and the schedulability. Lehoczky et al. showed that on average a utilization of 88% is the threshold for a random task set to be schedulable. Furthermore, they defined an exact schedulability test that does not rely on the worst-case analysis by Liu and Layland, which will be presented below. They define the demand of $i \leq n$ tasks on the processor as the following function:

$$W_i(t) = \sum_{j=1}^{i} C_j * \left\lceil \frac{t}{T_j} \right\rceil$$

If 0 is a critical instant, $W_i(t)$ is the demand of tasks $\tau_1, \ldots, \tau_i$ on the processor in the interval $[0, t]$. Furthermore, for convenience, they define

$$L_i(t) = \frac{W_i(t)}{t},$$
$$L_i = \min_{0<t\leq T_i} L_i(t),$$
$$L = \max_{1\leq i\leq n} L_i$$

and formulated the following Theorem:

Theorem 2.3.1 (Theorem 2 in [LSD89]) *Given the periodic task set* $\Gamma_n = \{\tau_1, \ldots, \tau_n\}$

1. *Task* τ_i *is schedulable using RM scheduling if and only if*

$$L_i \leq 1 \tag{2.1}$$

2. Γ_n *is schedulable using RM scheduling if and only if:*

$$L \leq 1 \tag{2.2}$$

Inequality (2.1) mandates that throughout the task period, there must be a point in time where the demand on the processor by τ_i and its higher priority tasks is ≤ 1. The inequality (2.2) then simply requires that this is the case for all tasks.
Furthermore, it is not necessary to check all time instants in the task's period, but only the scheduling points of τ_i, the deadline of τ_i, and the arrival times of tasks with higher priority. This is because $L_i(t)$ is piecewise monotonically decreasing, as $\lceil t/T_i \rceil / t$ is strictly decreasing except at a finite set S_i of time instants, which are the aforementioned scheduling points. They can be mathematically described in the following way:

$$S_i = \left\{ k * T_j | j = 1, \ldots, i; k = 1, \ldots, \left\lfloor \frac{T_i}{T_j} \right\rfloor \right\}$$

L_i can now be redefined to

$$L_i = \min_{t \in S_i} L_i(t)$$

The following example from [LSD89] visualizes these scheduling points.

- Task τ_1: $C_1 = 20; T_1 = 100; S_1 = \{100\}$
- Task τ_2: $C_2 = 40; T_2 = 150; S_2 = \{100, 150\}$
- Task τ_3: $C_3 = 100; T_3 = 350; S_3 = \{100, 150, 200, 300, 350\}$

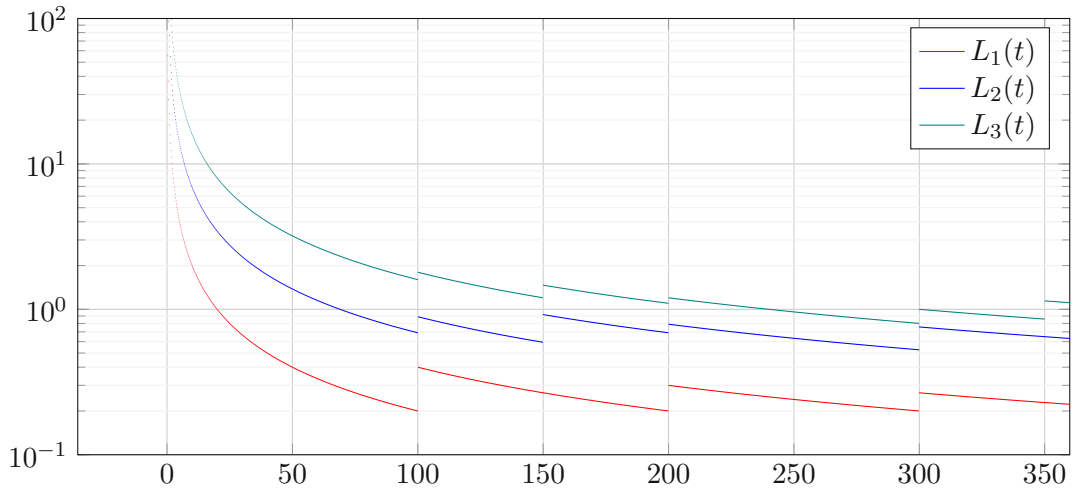


Figure 2.2: $L_i(t)$ example

The local minima of the functions in Figure 2.2 are the ones contained in S_i and also exactly those points where the functions "jump".

Formalization and Computationally Feasible Schedulability Test by Bini and Buttazzo [BB02]

Bini and Buttazzo formalized and improved this approach even further. They converted Theorem 2.3.1 by Lehoczky et al. into an expression using AND and OR operators.

Theorem 2.3.2 (Theorem 2 in [BB02]) *A periodic task set* $\Gamma_n = \{\tau_1, \ldots, \tau_n\}$ *is schedulable if and only if*

$$\max_{i=1,\ldots,n} \min_{t \in S_i} \frac{\sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j}{t} \leq 1 \Leftrightarrow \bigwedge_{i=1,\ldots,n} \bigvee_{t \in S_i} \frac{\sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j}{t} \leq 1$$

where

$$S_i = \left\{ k * T_j | j = 1, \ldots, i; k = 1, \ldots, \left\lfloor \frac{T_i}{T_j} \right\rfloor \right\}$$

Furthermore, they found a way to significantly reduce the number of equations that have to be checked by a schedulability test and developed a computationally feasible one, which is formalized in the following Theorem. This is necessary because the schedulability test proposed by Lehoczky et al. contains conditions that do not need to be checked, as they are OR-ed with a more relaxed one and just increase the computation time.

Theorem 2.3.3 (Theorem 3 in [BB02]) *Given the periodic task set* $\Gamma_n = \{\tau_1, \ldots, \tau_n\}$. *It is schedulable if and only if*

$$\bigwedge_{i=1,\ldots,n} \bigvee_{t \in \mathcal{P}_{i-1}(T_i)} \frac{\sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j}{t} \leq 1$$

where $\mathcal{P}_i(t)$ *is*

$$\begin{cases} \mathcal{P}_0(t) = \{t\} \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i \right) \cup \mathcal{P}_{i-1}(t) \end{cases}$$

To prove this theorem, Bini and Buttazzo defined a few new concepts.

Definition 2.3.1 (Definition 1 in [BB02]) *A job* τ_{ik} *is said to be active at time* t *if* $r_{ik} < t < f_{ik}$

Definition 2.3.2 (Definition 2 in [BB02]) *The processor is* i*-busy at time* t *if there exists a job of a task in* Γ_i *active at time* t. *The following set contains the points in the interval [0, b] where the processor is* i*-busy.*

$$Busy(\Gamma_i, b) = \{t \in [0, b] : \exists \tau_{jk} \text{ such that } \tau_{jk} \text{ is active at } t, \tau_j \in \Gamma_i\}$$

Definition 2.3.3 (Definition 3 in [BB02]) *The worst-case workload* $W_i(b)$ *of the* i *highest priority tasks in* $[0, b]$ *is the total time the processor is* i*-busy in* $[0, b]$. *By extension* $W_0(b) = 0$ *for all* b.

With the definition of the worst-case workload, the schedulability condition for τ_i can be reformulated to

$$C_i + W_{i-1}(T_i) \leq T_i$$

This inequality means that the computation time of τ_i together with the computation times of the higher priority tasks must not exceed the period of τ_i.

Definition 2.3.4 (Definition 4 in [BB02]) *Given the subset Γ_i of the i highest priority tasks, we define $\psi_i(b)$ to be the last instant in $[0, b]$ in which the processor is not i-busy.*

$$\psi_i(b) = \max_{t \in [0,b]} \{t \notin Busy(\Gamma_i, b)\}$$

With the definition of the last idle instant $\psi_i(b)$ in the interval $[0, b]$, they furthermore reformulated the worst-case workload in this interval in the following Lemma.

Lemma 2.3.1 (Lemma 1 in [BB02]) *Given a subset $\Gamma_i = \{\tau_1, \ldots, \tau_i\}$ of the i highest priority tasks. The workload $W_i(b)$ can be written as*

$$W_i(b) = \sum_{j=1}^{i} \left\lceil \frac{\psi_i(b)}{T_j} \right\rceil C_j + (b - \psi_i(b))$$

$\psi_i(b)$ is by definition the last instant at which the processor is not busy executing the i highest priority tasks. Therefore, the processor is busy in the interval $[\psi_i(b), b]$. The workload up to $\psi_i(b)$ is calculated by the sum of the lemma. It adds for every task how often it was activated and multiplies this amount with the computation time of the task. They also formulated the following second lemma.

Lemma 2.3.2 (Lemma 2 in [BB02]) *For any schedulable task set Γ_i*

$$W_i(b) = \min_{t \in [0,b]} \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j + (b - t)$$

To intuitively explain this lemma, we observe that $\sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j$ is the demand on the processor by the i highest priority tasks in the interval $[0, t]$. Therefore, $\forall t$ the following condition must hold

$$W_i(t) \leq \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j$$

because the actual workload cannot be greater than the demand. Moreover, in a feasible schedule, the workload in an interval cannot be greater than the length of the interval.

$$\forall t \in [0, b] \quad W_i(b) - W(t) \leq (b - t)$$

This can be rearranged to

$$\forall t \in [0, b] \quad W_i(b) \leq \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j + (b - t)$$

and shows that $\sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j + (b-t)$ is an upper estimate of the workload in $[0, b]$. Because $\psi_i(b)$ is in $[0, b]$ and because of lemma 2.3.1 there exists a t such that $t = \psi_i(b)$. This t must then also be the minimum of the upper estimation.
We now have shifted the problem from calculating the worst-case workload $W_i(b)$ of the i highest priority tasks for each task in the interval $[0, b]$ to finding $\psi_i(b)$.
However, according to the following lemma, also given by Bini and Buttazzo, $\psi_i(b)$ must be contained in the recurrently defined set $\mathcal{P}_i(t)$.

Lemma 2.3.3 (Lemma 3 in [BB02]) *Given a task subset Γ_i schedulable by RM, let $\psi_i(b)$ be the last idle instant in $[0, b]$ and let $\mathcal{P}_i$ be the set of points defined by the following expression:*

$$\begin{cases} \mathcal{P}_0(t) = \{t\} \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \cup \mathcal{P}_{i-1}(t) \end{cases}$$

Then,

$$\psi_i(b) \in \mathcal{P}_i$$

They prove this lemma by induction. For the initial step of $i = 1$ it is necessary to show that $\forall b\ \psi_1(b) \in \mathcal{P}_1(\mathrm{b})$. According to the recurrence relation

$$\mathcal{P}_1(b) = \mathcal{P}_0\left(\left\lfloor \frac{t}{T_1} \right\rfloor T_1\right) \cup \mathcal{P}_0(b) = \left\{\left\lfloor \frac{b}{T_1} \right\rfloor T_1, b\right\}$$

Because τ_1 is schedulable, the last idle instant $\psi_1(b)$ is either

1. at $\lfloor b/T_1 \rfloor T_1$ if the last instance of τ_1 in $[0, b]$ is active at b or
2. at b otherwise.

For the induction step, we need to show that if $\forall b\ \psi_i(b) \in \mathcal{P}_i(b)$, then $\forall b\ \psi_{i+1}(b) \in \mathcal{P}_{i+1}(b)$ and given a schedulable task subset Γ_{i+1}.
If we consider the time interval $[\lfloor b/T_{i+1} \rfloor T_{i+1}, b]$ we see that two things can happen:

1. The processor is $(i+1)$-busy in the whole interval or
2. there exists an instant of time at which the processor is not $(i+1)$-busy.

In the first case $\psi_{i+1}(b) = \psi_{i+1}(\lfloor b/T_{i+1} \rfloor T_{i+1})$, because in $[\lfloor b/T_{i+1} \rfloor T_{i+1}, b]$ the processor is always $(i+1)$-busy. Moreover, because τ_{i+1} is schedulable, $\psi_{i+1}(\lfloor b/T_{i+1} \rfloor T_{i+1}) = \psi_i(\lfloor b/T_{i+1} \rfloor T_{i+1})$. If this was not the case, τ_{i+1} would still be executing its last job in the interval $[0, \lfloor b/T_{i+1} \rfloor T_{i+1}]$ and therefore miss a deadline, but that cannot be the case, as τ_{i+1} is schedulable.
In the second case, we can take an instant $x \in [\lfloor b/T_{i+1} \rfloor T_{i+1}, b]$ where the processor is

not $(i+1)$-busy. Since the last job of τ_{i+1} in $[0, b]$ has already terminated at instant x, τ_{i+1} is not active anymore in $[x, b]$ and therefore $\psi_{i+1}(b) = \psi(b)$
When merging the two cases we get:

$$\psi_{i+1} = \psi_i(\lfloor b/T_{i+1} \rfloor T_{i+1}) \vee \psi_i(b)$$

The inductive hypothesis is $\psi_i(\lfloor b/T_{i+1} \rfloor T_{i+1}) \in \mathcal{P}_i(\lfloor b/T_{i+1} \rfloor T_{i+1})$ and $\psi_i(b) \in \mathcal{P}_i(b)$. Therefore,

$$\psi_{i+1}(b) \in \mathcal{P}_i(\lfloor b/T_{i+1} \rfloor T_{i+1}) \cup \mathcal{P}(b)$$

and

$$\psi_{i+1}(b) \in \mathcal{P}_{i+1}(b)$$

The following lemma combines lemmas 1, 2, and 3.

Lemma 2.3.4 (Lemma 4 in [BB02]) *Given a task subset Γ_i schedulable by RM and the set $\mathcal{P}_i(b)$ as defined in theorem 2.3.3, the workload $W_i(b)$ is*

$$W_i(b) = \min_{t \in \mathcal{P}_i(b)} \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j + (b - t)$$

With this at hand, we can now show why theorem 3 holds.
As the schedulability condition for τ_i can be written as

$$C_i + W_{i-1}(T_i) \leq T_i$$

we can substitute $W_{i-1}(T_i)$ which results in

$$C_i + \min_{t \in \mathcal{P}_{i-1}(T_i)} \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j + (T_i - t) \leq T_i$$

This can be transformed into

$$\min_{t \in \mathcal{P}_{i-1}(T_i)} C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j - t \leq 0$$

And because $\forall t \in \mathcal{P}_{i-1}(T_i) \quad \lceil t/T_i \rceil = 1$

$$\bigvee_{t \in \mathcal{P}_{i-1}(T_i)} \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j \leq t$$

As this needs to hold $\forall \tau_i \in \Gamma_n$, the formula stated in Theorem 3 results:

$$\bigwedge_{i=1,\ldots,n} \bigvee_{t \in \mathcal{P}_{i-1}(T_i)} \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j \leq t$$

This schedulability test is still state of the art and was used by Struhar et al. in in [SCA+23]. However, to apply this concept to the systems we deal with in this thesis, we need another concept called resource partitioning. This will be covered in Section 2.3.4

2.3.2 Dynamic Priority Scheduling

Dynamic priority scheduling assigns the priorities to individual jobs and not to tasks as a whole [SAÅ+04]. In EDF scheduling the job priorities are fixed and in Least Laxity First (LLF) scheduling, introduced in [Mok83], even the job priority is dynamic.

Earliest-Deadline-First Scheduling

The Earliest-Deadline-First (EDF) algorithm is one of the most popular ones in this category. The deadlines of the jobs depend on their execution time left in the current period [SAÅ+04]. Liu and Layland showed in their famous paper [LL73] that, given the same assumptions as for RM scheduling,

$$\sum_{i=1}^{n} U_i \leq 1$$

is a necessary and sufficient condition for the schedulability of a periodic task set Γ_n.

Optimality of Earliest-Deadline-First scheduling by [LL73]

They call the algorithm *deadline driven scheduling algorithm*, but there is essentially no difference to the EDF algorithm. Therefore, we call it EDF in this subsection.
Before they prove the previously stated utilization inequality, they define the following theorem.

Theorem 2.3.4 (Theorem 6 in [LL73]) *When the EDF algorithm is used to schedule a set of tasks on a processor, there is no processor idle time prior to an overflow.*

They prove this theorem by contradiction. Assume that there is an idle period before an overflow. The time interval considered starts at 0. They let $t_3 > 0$ denote the instant at which an overflow occurs, and t_1 and t_2 are the beginning and end of the idle period closest to t_3, respectively. Between t_2 and t_3 are multiple requests at random instants such that the processor is always busy in the interval $[t_2, t_3]$. Then they move all requests of all tasks in this interval to t_2. By doing so, the overflow now occurs still at t_3 or even earlier. They conclude that after all tasks are moved, there is an overflow without a processor idle period before it. However, this is a contradiction to the assumption that there is an idle period before an overflow. To further clarify this theorem: The important point is that there is no processor idle time between the first request of a task and an overflow. Before the first request, the processor can, of course, be idle. This theorem is then used to prove the following theorem

Theorem 2.3.5 (Theorem 7 in [LL73]) *For a given set of n tasks, the EDF algorithm is feasible if and only if*

$$\sum_{i=1}^{n} U_i \leq 1$$

The necessity follows immediately because it is impossible to schedule a task set if the utilization of the processor is greater than 1.
The sufficiency is not trivial, and again they prove it by contradiction. Assume $\sum_{i=1}^{n} U_i \leq 1$, but there is still an overflow between $t = 0$ and $t = T_1 T_2 \dots T_n$ because the scheduling is not feasible. According to Theorem 2.3.4 there must not be an idle period before the overflow at time $t = T$. Then they let $a_1, a_2, \dots, b_1, b_2, \dots$ denote the request times of the n tasks directly before T where $a_1, a_2, \dots$ denote the requests times of the tasks that have their deadline at T and $b_1, b_2, \dots$ the request times of the tasks that have their deadline after T.
Then they examine the two possible cases:
Case 1: Only computations requested at $a_1, a_2, \dots$ were carried out before T. Then the total demand on the processor is

$$\left\lfloor \frac{T}{T_1} \right\rfloor C_1 + \left\lfloor \frac{T}{T_2} \right\rfloor C_2 + \dots + \left\lfloor \frac{T}{T_n} \right\rfloor C_n$$

The computation times of the tasks that have a period greater than T, the ones that were requested at $b_1, b_2, \dots$ are not added to this sum.
Because there is no idle period and there is an overflow, the demand must be greater than the length of the interval.

$$\left\lfloor \frac{T}{T_1} \right\rfloor C_1 + \left\lfloor \frac{T}{T_2} \right\rfloor C_2 + \dots + \left\lfloor \frac{T}{T_n} \right\rfloor C_n > T$$

Additionally, because $x \geq \lfloor x \rfloor$

$$\frac{T}{T_1} C_1 + \frac{T}{T_2} C_2 + \dots + \frac{T}{T_n} C_n > T$$

which can be transformed to

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \dots + \frac{C_n}{T_n} > 1$$

which is a contradiction to the trivial necessary condition.
Case 2: Some of the computations requested at $b1, b_2, \dots$ have been executed before T. Because an overflow happens at T there must be an instant $T' < T$ such that none of the computations requested at $b1, b2, \dots$ are executed between T' and T. This is because of the dynamic EDF scheduling algorithm. Additionally, those tasks of the b_i's that were executed before T' must be completed. This is again because of EDF. The demand on the processor is therefore

$$\left\lfloor \frac{T - T'}{T_1} \right\rfloor C_1 + \left\lfloor \frac{T - T'}{T_2} \right\rfloor C_2 + \dots + \left\lfloor \frac{T - T'}{T_n} \right\rfloor C_n$$

Again, because an overflow occurs at T, the demand in $T - T'$ must be greater than the time available and therefore

$$\left\lfloor \frac{T - T'}{T_1} \right\rfloor C_1 + \left\lfloor \frac{T - T'}{T_2} \right\rfloor C_2 + \dots + \left\lfloor \frac{T - T'}{T_n} \right\rfloor C_n > T - T'$$

which again leads to the contradictive inequality

$$\frac{C_1}{T_1} + \frac{C_2}{T_2} + \cdots + \frac{C_n}{T_n} > 1$$

Any feasible scheduling can be converted to a scheduling that would have been generated by EDF. Therefore, EDF is optimal. The proof was given in [Der74] [Lip08].

2.3.3 Aperiodic Servers

Apart from the periodic task sets considered so far, there can also exist aperiodic tasks in a system. These are, for example, tasks that handle operator requests. They are characterized by unknown arrival times without any constraints.
When dealing with such systems the goal is to improve QoS properties of the aperiodic tasks while not violating the hard deadlines of the periodic tasks.
In the early days of RT systems two mechanisms were used to handle aperiodic tasks: *Background processing* and *polling*. Background processing executes aperiodic requests when the processor is idle because no periodic task is executing. However, this leads to bad response times, because depending on the utilization, it may take a long time until the processor is idle. Polling means creating a periodic task that repetitively checks if an aperiodic task is ready to execute and which lends its execution time to the aperiodic task. This provides better response times, but there is still room for improvement because, in the worst case, the aperiodic task has to wait until the next poll. This is why Lehoczky et al. introduced the Priority Exchange and Deferrable Server algorithms [SSL89].
The following paragraphs introduce some servers that have historical value and influenced some papers relevant to this thesis. As the servers themselves are not the topic of this thesis, their correctness will not be shown.

Priority Exchange

The Priority Exchange algorithm tries to provide a high average response time to aperiodic tasks. To achieve this, the server has execution capacity on different priority levels and tries to execute an aperiodic task on the highest priority possible. The server's execution capacity is replenished every period on the highest priority. If there is no aperiodic task to execute, the capacity gets exchanged to the next lower priority, and the lower priority periodic task can execute with the higher priority. The execution time for aperiodic tasks is in this way accumulated at a lower priority level until an aperiodic request occurs. This way it can be serviced immediately at the request, and the algorithm therefore provides a high average response time.

$$U_p = \ln \frac{2}{U_s + 1}$$

expresses the highest processor utilization for which RM can always schedule the tasks, U_p, via the server size U_s for this algorithm. The server size is its execution budget divided by its period. This equation can also be interpreted inversely. For example, for a periodic load, $U_p = 0.6$ the server size is $U_s = 0.1$. [SSL89]

Deferrable Server

The Deferrable Server (DS) is also a so-called *bandwidth preserving algorithm*. If the periodic task of the server has some of its execution time available when an aperiodic task is requested, it is executed immediately. This algorithm is very easy to implement. At the beginning of every period, the server quota is replenished. If an aperiodic task needs more CPU time than the server has available, it needs to wait until the next period. The downside of this simplicity is a smaller server size, which means it can not provide as much execution time for aperiodic tasks as the priority exchange server.

$$U_p = \ln \frac{U_s + 2}{2U_s + 0.1}$$

$U_p = 0.6$ results in $U_s = 0.07$.
For this and the previous server to guarantee deadlines of sporadic tasks, it is required that the minimum inter-arrival time of the aperiodic tasks does not exceed the period of the server [SSL89].

Sporadic Server

The sporadic server (SS) combines the benefits of the previous two servers: The simplicity of the DS and the server size of the PE. The sporadic server maintains its priority level as the DS does, but the replenishment does not happen regularly, but only after an aperiodic task is requested. If the server has a period of T_s and an aperiodic request occurs at time t, the server is replenished at time $t + T_s$. This also reduces the replenishment overhead [SSL89].

Dynamic Sporadic Server

In [SB96] Spuri and Buttazzo extended the Sporadic Server by Sprunt et al. in [SSL89] to work also with an EDF scheduler. The Dynamic Sporadic Server (DSS) can be in three different states called IDLE, in which the server does not need to be executed, READY, in which the server is not executed, but it could be, and EXE, in which the server is executed. When the server starts, it is in the READY state. If there is an aperiodic task and the server has a budget available, it switches to the EXE state. If the server gets preempted by a higher-priority task, it switches back into the READY state. If the budget is empty, it switches into the IDLE state. If the server is in the IDLE state, an aperiodic request occurs, and the system has a budget available, or if the budget gets replenished, it switches to the READY state and, in addition to replenishing its budget at $t + T_s$, also sets its deadline to this time instant.

Total Bandwidth Server

Using a SS it still can happen that the response time of an aperiodic task can be significantly long. Spuri and Buttazzo, therefore, introduce in [SSL89] a new algorithm called the Total Bandwidth Server. They try to decrease the response time of the server

not by decreasing its period T_s, but by decreasing the periods of the aperiodic tasks. Instead of setting the deadline d_k of the k-th aperiodic request equal to the time of replenishment of the server, it is now calculated with the following formula:

$$d_k = \max(r_k, d_{k-1}) + \frac{C_k}{U_s}$$

where C_k is the computation time of the request and U_s is the utilization of the server. The first part of the sum makes sure that the computation time of the aperiodic tasks never exceeds U_s if the aperiodic tasks respect their declared C_k. This is also where this server has its name from because the total bandwidth available is not exceeded.

Constant Bandwidth Server

The DSS and TBS were influential for the Constant Bandwidth Server (CBS) introduced by Abeni and Buttazzo in [AB98] to schedule soft RT tasks. The CBS guarantees, compared to TBS, that U_s is also not exceeded when there is an overload on the system because the aperiodic tasks exceed their C_k.
A CBS has a computation budget c_s, a maximum budget Q_s, and a period T_s. The server size $U_s = Q_s/T_s$. c_s is only recharged when it reaches 0 and from a theoretical point of view, there is no finite instant where the budget is equal to zero. After recharging, the deadline of the server is set to $d_{s,k+1} = r_{i,j} + T_s$. The server is said to be active at time t, if there exists a job $\tau_{i,j}$ such that $r_{i,j} < t < f_{i,j}$. Otherwise, the server is idle. When a job $\tau_{i,k}$ arrives and the server is active, it is added to a queue. If the server is idle and $c_s \geq (d_{s,k} - r_{i,j})U_s$, the server generates a new deadline $d_{s,k+1} = r_{i,j} + T_s$ and c_s replenishes to Q_s. This is because the current budget is greater than what the server is allowed to use until the deadline according to the server size U_s. To prevent an overload, the deadline of the server is extended. This may extend the response time of the job, but as Abeni and Buttazzo developed this server for soft RT tasks, this is no problem. Otherwise, the job is served with the current server deadline $d_{s,k}$.

2.3.4 Resource Partitioning

When implementing an RT system, it is necessary to check the schedulability of the tasks in the system. When all tasks that will ever run in the system are known at design time, it is possible to conduct a global schedulability analysis as described in Section 2.3.1. However, in an open environment where new tasks may be added during runtime, it would be necessary to conduct the schedulability analysis again for every new task [DL97]. To avoid this, hierarchical scheduling frameworks have been developed. One example is explained in detail in the following paragraph.

Periodic Resource Model by Shin and Lee [SL03]

In the periodic resource model, the resource, in this case the CPU, is partitioned among the tasks of the system. Each task then receives a partition $\Gamma(P, Q)$ of the resource where

Q is the time for which the task will have access to the resource every period P. As in one period the task may get the resource at the beginning and in the next period at the end, the availability can jitter significantly. To analyze the schedulability of a new task in the system, two functions are of interest that describe two somewhat opposite things: The supply bound function

$$\mathrm{sbf}_\Gamma(t) = \left\lfloor \frac{t-(P-Q)}{P} \right\rfloor Q + e_s$$

where

$$e_s = \max(t - 2(P-Q) - P\left\lfloor \frac{t-(P-Q)}{P} \right\rfloor, 0)$$

which shows how much supply the task gets at least in a time interval of length t and the linear service time-bound function

$$\mathrm{tbf}_\Gamma(t) = (P-Q) + P\left\lfloor \frac{t}{Q} \right\rfloor + e_t$$

where

$$e_t = \begin{cases} P - Q + t - Q\left\lfloor \frac{t}{Q} \right\rfloor & \text{if } t - Q\left\lfloor \frac{t}{Q} \right\rfloor > 0 \\ 0 \quad \text{otherwise} \end{cases}$$

which shows how long the service time needs to be to get a supply time of t. The *service time* is defined as the duration it takes for the resource to provide a certain supply.
$\mathrm{sbf}_\Gamma(t)$ can be intuitively explained in the following way. To get the minimum supply, we take t, the length of the interval, and subtract $(P-Q)$, which is the time of a period where the task does not have the resource. This subtraction is necessary to get the most pessimistic view on the interval t. $\left\lfloor \frac{t-(P-Q)}{P} \right\rfloor$ gives us the number of complete periods in $t-(P-Q)$ and multiplied with Q the amount of supply in $t-(P-Q)$. e_s is then added to get any additional supply in a period that is not fully in the interval t. It is calculated by subtracting the duration of the full periods $P\left\lfloor \frac{t-(P-Q)}{P} \right\rfloor$ from the interval and, additionally, twice $(P-Q)$ which is time outside the full periods where the task potentially does not have the resource. Everything left in the interval must be a time when the task has the resource.
For $tbf_\Gamma(t)$ the calculation is similar. $(P-Q)$ again stands for the time the task has to wait for the resource. $\lfloor t/Q \rfloor$ is the number of full periods needed to get the desired supply time and is then multiplied by P. The term e_t is necessary if the interval t is not a multiple of Q. In that case another $(P-Q)$ must be added as well as the remaining computation time itself $t - Q\lfloor t/Q \rfloor$ to get the maximum service time.
These two functions are non-decreasing step functions, as can be seen in Figure 2.3. Linear lower and upper bounds can be given, respectively, by the following two functions.

$$\mathrm{lsbf}_\Gamma(t) = \frac{Q}{P}(t - 2(P-Q)) \leq sbf_\Gamma(t)$$

$$\mathrm{ltbf}_\Gamma(t) = \frac{P}{Q}t + 2(P-Q) \geq tbf_\Gamma(t)$$

These bounds can be proven by simple mathematical transformations, as done in [SL03].

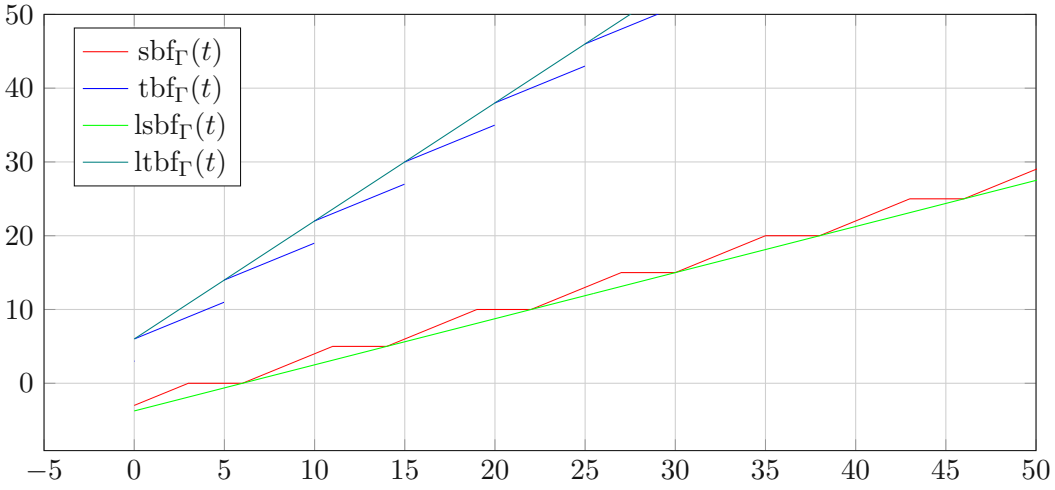


Figure 2.3: $sbf_\Gamma(t)$ and $tbf_\Gamma(t)$ functions and their linear bounds for $P = 8$ and $Q = 5$

Optimal Server Parameters by Lipari and Bini [LB03]

When using such a periodic resource model, it is necessary to know for a given task set Γ_n: For which server parameters (P, Q) is the task set schedulable. Lipari and Bini went in [LB03] even a step further and formally defined how the optimal server parameters can be found.
To get the optimal parameters for a task set, it is necessary to modify the schedulability condition.
They model the sharing of resources in a resource partitioning system by reducing the processor speed or by increasing the computation times of the applications, which is effectively the same thing. To find the factor by how much the processor can be slowed down at most, the equation describing the scheduling condition (Theorem 2.3.3) can be modified to reflect the throttled processor.

$$\bigwedge_{i=1,\dots,n} \bigvee_{t \in \mathcal{P}_{i-1}(D_i)} \sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil \frac{C_j}{\alpha} \leq t$$

$$\bigwedge_{i=1,\dots,n} \bigvee_{t \in \mathcal{P}_{i-1}(D_i)} \alpha \geq \frac{\sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j}{t}$$

$$\alpha \geq \alpha_{min} = \max_{i=1,\dots,n} \min_{t \in \mathcal{P}_{i-1}(D_i)} \frac{\sum_{j=1}^{i} \left\lceil \frac{t}{T_j} \right\rceil C_j}{t}$$

In the final inequality α_{min} is the minimal factor by which the processor can be slowed down for the task set to still be schedulable.
Additionally, an initial delay Δ has to be considered when optimizing the server parameters. This is because it is possible that a task has to wait for the execution after its request

because the server has no budget left in its current period. The worst case is if the task then also has to wait for the maximum time in the next period until it gets executed. Now they take α and Δ into consideration regarding the schedulability of a task τ_i. From Lemma 2.3.4 they get the schedulability condition of a task using the workload of the i highest priority tasks for an environment without a server.

$$C_i + \min_{t \in \mathcal{P}_{i-1}(D_i)} \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j + (D_i - t) \leq D_i$$

When using a server, the parameters defined previously α and Δ need to be added to the equation.

$$\Delta + \frac{C_i}{\alpha} + \min_{t \in \mathcal{P}_{i-1}(D_i)} \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil \frac{C_j}{\alpha} + (D_i - t) \leq D_i$$

This condition can be transformed into

$$\Delta \leq D_i - \left(\frac{C_i}{\alpha} + \min_{t \in \mathcal{P}_{i-1}(D_i)} \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil \frac{C_j}{\alpha} + (D_i - t) \right)$$

After simplifying we obtain an equation to directly calculate the maximum initial delay possible.

$$\Delta \leq \max_{t \in \mathcal{P}_{i-1}(D_i)} t - \frac{1}{\alpha} \left(C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j \right)$$

Theorem 2.3.6 (Theorem 3 in [LB03]) *A task set Γ_n is schedulable by a server characterized by the lower bound supply function $lsbf_\Gamma(t)$ if:*

$$\Delta \leq \min_{i=1 \ldots n} \max_{t \in \mathcal{P}_{i-1}(D_i)} t - \frac{1}{\alpha} \left(C_i + \sum_{j=1}^{i-1} \left\lceil \frac{t}{T_j} \right\rceil C_j \right)$$

where

$$\begin{cases} \mathcal{P}_0(t) = \{t\} \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1} \left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i \right) \cup \mathcal{P}_{i-1}(t) \end{cases}$$

This theorem only adds the condition that for all tasks the length of the interval $[0, t]$ minus the response time must be greater than the initial delay.
With these formulas, it is now possible to calculate α and Δ explicitly. When designing a server, that is, choosing the parameters (P, Q), they can be calculated as follows.

$$\Delta = 2(P - Q) \quad \alpha = \frac{Q}{P}$$

$$P = \frac{\Delta}{2(1 - \alpha)} \quad Q = \alpha P$$

However, there are no optimal parameters in the sense that they fit all scenarios because there are two different aspects that have to be considered.

1. The server period should be small to get a low bandwidth, meaning that CPU usage is low.
2. The server period should be large to reduce the time spent on context switches.

The consideration of context switch time removes constraint number 6 in the list defined by Liu and Layland.
This optimization problem can be expressed in the following cost function, where $T_{overhead}$ is the time lost during a context switch.

$$c_1 \frac{T_{overhead}}{P} + c_2 \alpha$$

The two constants c_1 and c_2 need to be defined at design time and allow tuning of the system. To obtain optimal server parameters, the previous function needs to be minimized.

The formula in Theorem 2.3.6 and the cost function are also used by Struhár in [SCA+23]. Therefore, we have explained all the theoretical concepts relevant to this thesis.

CHAPTER 3

Offline Phase

The offline phase aims to find an initial placement of containers to nodes and define which container gets how much execution time over which period. This problem is similar to the bin-packing problem and is hence NP-complete in complexity. In [SCA+23] the initial placement and dimensioning of containers were done with the use of an SMT solver that does not scale for larger systems. This thesis contains the description and implementation of three alternative solvers. Two of them are based on heuristics and one is also based on an SMT solver. All of them provide more scalability for increasing system sizes at the expense of sacrificing some of the optimality of the obtained solutions.

3.1 Theory

In Section 2.3 we analyzed RT scheduling from a theoretical point of view and found formulas on how to fulfill the RT requirements of the tasks in the system and how the system itself has to be parametrized. However, when implementing these formulas in the form of algorithms on a computing device, the resulting complexity of these algorithms is often NP-hard or NP-complete [LKB77]. Therefore, we can expect computation times that grow exponentially with the problem size, if we try to solve them with currently known algorithms. This is why for NP-hard and NP-complete problems or any problem in NP it is in general not possible to find an optimal solution. Only for small instances of these problems, algorithms might find one in a feasible amount of time. [HF04]

Instead of designing custom algorithms to solve the problem, it is also possible to use more general approaches, e.g. a satisfiability modulo theories (SMT) solver or integer linear programming (ILP), to find optimal solutions. Usually at the cost of impractically long computation times. Heuristics, on the other hand, often provide faster computation times by sacrificing optimality. These two approaches are described in the following two subsections.

3.1.1 Optimal solutions

The most basic approach to finding optimal solutions for the allocation and dimensioning problem at hand would be a brute-force algorithm, which tries to find a solution by iterating over all possible parameters. However, this approach is well-known to be inefficient. A different approach is ILP where the scheduling constraints need to be converted into inequalities which can then be solved by an ILP solver. However, it can be shown that ILP is NP-hard and, therefore, as described in Section 3.1, solutions can usually not be computed in a feasible amount of time. Alternatively, an SMT solver can be used. This requires the problem to be formulated as first-order logic constraints. SMTs are a generalization of the problem of boolean satisfiability (SAT). The SAT problem asks the question: Is there an interpretation that satisfies the given boolean formula? SMT extends this problem by allowing variables to obtain integers and real values and other concepts such as arrays or lists. Furthermore, the formulas can be expressed in first-order logic, also allowing quantifiers instead of propositional logic only [BT18]. As SAT is already NP-complete, SMT is typically NP-hard or even undecidable, depending on the theory. By restricting the formulas to certain theories, for example only integers or no quantifiers, the computation time can be reduced [BT18], but in the general case, it is still not possible to find optimal solutions to the problem in a feasible amount of time. There exist multiple efficient SMT solvers (e.g. CVC5 [BBB+22], openSMT [BPST10], Z3 [DMB08], Yices [DDM06, Dut14]). They compete against each other in the yearly SMT-COMP ([WCD+19]) and are also used in many application areas such as model checking [KGC16], scheduling [CO16], optimization [BT18], automated test case generation [PVL11], and static analysis [SK11].

3.1.2 Heuristics

Heuristic Algorithms can provide results for optimization problems within short computation times. This is achieved by not searching the entire search space. However, this does not guarantee a short computation time but only makes it more likely. Additionally, heuristics do not provide a guarantee that the result is within a certain range of the optimal result.
Heuristic algorithms are typically analyzed experimentally [Sha04]. There are heuristics specially tailored to certain problems, e.g., the traveling salesman problem [LK73] and so-called metaheuristics. They make use of concepts from other domains such as physics or biology and provide general algorithms for different unrelated problems. Simulated Annealing is an example of a metaheuristic and is used in this thesis to replace the SMT solver-based approach in the container allocation problem [OK96].

Simulated Annealing

The algorithm was initially introduced by Kirkpatrick et al. in [KGJV83] and emulates the physical process of slowly cooling a solid. The goal of the real process is to transform a solid into a low-energy state, which is a state where the atoms are highly ordered,

e.g. a crystal lattice. This is for example important in semiconductor manufacturing to get defect-free silicon crystals. To achieve this structure, the material is heated to a temperature where many atomic reorderings can occur. Then it is slowly cooled until it arrives at a state where the structure is considered good. The algorithm simulates this process by transforming a bad, unordered solution into an optimized one [Rut89].
The algorithm 3.1 describes simulated annealing in pseudocode. The algorithm tries to

Algorithm 3.1: Simulated Annealing Algorithm

```
1  s ← s_0;
2  for i = 0; i < i_max; i++ do
3  |   T ← temperature(i);
4  |   s_new ← neighbor(s);
5  |   if f(s_new) ≤ f(s) then
6  |   |   s ← s_new;
7  |   else
8  |   |   if random(0,1) < exp(−(f(y)−f(x))/T) then
9  |   |   |   s ← s_new;
10 |   |   end
11 |   end
12 end
13 return s
```

minimize a cost function. Therefore, the problem at hand must be expressed in such a function, custom to the problem. This is also why the heuristic applies to all different kinds of problems. In each step, a new neighbor solution (`neighbor(s)`), depending on the current solution s, is computed, and the cost function is evaluated. A new neighbor means to not choose a completely random solution but to alter for example only one parameter in the domain of the function or multiple parameters by small amounts. If the result is better, the new solution is saved and otherwise discarded. However, to overcome local minima, with a certain probability worse solutions are also accepted. This probability depends on the temperature (`temperature()`) and two parameters for the exponential function x and y. As the temperature decreases with every step, the probability of accepting worse solutions decreases as well. The algorithm can either stop after a fixed amount of iterations or after the function evaluates to a value below a threshold.
To successfully utilize this approach, the following parameters have to be tuned depending on the application [Rut89]:

1. Domain: The values s can obtain need to be specified.

2. Neighbor function: `neighbor(s)` returns a solution similar but not equal to its input.

3. Cost function: $f(s)$ needs to reflect how good the current solution s_{new} is.

4. Cooling schedule: `temperature(i)` defines the development of the temperature over time. Its only constraint is that it must be monotonically decreasing. The crucial aspects are the starting temperature when the temperature should be lowered by how much, and when to stop the annealing.

In [XGSH13] Tsallis presented an implementation of the Generalized Simulated Annealing algorithm. In this algorithm, the neighbor function is not exclusive to the application domain, but a general function choosing integer or real numbers from a given space. In detail, a distorted Cauchy-Lorentz distribution generates jumps across the search space.

$$g_{q_v}(\Delta x(t)) \propto \frac{(T_{q_v}(t))^{-\frac{D}{3-q_v}}}{\left(1+(q_v-1)\frac{(\Delta x(t))^2}{(T_{q_v}(t))^{-\frac{2}{3-q_v}}}\right)^{\frac{1}{q_v-1}+\frac{D-1}{2}}}$$

t is the artificial time, $\Delta x(t)$ specifies to distance of the trial jump of variable $x(t)$, and $T_{q_v}(t)$ is the artificial temperature. If the jump worsens the solution, it is only accepted by a probability calculated using the generalized Metropolis algorithm:

$$\mathcal{P}_{q_a} = \min 1, (1-(1-q_a)\beta\Delta E)^{\frac{1}{1-q_a}}$$

where q_a is a parameter of the algorithm. The temperature is decreased according to

$$T_{q_v}(t) = T_{q_v}(1)\frac{2^{q_v-1}-1}{(1+t)^{q_v-1}-1}$$

The parameters q_v and q_a allow tuning the algorithm's behavior, from Classical Simulated Annealing (CSA) to Fast Simulated Annealing (FSA [TS96]) and variants with even faster cooling rates. The benefit of Generalized Simulated Annealing is the large probability of long jumps and therefore a higher chance of finding the global minimum compared to CSA and FSA.

Generalized Simulated Annealing is also used in this thesis.

Greedy Algorithms

According to Curtis ([Cur03]), greedy algorithms make choices that are the best at the time and never revert them or go back to an earlier point in time. They are often simple, and for some problems, they can also yield optimal results. For example when searching for minimum-cost spanning trees or shortest paths in graphs. Furthermore, greedy algorithms can be used as heuristics in optimization problems where no algorithm exists that yields an exact solution in a feasible amount of time.
The following paragraph describes the custom greedy algorithm we developed to compute a feasible container allocation.

Custom Greedy Algorithm The main idea of the greedy algorithm is to drastically reduce the search space further, considering only a fraction of the possible periods and quotas for the containers. This way the solutions may be far from optimal in terms of the cost function, but hopefully a feasible allocation can be found much faster.
The Pseudo code for the algorithm is given in algorithm 3.2.
The variable `possible_periods` contains the periods which the algorithm can choose from. How these periods are chosen in our implementation is described in Section 3.2.2. Generally, it is possible that the lists do not have the same length for every container as for some containers more periods might be possible than for others. Therefore, the variable `max_possible_periods` contains the maximum length of all lists. The `task_utilizations` variable contains a list consisting of the utilization of the containers.
At the start of each search run, a random container-to-node assignment is generated. If it is infeasible in terms of the memory and stack requirements of the containers, a new assignment is generated until the constraints are satisfied. This is the only part of this algorithm that is indeterministic and also the reason for the outmost while loop. If the initial assignment is feasible in terms of memory and stack, it is still possible that it is impossible to fulfill the timing requirements of the tasks.
The outer for-loop iterates over the possible periods. The inner for-loop over the containers in the system. Therefore the algorithm tries to assign the same period to all containers. Due to the difference in length of the `possible_periods` list, the `index` variable is introduced to prevent an index-out-of-bounds error. Most of the logic serves the purpose of computing the quota for the containers. An initial quota is guessed by setting it to the utilization of the container. Followingly, the algorithm tries to decrease it as much as possible. This is achieved by subtracting the maximum quota found to this point that does not work and halving the result. Followingly, the algorithm checks if the tasks inside the container are schedulable with the given quota. If so, the `min_working` variable is updated, which contains the smallest quota found that allows feasible scheduling. If the quota does not allow a feasible scheduling the quota is doubled. This chain continues until a quota is hit, which has already been considered. In the end, the quota is set to `min_working`. This is done for all containers. If the whole system is schedulable, then the algorithm terminates. Otherwise, a new search run is started. If no scheduling is found after `SEARCH_RUNS` runs, the algorithm returns -1.

3.2 Implementation

This section goes into detail on how the approach in [SCA+23] was replaced in this thesis. The main goal was to show that an SMT solver can not create server parameters for bigger systems, but that heuristics are necessary to do so. Three different heuristics, Simulated Annealing, a custom greedy heuristic, and a different SMT solver-based solution were then built into the scheduling tool and evaluated. The results are shown in Section 5.1. The following subsections provide insights into how the systems are modeled in the tools, how the different solvers were implemented, and how the tool was generally improved.

Algorithm 3.2: Custom Greedy Algorithm

```
1  SEARCH_RUNS ← 100;
2  possible_periods ← list of list of possible periods per container;
3  max_possible_periods ← max(length of all lists in possible_periods);
4  task_utilizations ← list of utilizations of every container;
5  count ← 0;
6  while count < SEARCH_RUNS do
7      count = count + 1;
8      system.assign_containers_randomly();
9      while system.evaluate_memory_and_stack() == False do
10         system.assign_containers_randomly();
11     end
12     for i = 0; i < max_possible_periods; i++ do
13         for j, container in enumerate(system.containers) do
14             index ← min(i, len(possible_periods) − 1);
15             container.period ← possible_periods[j][index];
16             container.quota ← max(1, int(container.period∗task_utilization[j]));
17             max_not_working ← 0;
18             visisted ← [];
19             min_working ← container.period;
20             while container.quota not in visited do
21                 visited.append(container.quota);
22                 if system.check_scheduling_one_container(container) == 0
                    then
23                     min_working ← min(container.quota, min_working);
24                     container.quota ← container.quota − int((container.quota −
                        max_not_working)/2);
25                 else
26                     max_not_working ←
                        min(container.quota, max_not_working);
27                     container.quota ← min(container.quota, container.quota ∗ 2);
28                 end
29             end
30             container.quota ← min_working;
31         end
32         if system.evaluate() == False then
33             return system;
34         end
35     end
36 end
37 return − 1;
```

	Type	Interval
name	str	-
memory	int	[80000, 100000]
stack	int	[1000, 2000]
assigned_containers	[Container]	-

Table 3.1: Node Attributes

	Type	Interval	Set
name	str	-	-
wcet	int	[2, 20]	-
period	int	-	[100, 200, 500, 1000, 2000, 5000, 10000, 20000, 100000]
deadline	int	equal to period	-
memory	int	[16, 64]	-
stack	int	[2, 8]	-

Table 3.2: Task Attributes

3.2.1 Model

The system is represented in a model with a different file for each part, e.g. Node, Container, Task. The time granularity used in the tool is $1\mu s$.

Node

A physical computing node in the system is represented via a simple Python dataclass. Table 3.1 gives an overview of the attributes of a node. The first three attributes are set when the system is read from a file and therefore generated by the system generator, described in Section 3.2.3. The generator sets the available memory and stack space on each node to a random value in the interval specified in the interval column. This range can be specified in the `config.json` file in the `systems/` folder. Throughout this thesis, the values given in the table were used.

Task

A task is also represented via a Python dataclass. Its attributes are a name, the WCET, the period, the deadline, which always equals the period in this thesis, and the memory disk storage it requires. All of these attributes are set by the system generator and apart from the name, chosen randomly. Table 3.2 gives an overview of the attributes. The periods are chosen randomly from the set given in the table. These periods are according to [KZH15] common to automotive RT systems.

Container

A container object represents a Docker container and contains multiple tasks. The tasks are fixed and can not switch between containers. To reduce the complexity of the evaluation of the tool, a container always contains four tasks. The attributes of a container are its name, the quota, and the period, which need to be calculated by the tool, the node it is assigned to, also computed by the tool, and the list of task objects assigned to it. Two properties of the class return the memory and stack requirements of the container, which is equal to the sum of the requirements of the tasks.
The container class provides a method (`get_intervals()`) to compute the set of intervals $\mathcal{P}$ explained in theorem 2.3.6. To reduce the computation time of the heuristics, they are computed once upfront and used for every set of parameters the heuristic tries. The method `higher_priority_demand(index, p_in_time)` returns the demand of all higher priority tasks until the point in time given. This is used to check schedulability.
The container class has additional attributes and methods to add constraints to the SMT solver.

System

A system object contains the nodes and the containers of the system. Additional attributes are the name of the system and the seed with which the random number generator was seeded when the system was created. The system class has properties to get the schedulability of the system and how many nodes and containers it contains. The `check_system_detailed()` prints information on why the system is not schedulable, if this is the case. The `print_current()` function prints the system as a human-readable string to the console.

3.2.2 Solvers

The goal of the three solvers presented in this thesis is to find an assignment of containers to nodes and a pair of quota and period values for every container. This optimization problem has been defined in [SCA+23] and is listed here for completeness.
The notation is as follows: Every container $\pi_k, k = 1, \ldots, m$ has a set of boolean variables $v_1^k, \ldots, v_n^k$ specifying whether the container is assigned to node n or not. P_k is the containers period and Q_k its budget. Superscript S denotes the stack requirements of a container and superscript M the memory requirements. $f_j, j = 1, \ldots, n$ represents a node in the system. C_i^j is the WCET of task i on node j, which means the tasks can have different WCET on different nodes.

$$\min_{\{Q_k, P_k, v_1^k, \ldots, v_n^k | k=1,\ldots,m\}} c_1 * \sum_{k=1}^{m} \frac{\sum_{j=1}^{n} v_j^k * \delta_j}{P_k} + c_2 * \sum_{k=1}^{m} \frac{Q_k}{P_k} \tag{3.1}$$

subject to:

$$\forall j = 1, \ldots, n : \sum_{k=1}^{m} v_j^k * \pi_k^M \leq f_j^M \tag{3.2}$$

$$\forall j = 1, \ldots, n : \sum_{k=1}^{m} v_j^k * \pi_k^S \leq f_j^S \tag{3.3}$$

$$\forall j = 1, \ldots, n : \sum_{k=1}^{m} v_j^k \frac{v_j^k * Q_k}{P_k} \leq 1 \tag{3.4}$$

$$\forall k = 1, \ldots, m, \forall j = 1, \ldots, n : v_j^k = \{0, 1\} \tag{3.5}$$

$$\forall k = 1, \ldots, m : \sum_{j=1}^{n} v_j^k = 1 \tag{3.6}$$

$$\forall k = 1, \ldots, m : \frac{Q_k}{P_k} \geq \sum_{\tau_i \in \mathcal{T}^k} \frac{\sum_{j=}^{n} v_k^j * C_i^j}{T_i} \tag{3.7}$$

$$\forall k = 1, \ldots, m : \bigwedge_{\tau_i \in \mathcal{T}^k} \bigvee_{t \in \mathcal{P}_{i-1}(D_i)} t - 2 * (P_k - Q_k) - \frac{P_k}{Q_k} (\sum_{j=1}^{k} v_j^k * C_i^j + \sum_{\substack{\tau_l \in \mathcal{T}^k \\ p_l \geq p_i}} \sum_{j=1}^{n} \left\lceil \frac{t}{T_l} \right\rceil * v_j^k * C_l^j) \geq 0 \tag{3.8}$$

Equation (3.1) is the cost function also mentioned in 2.3.4. c_1 and c_2 are design time constants and δ_j is the task switching overhead on node j. Equations (3.2) and (3.3) ensure that the available memory and stack are not exceeded by the assignment and equation (3.4) ensures that bandwidth on every node is less than or equal to 1. Equation (3.5) defines the variables v_j^k as boolean and equation (3.6) makes sure that every container is assigned to exactly one node. Equation (3.7) ensures that the bandwidth of a container is sufficient for all the tasks inside it. Equation (3.8) is the scheduling condition introduced in Section 2.3.4.

SMT

The SMT-based solver uses Z3 and its Python API. To ensure determinism when running the benchmarks, which contain hundreds of systems, a new context is created for every system. This way there is no caching between the individual runs and it takes Z3 the same amount of time to compute a solution for a certain system. Independently of the system(s) that ran before it.
The `auto_config` option of Z3 is disabled and the solver is manually set to `QF_NRA` which stands for *Quantifier-Free Non-linear Real Algebra*. This is necessary because the cost function in equation (3.1) is not linear. Additionally, the real numbers are necessary, because summands such as the utilization of the nodes are no integers.
The `memory_high_watermark_mb` option is set to 4000 to try to limit the memory

usage of Z3 to 4GB. However, this is no strict limit and it uses more if available. The timeout is set to five minutes to make sure that the benchmarks finish in a somewhat feasible time.
The SMT solver can be started in the following two different modes:

- Unoptimized: The solver does not try to minimize the cost function (3.1). It runs once, tries to find a scheduling before the timeout, and outputs it.
- Optimized: Because the Optimizer built into Z3 cannot optimize non-linear functions, a little workaround was necessary to be still able to optimize the cost function. After the initial scheduling is found by the solver, an additional constraint is added to the solver to enforce a better solution in the next run. This happens after every run until either the optimal solution is found and the solver times out or the limit of optimization iterations is hit.

Z3 then tries to satisfy all constraints previously explained and, depending on the mode, may also try to minimize the cost function.

Compositional SMT

This solver also utilizes Z3, but in a different way than the original solver which also utilizes an SMT solver. Instead of adding all constraints simultaneously, the solver first searches for quota and period for every container independently. The final scheduling constraint in Section 2.3.4 is sufficient to find such a pair. To find pairs with a low processor utilization, the cost function, also from Section 2.3.4, is used. This makes it easier to find container allocations, which is the second step of this solver. Because of the decreased number of constraints given to Z3 and the, therefore, reduced complexity of each run, this approach drastically reduces the runtime compared to all other solvers, as can be seen in Section 5.1.
If the cluster is heterogeneous, it is necessary to compute the quota pair for each node type individually to account for the individual computation times and scheduling overheads. The resulting list of pairs can then be used in the second step for the allocation. However, as we focus on homogeneous clusters in this thesis, our implementation of this solver is also only capable of computing allocations for homogeneous systems.
This solver also provides a parameter that allows tuning the optimality of the assignment. It specifies the maximum error of the cost function per container compared to an optimal solution. This can be helpful for systems with high utilization where it may be necessary to reduce the bandwidth of every container as much as possible. Assignments with a lower cost, however, require more computation time on the other hand, because the SMT solver has to do more iterations until it arrives at an assignment with less cost as Section 5.1 shows.

We developed this solver, to generate allocations for evaluating the online phase in Section 5.2.

Simulated Annealing

The Simulated Annealing-based solver uses the Dual Annealing function from the `scipy` package ([sci]), which again utilizes the Generalized Simulated Annealing algorithm presented in [XGSH13].
To limit the search space, the function requires bounds for every variable. The number of variables is three times the amount containers in the system. Every container has separate variables for its node, quota, and period. The implementation of Dual Annealing from the `scipy` package can only work with real numbers. As the variables should always obtain integers, there are a few countermeasures necessary. The variables are always rounded after they are fed into the system to check for schedulability. Additionally, the variable for the node has a range of $[0, \#nodes - \epsilon]$. This way it is ensured that the variable for the nodes does not obtain the illegal value $\#nodes$. This is also why the local search provided by the Dual Annealing function is not used. The parameters of the additional calls to the evaluation function done by the local search are sometimes so close to each other that after rounding them to integers they result in the same number. The evaluation function would therefore be called multiple times with the same parameters. The Simulated Annealing solver provides the following two variants.

- Minimum: The search space limit for the quota and period of the container is set to the minimal period of the tasks of the container.
- Hyper Period: The search space limit for the quota and period of the container is set to the hyper period of the tasks of the container.

The difference between these two is the runtime as will be seen in Section 5.1. Theoretically, the hyperperiod variant could yield better results, because all possible periods are considered.
The function that the Simulated Annealing algorithm tries to optimize is implemented in the system class. For a set of input variables, the function returns an integer representing the value of the proposed variable assignment. The smaller the return value, the better the solution. The previously mentioned constraints (equations 3.2 - 3.8) are checked in Python. For every constraint that is not fulfilled, the return value is increased by 1000. To prevent plateaus in the multidimensional function space, Simulated Annealing always tries to minimize the cost function (equation 3.1). This helps to guide the algorithm in the right direction, as small changes to the variables also yield a different return value. If the optimization is omitted, the return value would be the same for a lot of different input values until it makes a jump, when suddenly a constraint is fulfilled.

Greedy

The third solver utilizes the custom greedy algorithm. Again the search space is limited. For this solver, the upper limit of quota and period of a container is the hyper period of the tasks. The step distance is set to $50\mu s$ which is necessary to get practical results. If

the step was smaller, the heuristic would find quotas and periods with a smaller value which would introduce too many task switches.
The greedy solver provides two variants as well:

- Standard: The algorithm begins its search for solutions at the lower end of the search space.
- Inverted: The algorithm starts to search for solutions at the upper end of the search space. This way the cost function can be minimized further if the context-switching overhead is considered because the resulting periods are greater this way.

3.2.3 Tool

The tool is written in Python and requires version 3.10 or higher. The architecture is modular. There is a script that generates random systems according to the arguments specified in the `config.json` file. The systems are then saved to a `.json` file each, which can then be read by the allocator tool. The allocator tool has multiple modes that can be started. There is a single-run option where an allocation for one system specified via the command line is computed. The result is then printed to the command line. Furthermore, there is the benchmark mode, where solutions for several systems are computed after each other. The tool needs the name of the benchmark set and then iterates over all its subfolders containing different system sizes. It is also possible to stop the script and continue the benchmark later on. This is helpful because it takes multiple hours to get statistically relevant data for one system size.

System generator

The system generator does not have any command line arguments. Instead, it is configured solely via the `config.json` file. In this file, it is possible to specify the parameters of nodes and tasks, described in Section 3.2.1. Additionally, the following parameters can be specified in the `config.json`:

- Utilization Factor: When generating a system, for each container the periods of the tasks are selected randomly first. The WCETs are then chosen in such a way that depending on the number of tasks on every node, the utilization is at the given factor.
- Standard Deviation: The WCETs are chosen from a normal distribution with the given standard deviation. We chose a standard deviation of two to ensure that the utilization varies a bit.
- Number of Nodes
- Number of Containers
- Number of Tasks per Container

- Number of Systems: The tool allows the generation of multiple systems of the same size in one run. Each system is then saved in the same folder in a separate file.
- Seed: If -1, the current Unix timestamp is used as a seed for the random number generator. Alternatively, a seed can also be specified directly.

For the systems we generated, we set the number of containers to three times the number of nodes and the number of tasks per container to four.
The system generator also copies the configuration file used for the run to the folder where the systems are saved. This way, it is possible to see with which configuration the systems in the folder were generated.

CHAPTER 4

Online Phase

After an initial container-to-node placement and container dimensioning have been calculated during the offline phase, the containers can be started on the nodes. During the online phase, multiple controllers need to check the temporal behavior of the containers and react if deadlines are missed. The main idea for the architecture of the monitoring system was introduced by Struhar et al. in [SCA+23]. However, the work presented here improves and refines their approach. The most significant change from a theoretical point of view is the use of time-utility (TU) functions when adapting the parameters of the containers. These functions have been explained in Section 4.1.1.

4.1 Theory

When it comes to adapting a container at runtime, there are two possibilities.

1. A container's quota was sufficient during the last interval, such that all its tasks completed their execution before their deadline.
2. A container's quota was insufficient during the last interval, such that one or more tasks did not complete their execution before their deadline. Therefore, additional resources are needed to maintain the real-time behavior of the tasks within the given container.

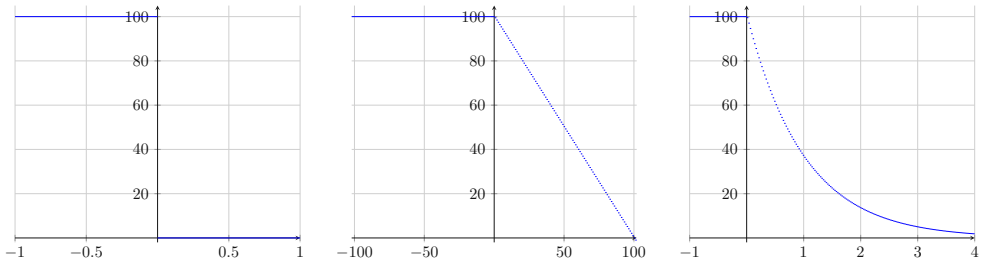
The interval in which this adaptation is takes place is application-dependent and determined by the periods of the tasks and the containers. If the interval is too small, the system is too reactive and may perform unnecessary changes. If the interval is too long, the performance of the tasks suffers because they miss too many deadlines until the container is adapted.
In [SCA+23] Struhar et al. presented an adaptation controller that uses the difference

between the actual and the expected response times of the tasks inside the containers. This so-called temporal error of the tasks indicates if a deadline has been met or not. If it is positive, the absolute timestamp of the expected deadline has been greater than the timestamp of the actual finishing time, and the deadline has been met. Otherwise, a negative temporal error indicates a missed deadline.
The tasks in the containers running on the same node often have different temporal requirements, i.e. for some, it might be less problematic than for others if a deadline is not met. With the use of temporal errors for container adaptations, it is impossible to reflect these task properties. To achieve this, assigning a semantical value to the temporal errors is necessary. This can be done with a time-utility (TU) function that is assigned to each task. These functions are described in the following section.

4.1.1 Time-Utility Functions

A time-utility function (TUF) assigns usefulness to the result of a task's job depending on its completion time, even after the deadline has expired. The utility of the result, therefore, depends on the response time of the task [Jen93]. In the initial phase of TUF-based scheduling, only maximal utility accrual (UA) concerning the timeliness of the tasks has been considered. However, further parameters, such as energy consumption or predictability, have been added over time [RJL05].
TUFs had their origin in software in the military environment, e.g., missile defense [GHJ+77]. Various further developments of the original TUF/UA model have been studied in the academic literature, e.g., [AB99, BPB+00] and have been applied in civilian contexts.

The TUFs used in this thesis are common in real-world applications [RJL05]. They yield 100 if the task finishes before its deadline. Otherwise, they yield a value in the interval $[0, 100]$. Their course from 100 to 0 follows one of the function types below. Figure 4.1 contains one example for each TUF. When calculating the TU, we define the deadline as instant 0. If a task is completed before the deadline, the argument for the TUF is negative. Otherwise, it is positive.



(a) Hard TUF (b) Linear TUF (c) Exponential TUF

Figure 4.1: Example TUFs

Hard TUF The utiltiy equals 0 after the deadline.

$$\text{hard}(x) = \begin{cases} x <= 0 & 100 \\ else & 0 \end{cases}$$

Linear TUF The utility decreases evenly to 0.

$$\text{linear}(x, m) = \begin{cases} x <= 0 & 100 \\ else & \max(m * (-x) + 1, 0) \end{cases}$$

Exponential The utility decreases exponentially to 0.

$$\text{exponential}(x, m) = \begin{cases} x <= 0 & 100 \\ else & \exp(m * (-x)) \end{cases}$$

The parameter m can be chosen freely for each task, determining the shape of the function by establishing the interval between the deadline and the moment when the Time-Utility Function (TUF) returns 0, known as the expiration time [Jen93]. However, it's crucial to note that both the function type and the parameter m depend significantly on the specific application. For example, in a technical report by Maynard et al. [MSC+88], the authors illustrate how they derived application-specific time-utility functions for an air-defense application while developing a command, control, and battle management system.

4.2 Design and Implementation

The following subsections describe the architecture and implementation of the software used in the online phase. Section 4.2.1 provides a diagram and a description of the overall architecture. The subsequent sections contain the details of the individual architectural components.

4.2.1 Overview

As Figure 4.2 shows, three main components contain the logic of the online phase. The Cluster-Level-Controller (CLC) is responsible for starting and stopping the containers on the nodes. If a container needs to be relocated, it is also the task of the CLC to find a suitable new node to move the container to. The implementation of the CLC does not allow multiple running instances of it in the same cluster. This is because reliability concerns such as redundancy are not within the scope of this thesis.

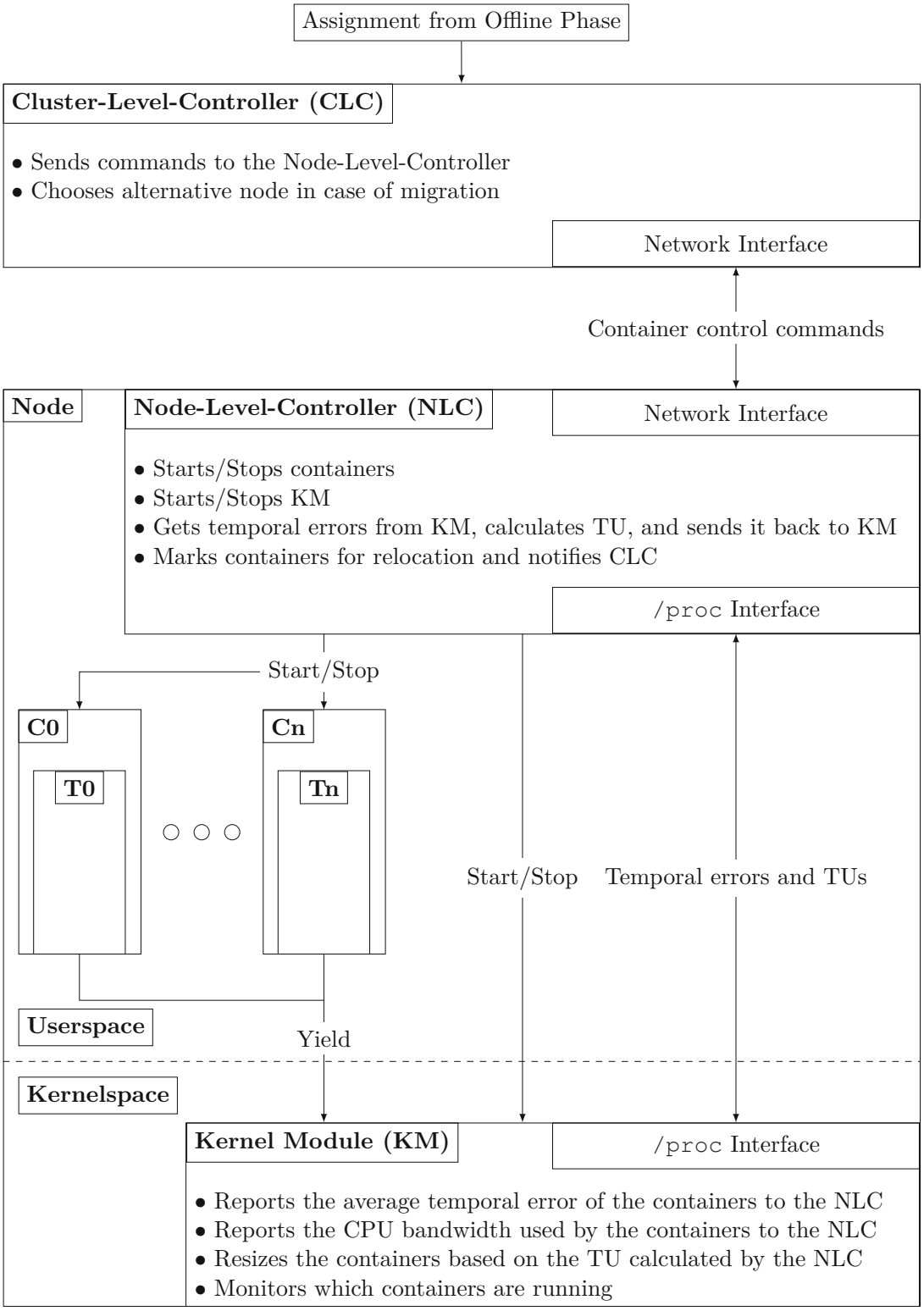


Figure 4.2: Architecture Diagram Online Phase

The CLC sends its commands to the Node-Level-Controller (NLC) on a node in the cluster. For simplicity, communication between the CLC and the NLC is based on HTTP messages. The NLC executes the commands it receives from the CLC, starts the kernel module, which will be explained in Section 4.2.4, and communicates with it. This communication utilizes the `/proc` file system. Because the floating-point unit is not available in kernel mode, the TU is calculated in the NLC. In a customizable interval, the NLC reads the average temporal error of every task during the last period. Based on this error, the TU is calculated and sent to the kernel module.

The kernel module (KM) was introduced to the architecture to aid flexibility of the system design. It is easier to add or change the functionality of the controller system if it is not implemented in the Linux kernel directly, as is the case in [SCA+23].
The KM not only relies on the NLC to get information about the containers but also monitors the running containers. Every task inside a container must yield to notify the KM that it has completed its work. The Linux kernel has been patched with a hook function such that a function inside the KM is called with every yield. After the first yield, the container and its task are registered inside an internal data structure in the KM; after the second yield, the task and the KM are synchronized, and after every consecutive yield, the KM compares the actual completion time of the task with the desired completion time. This temporal error, whether positive or negative, is then accumulated and is the basis for the TU calculation. When the NLC sends the TUs to the KM, the KM changes the scheduling parameters of the containers and, therefore gives them more or less CPU time.

4.2.2 Cluster-Level-Controller (CLC)

The CLC is implemented as a Python script. In the beginning, it parses two `.json` files: The one created by the offline phase, containing the container assignment, and the one containing the configuration of the NLCs. Then, it launches a thread for every node in the system and starts to send commands. At first, the configuration is sent to the NLC. Subsequently, the start commands for the containers are sent. After all containers have been started, the thread connects to the `/relocate_monitor` endpoint. This connection is kept open. With the use of Server-Sent Events (SSE), defined in the HTML specification [1], the NLC can send a message to the CLC every time a container needs to be relocated.

When a container needs to be relocated, the CLC retrieves the currently used bandwidth from every node. Based on these values, the CLC then starts the container on a different node if it can find one with more CPU bandwidth available and stops the container on the old node. If no node is found that can provide a better environment for the container, it is not relocated. Hence, we try to start the container on the node which has the most bandwidth available.

[1] https://html.spec.whatwg.org/multipage/server-sent-events.html#server-sent-events

When the CLC is stopped, it also sends commands to the NLCs to stop all running containers and clean up the used resources. After all containers have been stopped, the CLC fetches the log messages from the nodes to create statistics and graphs of the run.

As the goal of this implementation was to provide a proof-of-concept, no special error handling is implemented if any of the commands fail. Errors are only reported to the user.

4.2.3 Node-Level-Controller (NLC)

The NLC is also implemented as a Python script and provides six HTTP endpoints for communication with the CLC:

- **`/configure`** expects a JSON structure containing the configuration information.
- **`/start`** expects a JSON structure containing information about the container to start.
- **`/stop`** expects a JSON structure containing information about the container to stop.
- **`/bandwidth`** returns the current CPU bandwidth used on the node.
- **`/relocate_monitor`** keeps the connection open and sends a message to the client every time a container should be relocated. The message also contains the JSON structure with information about the container.
- **`/logs`** returns all log messages that were generated by the KM or a task inside a container.

The JSON structures representing containers, tasks, TUFs, and configurations are defined in Table 4.1. Each time a message containing any of these structures is sent, their validity is checked.

After the NLC has started, it waits for its configuration. Therefore, the first endpoint that is called must be the `/configure` endpoint. This is enforced by adding a cookie with a random string to the response of this endpoint. This string then has to be added to all further requests. Other endpoints can then only be called successfully if the string matches.
The configuration contains, for example, the name and parameter of the KM or intervals between certain operations. All fields are listed in Table 4.1. Aside from saving the configuration, the `/configure` endpoint also starts the KM and a thread that communicates with the KM. The interval in which this communication takes place depends on the periods of the tasks on the node and can be specified with the `interval_tu` field in the configuration file. There are three options:

- **min**: The interval is equal to the smallest task period on the node.
- **max**: The interval equals the greatest task period on the node.
- **medium**: The interval equals the difference between the greatest and smallest period divided by 2.

The communication consists of two steps. First, the NLC reads the average temporal error of each task, the bandwidth and quota used during the last period, and whether any container should be relocated. This is where the bandwidth value originates from, which is returned when calling the `/bandwidth` endpoint. The average temporal error values are used to calculate the TUs of the tasks. After all the TUs have been computed, the NLC sorts them in increasing order and writes them to the KM (`sendTU` in Figure 4.2). This way, the tasks with the lowest utility get a quota increase first, and only those with a higher utility may fail to get more quota if there is not enough left to serve the demand of all containers.

When the `/start` endpoint is called, the NLC starts the Docker container specified in the parameter field. Before continuing, the NLC ensures that the tasks inside the container have yielded and the kernel module is aware of its existence by waiting until the task is visible as a subprocess via the `psutil` Python module. The NLC then sends information about the new container and tasks to the KM. This is done by a message sent via the `/proc` file system that contains the period and deadline of every new task and the minimal quota of the container. The minimal quota is, in our case, always 100

Container	**NodeConfig**	**TUF**
image_name: str	kernel_module_name: str	type: str
name: str	interval_tu: str	param: float
quota: int	interval_relocate: str	
period: int	gain_reduce: int	
relocate: Optional[int]	gain_increase: int	
bw_used: Optional[int]	tuf_active: int	
quota_used: Optional[int]		
tasks: list[Task] | dict[str, Task]		
injector: Optional[Injector]		
Task	**Injector**	
name: str	TASK_WCET_US: int	
period: int	TASK_PERIOD_NS: int	
deadline: int	SEED: Optional[int]	
avg_temporal_error: Optional[int]	WCET_SCALER: Optional[float]	
tuf: TUF		

Table 4.1: JSON Datastructures

µs. To be able to compose this message, it is necessary first to get the mapping between the task name and the process ID (PID). This mapping is also saved for later use in the NLC. Thus, the task list of the container is converted into a dictionary that contains the relation between PID and task. After these steps, the NLC sends its HTTP response, signalizing either a successful or failed container start.

When the `stop/` endpoint is called, the NLC gets the reference to the container from the client via the container name. If the container has already terminated on its own, no further action is taken. Otherwise, the container is killed and removed from the internal list of the NLC.

When the `relocate_monitor` endpoint is called, the NLC creates a Python generator that runs until the CLC closes the connection. The generator checks at a given interval if there are one or more containers that should be relocated. There are again three possibilities regarding the length of this interval, depending on the `interval_relocate` field in the configuration file (cf. Table 4.1):

- **fixed**: The interval is set to 5 seconds. This value was chosen because, during the development and testing of the controllers, it proved to be a suitable intermediate value. Neither is it too short, leading to a lot of relocations, nor too long, preventing any relocation.
- **min**: The interval is set to be equal to the smallest task period on the node.
- **max**: The interval is set to be equal to the greatest task period on the node.

If multiple containers should be relocated, a policy chooses one of them. We implemented a *Lowest Utility First* policy. As the name suggests, only the container with the lowest utility is chosen for relocation.

4.2.4 Kernel Module (KM)

As shown in Figure 4.2, the KM has two main tasks: To keep track of the containers and tasks that are running and to adapt the current CPU quota (Q_k) of the running containers. To be able to resize the containers, the NLC periodically sends the TUs of every task to the KM. The KM then calculates the new quota (Q_k^*) with the following formulas. If the TU is equal to 100, i.e., the task is finished earlier than its deadline, the quota is reduced:

$$Q_k^* = \max\left(Q_k - \frac{g_r * \Delta_k^{avg}}{100}, \text{min\_quota}\right)$$

g_r is a parameter, given in percentage, that specifies how much the quota is modified. The average temporal error Δ_k^{avg} is calculated in the following way:

$$\Delta_k^{avg} = \frac{1}{|\mathcal{T}^k|} \sum_{\tau_i \in \mathcal{T}^k} \frac{\Delta_i^{avg}}{\max(\frac{T_i}{P_k}, 1)}$$

The average temporal error of a task in a random interval Δ_i^{avg} is the sum of the temporal errors of the task periods divided by the number of periods in the interval. This value is then scaled to the period of the container using the previous formula. This is necessary because the periods of the tasks in a container have no relation to the period of the container. The temporal errors, therefore, need to be scaled accordingly and we divide the average temporal error of the task by the period of its container. As there can be multiple tasks in a container, we average all the average temporal errors of all tasks in the container to get the overall average temporal error of the container.
If the TU is in the interval $[0, 100)$, the quota is increased:

$$Q_k^* = Q_c + \frac{g_i * \Delta_k^{avg} * (100 - \frac{1}{|\mathcal{T}^k|} \sum_{\tau_i \in \mathcal{T}^k} TU_i)}{100 * 100}$$

g_i again is a parameter for the intensity of the quota change. This formula takes into account not only the TU but also the temporal error of the container, which acts as a tie-breaker. If multiple tasks have the same utility, the quota of those tasks with a high average temporal error is increased more.
As the quota is set for the container, but the utilities are calculated per task, it is necessary to have a mechanism to calculate the utility of the container. Contrary to the offline phase, we have only one task in every container in the offline phase to reduce the complexity. Therefore, the container's utility equals the task's utility.

To make sure that the KM calculates the temporal error correctly, the KM must know the task's deadline. The following procedure allows the KM to derive the deadline.
After the task has switched to the FP scheduler, it has to yield once. This initial yield is used in the KM to create a structure for the new container and task. After this yield, it is possible to send the information about the task to the KM by the NLC. As the NLC sends this information as soon as the task is known to the OS as a running process, this first yield must not be omitted, because the information would be lost, and the container could not be adapted. In the task itself, it is possible to execute a one-time setup code after this yield. Before the start of the periodic load of the task, it must yield a second time. This is important to catch the start time of the periodic load in the KM. The KM can now add the deadline and then the period of the task to this start time to calculate all further deadlines.
From now on, the task only has to yield after every periodic load has been completed. These yield times are then compared to the deadline to calculate the temporal error of the period. To ensure that the kernel and the task calculate the same next deadline, if the last one was missed, the task must calculate its next activation time before it yields.

If a container is stopped, the de-registering function in the KM gets called from the scheduler. The KM then removes this container from its internal data structure.

4.2.5 Modifications to the Linux Kernel

We kept the modifications to the Linux kernel to a bare minimum. A register function was added to inject three function pointers, which refer to functions in the KM. This

register function is called by the KM at its start-up. The three functions, passed as parameters, are called at the following locations in the kernel:

- In the `SYSCALL_DEFINE0` of the yield: A timestamp is taken immediately after the call and passed into the KM. This ensures that the offset between the KM and the user-space task is as small as possible.
- In the `sched_yield` function of the fixed-priority (FP) RT scheduler: This is due to the hierarchical scheduling ([ABC19]) we use in this thesis. The deadline scheduler schedules the containers, but the FP scheduler schedules the tasks inside the containers. By inserting our custom hook function into the yield function of the FP scheduler, we get a notification every time a task yields. This is necessary because there is no other way to know if a task has finished its work. We can then measure the temporal error by comparing its actual and expected finishing time. The actual finishing time is available in the KM because of the timestamp that was passed into it by the previously described hook function.
- In the `free_rt_sched_group` of the FP RT scheduler: This is necessary to know when a container has stopped. Each time this happens, the struct representing this container in the KM is deleted.

4.2.6 Stability Problem

Unfortunately, we are facing a bug in the Linux kernel that causes our setup to crash sporadically. We were able to locate the problematic point in the source code, but were not able to find the cause of the error and followingly also not able to fix it. The BUG_ON in the *pick_next_rt_entity* function in the *kernel/sched/rt.c* file is the problem.
As we need to use Linux kernel version 5.10 to ensure compatibility with the HCBS patch, which was published for version 5.8, it is possible that bugs, that were fixed in later versions, are still present. However, bumping the patch to a newer kernel version would have been out of the scope of this thesis.

CHAPTER 5

Evaluation

In this Chapter, we present the evaluation of our mechanisms and approaches from the offline and online phases using a real-world hardware setup and synthetic use cases. For the offline phase, we study the scalability of our approaches in terms of runtime, as well as the quality of the produced results compared to the state-of-the-art. For the online phase, we show how the addition of a time utility function to the problem space affects the decision-making of the hierarchical control and how the system can be optimized at runtime to increase overall system utility.

5.1 Offline Phase

The different solvers presented in Section 3.2.2 were evaluated to show how their runtime and the quality of the results compare to each other. All experiments were carried out on a *Dell Latitude 5420* with an *11th Gen Intel(R) Core(TM) i7-1185G7 @ 3.00GHz, 1805 Mhz, 4 Core(s), 8 Logical Processor(s)* Processor, 32 GB RAM, and Windows 10.

Experimental Setup Python 3.11.2 inside a *venv* virtual environment was used throughout the experiments together with the following packages:

- numpy - 1.24.2
- pandas - 2.0.1 (Only for processing the measured data)
- psutil - 5.9.5
- scipy - 1.10.1
- z3-solver - 4.12.1.0

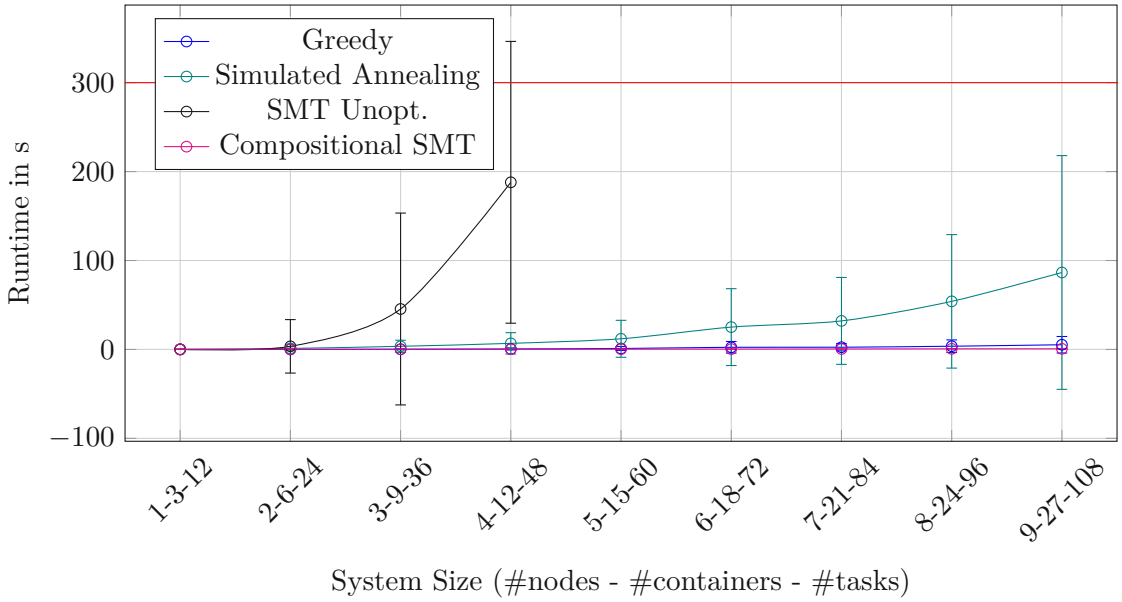


Figure 5.1: Solver Runtimes Comparison

Python 3.10+ is required to execute the tool.
The process is fixed to one core using the *psutil* package at the beginning of each run. This improves the precision of the timeout of Z3 because the specified amount of time is CPU time. If the process is not fixed, it may switch between cores and also run on cores where it gets preempted. This can lead to longer execution times because the absolute execution time of the process does not equal the CPU time anymore. However, to prevent long preemption times, the priority of the process is set to high.

For the following results, the benchmark set *benchmark_50* was used, meaning that the system utilization is in every system at about 50%. For every system size (data point on the x-axis in the plots), we aggregate 100 runs. To get the average run time for the SMT solver, a solution for every one of the 100 system runs was computed, and the runtimes were then averaged. Because the Simulated annealing and greedy solvers contain indeterminisms, both solvers computed ten different allocations for every one of the 100 systems. If run multiple times on the same system, Z3 returns the same result in the same amount of time because the default seed, used by its internal heuristics, is 0 [z3d].

5.1.1 Runtime

Figure 5.1 contains a line plot with error bars of the four solvers showing their average runtime and standard deviation. For the SMT solver, the unoptimized version was used; for the Simulated Annealing the minimum version; for the greedy solver, the non-inverted one; and for the Compositional SMT, the maximum error was set to 0.1 to get a feasible solution quickly without much optimization. There will be no plots for the Simulated

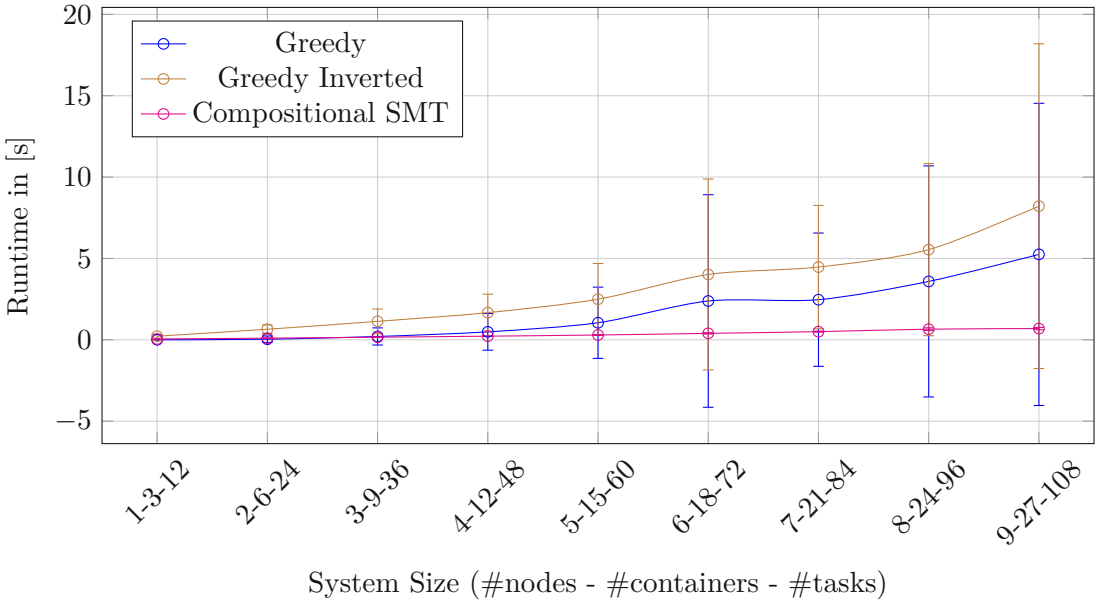


Figure 5.2: Greedy and Compositional SMT Runtimes Comparison

Annealing solver in the hyperperiod configuration because it was impossible to solve any of the smallest systems with one node and three containers in under 10 minutes. This is because the search space gets too great when using the hyperperiods of the containers as the search limits for quota and period. In the *benchmark_50* set for the smallest systems with size *1-3-12*, the search space is on average $1.75714 * 10^{11}$ times greater when using the hyper period as a search limit instead of the minimum.

SMT Solver The SMT solver has the worst performance in terms of runtime, which is to be expected. The timeout was set to five minutes, corresponding to 300 seconds. Because the Python process cannot be completely isolated on one core, the CPU time differs from the absolute time. Furthermore, the larger the system is the more memory Z3 needs. For systems with Allocating the memory also takes time, during which it is unable to compute anything. Therefore, the line representing the average runtime of the SMT solver overshoots the red line representing the timeout. For the system size *3-9-36*, there are already instances where five minutes are not enough to find a container allocation using the unoptimized SMT solver.

Simulated Annealing The runtime performance of the Simulated Annealing solver is better, but it is also clearly visible that it does not scale that well either. The reason for this is that we were not able to find a neighbor function that fits the problem well. Our implementations were unable to achieve runtimes better than those presented in Figure 5.1. Those numbers were achieved with the generic neighbor function of the *scipy* library as described in Section 3.2.2. However, this function has the downside that it is

not possible to specify any constraints on the variables. Therefore, the library proposes solutions, where the quota exceeds the period. As these solutions need to be checked nonetheless, they degrade the performance.
Additionally, we profiled some runs using *cProfile*. The results showed that a lot of time is spent converting the solutions proposed by the algorithm as a list of values into our internal Python objects which can then be checked for feasibility.

Greedy The greedy solver provides much better performance. To get a better insight into how the greedy solver and the Compositional SMT perform, Figure 5.2 provides a comparison between the normal and the inverted greedy and the Compositional SMT solvers. Because of the inverted search direction of the Greedy Inverted, its runtime is a bit greater.

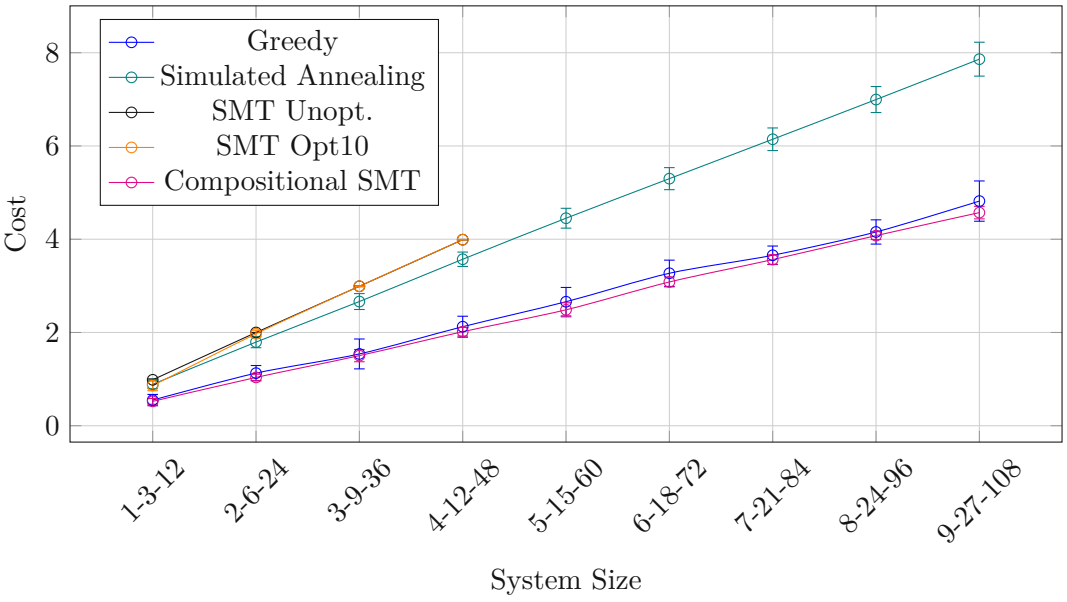
Limitations

As clearly visible from Figure 5.1, the unoptimized SMT solver is unsuitable for calculating allocations. For small systems with a small number of containers, it can not find solutions in under five minutes. The Simulated Annealing and the Greedy solver both promise better performance, but further limitations became visible when the solvers were used to generate systems for the evaluation of the online phase. For benchmark runs, the solvers were allowed to generate solutions where the quota and period of the containers were as small as one microsecond. However, this is not feasible for a real system as the resulting task switching overhead would be greater than the execution time of the containers, and, therefore, the tasks could not progress and miss a lot of deadlines. To prevent this, we forced them to generate solutions with a minimum quota and period of at least 100 microseconds. This additional constraint made the problem even harder, and the Greedy solver was not able to come up with a solution in the given amount of time. It fails because its approach is to choose the same period for all containers. A closer look at the scheduling constraint in equation (3.8) shows that for tasks with a great range of periods, it is not possible to find a period that fits all containers. The Simulated Annealing solver was able to compute solutions, however, only for systems with utilization up to 30% and 12 containers.

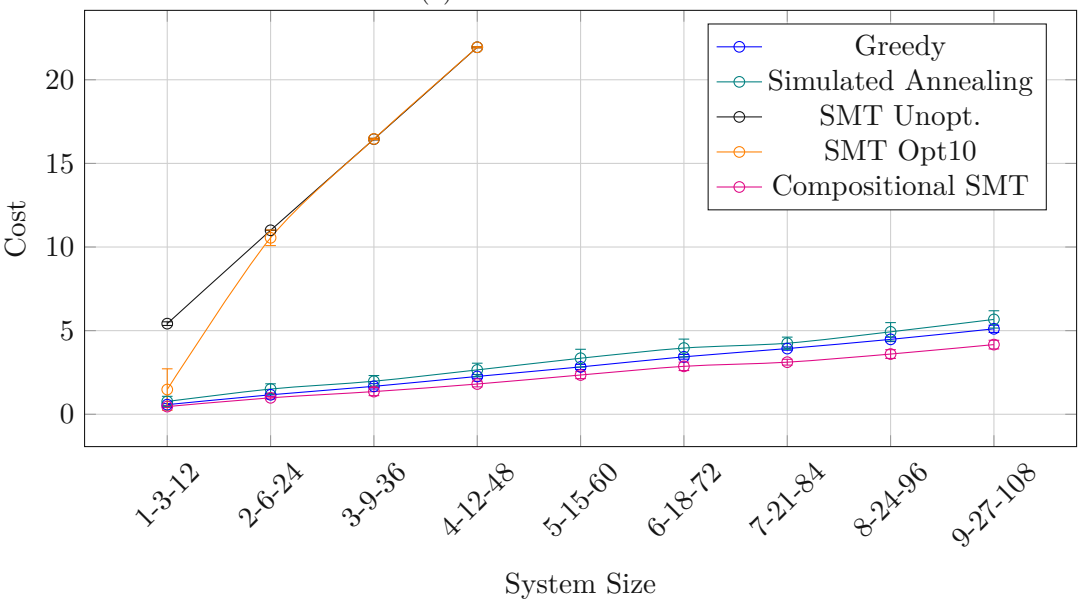
Compositional SMT As a result, we developed the Compositional SMT solver. Unlike all other solvers, it can compute allocations for all kinds of systems with different sizes and constraints and can also compute allocations for systems with a significant number of containers and a small number of nodes in a short amount of time as Figure 5.2 shows.

5.1.2 Quality

Additionally to how quickly a solution can be found, the quality of the solution is also of interest. This can be measured using the cost function in equation (3.8). Figure 5.3 compares the average cost of the four solvers when computing solutions for the *benchmark_50* set. Figure 5.3a depicts the cost in an optimal scenario without any scheduling

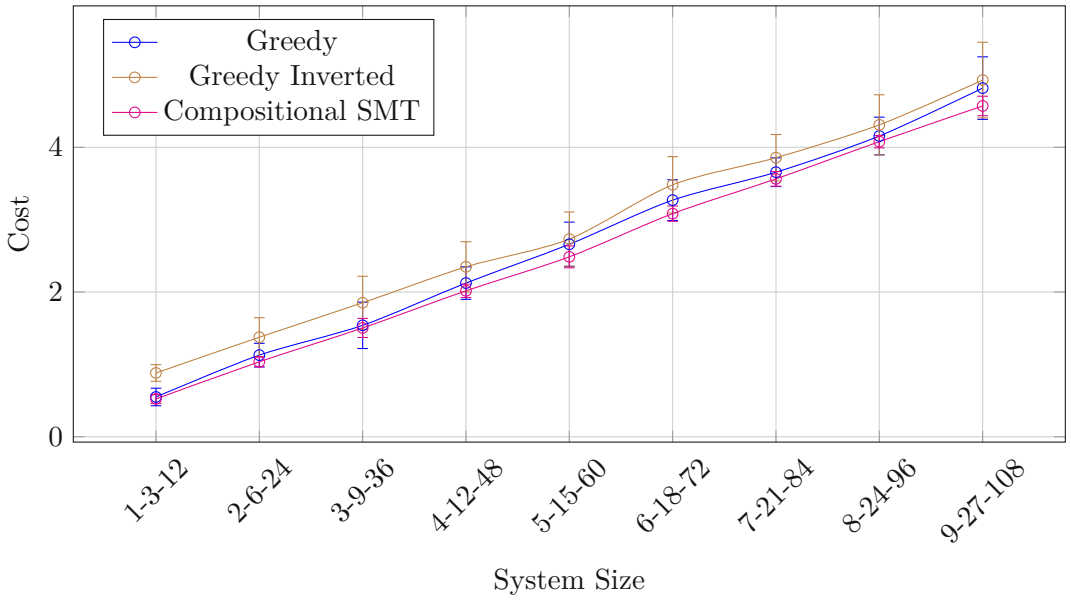


(a) Without Overhead

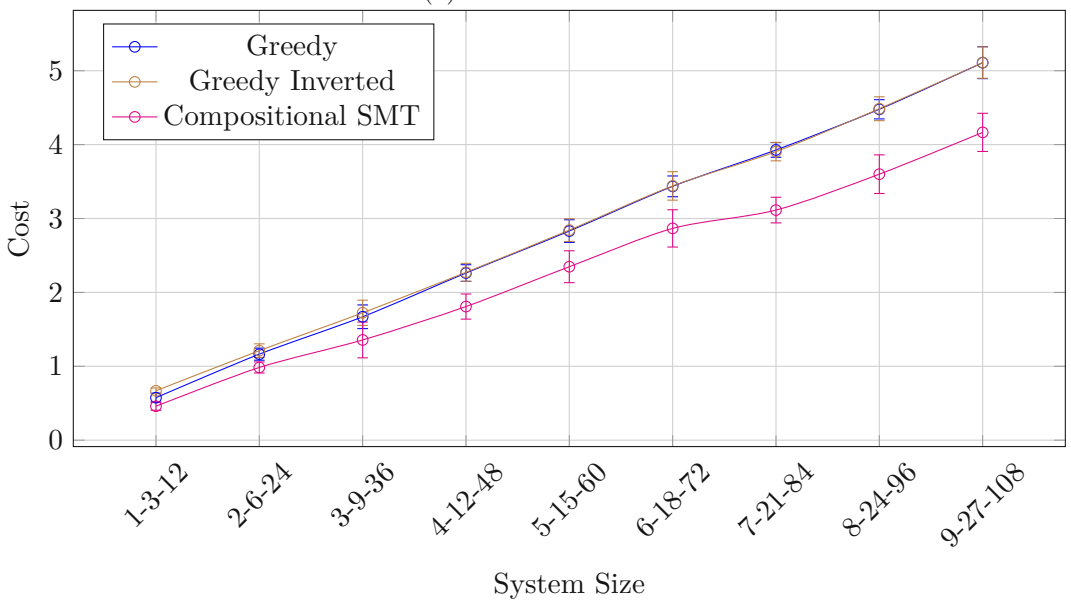


(b) With Overhead

Figure 5.3: Solver Qualities with and without overhead



(a) Without Overhead



(b) With Overhead

Figure 5.4: Greedy Qualities with and without overhead

overhead. Figure 5.3b depicts the cost where the two constants $c_1 = c_2 = 0.5$ and scheduling overhead δ is set to 10.

SMT Solver Unoptimized As expected, the SMT solver performs the worst in both scenarios, because the cost function is not added as a constraint, and, therefore, it chooses the first solution it finds. This is often one with very small numbers, resulting in a small period and a quota, which is relatively large compared to the period. This results in high bandwidth usage. The plots also show that the SMT solver did not find any solution for systems with more than four nodes within five minutes.

SMT Solver Optimized For completeness, we also conducted experiments with our optimized SMT solver. As explained in Section 3.2, this solver adds a constraint to the unoptimized SMT solver to enforce a solution with a lower cost in each additional run. In our experiments, we set the limit of 10 runs, each with a maximum duration of five minutes. Figures 5.3a and 5.3b show that for small instances this solver can compute solutions with a lower cost. For bigger systems, five minutes are enough to find the same initial solution as the unoptimized version, but an additional five minutes is not enough to improve on this solution. Therefore, the average cost of the optimized version converges to the cost of the unoptimized version.

Simulated Annealing The Simulated Annealing solver performs better. As explained in Section 3.2.2, the cost function is built into the evaluation function of the algorithm, and, therefore, some optimization is done automatically on the way toward finding a feasible allocation. However, this also means that a different cost function, e.g. one that considers the scheduling overhead, can lead to a different runtime. In our experiments, the runtime differed on average by a factor of 1.46. The two plots also show the difference in the solutions between the SMT solver and the Simulated Annealing solver. In terms of the ratio between quota and period, they are quite similar, as Figure 5.3a shows. However, the Simulated Annealing solver finds quotas and periods that are in absolute numbers well above those found by the SMT solver. This is reflected by Figure 5.3b. The short periods of the SMT solutions lead to high scheduling overhead. The comparison of longer periods of the Simulated Annealing allows this approach to generate solutions in quality similar to those generated by the greedy solver.

Greedy The solutions calculated by the greedy solver have a lower cost than those of SMT and Simulated Annealing. The periods found by this approach are a bit shorter than those of the Simulated Annealing, but the quotas are substantially shorter as well. Therefore, the cost is also lower when the overhead is considered.
Figure 5.4 compares the quality of the allocations of the two greedy solver versions. When no overhead is considered, the standard implementation yields better performance because the ratio between quota and period is greater. The inverted greedy, which starts the search with higher periods, finds feasible allocations with higher periods. However, the quota must also obtain a greater value to fulfill the scheduling requirements. As

the factor by which the quota has to be increased is greater than the factor by which the period is increased, the overall cost is greater too. When the scheduling overhead is considered, the longer periods lead to a better quality of the allocations. This is visible in Figure 5.4b. The small difference between the two solvers lies in the nature of the algorithm itself. Because it tries to find a period that fits all containers, the interval of possible periods is rather short and both versions find either the same periods or periods that are close to each other.

Compositional SMT For the Compositional SMT solver, it is again necessary to compare not only the quality of the results but also the time required to compute them. For the results shown in Figure 5.3a and 5.3b we set the maximum error of the cost function to 0.000001. This led to the best results compared to all other solvers but at the cost of additional runtime. When the scheduling overhead was not considered, the runtime increased on average by a factor of about 7 compared to experiments of section 5.1.1 where the maximum error per container was set to 0.1. The computation took about 4 seconds for the biggest systems.
When the scheduling overhead was considered, the runtime increase was more drastically compared to the runtime experiments. The runtime increased on average by a factor of 56 with the computation for the biggest systems taking about 33 seconds on average.

5.2 Online Phase

We evaluated our changes to the online phase in multiple scenarios. The goal was to show that with the use of the implemented controller system, more tasks meet their deadline than if only the fixed container assignment from the offline phase is used.

Experimental Setup The CLC was executed on an Ubuntu 22.04.3 LTS virtual machine in VMWare Workstation 16.2.5 build-20904516 on the same *Dell Latitude* laptop as was used for the experiments of the offline phase.

The cluster consisted of two Nerve MFN100 [AG], which also run Ubuntu 22.04.3 LTS. Node 1 was configured with an *Intel(R) Atom(TM) Processor E3940 @ 1.60GHz* and 4 GB of RAM and node 2 with an *Intel(R) Atom(TM) Processor E3950 @ 1.60GHz* and 8 GB of RAM. The built-in switch of node 1 was used to route traffic between the VM and node 2.

All Python interpreters used had version 3.10.12. The following packages were used by the CLC:

- httpx - 0.25.0
- httpx_sse - 0.3.1
- matplotlib - 3.8.0

- pandas - 2.1.1
- pydantic - 2.4.2

The following packages were used by the NLC:

- docker - 6.1.3
- fastapi - 0.103.1
- psutil - 5.9.5
- pydantic - 2.3.0
- setuptools - 59.6.0
- sse_starlette - 1.6.5

5.2.1 Workload

A C program within a Docker container simulated the periodic workload, by incrementing a variable in a loop. Environment variables can set the period and WCET of the task at the container's startup. The WCET is then converted into the number of loop iterations necessary to achieve the desired amount of time.

Initially, the task sets its scheduler to the FIFO RT scheduler so that it is detectable by the KM and schedulable by the hierarchical scheduling. It also contains the yields at the required points in the program as described in Section 4.2.4. The `clock_nanosleep` function from the C time library is used to wake up at the beginning of each period. After the workload is executed, the task writes its measured response time of the periodic load into an internal circular buffer and calculates its next activation time. This absolute timestamp is then used as a parameter for the sleep function. The `clock_nanosleep` function does not guarantee that the task wakes up at the given timestamp, but only the task sleeps at least until the given timestamp. As interrupts can delay the wakeup, the nodes were modified such that all interrupts were handled on core 0, and the tasks were only executed on core 1. This restriction to a single core does not affect the experiments because when using the RT group scheduling feature, Docker does not allow more RT containers than one core can handle. More cores could be utilized when the PREEMPT_RT patch is used instead of the RT group scheduling, but this was no option because the HCBS patch used in this thesis ([ABC19]) utilizes RT group scheduling.

To make the execution times of the task more realistic, the actual execution time is derived from the given upper bound (offline_bound) in every period and follows an exponential tail Gumbel distribution. This distribution is, according to [SADO17, BE00], suitable for modeling task execution times and was also used in [KRC23] to evaluate different time-triggered schedulers.
In theory, complex static analysis or symbolic execution methods are used to derive a

safe WCET for tasks in a real-time system such that all possible execution paths with all possible input vectors as well as all system aspects, i.e., preemption cost, cache miss penalty, etc, are considered. However, in reality, this is a very complex and costly process, and most WCET bounds are found through limited experimental tracing. Hence, the upper bound that is given is not a safe upper bound, i.e., an actual WCET, but rather the highest measured execution time from a limited set of traces. To depict this, we have added a parameter called `WCET_scaler`. This parameter can obtain values between 0 and 1 and specifies the relation between the given upper bound and the actual WCET. If the parameter is, for example, 0.8 the task may need up to 20% longer than the given upper bound to execute. The NLC does not know this parameter and assumes that the given upper bound of the offline phase is the actual WCET. This mechanism allows us to model something commonly used in RT systems. As it is often too complex to calculate the actual WCET, the application is profiled with many different inputs. The resulting WCET is then only correct for the tested inputs. Due to the impossibility of testing all inputs, there is still a chance that a different input results in a greater execution time. Our parameter, therefore, allows us to specify how well each task was profiled.

$$\text{true\_WCET} = \frac{\text{offline\_bound}}{\text{WCET\_scaler}}$$

5.2.2 Scenarios

We evaluated the online phase with the following scenarios:

1. 10 containers, `WCET_scaler` = 0.8, System utilization is at 70%, Lax TU.
2. 10 containers, `WCET_scaler` = 0.6, System utilization is at 70%, Lax TU
3. 10 containers, `WCET_scaler` = 0.261856, System utilization is at 70%, Lax TU
4. 10 containers, `WCET_scaler` = 0.261856, System utilization is at 70%, Strict TU
5. 7 containers, `WCET_scaler` = 0.261856, System utilization is at 50%, Strict TU, one node is full, the other one is empty.
6. 7 containers, `WCET_scaler` = 0.261856, System utilization is at 50%, Strict TU, low relocate interval

The value 0.261856 for the `WCET_scaler` is the expected value of the Gumbel distribution. The scenarios that use this value represent those cases in which tasks were poorly profiled.

Scenario 1 This scenario did not yield any interesting cases. Since we set the deadline of a task equal to its period in this thesis and `WCET_scaler` = 0.8, there are no deadline misses. Furthermore, the system overhead was not enough interference to lead to any deadline miss in the case of 70% system utilization. Therefore, the TU of all tasks is equal to 100% at all times, and no relocation or container adaptation was necessary.

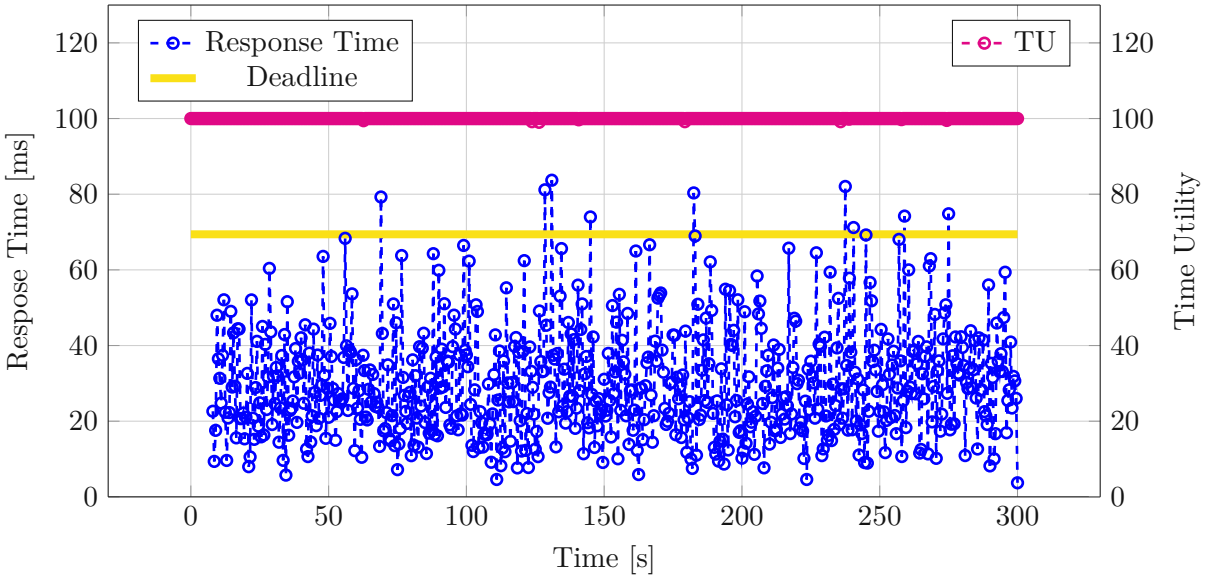


Figure 5.5: Scenario 2 - Task 9: Maximum Response Time per half-second and Average TU per half-second over Time

Scenario 2 The reduction of the `WCET_scaler` led to deadline misses in some tasks and, as a result, also to a reduced TU. Figure 5.5 shows the graph of the response time of task 9 and its TU over time. For this scenario, the NLC was configured in such a way that the interval of the TU evaluation was set to *medium*. For node 1, on which task 9 was running, this was about 60 ms. To keep the plots meaningful, we aggregated the data points for each half-second interval. For the response times, we plot the maximum value of the interval; for the TU, we plot the average value.
It is also possible that the response time graph indicates missed deadlines, but the TU is still 100% because the TU is, as already mentioned, not calculated after each period of the task. If the response times around a missed deadline are short enough, they can compensate for the miss, and the TU is still maximal.

Scenario 3 A further reduction of the `WCET-scaler` led to constant deadline misses and a significantly reduced TU. Figure 5.6 shows the response time and TU graph of the same task. However, because of the high system utilization, the CLC could not relocate any container.

Scenario 4 The switch from lax to strict TUs worsened the situation even more. Figure 5.7 shows the response time and TU graph of task 4, which suffered the most in this run because its WCET is already quite close to the period without scaling it. In this run, container C5 was relocated from node 1 to node 2 at the very beginning. Due to the indeterminism of the tasks themselves, it is possible that the containers did not immediately require a larger budget after startup, and therefore, C5 could be relocated to node 2.

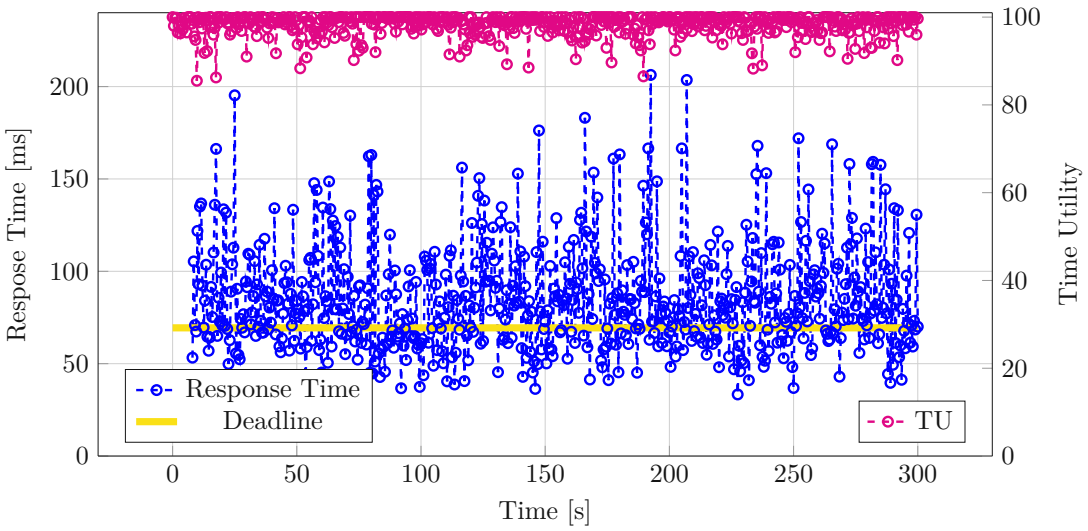


Figure 5.6: Scenario 3 - Task 9: Maximum Response Time per half-second and Average TU per half-second over Time

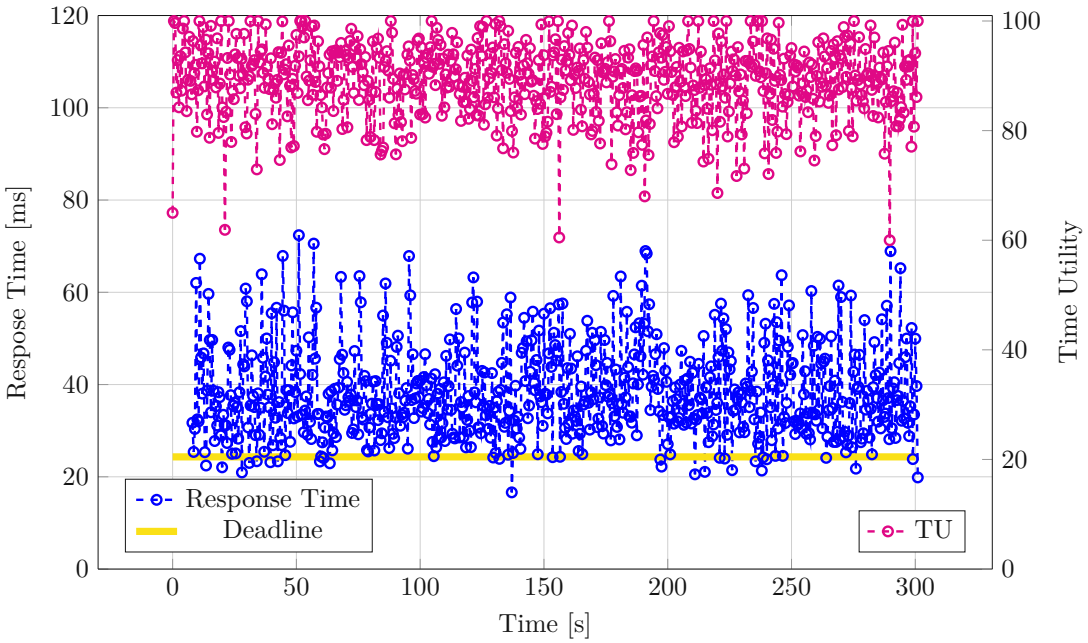


Figure 5.7: Scenario 4 - Task 4: Maximum Response Time per half-second and Average TU per half-second over Time

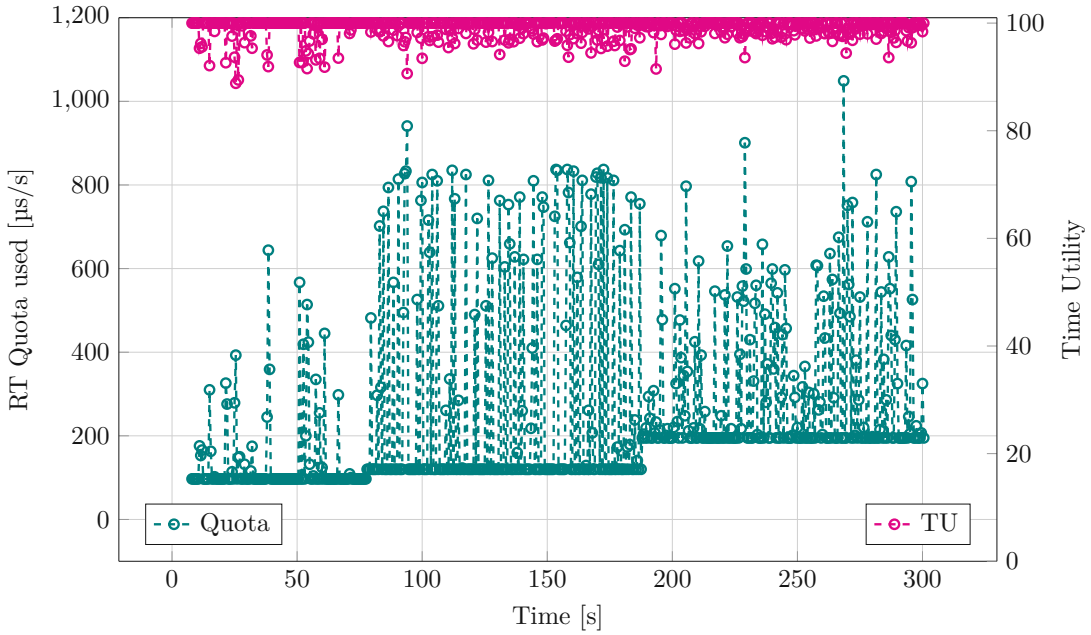


Figure 5.8: Scenario 5 - Node 2: Maximum Quota per half-second and Average TU per half-second over Time

Scenario 5 To demonstrate the relocation feature of the online phase, we started containers only on node 1. Because of the low `WCET_scaler` value many tasks missed their deadlines. Container C6 was relocated immediately after startup. Container C5 was relocated after about 70 seconds (Figure 5.9) and C3 after about 180 seconds. Figure 5.8 contains a graph of the quota used on node 2. The quota jumps indicate that a container was relocated to the node.

To understand why it took more than a minute until C5 was relocated even though it missed deadlines since the beginning, we need to have another look at the relocation mechanism. In an interval different on each node, the time utility of all containers on the node is calculated. If a task misses one or multiple deadlines during this interval, it may or may not lead to a reduced TU of the task; depending on the response times of the other jobs during the interval. If the evaluation results in a reduced TU, the KM attempts to increase the container quota. This may be successful or unsuccessful, depending on the quota requirements of the other containers. If it fails, the relocate flag is set for this container. In an interval different from the one that calculates the TUs, the relocate flags of all containers are checked. If the flag of one container is set, the NLC sends a message to the CLC, and the CLC then tries to relocate the container. However, it is also possible that the relocate flag of a container is set after one period of the TU interval, and during the next interval, all TUs are at 100% again because the CPU demand decreased.
The relocation interval can be used in two different ways. To try to catch every relocate

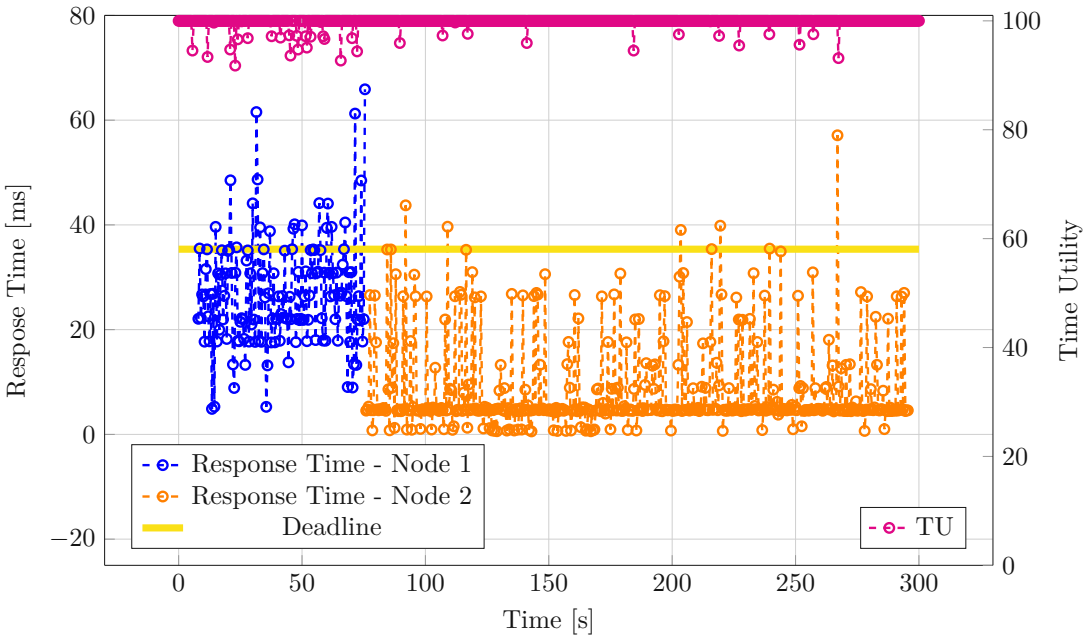


Figure 5.9: Scenario 5 - Task 5: Maximum Response Time per half-second and Average TU per half-second over Time

flag that is true, it can be set to the same value as the TU interval, as done in scenario 6 (5.2.2). A different approach would be to set it to a higher value and relocate only those containers that suffer over a longer timespan as done in this scenario.

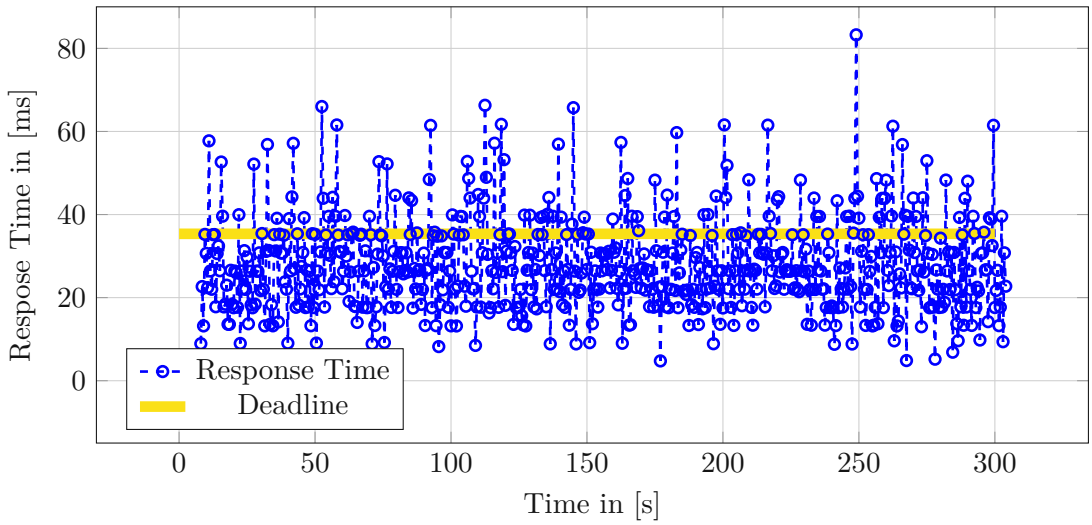


Figure 5.10: Scenario 5 - Task 5: Maximum Response Time per half-second without the new controllers

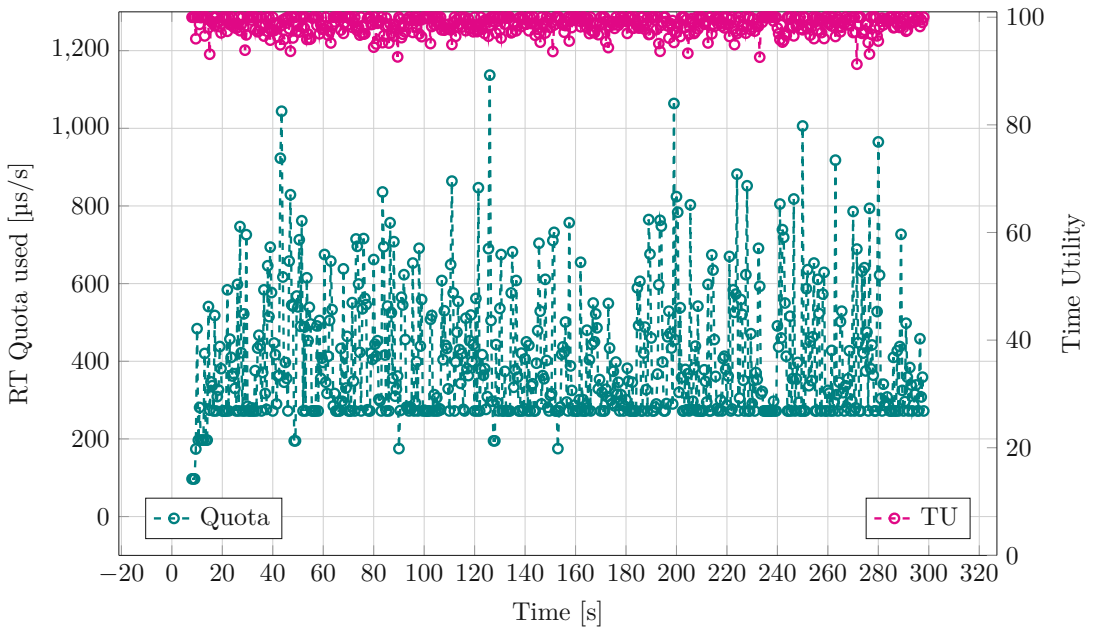


Figure 5.11: Scenario 6 - Node 2: Maximum Quota per half-second and Average TU per half-second over Time

To provide an example of the effect on the response time of a task, we repeated this run with the same configuration but without using the controllers of the online phase. Therefore, the quotas of the containers are not adapted, and no container is relocated. Figure 5.10 provides the response time graph of the same task as in Figure 5.9.

Scenario 6 The lower relocation interval led to the expected quicker relocation of some containers after their startup. Also, at runtime, containers are relocated more quickly. Figure 5.11 shows spikes below 300 µs/s in the used quota of node 2. At these spikes, a container was relocated to node 1 for a short period and then relocated back to node 2, because it could not meet the deadlines on node 1 too.

5.3 NLC Overhead

To evaluate the CPU demand of the NLC itself, we executed runs with different system utilization and different TU categories. We also disabled the logging of the task and the KM, to reduce the artificial overhead necessary for the experiments. For utilities ranging from 10 to 60%, the systems consisted of 12 containers, for the utility of 70% the system consisted of 10 containers.
Configuration 1 in Figure 5.12 had the TU interval set to *medium* and the relocation interval set to 5 seconds. The figure shows that the usage is substantial but can also vary widely. This is because it is heavily dependent on the number of tasks running on the node and their period. A lower number of tasks leads to fewer and shorter interactions with the KM and fewer TUs that need to be calculated. Higher periods of tasks lead to less frequent interactions with the KM. Additionally, the controller is implemented in Python which contributes to the high CPU overhead, as it is an interpreted language. An analysis with the *pyinstrument*[1] profiler showed that the *periodic_worker* method is responsible for about 90% of the overall CPU time. This method contains the communication with the KM. This is mainly due to the read and write operations via the */proc* file system which consume about 25% of the overall CPU time. The remaining 65% are calculations and read and write accesses to the Python objects required in the NLC.
For each data point, we executed three runs with the same system, but different TU functions, and plotted the average of these three values. As expected, the different TU functions did not affect the CPU usage of the NLC, as they did not affect the computation effort.

[1] https://github.com/joerick/pyinstrument

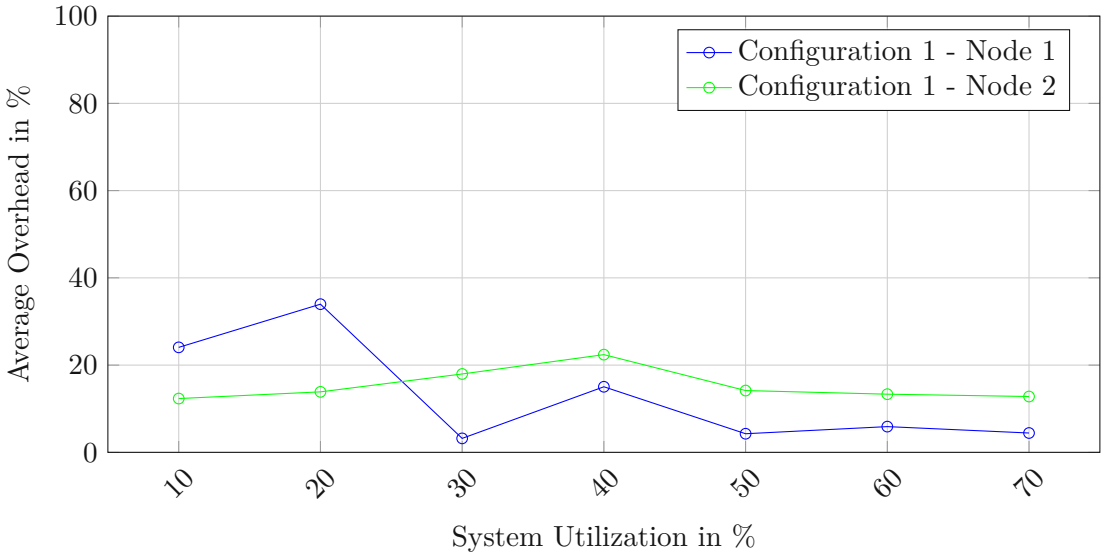


Figure 5.12: CPU usage of the NLC with configuration 1

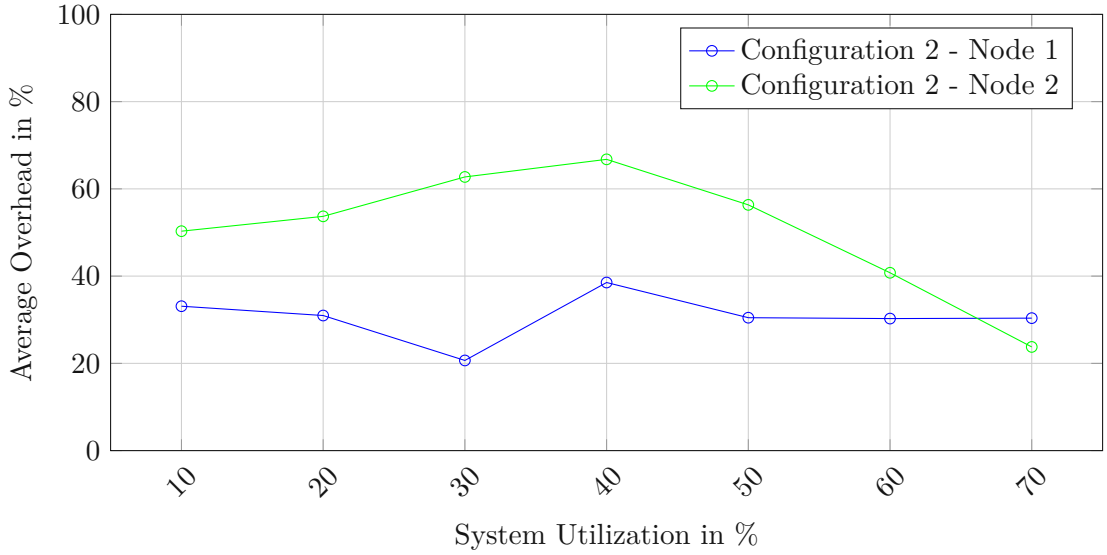


Figure 5.13: CPU usage of the NLC with configuration 2

Additionally, we executed the same experiment with a different configuration of the NLCs. The interval for calculating the TUs and checking the relocation flag were both set to the value *min*. Therefore, both had the same period as the task with the lowest period. Figure 5.13 contains a graph with the average CPU values. As expected, the absolute values are greater than in configuration 1 due to the lower intervals in the NLC.

CHAPTER 6

Conclusion

The thesis improves upon the state-of-the-art in real-time container orchestration in two directions. First, it solves the scalability issues of the SMT-based offline assignment and initial dimensioning phase of the Hierarchical Resource Orchestration Framework [SCA+23]. Specifically, the objective was to efficiently determine an assignment of containers to nodes while establishing an RT interface comprising a quota and period pairing for each container. We proposed, implemented, and evaluated three different heuristic algorithms: one based on a Greedy approach, a Simulated Annealing meta-heuristic, and a heuristic approach with an SMT-based step within. Second, we redesigned and reimplemented the Hierarchical Resource Orchestration Framework from [SCA+23], reducing the number of control levels and enhancing the control mechanism of the online phase with time-utility functions (TUFs) to better capture the usefulness of results in case of deadline misses.

6.1 Offline Phase

Concerning the offline phase, all solvers demonstrated superior performance compared to the existing SMT approach. However, it is noteworthy that the Simulated Annealing solver faced challenges in terms of scalability, which originated mainly from the use of a generic neighbor function. The runtimes of our greedy solver yielded scalable results, yet in real-world scenarios with more stringent restrictions on quota and period values and reduced maximum bandwidth on the nodes, the greedy solver was unable to solve any instances, and the Simulated Annealing solver was able to only solve smaller instances with reduced utilization of the nodes. In contrast, the Compositional SMT solver exhibited the best performance among our solvers in terms of runtime and quality of the results, measured by a cost function. Additionally, it was capable of computing real-world allocations for the evaluation of the online phase. The Compositional SMT approach breaks down the equations necessary for a feasible allocation, significantly simplifying the problem and allowing Z3 to solve it efficiently. However, our implementation is only

able to compute allocations for homogeneous systems, where all nodes have the same computing properties.

Our evaluations also show that there always is a trade-off between fast results and good results. Simulated Annealing is an algorithm that can overcome local minima and can compete with the greedy and Compositional solver in terms of quality when a scheduling overhead is considered. The greedy solver provides solutions quickly but has no option for optimization and can only find optimal solutions by chance. The Compositional SMT solver is the fastest when it comes to computing feasible allocations. However, when the quality of the solutions is relevant, more time must be spent on computation.

6.2 Online Phase

Concerning the online phase, we implemented a mechanism based on Time-Utility Functions (TUFs) that reduces deadline misses for soft real-time tasks within containers. Our hierarchical system of controllers operates at different levels. At the cluster level, a Cluster-Level Controller (CLC) was responsible for starting and stopping containers on the nodes designated during the offline phase. Commands from the CLC were transmitted to the node-level controller (NLC) via the network, which, in turn, interacted with the Docker instance on the node. Additionally, the NLC communicated with a Kernel Module (KM) that we developed. This module tracked the missed deadlines of all tasks on the node, reporting temporal errors (the difference between actual and expected task completion time) back to the NLC. Subsequently, the NLC computed time utilities (TUs) for each task and relayed this information to the KM. The KM then adapted the quota and period of the container based on the computed TU. If a container required more quota than currently available on the node, the NLC informed the CLC, which, in turn, sought an alternative node in the system for potential relocation of the container.

We have also performed experiments using a real-world test-bed to demonstrate the effectiveness of our online orchestration framework. We showed that it is capable of maximizing the system utility while trying to optimize resource usage. This is achieved by increasing the CPU quota of those containers which miss their deadlines and decreasing the quota of those containers which finish early. Additionally, containers are relocated to a different node, if their current node can not provide enough CPU time.

CHAPTER 7

Future Work

There are various possibilities for expanding upon the findings presented in this thesis.

7.1 Offline Phase

While we successfully developed an efficient solver for the container allocation problem during the offline phase, some of our heuristics still have room for improvement. For example, the Simulated Annealing may yield more optimal results in the same time frame with a more customized neighbor function.

As the runtimes for the computation of individual quota period pairs are low using the Compositional SMT solver, it also seems feasible to use this approach in a heterogeneous cluster, where it is necessary to compute a pair for each node. However, as we considered only homogeneous clusters in this thesis, we did not evaluate such an extension.
It would also be possible to parallelize the computation of the quota and period pairs of the containers, which could yield an additional benefit in computation time. This is also true for heterogeneous systems where the quota and period pairs of the same container but for different nodes could also be computed in parallel. For this reason, we believe that this approach is the most promising for solving this allocation problem.

7.2 Online Phase

In terms of the online phase, there are opportunities for enhancing our solution. To allow deployment of the implementation in a real-world setting, it is necessary to fix a bug in the Linux kernel related to real-time scheduling and the HCBS patch. We pointed out the location in the kernel code where the bug occurs, but it was out of the scope of this thesis to search for its origin.
Regarding fault tolerance of the Cluster-Level-Controller (CLC), the implementation

could be enhanced with either one additional redundant CLC to tolerate a fail-silent behavior or with $3f + 1$ additional CLCs to tolerate f byzantine faults [LSP19].

If a solution is desired that does not necessitate a KM or patched kernel, an approach could involve implementing all the required functionality in user space. The caveat here is that tasks would need to provide their execution times via messages on the container's log stream. The NLC would then process these messages and take appropriate measures based on task completion times. The *docker update* command could be used to adjust the CPU quota of the containers. However, the overhead can be expected to be significant, and it would be interesting to see if it justifies the simplifications.

For those seeking minimal kernel modifications, an alternative is to utilize the PREEMPT_RT patch instead of the hierarchical scheduling based on GROUP_RT scheduling. The PREEMPT_RT patch, loaded as a kernel module, enhances preemptiveness in the Linux kernel, reducing latencies compared to hierarchical scheduling. However, this approach would necessitate adjustments to the controller system theory since containers, with the PREEMPT_RT patch, no longer provide the RT interface comprising CPU quota and period. This alternative might be preferable in scenarios where containers contain only one task and hierarchical scheduling is not utilized.

Addressing the CPU overhead of the NLC, it might be advantageous to implement a mechanism that adapts the intervals at which the NLC reads temporal errors from the KM. These interactions contribute significantly to the overall overhead, and adjusting intervals based on the utility levels could optimize performance.
A different approach would be to shift more functionality to the kernel space. The TU utilities could be calculated by approximating the exponential functions and fixed point arithmetic to prevent the necessity of an FPU. However, this has the downside of limited portability and stability, due to dependencies on kernel mechanisms. Furthermore, the development of Linux kernel functionalities requires knowledge about the architecture of the kernel and the build process and is not as straightforward as the development of a user space application. A user-space implementation is, therefore, more stable and easier to maintain.

List of Figures

List of Tables

List of Algorithms

Bibliography

[AB98] Luca Abeni and Giorgio Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings 19th IEEE Real-Time Systems Symposium (Cat. No. 98CB36279)*, pages 4–13. IEEE, 1998.

[AB99] Saud Ahmed Aldarmi and Alan Burns. Dynamic value-density for scheduling real-time systems. In *Proceedings of 11th Euromicro Conference on Real-Time Systems. Euromicro RTS'99*, pages 270–277. IEEE, 1999.

[AB04] Luca Abeni and Giorgio Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Systems*, 27:123–167, 2004.

[ABC19] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the linux kernel. *ACM SIGBED Review*, 16(3):33–38, 2019.

[AG] TTTech Computertechnik AG. Nerve. https://www.tttech-industrial.com/nerve. Accessed: 07.11.2023.

[AHGZ16] Sherif Abdelwahab, Bechir Hamdaoui, Mohsen Guizani, and Taieb Znati. Cloud of things for sensing-as-a-service: Architecture, algorithms, and use case. *IEEE Internet of Things Journal*, 3(6):1099–1112, 2016.

[AZH18] Mohammad Aazam, Sherali Zeadally, and Khaled A Harras. Deploying fog computing in industrial internet of things and industry 4.0. *IEEE Transactions on Industrial Informatics*, 14(10):4674–4682, 2018.

[BB02] Enrico Bini and Giorgio C Buttazzo. The space of rate monotonic schedulability. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 169–178. IEEE, 2002.

[BBB+22] Haniel Barbosa, Clark Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, et al. cvc5: A versatile and industrial-strength smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 28th International Conference, TACAS 2022, Held as Part of the European Joint Conferences on Theory and Practice of Software,*

ETAPS 2022, Munich, Germany, April 2–7, 2022, Proceedings, Part I, pages 415–442. Springer, 2022.

[BE00] Alan Burns and Stewart Edgar. Predicting computation time for advanced processor architectures. In *Proceedings 12th Euromicro Conference on Real-Time Systems. Euromicro RTS 2000*, pages 89–96. IEEE, 2000.

[BMZA12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, pages 13–16, 2012.

[Bon11] Flavio Bonomi. Connected vehicles, the internet of things, and fog computing. In *The eighth ACM international workshop on vehicular inter-networking (VANET), Las Vegas, USA*, pages 13–15. sn, 2011.

[BPB+00] Alan Burns, Divya Prasad, Andrea Bondavalli, Felicita Di Giandomenico, Krithi Ramamritham, John Stankovic, and Lorenzo Strigini. The meaning and role of value in scheduling flexible real-time systems. *Journal of systems architecture*, 46(4):305–325, 2000.

[BPST10] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The opensmt solver. In *TACAS*, volume 6015, pages 150–153. Springer, 2010.

[BT18] Clark Barrett and Cesare Tinelli. *Satisfiability modulo theories*. Springer, 2018.

[CHR+17] Mung Chiang, Sangtae Ha, Fulvio Risso, Tao Zhang, and I Chih-Lin. Clarifying fog computing and networking: 10 questions and answers. *IEEE Communications Magazine*, 55(4):18–20, 2017.

[CJ] Sean Campbell and Michael Jeronimo. An introduction to virtualization.

[CO16] Silviu S Craciunas and Ramon Serna Oliver. Combined task-and network-level scheduling for distributed time-triggered systems. *Real-Time Systems*, 52:161–200, 2016.

[Cur03] Sharon A Curtis. The classification of greedy algorithms. *Science of Computer Programming*, 49(1-3):125–157, 2003.

[Dav] Alex Davies. Cisco pushes iot analytics to the extreme edge with mist computing. https://rethinkresearch.biz/articles/cisco-pushes-iot-analytics-extreme-edge-mist-computing-2/. Accessed: 03.08.2023.

[DDM06] Bruno Dutertre and Leonardo De Moura. The yices smt solver. *Tool paper at http://yices. csl. sri. com/tool-paper. pdf*, 2(2):1–2, 2006.

[Der74] Michael L Dertouzos. Control robotics: The procedural control of physical processes. In *Proceedings IF IP Congress, 1974*, 1974.

[DL97] Zhong Deng and JW-S Liu. Scheduling real-time applications in an open environment. In *Proceedings Real-Time Systems Symposium*, pages 308–319. IEEE, 1997.

[DLNW13] Hoang T Dinh, Chonho Lee, Dusit Niyato, and Ping Wang. A survey of mobile cloud computing: architecture, applications, and approaches. *Wireless communications and mobile computing*, 13(18):1587–1611, 2013.

[DMB08] Leonardo De Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In *Tools and Algorithms for the Construction and Analysis of Systems: 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings 14*, pages 337–340. Springer, 2008.

[Dut14] Bruno Dutertre. Yices 2.2. In *Computer Aided Verification: 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings 26*, pages 737–744. Springer, 2014.

[Ede16] Michael Eder. Hypervisor-vs. container-based virtualization. *Future Internet (FI) and Innovative Internet Technologies and Mobile Communications (IITM)*, 1, 2016.

[FAC22] Stefano Fiori, Luca Abeni, and Tommaso Cucinotta. Rt-kubernetes: containerized real-time cloud computing. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, pages 36–39, 2022.

[Fin18] Norman Finn. Introduction to time-sensitive networking. *IEEE Communications Standards Magazine*, 2(2):22–28, 2018.

[G+73] Robert P Goldberg et al. *Architectural principles for virtual computer systems*. PhD thesis, Harvard University Cambridge, MA, 1973.

[GHJ+77] MG Gouda, YW Han, ED Jensen, WD Johnson, RY Kain, HONEYWELL INC MINNEAPOLIS MINN SYSTEMS, and RESEARCH CENTER. Distributed data processing technology. volume iv. application of ddp technology to bmd: Architectures and algorithms. Technical report, 1977.

[HF04] David Harel and Yishai A Feldman. *Algorithmics: The spirit of computing*. Pearson Education, 2004.

[HGLBV01] J-P Hubaux, Thomas Gross, J-Y Le Boudec, and Martin Vetterli. Toward self-organized mobile ad hoc networks: The terminodes project. *IEEE Communications Magazine*, 39(1):118–124, 2001.

[HSI+19] Florian Hofer, Martin A Sehr, Antonio Iannopollo, Ines Ugalde, Alberto Sangiovanni-Vincentelli, and Barbara Russo. Industrial control via application containers: Migrating from bare-metal to iaas. *arXiv preprint arXiv:1908.04465*, 2019.

[HT97] C-C Han and H-Y Tyan. A better polynomial-time schedulability test for real-time fixed-priority scheduling algorithms. In *Proceedings Real-Time Systems Symposium*, pages 36–45. IEEE, 1997.

[Jen93] E Douglas Jensen. A timeliness model for asychronous decentralized computer systems. In *Proceedings ISAD 93: International Symposium on Autonomous Decentralized Systems*, pages 173–182. IEEE, 1993.

[JLF11] Cotye Julian, Tana Lucy, and John Farr. Commercial-off-the-shelf selection process. *Engineering Management Journal*, 23(2):63, 2011.

[KGC16] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. *Formal Methods in System Design*, 48:175–205, 2016.

[KGJV83] Scott Kirkpatrick, C Daniel Gelatt Jr, and Mario P Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[Koz15] Nectarios Koziris. Fifty years of evolution in virtualization technologies: from the first ibm machines to modern hyperconverged infrastructures. In *Proceedings of the 19th Panhellenic Conference on Informatics*, pages 3–4, 2015.

[KRC23] Paraskevas Karachatzis, Jan Ruh, and Silviu S Craciunas. An evaluation of time-triggered scheduling in the linux kernel. In *Proceedings of the 31st International Conference on Real-Time Networks and Systems*, pages 119–131, 2023.

[KS22] Hermann Kopetz and Wilfried Steiner. *The Real-Time Environment*, pages 1–29. Springer International Publishing, Cham, 2022.

[KZH15] Simon Kramer, Dirk Ziegenbein, and Arne Hamann. Real world automotive benchmarks for free. In *6th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS)*, volume 130, 2015.

[LB03] Giuseppe Lipari and Enrico Bini. Resource partitioning among real-time applications. In *15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.*, pages 151–158. IEEE, 2003.

[Lip08] Giuseppe Lipari. Edf scheduling. `http://retis.sssup.it/~lipari/courses/str07/edf-handout.pdf`, May 2008. Accessed: 13.06.2023.

[LK73] Shen Lin and Brian W Kernighan. An effective heuristic algorithm for the traveling-salesman problem. *Operations research*, 21(2):498–516, 1973.

[LKB77] Jan Karel Lenstra, AHG Rinnooy Kan, and Peter Brucker. Complexity of machine scheduling problems. In *Annals of discrete mathematics*, volume 1, pages 343–362. Elsevier, 1977.

[LL73] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[LSD89] John Lehoczky, Lui Sha, and Yuqin Ding. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *RTSS*, volume 89, pages 166–171, 1989.

[LSP19] Leslie Lamport, Robert Shostak, and Marshall Pease. The byzantine generals problem. In *Concurrency: the works of leslie lamport*, pages 203–226. 2019.

[LWT] A realtime preemption overview. https://lwn.net/Articles/146861/. Accessed: 24.03.2023.

[LXRD19] Hao Li, Xuefei Xu, Jinkui Ren, and Yaozu Dong. Acrn: a big little hypervisor for iot development. In *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*, pages 31–44, 2019.

[LZCC17] Kai Liang, Liqiang Zhao, Xiaoli Chu, and Hsiao-Hwa Chen. An integrated architecture for software defined and virtualized radio access networks with fog computing. *IEEE Network*, 31(1):80–87, 2017.

[MFIT17] Roberto Morabito, Ivan Farris, Antonio Iera, and Tarik Taleb. Evaluating performance of containerized iot services for clustered devices at the network edge. *IEEE Internet of Things Journal*, 4(4):1019–1030, 2017.

[MG+11] Peter Mell, Tim Grance, et al. The nist definition of cloud computing. 2011.

[Mok83] Aloysius Ka-Lau Mok. *Fundamental design problems of distributed systems for the hard-real-time environment*. PhD thesis, Massachusetts Institute of Technology, 1983.

[MSC+88] David P Maynard, Samuel E Shipman, Raymond K Clark, J Duane Northcutt, Russell B Kegley, Betsy A Zimmerman, and Peter J Keleher. An example real-time command, control, and battle management application for alpha. *Archons Project TR-88121, CMU*, 1988.

[MTS+20] José Martins, Adriano Tavares, Marco Solieri, Marko Bertogna, and Sandro Pinto. Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems. In *Workshop on next generation real-time embedded*

systems (NG-RES 2020). Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[OK96] Ibrahim H Osman and James P Kelly. Meta-heuristics: an overview. *Meta-heuristics: Theory and applications*, pages 1–21, 1996.

[ope] The openfog consortium reference architecture: Executive summary. `TheOpenFogConsortiumReferenceArchitecture:ExecutiveSummary`. Accessed: 05.07.2023.

[Org72] Elliott I Organick. *The Multics system: an examination of its structure*. MIT press, 1972.

[PG74] Gerald J Popek and Robert P Goldberg. Formal requirements for virtualizable third generation architectures. *Communications of the ACM*, 17(7):412–421, 1974.

[PVL11] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated test case generation with smt-solving and abstract interpretation. In *NASA Formal Methods: Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings 3*, pages 298–312. Springer, 2011.

[RJL05] Binoy Ravindran, E Douglas Jensen, and Peng Li. On recent advances in time/utility function real-time scheduling and resource management. In *Eighth IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC'05)*, pages 55–60. IEEE, 2005.

[Rut89] Rob A Rutenbar. Simulated annealing algorithms: An overview. *IEEE Circuits and Devices magazine*, 5(1):19–26, 1989.

[SAÅ+04] Lui Sha, Tarek Abdelzaher, Karl-Erik Årzén, Anton Cervin, Theodore Baker, Alan Burns, Giorgio Buttazzo, Marco Caccamo, John Lehoczky, and Aloysius K Mok. Real time scheduling theory: A historical perspective. *Real-time systems*, 28:101–155, 2004.

[SADO17] Karila Palma Silva, Luis Fernando Arcaro, and Romulo Silva De Oliveira. On using gev or gumbel models when applying evt for probabilistic wcet estimation. In *2017 IEEE Real-Time Systems Symposium (RTSS)*, pages 220–230. IEEE, 2017.

[Sat96] Mahadev Satyanarayanan. Fundamental challenges in mobile computing. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, pages 1–7, 1996.

[SB96] Marco Spuri and Giorgio Buttazzo. Scheduling aperiodic tasks in dynamic priority systems. *Real-Time Systems*, 10(2):179–210, 1996.

[SBAP20] Václav Struhár, Moris Behnam, Mohammad Ashjaei, and Alessandro V Papadopoulos. Real-time containers: A survey. In *2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2020.

[SBCD09] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing*, 8(4):14–23, 2009.

[SCA+21] Václav Struhár, Silviu S Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V Papadopoulos. React: Enabling real-time container orchestration. In *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8. IEEE, 2021.

[SCA+23] Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro Vittorio Papadopoulos. Hierarchical resource orchestration framework for real-time containers. *ACM Transactions on Embedded Computing Systems*, 2023.

[sci] Dual annealing documentation. https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.dual_annealing.html. Accessed: 20.06.2023.

[Sha04] N. Shakhlevich. Handout 5: Computational complexity; approximation algorithm, 2004.

[SK11] Sanjit A Seshia and Jonathan Kotker. Gametime: A toolkit for timing analysis of software. In *Tools and Algorithms for the Construction and Analysis of Systems: 17th International Conference, TACAS 2011, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2011, Saarbrücken, Germany, March 26–April 3, 2011. Proceedings 17*, pages 388–392. Springer, 2011.

[SL03] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *RTSS 2003. 24th IEEE Real-Time Systems Symposium, 2003*, pages 2–13. IEEE, 2003.

[SRS+17] Pedro M Pinto Silva, Joao Rodrigues, Joaquim Silva, Rolando Martins, Luís Lopes, and Fernando Silva. Using edge-clouds to reduce load on traditional wifi infrastructures and improve quality of experience. In *2017 IEEE 1st International Conference on Fog and Edge Computing (ICFEC)*, pages 61–67. IEEE, 2017.

[SSL89] Brinkley Sprunt, Lui Sha, and John Lehoczky. Aperiodic task scheduling for hard-real-time systems. *Real-Time Systems*, 1:27–60, 1989.

[TMV18] Timur Tasci, Jan Melcher, and Alexander Verl. A container-based architecture for real-time control applications. In *2018 IEEE International Conference on Engineering, Technology and Innovation (ICE/ITMC)*, pages 1–9. IEEE, 2018.

[TRA15] Andrea Tosatto, Pietro Ruiu, and Antonio Attanasio. Container-based orchestration in cloud: state of the art and challenges. In *2015 Ninth international conference on complex, intelligent, and software intensive systems*, pages 70–75. IEEE, 2015.

[TS96] Constantino Tsallis and Daniel A Stariolo. Generalized simulated annealing. *Physica A: Statistical Mechanics and its Applications*, 233(1-2):395–406, 1996.

[VRMCL08] Luis M Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: towards a cloud definition, 2008.

[WCD+19] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The smt competition 2015–2018. *Journal on Satisfiability, Boolean Modeling and Computation*, 11(1):221–259, 2019.

[WEE+08] Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, et al. The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems (TECS)*, 7(3):1–53, 2008.

[XFJ16] Bruno Xavier, Tiago Ferreto, and Luis Jersak. Time provisioning evaluation of kvm, docker and unikernels in a cloud platform. In *2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*, pages 277–280. IEEE, 2016.

[XGSH13] Yang Xiang, Sylvain Gubian, Brian Suomela, and Julia Hoeng. Generalized simulated annealing for global optimization: the gensa package. *R J.*, 5(1):13, 2013.

[YFN+19] Ashkan Yousefpour, Caleb Fung, Tam Nguyen, Krishna Kadiyala, Fatemeh Jalali, Amirreza Niakanlahiji, Jian Kong, and Jason P Jue. All one needs to know about fog computing and related edge computing paradigms: A complete survey. *Journal of Systems Architecture*, 98:289–330, 2019.

[z3d] Z3 guide parameters. `https://microsoft.github.io/z3guide/programming/Parameters/#sat`. Accessed: 26.06.2023.