# Sammanfattning

Containerbaserad virtualisering har blivit en fördelaktig driftsmodell för heterogena distribuerade beräkningsmiljöer, inklusive moln- och foggplattformar, tack vare dess lättviktiga isolering och prestanda nära inbyggd hårdvara. Genom att möjliggöra effektiv konsolidering av fysiska resurser kan containrar minska infrastrukturens fotavtryck, sänka driftkostnader och förbättra systemutnyttjandet. Trots dessa fördelar ställer många industriella domäner strikta domänspecifika krav, framför allt avseende temporal förutsägbarhet (realtidsbeteende). Konventionella containerteknologier garanterar inte i sig sådan förutsägbarhet. I praktiken uppvisar containeriserade arbetslaster ofta variationer i exekverings- och svarstider, vilket orsakas både av deras egna resursbehov och av interferens från samlokaliserade containrar. Denna tidsmässiga instabilitet härrör från konkurrens om delade hårdvaruresurser, såsom cacheminne på sista nivån (Last-Level Cache, LLC), primärminne och databussar.

Denna doktorsavhandling syftar till att förbättra den temporala förutsägbarheten hos containerbaserad virtualisering genom att justera containerreserveringar av resurser. Avhandlingen undersöker ingående flera aspekter relaterade till mjuka realtidssystem baserade på containerbaserad virtualisering. Följande bidrag presenteras specifikt: (i) en systematisk litteraturöversikt som sammanfattar befintliga forskningsinriktningar för att möjliggöra realtidsanpassad containerbaserad virtualisering; (ii) en containerorkestrerare utformas för att underlätta driftsättning och anpassning av realtidscontainrar, med implementation i Kubernetes; (iii) ett hierarkiskt resursorkestreringsramverk utvecklas för realtidscontainrar, bestående av en flerskiktshierarki för resurshantering som möjliggör resursanpassning; samt slutligen (iv) en virtualisering av dynamiska styrsystem introduceras, vilken anpassar exekveringsfrekvenser efter systemdynamiken och optimerar den övergripande prestandan.

# Abstract

Container-based virtualization has become an advantageous deployment model for heterogeneous distributed computing environments, including cloud and fog platforms, due to its lightweight isolation and near-native performance. By enabling efficient consolidation of physical resources, containers can reduce infrastructure footprint, lower operational costs, and improve system utilization. Despite these advantages, many industrial domains impose stringent domain-specific constraints, most notably temporal predictability (real-time behavior). Conventional container technologies do not inherently guarantee such predictability. In practice, containerized workloads often exhibit variability in execution and response times, driven by both their own resource demands and interference from co-located containers. This timing instability stems from contention over shared hardware resources, such as last-level caches (LLC), main memory, and data buses.

This doctoral thesis aims to enhance the temporal predictability of container-based virtualization by adjusting container resource reservations. It comprehensively investigates various aspects pertaining to soft real-time container-based virtualization. Specifically, the following contributions are presented: (i) a systematic literature review summarizes existing research directions aimed at enabling real-time container-based virtualization; (ii) a container orchestrator is designed to facilitate the deployment and adaptation of real-time containers with its implementation in Kubernetes; (iii) a Hierarchical Resource Orchestration Framework is developed for real-time containers, comprising a multi-layer hierarchy of resource management to facilitate resource adaptation; finally (iv) a virtualization of dynamic control systems is introduced that adapt the execution rates to system dynamics and optimize the overall performance.

# Acknowledgments

I would like to express my sincere gratitude to all those who supported me throughout this research on my PhD path. First and foremost, I extend my deepest appreciation to my supervisors, Moris Behnam, Alessandro Papadopoulos, Mohammad Ashjaei, and Silviu Craciunas, for their guidance, encouragement, and insightful feedback during every stage of this work.

I am grateful to the many colleagues I have met at Mälardalen University, many of whom have become my close friends. Special thanks to Robbert Jongeling, my companion on tens of thousands of kilometers of road cycling across Sweden, and to Salman Sheik and Lan Van Dao for our numerous conversations. Additionally, thanks to Muhammad Abbas Khan, Shahriar Hasan, Arnab Barua, Nitin Desai, Zeinab Bakhshi, Mirgita Frasheri, Auday Al-Dulaimy, Leo Hatvani, Jean Malm, Adnan Ghaderi, Taufik Akbar Sitompul, and Malvina Latifaj for bringing positive energy and good spirit to my work life and beyond.

I have always been lucky to have great teachers and colleagues from whom I could learn a lot and who always gave me the support I needed. Starting with my primary school math teacher, Mrs. Miroslava Pritzlova, who guided me through my first near-impossible math tasks. Pavel Muki Muknsnabl, for introducing me to the blue screen of Turbo Pascal and helping me write my very first "Hello, World!" program. Jindrich Bohm, Jiri Svetlik, and Jiri Dobry, for giving me the opportunity to take my first steps as a software developer and guiding me through practical aspects of software engineering at Kadel Software. Jakub Stochl, Peter Stanovsky, Shukhrat Raimov, Bakytzhan Serikbayev, and Pavol Kuslita, for giving me the opportunity to work at IBM and Vodafone and helping me understand the challenges of developing software at scale. Finally, I would like to express my appreciation to my current col-

leagues at Hitachi Energy for their constant support and inspiring teamwork, namely Ramon Cuesta-Perez, Henrik Pind, Lasse Kinnunen, Richard Lovgren, Patrik Jonsson, Patrik Forst, John Backstrand, William Mokhlef, Jakub Sipowicz, Nagesh H Hegde, and Mohan Reddy Putluru.

I am deeply thankful to my family for their constant encouragement and support.

Václav Struhár, Västerås, May 2026

# List of Publications

## Papers included in this thesis[1]

**Paper A:** *Václav Struhár*, Moris Behnam, Mohammad Ashjaei, Alessandro V. Papadopoulos, "*Real-time containers: A survey*",
In the 2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020).

**Paper B:** *Václav Struhár*, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos, "*REACT: Enabling real-time container orchestration*",
In the 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2021).

**Paper C:** *Václav Struhár*, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos, Silviu S. Craciunas, "*Resource Adaptation for Real-Time Containers Considering Quality of Control*",
In the 28th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2023).

**Paper D:** *Václav Struhár*, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos, "*Hierarchical resource orchestration framework for real-time containers*",
In the ACM Transactions on Embedded Computing Systems (TECS 2023).

**Paper E:** *Václav Struhár*, Alessandro V. Papadopoulos, Inmaculada Ayala,

[1] The included publications have been reformatted to comply with the thesis layout.

Mercedes Amor, Lidia Fuentes, "*Container Orchestration in Edge Computing with Fluctuating Green Energy: A Multi-Armed Bandit Approach*",
In the 17th International Conference on Ubiquitous Computing and Ambient Intelligence (UCamI 2025).

**Paper F:** *Václav Struhár*, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos, "*RT-SCALER: Adaptive Resource Allocation Framework for Real-Time Containers*",
In the 1st International Workshop on Real-time and intelliGent Edge computing (RAGE 2022).

# Contents


**I Thesis** **1**

**1 Introduction** **5**

1.1 Performance isolation of container-based virtualization . . . . 7

1.2 Thesis outline . . . . . 11

**2 Research Overview** **13**

2.1 Context & research goals . . . . . 13

2.2 Research process . . . . . 15

**3 Background & Related Work** **19**

3.1 Virtualization . . . . . 19

3.2 Hypervisor-based virtualization . . . . . 23

3.3 Container-based virtualization . . . . . 24

3.3.1 Architecture of container-based virtualization . . . . . 26

3.3.2 Open Container Initiative . . . . . 29

3.4 Real-time systems . . . . . 30

3.4.1 Real-time virtualization . . . . . 32

3.4.2 Performance interference and Resource adaptation in container-based virtualization . . . . . 36

3.5 Container orchestration . . . . . 37

3.5.1 Container orchestrators . . . . . 38

3.5.2 Architecture of container orchestrator . . . . . 39

3.5.3 Real-time container orchestration . . . . . 41

**4 Research Results** **43**
4.1 Results overview . . . . . 43
4.1.1 Real-time container-based virtualization: main enablers, limitations, and open challenges . . . . . 43
4.1.2 Dynamic resource allocation for real-time containers . 44
4.1.3 Enabling real-time container orchestration . . . . . 48
4.2 Thesis Contributions . . . . . 55
4.3 Mapping of contributions, research questions, and publications 56
4.4 Paper contributions . . . . . 57
4.4.1 Personal contributions . . . . . 57
4.4.2 Included papers . . . . . 58
4.4.3 Not Included Papers . . . . . 65

**5 Conclusion** **67**
5.1 Conclusion and summary . . . . . 67
5.2 Discussion and future work . . . . . 68
5.2.1 Future Work . . . . . 69

**Bibliography** **71**

**II Included Papers** **85**

**6 Paper A:**
**Real-time containers: A survey** **87**
6.1 Introduction . . . . . 2
6.2 The Review Process . . . . . 3
6.2.1 Question formalization . . . . . 3
6.3 Containers technology . . . . . 3
6.3.1 Container platforms . . . . . 4
6.3.2 Real-Time Containers . . . . . 4
6.3.3 Real-time support of Linux . . . . . 4
6.4 Survey Results . . . . . 5
6.4.1 Real-time patch based . . . . . 5
6.4.2 Co-Kernel based . . . . . 8
6.4.3 Hierarchical Scheduling based . . . . . 9
6.4.4 Custom real-time approaches . . . . . 9

6.5 Weaknesses of real-time containers . . . . . 10
6.6 Conclusion . . . . . 11
Bibliography . . . . . 13

**7 Paper B:**
**REACT: Enabling real-time container orchestration** **17**
7.1 Introduction . . . . . 2
7.2 Background and Prior Work . . . . . 3
7.3 Orchestration of real-time containers . . . . . 6
7.3.1 System Model . . . . . 6
7.3.2 Performance Metrics . . . . . 7
7.3.3 Container Level Metrics . . . . . 8
7.4 Design of the Real-Time (RT) Orchestrator . . . . . 9
7.4.1 RT Extension of the Master Node . . . . . 9
7.4.2 RT Extension of Compute Nodes . . . . . 12
7.5 Implementation . . . . . 16
7.6 Evaluation . . . . . 18
7.7 Conclusion . . . . . 21
Bibliography . . . . . 23

**8 Paper C:**
**Resource Adaptation for Real-Time Containers Considering Quality of Control** **27**
8.1 Introduction . . . . . 2
8.2 Related Work . . . . . 4
8.3 Technical Background . . . . . 5
8.3.1 Control with flexible timing constraints . . . . . 5
8.3.2 Resource reservation for real-time containers . . . . . 6
8.4 QoC Controller . . . . . 7
8.4.1 Control Period Controller . . . . . 10
8.4.2 Resource Reservation Controller . . . . . 11
8.4.3 Interference Mitigation . . . . . 11
8.4.4 Communication with other controllers . . . . . 12
8.5 Experimental Evaluation . . . . . 12
8.6 Implementation on Linux . . . . . 17
8.7 Conclusions and future work . . . . . 17

Bibliography . . . . . 19

**9 Paper D:**
**Hierarchical resource orchestration framework for real-time containers 23**
9.1 Introduction . . . . . 2
9.2 Background and Related Work . . . . . 6
9.3 System architecture and System model . . . . . 7
9.4 Hierarchical Resource Orchestration Framework . . . . . 12
9.4.1 Offline Phase . . . . . 13
9.4.2 Online Phase . . . . . 17
9.4.3 Implementation . . . . . 23
9.5 Evaluation of the framework . . . . . 27
9.5.1 Demonstration of performance interference between containers . . . . . 27
9.5.2 Evaluation of the offline phase . . . . . 27
9.5.3 Evaluation of container-level and node-level controllers 29
9.5.4 Evaluation of cluster-level controller . . . . . 32
9.6 Discussion . . . . . 34
9.7 Conclusion . . . . . 35
Bibliography . . . . . 37

**10 Paper E:**
**Container Orchestration in Edge Computing with Fluctuating Green Energy: A Multi-Armed Bandit Approach 43**
10.1 Introduction . . . . . 2
10.2 Background: Multi-Armed Bandits . . . . . 4
10.3 Related Work . . . . . 5
10.4 System Model . . . . . 7
10.5 Modeling . . . . . 7
10.5.1 Learning algorithm for optimal policy: LinUCB . . . . 9
10.6 Orchestrator Architecture . . . . . 10
10.7 Experimental evaluation . . . . . 11
10.8 Conclusion . . . . . 15
Bibliography . . . . . 17

**11 Paper F:**
**RT-SCALER: Adaptive Resource Allocation Framework for Real-Time Containers** **21**
11.1 Introduction . . . . . 2
11.2 Container orchestration . . . . . 4
11.2.1 Offline phase . . . . . 5
11.2.2 Online phase . . . . . 6
11.3 Practical Insights . . . . . 8
11.4 Conclusion . . . . . 11
Bibliography . . . . . 13

# I

# Thesis

# List of Acronyms

| | |
|---|---|
| **API** | Application Programming Interface |
| **BE** | Best Effort |
| **cgroups** | Control Groups |
| **CLM** | Container Level Metrics |
| **CMAB** | Contextual Multi Armed Bandit |
| **COTS** | Commercially available Off-The-Shelf |
| **CPU** | Central Processing Unit |
| **DMA** | Direct Memory Access |
| **FEC** | Fog and Edge Computing |
| **HCBS** | Hierarchical Constant Bandwidth Server |
| **IACS** | Industrial Automation and Control System |
| **IED** | Intelligent Electronic Devices |
| **IoT** | Internet of Things |
| **IPC** | Interprocess Communication |
| **LLC** | Last Level Cache |
| **OCI** | Open Container Initiative |
| **OSLM** | Operating System Level Metrics |
| **PID** | process ID |
| **QoC** | Quality of Control |
| **QoS** | Quality of Service |
| **RTAI** | Real-Time Application Interface |
| **RTDS** | Real-Time Deferrable Server |
| **RTOS** | Real-Time Operating System |
| **RT** | Real-Time |
| **SMMU** | System Memory Management Unit |
| **TSN** | Time Sensitive Networks |
| **vCPU** | virtual CPU |

| | |
|---|---|
| **VM** | Virtual Machine |
| **vPAC** | virtual Protection, Automation, and Control |
| **WCET** | Worst-Case Execution Time |

# Chapter 1

# Introduction

Container-based virtualization and container orchestration have become the de facto standard for packaging and deploying applications in cloud environments [1, 2, 3]. Virtualization provides isolation and resource control [4], enabling the co-location of isolated applications on shared hardware platforms, while orchestration automates deployment, manages containerized applications, and scales them across compute clusters. Together, virtualization and orchestration are enabling recent paradigms such as cloud, fog, and edge computing, in which applications are distributed across the entire cloud continuum [5]. Across several industrial domains, including Industrial Automation and Control System (IACS) and virtual Protection, Automation, and Control (vPAC) [6], there is a growing interest in adopting container-based virtualization and container orchestration to support the transition towards Industry 4.0 [7, 8, 9, 10]. These technologies enable key objectives of industrial domains, including greater flexibility in manufacturing processes, decentralized and distributed functions, higher degrees of automation, and more efficient utilization of computational resources.

Industrial domains, however, impose stringent requirements on temporal behavior and timing precision (also denoted as real-time) [11, 12]. Applications must satisfy both functional correctness and timeliness to produce computation results [13, 8]. If real-time behavior is compromised, the utility of the results may degrade (soft and firm real-time) or lead to catastrophic failures (hard real-time) [14]. Achieving real-time behavior is challenging in systems that de-

pend on container-based virtualization (and virtualization in general) deployed on Commercially available Off-The-Shelf (COTS) heterogeneous platforms. The underlying architecture of commercially available platforms is extremely complex and non-amenable to straightforward timing analysis [15]. Real-time behavior can be undermined by non-determinism introduced by hardware (e.g., CPU caches, memory buses, memory caches, and interrupt controllers) and by software stacks (e.g., process scheduler, network stack, device drivers, and interrupt handling). Many of these components are heavily optimized to maximize average-case performance and throughput, often at the expense of strict temporal predictability and bounded-latency guarantees, as seen in features such as caches, pipelines, and branch prediction mechanisms [16]. As mentioned by Dasari et al. [15], *the presence of shared hardware resources, like the memory bus and the memory controller, governed by performance-oriented arbiters and schedulers, facilitates neither composable analysis nor the computation of upper bounds on the key timing parameters on these systems.* Additionally, Cucinotta et al. [17] note that *virtualization brings a number of challenging issues for real-time workloads. The increased level of resource sharing among multiple virtual systems makes it difficult to run software in predictable ways, as the performance of each virtual system depends on the amount of resources (e.g., computing, storage, or networking) other virtual systems are consuming* [17].

Accurate knowledge of upper bounds, such as Worst-Case Execution Time (WCET), is essential for systems that require predictable timing behavior and reliable resource allocation. However, obtaining tight upper bounds in virtualized systems is challenging due to interference caused by the sharing of hardware and software resources among co-located workloads. Underestimating these bounds can compromise real-time behavior, whereas overestimating them may lead to inefficient resource utilization.

Furthermore, unpredictable and time-varying workload patterns further hinder real-time behavior [18]. For instance, data requiring cache-intensive processing may simultaneously arrive at multiple containers. Consequently, processing is triggered concurrently across these containers, leading to contention and interference in shared resources; thus, co-located workloads on the same machine exhibit additional performance variability due to resource sharing [19]. As a result, maintaining real-time behavior in container-based virtualization becomes challenging given the system's dynamic nature,

where multiple co-located real-time containers operate on the same hardware platform and may interact in complex and unpredictable ways. This variability can lead to fluctuations in task execution times, increased scheduling delays, and reduced predictability, all of which pose significant challenges to meeting stringent real-time constraints.

The central premise of this thesis is that real-time behavior in container-based virtualization cannot be reliably achieved through static design alone due to resource contention and workload variability. Instead, it requires continuous, informed, and coordinated decision-making at runtime, which is enabled by dynamic resource management and orchestrator support.

## 1.1 Performance isolation of container-based virtualization

Due to container-based virtualization limited isolation mechanism, co-located containers can influence each other's execution time in unpredictable ways [20, 18]. This issue, commonly referred to as the noisy-neighbor problem [21, 22, 23, 19], arises when multiple containers, Virtual Machines (VMs), or services share a resource pool, and one consumes enough shared resources to degrade the quality of service of the others. A workload in a shared environment uses one or more resources in ways that affect other workloads operating around it [22, 19]. The unpredictability of, and limited visibility into, co-located workloads can lead to situations where multiple workloads contend for scarce shared resources.

Several factors have been reported that can hinder real-time performance due to concurrent resource access by the containers, ranging from CPU, disk access [24, 25], network [26], shared front-side bus [15], or shared caches. A comprehensive overview of the sources of unpredictability on COTS is presented by Dasari et al. [15]. As an illustrative example, at the hardware level, co-located containers share the Last Level Cache (LLC), which is accessed by CPU cores prior to fetching data from main memory. Intensive simultaneous use of the LLC by multiple containers may increase response time for containerized applications and negatively affect time predictability [27]. Even basic workloads like matrix multiplication can expose LLC constraints, causing frequent cache misses and memory bandwidth saturation [28]. In modern

CPUs, cores share the LLC to increase utilization, simplify coherence, and accelerate core-to-core communication [27, 29]. Figure 1.1 illustrates LLC and memory bandwidth contention on an 8-core processor. The resulting performance interference is shown empirically in our experiment in Figure 1.2. Figure 1.2a shows how the response time of a container is affected by a co-located container running on another core and periodically switching between cache-intensive and non-cache-intensive workloads. As a result, the measured container's response times increase during cache-intensive periods. Similarly, Figure 1.2b shows how response time changes as additional containers performing cache-intensive operations are co-located with the monitored application.

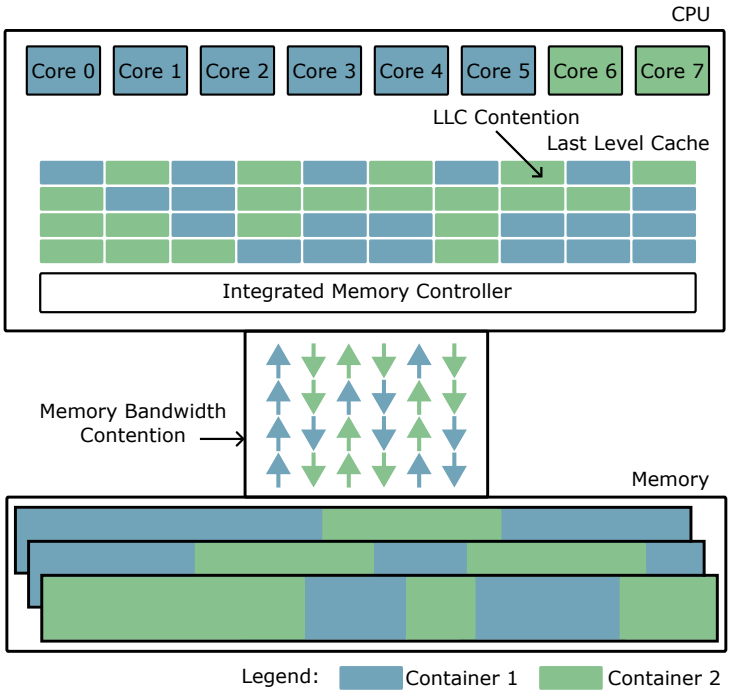
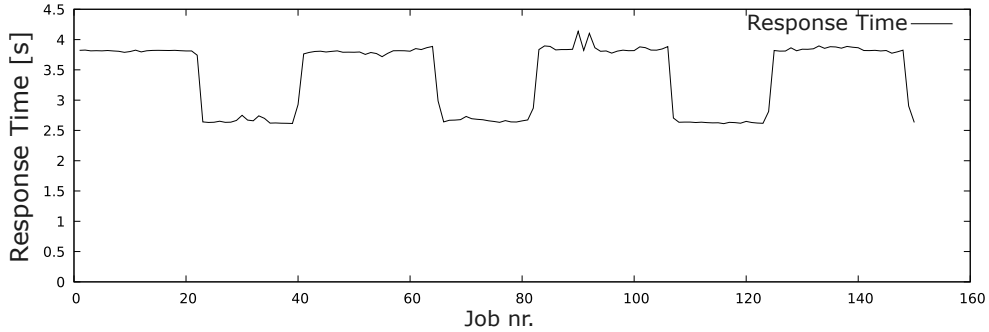



Figure 1.1: Illustration of LLC and memory bandwidth contention between two containers on an 8-core processor. Adapted from [30].

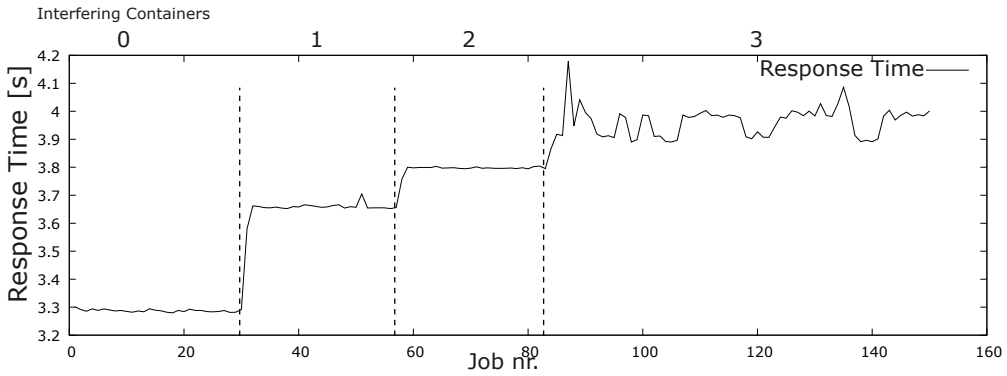
Additionally, the unpredictability of workload patterns and variable inter-

ference levels make it difficult to ensure real-time behavior in container-based virtualization [31], particularly on heterogeneous platforms where interference effects are workload- and architecture-dependent and cannot be fully characterized a priori. As container workloads fluctuate, they can introduce unforeseen impacts on neighboring containers, making interference difficult to anticipate and control. As stated in [25], the container-based system is not yet mature enough to ensure performance isolation.

This thesis examines multiple facets of soft real-time container-based virtualization. More precisely, we present a systematic literature survey that summarizes research on incorporating real-time properties into container-based virtualization. We design a container orchestrator that enables the deployment and resource adaptation of real-time containers. Finally, we connect resource adaptation to containers hosting dynamic control systems.




(a) The experiment consists of two containers: a monitored container in which response time is measured and shown in the figure, and a second container that periodically performs a cache-intensive workload (matrix multiplication). As shown, the response time of the monitored container degrades periodically, coinciding with the execution of the cache-intensive workload.




(b) The experiment shows the response time of the monitored container under varying levels of interference from co-running containers executing a cache-intensive workload. The number of interfering containers ranges from zero to three. As the number of interfering containers increases, a gradual degradation in response time can be observed.

Figure 1.2: Two experiments demonstrating performance interference between containers caused by contention of the LLC.

## 1.2 Thesis outline

The thesis is divided into two main parts. The first part introduces the overall scope of the work and comprises five chapters, while the second part presents all included publications, consisting of six research papers. The overview of the chapters included in this thesis is provided below.

**Chapter 1 - Introduction:** This chapter introduces the research area of real-time container-based virtualization and outlines our motivation and key challenges.
**Chapter 2 - Research Overview:** This chapter outlines the overall research goal, presents the research questions along with explanations of their significance, and describes the research process.
**Chapter 3 - Background & Related Work:** This chapter provides the technical background relevant to the research area addressed in this thesis, including virtualization (both hypervisor-based virtualization and container-based virtualization), general real-time systems, and real-time orchestration. Furthermore, we review the state of the art and position our contributions within the existing research.
**Chapter 4 - Research Results:** This chapter presents the contributions and maps them to the research questions they address. Additionally, the chapter provides a summary of the papers included in the thesis.
**Chapter 5 - Conclusion:** This chapter concludes the thesis by summarizing our findings and suggesting potential future work related to the presented research.

# Chapter 2

# Research Overview

In this chapter, we present the overall goals of the thesis and the research process used to achieve them.

## 2.1 Context & research goals

The overall goal of this thesis is to design an autonomous resource mechanism for real-time container-based virtualization to deal with workload variability, resource contention, and the resulting performance unpredictability in shared dynamic environments. We address this challenge in the context of soft real-time applications, with a particular focus on CPU resources. Although numerous studies have explored resource isolation in systems hosting multiple applications, many of them concentrate on specific aspects, such as cache [32] or bus [33] isolation, to improve real-time predictability.

To achieve the goal, we aim to advance the state of the art in real-time container-based virtualization by systematically analyzing existing real-time container technologies to identify their technical gaps, performance limitations, and remaining research challenges. Building on this foundation, we develop orchestration mechanisms that interpret and enforce the timing constraints and resource reservations for real-time container-based virtualization. To further enhance predictability in dynamic environments using container-based virtualization, we design mechanisms that continuously adapt resource

reservations, ensuring that real-time containers remain temporally correct even under workload variability and interference. Finally, we introduce a multi-level orchestration architecture that performs coordinated control across container-, node-, and cluster-levels, enabling local resource reservations adaptation, resource redistribution, and container migration while preserving real-time behavior and optimizing overall system resource utilization.

The overall research goal is decomposed into more specific research questions. We identify three specific research questions as outlined below.

**$RQ_1$: What are the main enablers, limitations, and open challenges in achieving real-time behavior in container-based virtualization?**

Real-time container-based virtualization is a novel research area, and a structured understanding of existing approaches is still limited. This research question seeks to identify approaches to enabling real-time behavior in container-based virtualization and to highlight the challenges that may impede successful implementation. The research question aims to provide insights into research and development efforts to address the current limitations of container-based virtualization and to identify potential areas for further improvement. At the time this research question was formulated, no systematic study provided a comprehensive overview of real-time container-based virtualization.

**$RQ_2$: How can resource allocation for real-time containers be dynamically adapted at runtime to ensure predictable performance under workload variability and interference, leveraging performance and Quality of Control (QoC) metrics?**

The container-based virtualization exhibits unstable processing times that vary widely with the executed workloads and the workloads of other co-located containers. In the literature, the adaptation is addressed only marginally, e.g., in [34], and the proposed solutions do not address real-time systems. This research question aims to investigate how computing resources (i.e., CPU reservation) can be dynamically adapted to containers based on their real-time requirements and observed performance.

Additionally, industrial domains employ control systems to regulate the operation of other devices or systems via control loops [35]. Control loops often involve rapid changes in response to varying conditions (e.g., shorter loops

when the controlled system is unstable). This has important implications for resource reservation mechanisms, as it requires dynamic resource allocation. In the scope of this research question, we aim to explore resource reservation for container-based virtualization driven by QoC performance.

**$RQ_3$: How can container orchestration frameworks be extended to incorporate real-time requirements, enabling schedulability-aware placement, and resource management of containerized workloads in distributed environments?**

Existing container orchestration platforms are primarily designed for best-effort, throughput-oriented workloads, and they lack direct support for real-time. In practice, container orchestrators decide where to place containers primarily by checking whether a compute node has enough resources and satisfies policy constraints. Once placed, the application is left to compete for CPU time, cache, and network bandwidth. The aim of this research question is to identify, design, and evaluate an extension to the existing container orchestration system (i.e., Kubernetes) to support real-time behavior.

## 2.2 Research process

In this thesis, the hypothetico-deductive method [36] is used, including steps depicted in Figure 2.1. To formulate the research questions and validate the proposed solutions, the study follows the steps outlined below. Part of the research was conducted in collaboration with industrial partners at TTTech[1], with ongoing discussions to align the research direction and outcomes with industry needs. TTTech focuses on safety-critical systems, with an emphasis on predictable networking, particularly Time Sensitive Networks (TSN). To fully leverage TSN capabilities, real-time computing support is essential, which can be potentially achieved through real-time container-based virtualization. In collaboration with TTTech, we conducted a systematic literature review and implemented most of the approaches reported in the published papers.

**Problem definition:** This step focuses on understanding the research area through a literature review, including state of the art and state of practice stud-

[1] https://www.tttech.com/

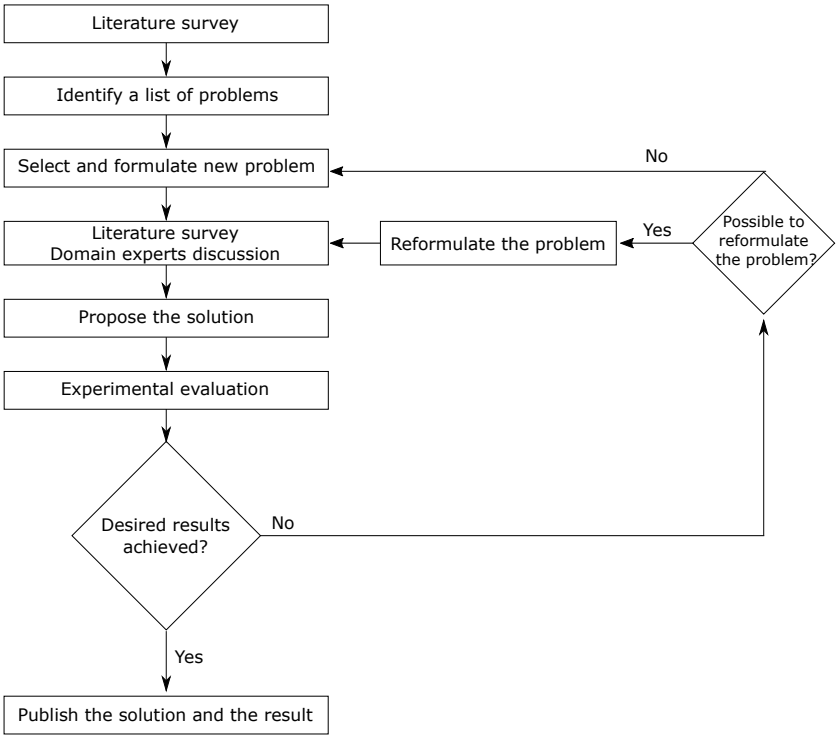



Figure 2.1: The research process adopted in this thesis.

ies, to identify existing gaps. Based on these findings, the scope of the problem is defined, setting boundaries for the research.

**Idea development:** This step focuses on developing solutions to the defined problem through an iterative process. It starts with brainstorming and generating ideas, followed by proposing and discussing possible approaches. These ideas are then refined and improved, taking into account the problem's constraints and requirements.

**Implementation or simulation:** In this step, we either implement our approach directly (e.g., extend a Linux kernel scheduler) to observe its real runtime behavior or evaluate it in a simulated environment to study how our proposed approach behaves under controlled conditions.

**Adjusting the solution or reformulating the problem:** This step occurs early

in the research due to, e.g., incomplete information or incorrect assumptions that arise during implementation or simulation. As preliminary results expose gaps between the expected and observed behavior, we iteratively refine either the proposed solution or the problem formulation.

**Evaluation and validation:** This step evaluates both the concept and its implementation. The solution is evaluated against predefined benchmarks to assess its effectiveness, and the results are analyzed to draw conclusions about its performance. Furthermore, the proposed solution is compared with existing alternatives to highlight its relative strengths and weaknesses.

# Chapter 3

# Background & Related Work

This section begins by introducing the concept of virtualization, followed by an examination of two widely adopted approaches to virtualization: hypervisor-based virtualization and container-based virtualization. Subsequently, this section presents the fundamentals of real-time systems, including important concepts such as real-time tasks and their characterization. Building on these foundations, this section establishes the relationship between virtualization and real-time systems. Finally, background information on container orchestration is presented.

## 3.1 Virtualization

Virtualization [37] plays a prominent role in many areas of today's computing, including cloud and fog computing, embedded systems, and Internet of Things (IoT). It involves creating a virtual version of a physical system [38], including but not limited to virtual computer systems, computer networks, operating systems, or storage devices [39] that are abstracted from the underlying physical hardware [40, 41]. Virtualization increases resource efficiency, reduces the physical hardware footprint, and lowers operational costs. This thesis focuses on container-based virtualization, a lightweight and efficient approach that shares host operating system components while preserving application-level isolation. However, to establish the appropriate background and enable

comparison, this section also briefly introduces hypervisor-based virtualization[1]. VMs and container-based virtualization are two main approaches within compute virtualization: VMs run on a hypervisor and encapsulate an entire system, including a full guest operating system, providing strong isolation and the ability to run different operating systems on the same host, but with higher resource overhead and slower startup times [42]. Container-based virtualization, in contrast, virtualizes at the operating system level by sharing the host kernel while packaging applications and their dependencies into isolated units called containers, making them lighter, faster to start, and more efficient for scaling than VMs. These two virtualization approaches are outlined in Figure 3.1.

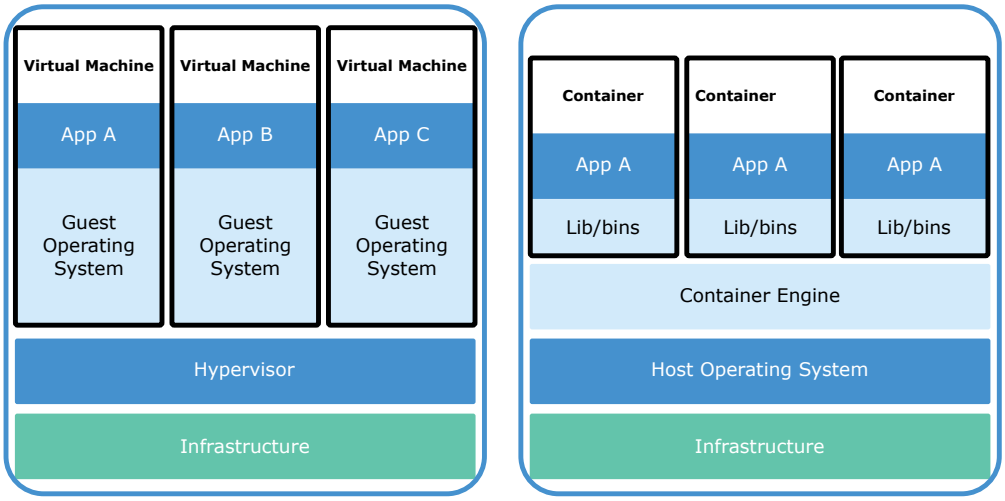



Figure 3.1: Comparison of hypervisor-based virtualization and container-based virtualization [41].

Virtualization introduces several advantages, including but not limited to:

- **Resource Efficiency**: Virtualized systems can run on a single physical system, reducing hardware footprint. Hardware resources can be dynamically distributed among the virtualized systems at runtime.

[1]Throughout this thesis, the terms hypervisor-based virtualization and VMs are used interchangeably to refer to virtualization approaches that rely on a hypervisor to provide isolation and execution environments.

- **Isolation and Security**: Virtualization isolates virtualized systems so that misbehavior in one instance cannot jeopardize the security of other virtualized systems or the host system [43].
- **Portability**: Virtualized systems are decoupled from the underlying hardware, allowing them to run on different machines. As a result, virtualized systems can be migrated easily between physical systems. It includes rapid cloning, backup, and deployment of the virtualized systems [43].
- **Costs Savings**: Virtualization enables organizations to reduce costs by consolidating multiple virtualized systems onto a smaller number of high-capacity physical systems, thereby improving resource utilization and minimizing hardware and operational expenses.
- **Reduce Downtime and Enhance Resiliency**: Virtualized systems can be migrated to a different physical system or cloud instance in the event of an outage or other disruption. This allows virtualized systems to remain running and accessible even in the event of hardware-related failures.
- **Sustainability**: Virtualization improves sustainability by reducing the need for hardware and lowering energy consumption [44].

Virtualization is becoming increasingly important in industrial domains, including IACS [45, 46] and vPAC [47, 6, 48], where real-time embedded systems are often involved. Virtualization enables the consolidation of multiple embedded systems, such as robot controllers, onto a single physical platform, leading to improved resource efficiency, lower operational costs, and greater sustainability. Additionally, productivity can be significantly enhanced during the development, deployment, and testing processes through software-based configuration rather than (often manual) hardware configuration. Also, virtualization supports advanced concepts, such as digital twins [49, 50], enabling simulation, monitoring, and optimization of physical systems in a virtual environment (digital copy of a physical system). In addition, virtualization enables novel technologies such as vPAC [47, 6, 48]. In the vPAC context, virtualization allows traditional electric substation protection functions (implemented in embedded Intelligent Electronic Devices (IEDs) [48]) to be implemented as virtualized real-time software components running inside VMs or containers, as illustrated in 3.2. The decoupling of software from hardware enables ven-

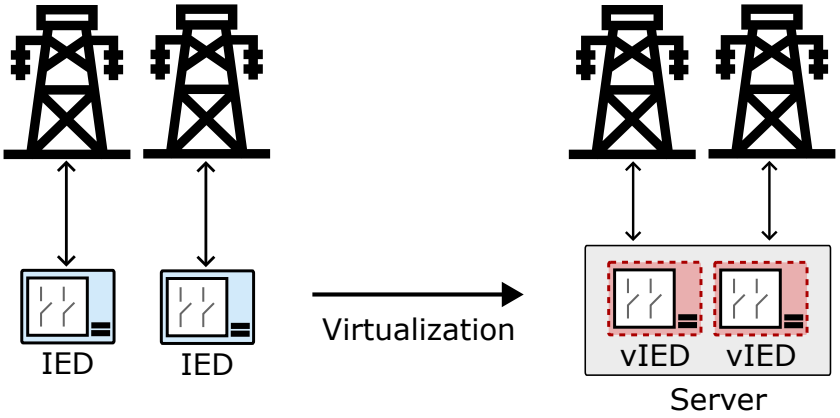



Figure 3.2: Virtualization of embedded systems in vPAC context. Multiple physical IEDs protecting electrical substations are virtualized and consolidated on a single physical server. This is an ongoing trend in electrical substation protection, e.g., by ABB [51] and Siemens SIPROTEC V [52].

dors to explore new delivery and business models, for example, by offering protection functions as deployable virtual systems rather than as fixed-function embedded devices [6].

However, industrial domains are subject to domain-specific timing constraints [53], including temporal predictability (also known as real-time behavior). For example, in motion control applications that move machines or mechanical components, cycle times can be as low as 1 ms, whereas fast control-loop applications typically require approximately 10 ms. For high-speed I/O operations, the cycle time required to execute this functionality can reach 100 $\mu$s [11]. Impaired real-time behavior can result in inaccurate motion. In vPAC, the required upper bound is 1 ms [51]. In addition to short control cycles, industrial domains demand low jitter. That is, tightly bound variation in latency so that timing remains deterministic and predictable.

As mentioned in [40], virtualization is all about isolation. The primary emphasis of virtualization is on spatial isolation. Spatial isolation means that each container or VM runs in its own separate view of system resources, so it cannot see or directly interact with resources belonging to other containers or VMs or the host (beyond what is explicitly shared). Temporal isolation

ensures that the temporal characteristics of an application are not affected by the execution of another co-located application [15]. Temporal isolation in virtualization is gaining increasing attention, driven by the growing need to support real-time systems.

The following sections provide an overview of the two main virtualization approaches [41]: hypervisor-based virtualization and container-based virtualization.

## 3.2 Hypervisor-based virtualization

A hypervisor (e.g., VMware, Hyper-V, KVM, Xen) is a software component [42] that creates and manages VMs by abstracting the physical hardware resources of a host system. It allows multiple operating systems to run concurrently on the same physical machine, each within its own isolated virtual environment. A hypervisor manages physical computing resources and makes isolated virtualized hardware resources available to VMs [54]. It serves as an abstraction layer between the physical hardware and the VMs, enabling multiple operating systems to run on the same physical hardware. Hypervisor-based virtualization incurs overhead that is difficult to mitigate [55][2]. Hypervisor-based virtualization can provide stronger temporal isolation than container-based virtualization, particularly when dedicated resource allocation and partitioning techniques are employed. First, hypervisors can allocate dedicated hardware resources to VMs, ensuring that each VM has an isolated, exclusive environment. Secondly, hypervisors occupy a more privileged position in the system architecture, enabling them to provide more comprehensive scheduling and resource allocation mechanisms for VMs. Finally, hypervisors provide finer-grained control over system-level operations, such as memory management and interrupt handling. This control enables hypervisors to effectively isolate the execution of different VMs, reducing the chances of interference and improving temporal isolation.

Hypervisors are commonly categorized into Type 0, Type 1, and Type 2 based on their placement in the system stack and their interaction with hardware. Type 0 hypervisors [57] are designed as part of the hardware or firmware and provide native partitioning with minimal software involvement, offering

[2]However, there is research aiming the lightweight VMs, e.g. [56].

extremely low overhead and strong determinism but limited flexibility. Type 1 hypervisors run directly on hardware without an intervening host operating system and manage VMs with strong isolation, high performance, and predictable behavior, making them suitable for cloud, real-time, and safety-critical systems [58]. Type 2 hypervisors operate on top of an operating system, relying on it for hardware access and resource management. While easier to deploy and more flexible, they incur higher overhead and weaker timing guarantees, which generally makes them less suitable for real-time or critical workloads.

## 3.3 Container-based virtualization

Container-based virtualization (also known as operating system virtualization [59]) traces its roots to the 1979 introduction of chroot [60], which provided basic filesystem isolation. Later, technologies such as FreeBSD Jails and Linux VServer extended this idea to include network and user isolation. A major breakthrough occurred with the development of Linux namespaces and control groups (cgroups) in the 2000s, which enabled process isolation and fine-grained resource management.

Container-based virtualization isolates applications and their associated components (e.g., binaries, libraries, data, and configuration files) into individual containers. Container-based virtualization relies on a shared host kernel, thereby using common system functionalities including process scheduling, memory management, and interrupt handling. In contrast to hypervisor-based virtualization, in the case of container-based virtualization, only the operating system kernel functionalities (and not all the hardware devices) are virtualized [42]. When container-based virtualization is used, the system relies on a host operating system kernel, which virtualizes its services for the guest containers and ensures isolation between them. Because all containers share this same kernel, they must use the same underlying hardware and software architecture. Container-based virtualization raises security concerns due to weaker isolation than VMs [61] potentially increasing the risk of privilege escalation, container escape, and cross-container attacks.

Container-based virtualization is more efficient than hypervisor-based virtualization, as it eliminates the need for multiple guest operating system instances and the associated hypervisor overhead. From the user's perspective, each container appears and executes as a standalone operating system [62].

Traditional VMs virtualize hardware and run separate guest operating systems, whereas containers do not require a full guest OS per instance. There are multiple container platforms, such as Docker[3], Podman[4], LXC/LXD[5]. The container platforms differ in design choices and features, e.g., daemon vs. daemon-less architecture, rootless support, user interface, and ecosystem integrations; they are all built upon the same underlying technologies and standards[6], for example:

- **Operating system mechanism:** Container-based virtualization relies on operating system mechanisms, particularly Linux namespaces for process, network, and filesystem isolation, and control groups (cgroups) for resource accounting and enforcement.
- **Standard container image formats and runtime interfaces:** Container-based virtualization relies on standardized image formats and runtime specifications defined by the Open Container Initiative (OCI). These standards ensure portability and interoperability, allowing container images to be built and executed across different container platforms and runtimes without modification.

Container-based virtualization offers several benefits, for example [45]:

- **Near native performance**: Container runtimes typically introduce minimal overhead, enabling execution speeds and resource efficiency close to running applications directly on the host [62].
- **Isolation and resource control**: Containers provide process-level isolation while allowing fine-grained resource governance (e.g., CPU limit, memory usage, I/O bandwidth, and network access). This helps prevent a misbehaving component from exhausting shared resources, supports workload prioritization, and improves overall system robustness.
- **Modular software architecture**: Containerization encourages decomposition into smaller, well-defined components with clear interfaces. This modularity simplifies deployment and orchestration, supports component reuse, and enables targeted updates or rollbacks of individual

[3] `https://www.docker.com/`

[4] `https://podman.io/`

[5] `https://linuxcontainers.org/`

[6] However, exceptions exist. For example, Kata Containers use lightweight VMs to isolate containers, providing stronger workload isolation and an additional security layer compared to traditional containers [63].

services without affecting the entire system, thereby reducing downtime and maintenance effort [45].
- **Lightweightness**: Compared to VMs, containers are smaller and start faster because they do not require a full guest operating system. This accelerates installation, replication, migration, and recovery.
- **Portability and reproducibility**: Applications packaged as container images can be deployed consistently across development, testing, and production platforms. This reduces configuration drift and supports more reliable validation, which is especially helpful in industrial systems where changes must be controlled and traceable [64].

In comparison to hypervisor-based virtualization, container-based virtualization has a negligible overhead, is more resource efficient (e.g., 29.4 times less RAM usage than a VM [65]), has higher flexibility, provides fast booting times [66], and enables near-native performance with negligible overhead [62, 67, 55, 68, 43].

### 3.3.1 Architecture of container-based virtualization

Container-based virtualization executes applications within isolated containers that share the host operating system kernel instead of virtualizing the entire hardware stack. Each container is typically created from a container image, a read-only package that bundles the application, its dependencies, and configuration. Container images are commonly stored and distributed via container registries, where they can be pulled (downloaded), pushed (uploaded), and reused across environments. A container engine is responsible for building images, pulling them from the registry, creating containers, and setting up isolation and resource limits. A container orchestrator coordinates multiple containers across multiple machines, performing placement, vertical and horizontal scaling, handling service discovery and networking, restarting failed instances, and rolling out updates. The overall architecture of Docker, one of the most prominent container platforms, is depicted in Figure 3.3[7].

The key components of container-based virtualization consist of:

- **Container:** A lightweight, isolated runtime environment that packages

[7] Adopted from `https://docs.docker.com/get-started/docker-overview/`

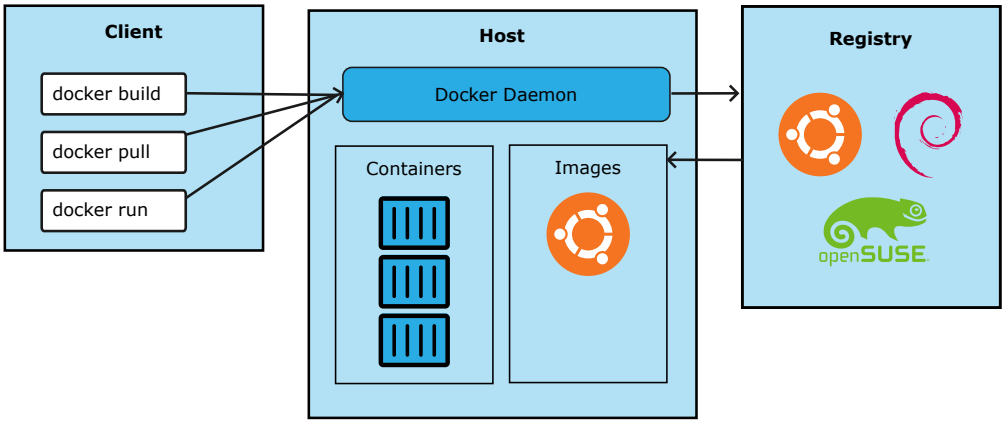



Figure 3.3: Docker Architecture.

an application together with its dependencies (libraries, binaries, configuration) so it can run consistently across different systems.

- **Container image:** A read-only package that bundles an application and its dependencies (libraries, binaries, configuration). A container is a runtime instance of a container image.
- **Registry:** A service for storing and distributing container images. It allows users and systems to push images to the registry and pull them as needed (e.g., from Docker Hub[8] or private registries).
- **Container engine:** The software responsible for creating, running, and managing containers on a host system. It manages images, container lifecycles, networking, and storage, and interacts with the daemon.
- **Container Daemon:** The container daemon is a long-running background service responsible for supervising container processes.
- **Container orchestrator:** A system that manages containers across multiple hosts, automating deployment, scaling, load balancing, health monitoring, and recovery of containerized applications (e.g., Kubernetes and Docker Swarm).

In Linux, container-based virtualization is implemented using Control Groups (cgroups) to account for resource usage and namespaces to control

[8] Available here: `https://hub.docker.com/`

visibility and provide isolation [42]. Namespaces restrict visibility and access to system resources, whereas cgroups regulate how much of those resources a container may consume. These two Linux features are described below.

### Linux cgroups

Linux cgroups[9] are a kernel-level mechanism for managing, limiting, and accounting for the resource usage of groups of processes in Linux systems. Rather than operating on individual processes, cgroups treat related processes as a single entity, enabling structured and fine-grained control over system resources such as CPU time, memory, and I/O bandwidth. This abstraction is important for modern systems that run multiple applications (or containers) concurrently on shared hardware.

Cgroups organize processes into a hierarchical structure in which resource constraints applied at a higher level are inherited by child groups. Resource management is implemented via dedicated controllers, each responsible for a specific resource type. These controllers allow the kernel to enforce hard limits, apply scheduling priorities, or collect detailed accounting information. As a result, cgroups enable resource allocation and help prevent one workload from monopolizing system resources. Cgroups can be modified at runtime. Unlike traditional static partitioning approaches, cgroup parameters can be adjusted dynamically while processes remain operational. For the purposes of this thesis, the CPU controller is of relevant, as it governs how CPU are allocated among containers. The CPU controller provides mechanisms for configuring the following parameters:

- **CPU Quota:** CPU quota limits the maximum amount of CPU time a container may consume during a scheduling period. This can be used for enforcing CPU reservations.
- **CPU Shares:** CPU shares define the relative priority of a container when CPU resources are contested. This can be used for fair resource distribution or workload prioritization.
- **CPU Affinity:** The cpuset controller restricts containers to specific processor cores. This can be used to limit interference between workloads.

[9] `https://man7.org/linux/man-pages/man7/cgroups.7.html`

#### Linux namespaces

Linux namespaces[10] enable process isolation in container-based virtualization. They partition kernel resources so that processes within a namespace perceive their own isolated instances of system components, such as process IDs (PIDs), network interfaces, mount points, user IDs, and Interprocess Communication (IPC) mechanisms. A namespace wraps a global system resource in an abstraction that makes it appear to processes within the namespace as if they have their own isolated instance of the global resource. Changes to the resource are visible to other processes that are members of the namespace, but are invisible to other processes outside the namespace [69].

### 3.3.2 Open Container Initiative

Open Container Initiative[11] (OCI) aims to define open standards for both container runtimes and container image formats, ensuring interoperability across different tools and platforms. The runtime specification defines how a container instance is configured and executed, while the image specification defines how container images are built and structured, including their layers, manifests, and metadata. OCI was created to standardize container technologies and prevent fragmentation in the container platform ecosystem, thus, container platforms such as Docker and Podman maintain compatibility. OCI defines these specifications:

- **Runtime Specification**: Defines how to unpack and launch a filesystem bundle as a container. Additionally, it defines how a container is started, the use of namespaces, cgroups, and isolation rules [70].
- **Image Specification**: Defines the structure and format of container images, container image layers, and metadata, ensuring uniformity in how images are built and stored. The goal of this specification is to enable the creation of interoperable tools for building, transporting, and preparing a container image to run [71].
- **Distribution Specification**: Standardizes the API protocols for pushing, pulling, and distributing container images from the container registry [72].

[10] `https://man7.org/linux/man-pages/man7/namespaces.7.html`
[11] `https://opencontainers.org/`

## 3.4 Real-time systems

Although container-based virtualization brings several benefits, sharing of hardware and operating-system resources poses challenges for real-time predictability. Therefore, the fundamentals of real-time systems are introduced.

Real-time systems are computing systems in which the correctness of system behaviors depends not only on the logical computation results, but also on the physical time when these results are produced [73]. Real-time systems must respond within precise time constraints to environmental events. As a consequence, the correct behavior of these systems depends not only on the value of the computation but also on the time at which the results are produced [14]. Real-time systems are commonly classified into three categories based on the severity of consequences when a real-time task fails to meet its deadline [14]:

- **Hard real-time:** A real-time system is said to be hard if producing the results after its deadline may cause catastrophic consequences on the system under control. Examples include an airbag deployment system and industrial safety controllers.
- **Firm real-time:** A real-time system is said to be firm if occasional deadline misses degrade system performance (the result has no value) but do not cause total failure. Examples include machine vision quality inspection.
- **Soft real-time:** A real-time system is said to be soft if producing the results after its deadline still has some utility for the system, although causing a performance degradation. Examples include video streaming and online gaming.

Desirable features of real-time systems [14]:

- **Timeliness:** Results must be correct not only in terms of their computed values but also with respect to their timing. Consequently, the operating system must provide dedicated kernel mechanisms for time management and for handling tasks with explicit timing constraints and varying levels of criticality.
- **Predictability:** To achieve the desired level of performance, the system must be analyzable so that the consequences of any scheduling decision can be predicted. In safety-critical applications, all timing requirements should be verified offline before the system is deployed. If a task cannot

be guaranteed to meet its timing constraints, the system must indicate this in advance, allowing appropriate mitigation measures to be planned.
- **Robustness:** Real-time systems must remain operational under peak-load conditions and therefore need to be designed to handle all anticipated load scenarios. Overload management and adaptation mechanisms are essential for systems with varying resource demands and significant fluctuations in workload.

### Real-time tasks

Real-time systems are composed of real-time tasks, whose correctness depends on meeting specified timing constraints. Several models have been proposed in the literature to represent real-time tasks and capture their temporal behavior. Real-time tasks are commonly classified as: (i) Periodic, activated regularly at fixed intervals; (ii) Sporadic, activated irregularly but with a known minimum inter-arrival time, (iii) Aperiodic, activated irregularly with no minimum inter-arrival time.

In this thesis, we adopt the standard Liu and Layland [74] periodic task model. Periodic tasks commonly stem from functions such as sensor data acquisition, low-level actuation, control loops, and system monitoring. These activities must be executed repeatedly at well-defined frequencies, which are determined by the timing requirements of the application [14]. Set $\tau$ consists of independent periodic tasks $\{\tau_1, \tau_2, ...\}$. A periodic task $\tau_i$ is characterized by two parameters, the WCET $C_i$ and Period $T_i$:

- **WCET** $C_i$: is the maximum amount of time a task can take to execute on a given hardware platform under the worst possible conditions. It represents an upper bound on execution time. WCET can be obtained through static analysis, measurement-based techniques, or formal methods. However, as stated in Section 1, shared hardware resources hinder composable analysis and the computation of upper bounds on the key timing parameters on these systems [15]. This means that the tight WCET bounds become increasingly difficult to derive. The WCET is a crucial parameter for CPU reservation, e.g., for Hierarchical Constant Bandwidth Server (HCBS) [75] used for experimental evaluation in this thesis.
- **Period** $T_i$ - of a task is the fixed time interval between two consecutive

releases (activations) of the task. It defines how often the task executes.

Real-time tasks can be characterized (and the performance of a real-time system can be evaluated) by parameters [14], for example:

- **Response time**: represents the difference between the finishing time and the start time of a task.
- **Lateness**: represents the delay of a task with respect to its deadline.

These task characterizations can be used to construct performance metrics for evaluating real-time systems, and we apply them in our container orchestration framework REACT described in Paper B (presented in Chapter 7).

### 3.4.1 Real-time virtualization

Traditional virtualization approaches are primarily engineered to maximize resource utilization, often at the expense of temporal predictability. Although they deliver elasticity, flexibility, resiliency, and cost-effective operation, these technologies prioritize consolidation over isolation [76, 17]. In contrast, real-time systems require temporal predictability, which standard virtualization does not inherently provide [77]. Conventional virtualization approaches often introduce undesirable latencies and jitter, making them unsuitable for real-time applications. Thus, real-time virtualization faces challenges in supporting real-time applications [17].

#### Real-time hypervisor-based virtualization

Research and development on hypervisor-based virtualization for real-time applications are underway. For example:

- **Jailhouse[12]:** Linux-assisted partitioning hypervisor; designed to run real-time, safety-critical, or bare-metal workloads alongside Linux on multicore systems. Instead of virtualizing resources dynamically, Jailhouse splits the existing hardware into isolated partitions called cells, assigning dedicated CPUs, memory regions, and devices to each cell at runtime. Linux initially boots the system and then transitions into the root cell, after which Jailhouse takes control of the hardware and

[12] `https://github.com/siemens/jailhouse`

enforces strong spatial and temporal isolation among cells. Jailhouse performs no scheduling, does not overcommit resources, and virtualizes only what cannot be partitioned in hardware, enabling predictable, near bare-metal performance.

- **Bao[13]:** lightweight, open-source, bare-metal static partitioning hypervisor designed for embedded and mixed-criticality real-time systems. Its primary goal is to provide strong spatial and temporal isolation with minimal overhead. Bao statically assigns CPUs, memory, and I/O devices to virtual machines at boot time, maps virtual CPUs (vCPUs) one-to-one to physical CPUs, and uses direct interrupt injection and pass-through I/O, thereby avoiding the need for a runtime scheduler and reducing interference between workloads.
- **ACRN[14]:** open-source, bare-metal hypervisor developed by Intel, primarily targeting embedded, IoT, and automotive systems. It is designed to support mixed-criticality workloads by running a real-time or safety-critical operating system alongside a general-purpose operating system such as Linux on the same multicore platform.
- **Xen with Real-Time Deferrable Server (RTDS)[15]:** Xen's Real-Time Deferrable Server supports CPU resource reservations on a per-vCPU basis by assigning each virtual CPU a budget and a replenishment period. It evolved from the research project into the Xen mainline.
- **PikeOS[16]:** Separation-kernel Real-Time Operating System (RTOS) + hypervisor with strict time and space partitioning; widely used in avionics, rail, and automotive.

Real-time hypervisors achieve predictability by prioritizing isolation over consolidation, typically via static partitioning with no overcommitment (e.g., one vCPU per physical core, direct assignment of memory/devices), as in Jailhouse and Bao, thereby minimizing cross-VM interference. When sharing is unavoidable, they use time-partitioning or budget/period servers (e.g., Xen's RTDS) with admission control to bound CPU availability. They further ensure deterministic I/O by favoring device pass-through (VT-d/SR-IOV, System

[13] http://www.bao-project.org/
[14] https://projectacrn.org/
[15] https://wiki.xenproject.org/wiki/RTDS-Based-Scheduler
[16] https://www.sysgo.com/pikeos

Memory Management Unit (SMMU)) and Direct Memory Access (DMA) containment, thereby reducing emulation overhead and jitter. Finally, they control interrupt/timer latency with lean kernels and verified paths, and provide assurance through certification in safety-critical domains, aligning engineering mechanisms with formal guarantees of timing behavior.

### Real-time container-based virtualization

Real-time container-based virtualization is a less mature area of research as shown in [78]. Compared to hypervisor-based virtualization, achieving real-time guarantees with containers is more challenging due to weaker temporal isolation among co-located containers sharing the same hardware and operating system kernel. *The interference of workloads in containers is much more serious than that in virtual machines due to the weak isolation implementation of containers* [19].

Moreover, real-time performance depends on the operating system's real-time capabilities. Containers lack dedicated hardware resources, and inter-container isolation is often achieved through namespace isolation and resource limits in cgroups. These mechanisms are less effective than the complete hardware-based isolation provided by hypervisors, resulting in weaker temporal isolation. Overall, while container-based virtualization offers benefits such as improved resource utilization and faster deployment, it typically provides weaker temporal isolation than hypervisor-based virtualization. The unpredictability and limited knowledge of co-tenant workloads often lead to scenarios in which multiple workloads compete for limited shared resources. Such scenarios are often accompanied by the performance degradation of some workloads when a co-tenant workload [23].

There are three main directions to improve temporal characteristics of container-based virtualization [78]:

- **Real-time co-kernel**: A real-time co-kernel (often called a dual-kernel approach) is a design where a small, high-priority real-time core runs side-by-side with the standard Linux kernel and takes control of time-critical work - especially interrupt handling and real-time thread scheduling - so that real-time tasks can execute with more deterministic latency even when Linux is busy. Linux continues to run for non-critical services, but the co-kernel sits above it in priority and arbitrates access to

the CPU for real-time activities. Examples include Xenomai's[17] Cobalt co-kernel, and other classic dual-kernel extensions such as Real-Time Application Interface (RTAI)[18] (historically related to Xenomai's evolution). This approach is used, for example, in RT-CASEs [79] that aims to explore the benefits of combining container-based virtualization and hard real-time co-kernels. The work of Hofer et al. [80] demonstrates the feasibility of running containerized time-critical applications using Xenomai.

- **PREEMPT_RT**[19]: The real-time patch (PREEMPT_RT) improves the kernel's locking primitives to maximize preemptible sections, and thus it improves the real-time performance of the operating system and deployed containers. The advantage of the patch is that there is no need for special libraries or Application Programming Interface (API) needed by the application developers.
  In practice, industrial real-time container deployments most commonly build on a PREEMPT_RT-enabled Linux host, reflecting the broad availability and maturity of the PREEMPT_RT approach in production Linux ecosystems. This will increase further with the integration of PREEMPT_RT into the mainline kernel. For example, Goldschmidt et al. [81] present an architecture and performance measurements for a multi-purpose industrial controller in industrial automation using PREEMPT_RT. The feasibility of using container-based virtualization together with PREEMPT_RT patch in the context of vPAC is shown by Schönborn et al. [82].
- **Hierarchical-scheduling**: Inspired by a similar concept in the hypervisor-based virtualization, where a global scheduler assigns Central Processing Unit (CPU) time for the VMs (containers), the second layer scheduler schedules the individual tasks of the VM (container). This approach is based on the Hierarchical scheduling framework [83]. The idea is based on a compositional (hierarchical) real-time scheduling framework that lets a complex embedded system be built from independently analyzed components, while preserving timing guarantees when components are integrated.

[17] `https://xenomai.org`
[18] `https://www.rtai.org/`
[19] `https://wiki.linuxfoundation.org/realtime/start`

Additionally, there are efforts to explore convergence between container-based virtualization and hypervisor-based virtualization [84, 76] (similar to Kata Containers [20] in the security context). Containers are widely adopted in cloud platforms due to their lightweight virtualization model, whereas partitioning hypervisors are gaining traction in industrial environments because they provide the strong isolation properties required by safety standards. Building on these strengths, the concept of Partitioned Containers has been proposed to combine container flexibility with hypervisor-level isolation, enabling the use of cloud-native technologies in safety-critical and industrial contexts.

### 3.4.2 Performance interference and Resource adaptation in container-based virtualization

Container-based virtualization is susceptible to temporal disruptions caused by the utilization of shared resources, which could result in considerable degradation of the real-time performance. Co-location of containers on shared platforms may introduce timing disturbances that are due to the presence of non-partitionable or difficult-to-partition resources. When multiple workloads execute concurrently on a server, they compete for shared resources [85]. For example, caches, memory controllers, and translation look-aside buffers [86]. The workloads for such resources are referred to as inference sources. An example of the interference between containers is shown in our experiment in Figure 1.2.

The study by Garg et al. [18] indicates that containers also interfere with shared resources. The interference is due to the host operating system support required for containers and can be disruptive if the mix of containers hosted on a machine is determined solely by each container's resource usage. The study [41] claims that the shared host operating system kernel can degrade container performance and may lead to denial-of-service attacks.

In the literature, several studies propose preventive strategies to mitigate noisy-neighbor effects on shared processor resources, such as in [87]. Several works propose machine-learning-based solutions; for instance, [88] and [89] employ support vector machines and random forests, respectively [90].

Cgroups are used in the study by Al-Dhuraibi et al. [34], the authors present a system for elastic scaling of containers that adjusts memory and CPU ac-

[20] `https://katacontainers.io/`

cording to the workload. The resource adjustment strategy is based on static response-time thresholds. The study presents a simple strategy for vertically controlling resources. However, real-time requirements are not considered in this study.

The study by Rossi et al. [91] proposes reinforcement learning techniques to adapt at runtime the deployment of container-based applications through horizontal and vertical elasticity. The system adapts without manual tuning. Furthermore, both vertical and horizontal scaling (which can introduce performance penalties) are explored. The paper [86] presents a data-driven, machine-learning technique based on Gaussian Processes to build a predictive runtime model of the system's performance that can adapt to workload variability. Again, real-time is not within the scope of the work.

## 3.5 Container orchestration

Container orchestrators automate the deployment, scaling, networking, and lifecycle management of containerized applications across distributed computing environments. As applications are decomposed into multiple loosely coupled containers, manual coordination quickly becomes impractical, particularly in cloud architectures. Orchestrators such as Kubernetes address this complexity by abstracting the underlying infrastructure and providing declarative mechanisms to manage the desired system state, handle service discovery, balance workloads, and automatically recover from failures. By enabling portability, resilience, and efficient resource utilization, container orchestration has become a foundational technology for modern cloud architectures and large-scale distributed systems [92].

In literature, the list of responsibilities of a container orchestrator is as follows:

- **Provision and rollout containers:** Refers to a container orchestrator's ability to automatically create the required infrastructure resources and deploy containerized applications in a controlled, repeatable manner. It manages scheduling, configuration, and progressive rollout of containers (e.g., updates or scaling) to ensure availability and consistency.
- **Manage configuration and placement:** Describes a container orchestrator's capability to centrally define application settings (such as environment variables, secrets, and resource requirements) and intelligently

decide where containers run within a cluster. It places workloads on suitable nodes based on constraints and policies.

- **Assign and optimize resources:** refers to a container orchestrator's ability to allocate CPU, memory, storage, and network resources to containers according to defined requirements and limits. It continuously monitors usage and adjusts placement or scaling to maximize efficiency while preventing resource contention and performance degradation.
- **Ensure service availability:** refers to a container orchestrator's responsibility to keep applications continuously accessible despite failures or changing demand. It achieves this through mechanisms such as health checks, automatic restarts, replication, load balancing, and self-healing to minimize downtime.
- **Scale container up, down, or terminate containers:** Describes a container orchestrator's capability to automatically adjust the number of running containers based on workload demand or defined policies. It can start additional instances to handle increased load, reduce replicas when demand drops, or terminate unhealthy or unnecessary containers to optimize resource usage.
- **Load balancing:** Refers to a container orchestrator's ability to route incoming requests across multiple container instances to prevent overload and reduce latency.
- **Secure container-to-container communication:** Refers to a container orchestrator's ability to protect data exchanged between services running in different containers. It uses mechanisms such as network isolation, authentication, encryption, and fine-grained access policies to ensure that only authorized containers can communicate securely.

### 3.5.1 Container orchestrators

There are several container orchestrating tools available, for instance, Kubernetes[21], Docker Swarm[22] [93]. Kubernetes and Docker Swarm both provide solutions for orchestrating containerized applications, but they differ significantly in scope, complexity, and adoption. Kubernetes is a highly extensible, production-grade orchestration platform designed to manage complex,

[21] `https://kubernetes.io`
[22] `https://docs.docker.com/engine/swarm`

large-scale container deployments with advanced scheduling, self-healing, and declarative configuration capabilities. In contrast, Docker Swarm emphasizes simplicity and tight integration with the Docker ecosystem, offering an easier learning curve but fewer features and limited scalability. As a result, Docker Swarm is best suited for small-scale or experimental environments, while Kubernetes has emerged as the dominant standard for cloud-native and enterprise-level container orchestration.

### 3.5.2 Architecture of container orchestrator

The architecture of a container orchestrator, depicted in Figure 3.4, typically comprises several components, including the control plane, worker nodes, container runtime, and the underlying compute cluster[23], described as follows:

- **Control Plane:** The main component that makes global decisions, maintains the desired state of applications, performs scheduling containers onto compute nodes, monitors health and restarts failed components, and manages configuration and secrets.
  The control plane consists of the following components: (i) Launcher, responsible for issuing actions to worker nodes; (ii) State Monitor, which maintains the global state of worker nodes and jobs and triggers corrective actions when the observed state diverges from the desired one; (iii) Resource Management and Accounting, which tracks resource availability across the compute cluster and monitors resource usage for both scheduling and accounting purposes; (iv) Scheduler, which determines job placement on nodes with sufficient available resources and is also invoked during job migration events, such as node failures, or when the hosting node no longer satisfies the requirements; and (v) a User API, which accepts deployment requests, validates them, and invokes the scheduler when necessary [94, 95].
- **Worker Nodes:** It runs the actual containers and is managed by the control plane. A worker node is a compute node responsible for running and managing application workloads assigned by the control plane. Each worker hosts the container runtime to execute containers, an agent (e.g., node daemon) that communicates with the control plane and en-

[23] Kubernetes architecture: `https://kubernetes.io/docs/concepts/architecture/`

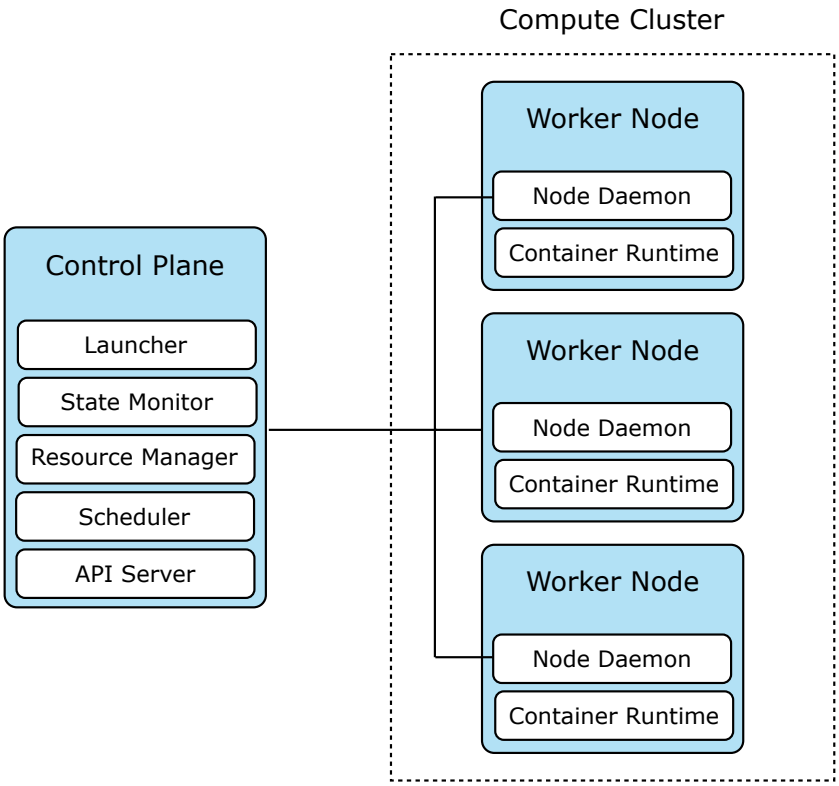



Figure 3.4: Container orchestrator architecture.

forces scheduling decisions, and supporting services such as networking and storage interfaces. The worker node reports its health and resource usage (CPU, memory, storage, and accelerators) to the control plane, executes lifecycle actions such as starting, stopping, or migrating containers, and ensures isolation, networking, and persistence for running workloads. Together, worker nodes provide the scalable execution layer of the cluster.

- **Container Runtime:** Software layer responsible for running containers on a node, handling tasks such as pulling container images, creating and starting containers, managing their lifecycle, and enforcing isola-

tion through operating-system features like namespaces and cgroups. It interacts with the underlying operating system kernel.
- **Compute Cluster:** A set of Worker Nodes, either on-premises or in the cloud, that run containerized applications.

### 3.5.3 Real-time container orchestration

Real-time container orchestrators extend traditional orchestration platforms by considering the timing constraints and real-time requirements of applications during deployment decisions. Instead of scheduling containers solely based on resource availability, these orchestrators place containers according to real-time requirements (following the CPU reservation, e.g., exploiting the multi-processor resource model (MPR) [96, 97]), ensuring predictable execution. By integrating real-time scheduling policies and resource isolation mechanisms, real-time orchestrators enable time-critical workloads to coexist with best-effort applications while maintaining real-time behavior.

Container orchestrators can be easily extended for use in fog computing [98, 99]. However, neither the orchestrators [92] nor their fog computing extensions support the deployment and maintenance of containers based on their real-time requirements. The challenge is to integrate the emerging real-time virtualization technology) with the orchestrating technology to enable matchmaking between real-time containers and compute nodes, accounting for the containers' real-time requirements and the compute nodes' real-time capabilities. Moreover, the orchestrators must account for the inherently low isolation levels of co-located containers.

While substantial research exists on real-time containers, resource isolation, and container orchestration, current approaches generally address these challenges independently. Existing orchestrators lack mechanisms for runtime adaptation based on observed interference and real-time performance degradation. Consequently, there remains a need for an autonomous orchestration framework capable of monitoring temporal behavior and dynamically adjusting resources to preserve real-time behavior.

# Chapter 4

# Research Results

This section provides an overview of the research results and contributions. It also outlines all the papers included in this thesis, specifying the individual contributions of each paper.

## 4.1 Results overview

This section provides an overview of the main results and the approaches by summarizing the contributions presented in the included publications. The purpose is to provide the reader with a consolidated understanding of the research outcomes without requiring a detailed review of each paper.

First, we address the main enablers, limitations, and open challenges of real-time containers. Second, we discuss an approach for dynamic resource allocation for real-time container-based virtualization. Third, we address container orchestration for real-time container-based virtualization.

### 4.1.1 Real-time container-based virtualization: main enablers, limitations, and open challenges

In Paper A (presented in Chapter 6), we provide a systematic literature survey on real-time containers. The survey describes methods and technologies employed and the main factors limiting the maturity of real-time container-based

virtualization. We categorize existing work into major research areas: (i) real-time patch-based, (ii) co-kernel based, (iii) hierarchical scheduling-based, and (iv) custom approaches, this is further detailed in section 3.4.1.

The survey reveals that existing approaches to real-time container-based virtualization primarily focus on CPU scheduling mechanisms, while orchestration support, communication determinism, and industrial-scale validation remain largely unexplored. The findings indicate that performance isolation is still a major obstacle preventing widespread adoption of containers in safety-critical systems. Although prior studies demonstrate that containerized real-time tasks can be executed across diverse hardware platforms and under concurrent workloads, the lack of strong isolation introduces uncertainties, making such approaches less appropriate for applications with stringent real-time requirements. To the best of our knowledge, this is the first systematic state of the art review of real-time container-based virtualization.

To support our arguments, in Paper D (presented in Chapter 9), we show interference among containers that act as noisy neighbors due to excessive LLC utilization. This indicates limited performance isolation between containers and highlights the challenge in real-time container-based virtualization. The illustration of the interference is shown in Figure 1.2.

These findings are important because they provide a comprehensive understanding of the current state of real-time container-based virtualization and evidence of its limitations. By systematically identifying research gaps and experimentally demonstrating the impact of shared-resource interference, the work highlights performance isolation as a fundamental challenge that must be addressed before containers can be confidently deployed in real-time systems.

### 4.1.2 Dynamic resource allocation for real-time containers

Static resource allocation may not be the efficient approach for container-based virtualization. Too many allocated resources lead to resource inefficiency, whereas too few can result in real-time performance degradation. To overcome this issue, we developed a hierarchical framework that continuously adapts CPU reservations based on system performance. Starting from an offline-derived resource allocation, the adaptation mechanism dynamically adjusts the allocated CPU based on real-time performance metrics to achieve the desired system performance. Furthermore, we extended dynamic resource allocation

for container-based virtualization by linking application-level performance, expressed as QoC, to resource management.

### Hierarchical Orchestration Framework for CPU resource adaptation

In Paper D and Paper F (presented in Chapter 9 and 11, respectively), we address timing disturbances in container-based virtualization and propose a hierarchical resource-orchestration framework for real-time containers to mitigate their effects. The framework comprises two phases: (i) an offline phase that determines the initial reservation of real-time containers (i.e., static resource allocation), and (ii) an online phase that complements the offline phase by dynamically readjusting resources when unexpected changes arise. The overall resource control is organized into a three-level (i.e., Container-, Node-, Cluster-level) hierarchy; this is shown in Figure 4.1. The hierarchy preserves a separation of concerns and enables the implementation of distinct policies at each control level. The individual controllers within the hierarchy and their respective responsibilities are described below.

- **Container-level Controller:** The primary goal of the container-level controller is to release unused resources by reducing the CPU budget of a respective container. This lowers container utilization and makes resources available for other co-located containers. A resource-release policy determines when and how many resources should be freed.
- **Node-level Controller:** The node-level controller aims to preserve the real-time properties of tasks by utilizing available unallocated computing resources and allocating them to containers that require additional resources. This is achieved by increasing their CPU reservation.
- **Cluster-level Controller:** The cluster-level controller monitors all nodes in the cluster and initiates container migrations when real-time performance requirements cannot be satisfied within a computing node. With a global view of the system, it can migrate containers to less loaded nodes. Migration is triggered when the node-level controller determines that local resource adjustments are insufficient to meet real-time requirements.

We implement the framework on a hierarchical real-time scheduling framework for Linux [75] HCBS introduced by Abeni et al., which relies on statically defined CPU reservations, where each container is assigned a fixed CPU

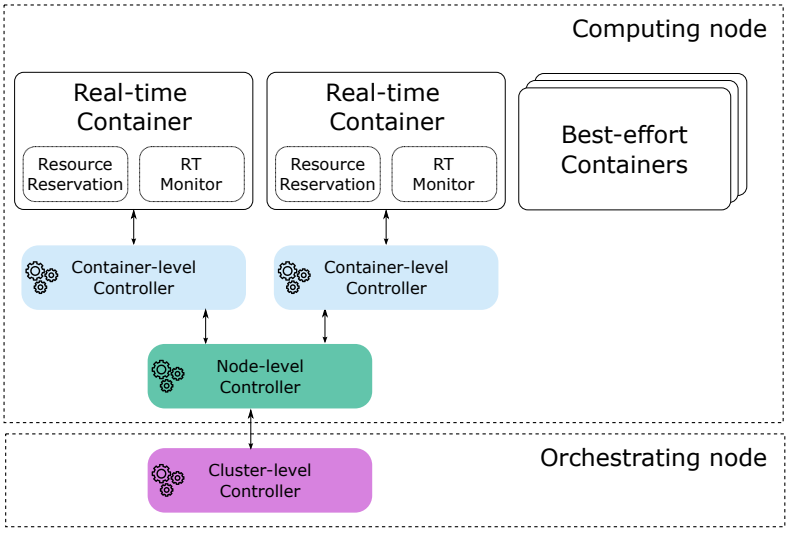



Figure 4.1: Overview of the Hierarchical Orchestration Framework.

budget over a fixed period. To overcome the static resource allocation, we extended HCBS with a runtime adaptation mechanism that dynamically adjusts CPU reservations based on the observed performance of real-time containers. Thus, containers can respond to workload variations and timing disturbances while maintaining the required real-time performance and improving resource utilization.

Our experiments illustrate the impact of performance interference among containers, resource control for individual containers, dynamic resource redistribution across containers, and container migration when total available resources are insufficient to satisfy the real-time requirements of containerized applications.

The importance of these findings lies in demonstrating that real-time container-based virtualization can move beyond static resource provisioning and adapt to changing conditions at runtime. By introducing a hierarchical orchestration framework that combines local resource adaptation with node-level resource redistribution and cluster-level container migration, the work provides a practical mechanism for maintaining real-time behavior while improving resource utilization. The results show that timing disturbances can

be mitigated through coordinated resource management.

### QoC-driven adaptive control of container resources

In Paper C (presented in Chapter 8), we linked the performance of control applications (i.e., QoC) with container resource adaptations. QoC refers to how well a control application achieves its desired objectives. It measures how effectively the system regulates, tracks, or stabilizes a process despite disturbances, noise, or system uncertainties. We show how to adjust the execution rate of the control application as system dynamics change while simultaneously adjusting the container's CPU resource reservation in HCBS. This approach reduces response errors without wasting CPU and treats container resources as actuable in a closed loop.

The Figure-4.2 illustrates a virtualized control application operating under varying levels of disturbance. In the region marked no error, the system experiences negligible perturbation, and therefore the control algorithm does not need to be executed frequently. Consequently, the container can operate with a lower CPU reservation. In contrast, in the region marked error, the system is subject to higher perturbation levels, requiring the control algorithm to run more frequently to maintain the desired performance. As a result, a higher CPU reservation must be allocated to the container to support the increased computational demand. As an example control application, we used an inverted pendulum simulator equipped with a virtual camera, depicted in Figure 4.3. The camera continuously streams images of the pendulum to the controller, which uses OpenCV-based image processing to estimate the pendulum's position and angle and compute the appropriate control actions to keep it balanced. Since real-time image processing is computationally demanding, the application requires sufficient CPU resources to maintain the desired control performance.

We propose a QoC-driven adaptation mechanism, overviewed in the Figure 4.4, which consists of four main components: (i) a real-time container hosting the control application, (ii) the control application, which is responsible for controlling the plant, (iii) the QoC controller, which dynamically adjusts the resource reservations allocated to the real-time container, and (iv) the plant, which represents the target system being controlled by the control application. The QoC controller monitors the control performance and adapts the

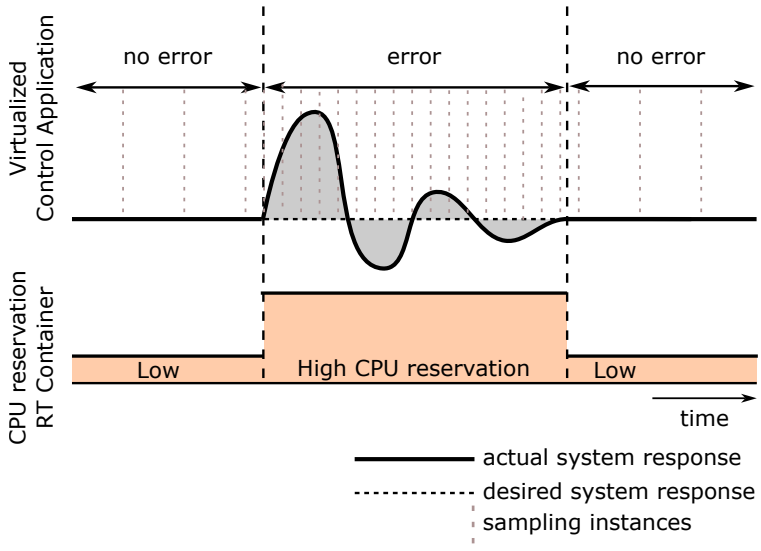



Figure 4.2: Illustration of the impact of virtualized control application QoC on resource reservation and control period.

container's resource allocation to ensure that the required quality of control is maintained.

The importance of these findings is that they establish a direct link between application-level control performance and resource management in container-based virtualization. Rather than adapting CPU reservations only based on real-time performance (as shown in the previous section), the proposed QoC-driven approach adjusts resources according to the actual needs of the controlled process. This enables more efficient resource usage during low-demand operating conditions while ensuring that sufficient computational resources are available when control performance becomes critical.

### 4.1.3 Enabling real-time container orchestration

Container-based virtualization is often used in conjunction with container orchestration. While containers provide application isolation and resource management, orchestration automates the deployment, scaling, and management

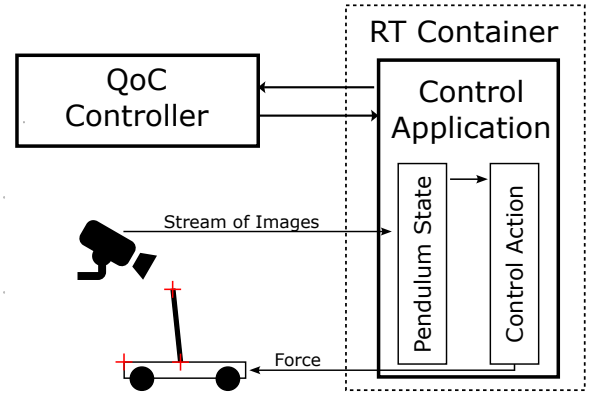



Figure 4.3: Overview of the inverted pendulum control application. A virtual camera captures images of the pendulum and streams them to the control application, which uses image processing to estimate the pendulum state (position and angle) and compute the control action required to keep the pendulum balanced.

of containers across distributed computing environments. With the emergence of real-time for container-based virtualization, there is an increasing need to enhance orchestration with real-time awareness. Such capabilities include support for real-time requirements, resource reservation, and monitoring and management of real-time performance to ensure real-time behavior.

In Paper B (presented in Chapter 7), we present a REACT framework for orchestrating real-time containers using HCBS by incorporating real-time requirements into the container-placement process. We define scheduling policies, runtime monitoring mechanisms, and performance metrics, along with an implementation that supports the execution of real-time containers. We present the architecture and design of a real-time container orchestrator based on Kubernetes. Moreover, we define metrics for evaluating real-time performance of containers and describe an initial model for allocating a mixture of real-time and non-real-time containers. This demonstrates that real-time-aware orchestration is feasible and enables schedulability-aware placement in containerized environments

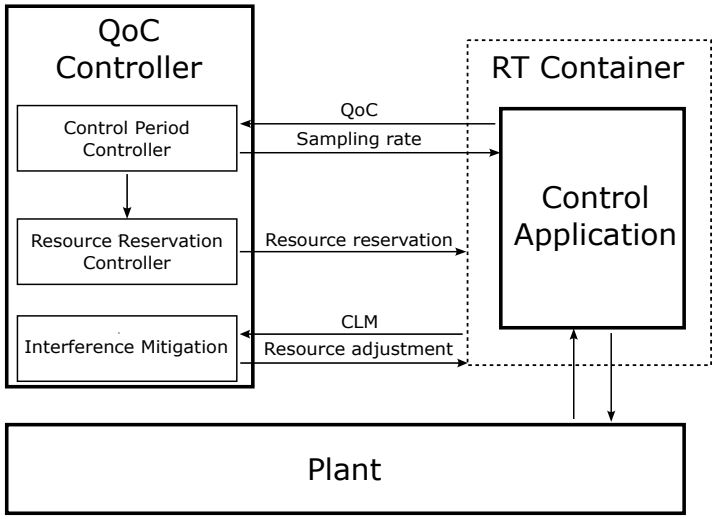



Figure 4.4: Architecture of the QoC-driven adaptation mechanism for real-time container-based virtualization.

### The proposed real-time orchestration extension

We follow the Kubernetes master-worker architecture, with the master node maintaining the global cluster state and the worker nodes hosting and executing containers. The proposed architecture is presented in Figure 4.5.

The master node (serving as real-time orchestrator) receives user-defined container deployment specifications extended with real-time interfaces, which define CPU reservations and task annotations specifying tasks periodicities and relative deadlines. Upon receiving a deployment request, the orchestrator first performs admission control to verify whether the real-time containers can be feasibly deployed in the cluster. It then executes the scheduling process to determine the most suitable nodes for container deployment. The orchestrator continuously monitors the performance of worker nodes and running containers to ensure that real-time requirements are satisfied. The collected monitoring metrics are used to support further admission control, scheduling decisions. To support scheduling decisions, the orchestrator collects two types of metrics:

- **Operating System Level Metrics (OSLM)** (shown by Jägemar et al.

in [100]) describes the execution environment and indicates whether a node is suitable for real-time workloads. These include CPU utilization, interrupt activity, non-preemptable sections, I/O request rates, and data read/write rates.

- **Container Level Metrics (CLM)**, which we introduced, is a set of metrics to monitor and characterize real-time behavior within containers. It includes the total number of deadline misses, the maximum observed lateness, and the maximum response time for each container.

The master node consists of three main components. (i) The Real-time Resource Monitor stores OSLM and CLM collected from worker nodes and makes them available to the admission controller and scheduler. (ii) The Real-time Admission Control checks whether a node has sufficient resources and verifies schedulability using utilization-based tests. (iii) The Real-time Container Scheduler then selects a feasible node based on monitored OSLM and CLM metrics.

The worker node, shown in Figure 4.6, continuously monitors and reports OSLM and CLM to the master node. It is also responsible for deploying containers with the appropriate resource allocation, i.e., CPU reservation and period settings.

The importance of these findings lies in demonstrating that real-time requirements can be integrated into container orchestration. By introducing real-time-aware admission control, scheduling, monitoring, and performance metrics, the proposed framework enables informed placement decisions based not only on resource availability but also on timing constraints and real-time performance. The findings show that schedulability-aware orchestration is feasible, providing a foundation for deploying and managing real-time containerized applications at scale while preserving their timing requirements.

#### Container orchestration in uncertain environments

In Paper E (presented in Chapter 10), we consider Container orchestration in uncertain environments, where (e.g., real-time) performance is not always a priori predictable. This may occur due to hardware-dependent susceptibility to the noisy-neighbor problem, or because interfering workloads are already present on the worker node. In such cases, the orchestrator must explore multiple deployment options to identify the node that provides the best real-time per-

formance. To address this, we employ a Contextual Contextual Multi Armed Bandit (CMAB) framework, enabling the orchestrator to learn and adaptively select optimal deployment decisions over time. Although the work primarily targets sustainable energy utilization as the main optimization objective, the proposed framework can be extended to incorporate real-time performance metrics (e.g., CLM), enabling deployment decisions that jointly consider energy efficiency and timing requirements.

CMAB is a simple yet effective framework for sequential decision-making under uncertainty [101]. At each decision step (e.g., a container is experiencing a performance degradation), the algorithm selects one of several possible actions (choosing a target node for container migration) and receives a reward based on that choice (new value of CLM). Since the rewards of alternative actions remain unknown, the algorithm must balance exploration (trying new actions to gather information) and exploitation (selecting the action currently expected to yield the highest reward). The overview is presented in Figure. 4.7.

The importance of these findings is that they address a practical challenge in container orchestration: making deployment decisions when performance cannot be accurately predicted in advance. By leveraging a CMAB framework, the orchestrator can learn from previous deployment outcomes and continuously improve its placement decisions in dynamic and uncertain environments.

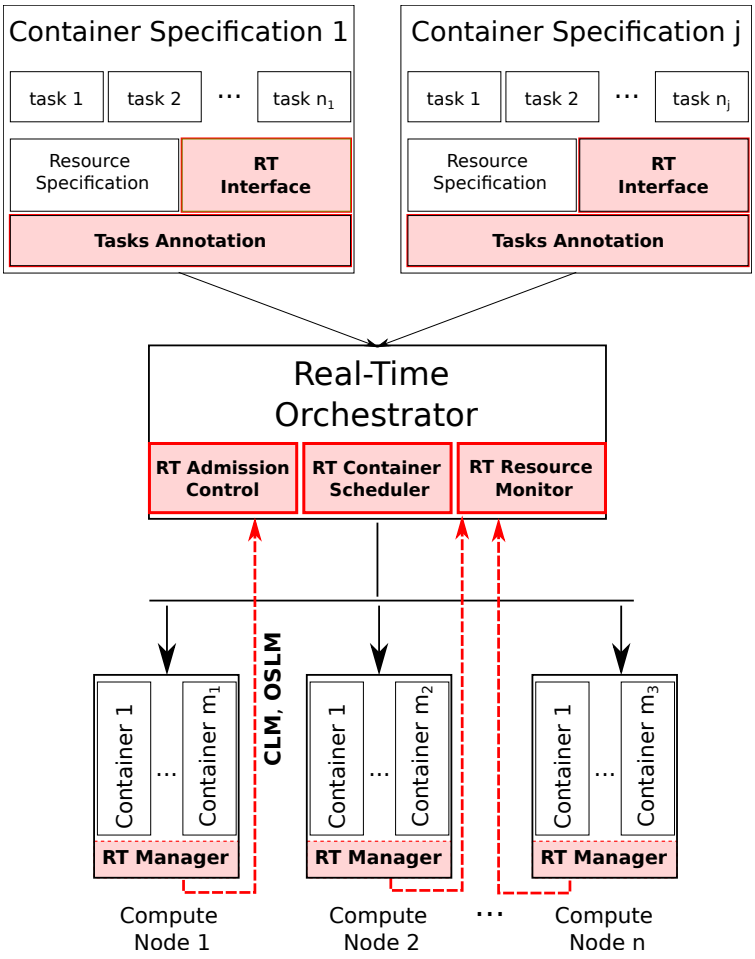



Figure 4.5: High-level architecture of the real-time container-based orchestration Framework.

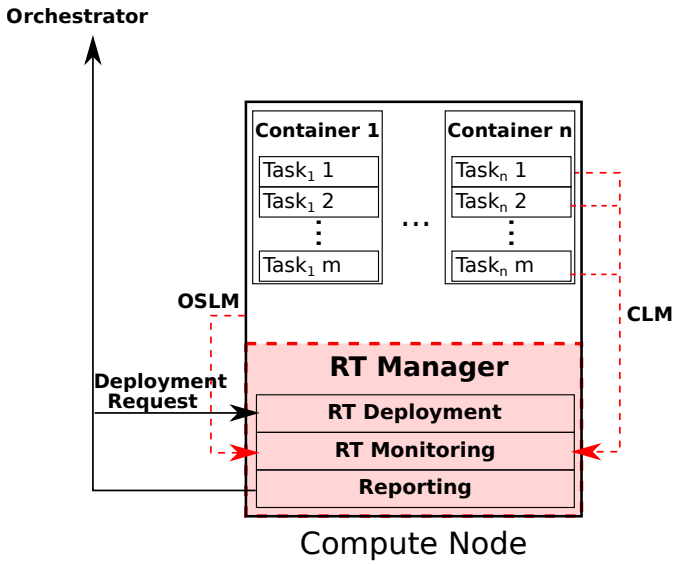



Figure 4.6: Overview of a worker node extended with the RT manager and CLM and OSLM.

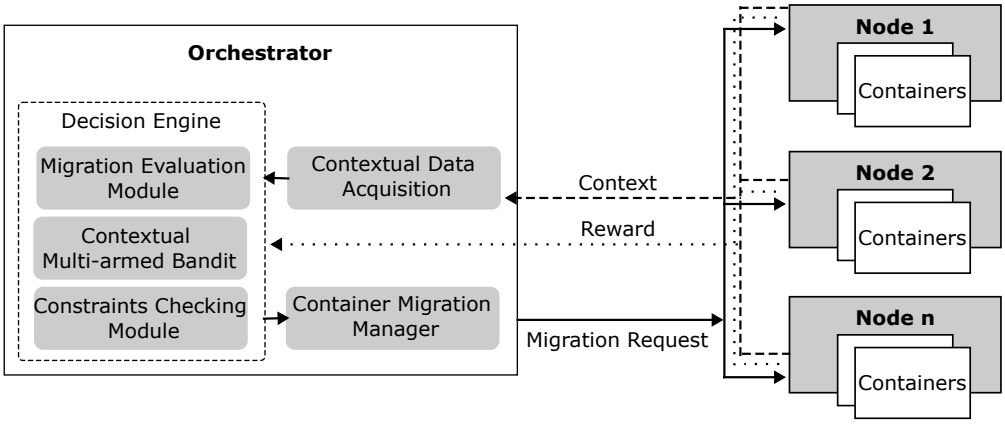



Figure 4.7: Overview of the orchestrator.

## 4.2 Thesis Contributions

This subsection lists the contributions $C_1$ to $C_4$ included in the thesis and the papers that constitute these contributions.

**$C_1$: Systematic review of real-time container-based virtualization.**

We provide the first systematic and structured survey of research on real-time container-based virtualization, identifying key architectural approaches, evaluating their effectiveness, and exposing critical gaps, particularly in orchestration, communication support, and industrial validation, that hinder adoption in real-time domains. We organize existing solutions into categories (i.e., real-time patch-based, co-kernel-based, hierarchical scheduling, and custom approaches), offering a clear overview of research directions. The survey explains how different techniques contribute to achieving time predictability in containerized environments. Finally, the survey highlights research gaps, including a lack of orchestration tools, real-time communication support, and insufficient validation for industrial adoption. At the time this review was conducted, no systematic overview of real-time containers was available.

**$C_2$: Runtime resource adaptation for container-based virtualization.**

We introduce a novel feedback-driven runtime resource adaptation mechanism that dynamically adjusts CPU reservations of real-time containers, improving timing predictability under workload variability and mitigating performance interference in multi-tenant environments. The approach enables containers to respond to workload variability and performance interference, mitigating temporal instability introduced by shared resource contention. By integrating monitoring and feedback-driven control, the mechanism improves timing predictability while maintaining efficient resource utilization in multi-tenant environments.

**$C_3$: Real-time aware container orchestration framework.**

We propose a real-time-aware container orchestration framework that extends conventional platforms with schedulability-driven placement, admission control, and runtime monitoring, enabling predictable execution of mixed real-

time and best-effort workloads. The framework incorporates real-time performance metrics to continuously monitor the system's real-time behavior. By integrating resource reservation and monitoring into orchestration decisions, the framework ensures predictable execution and improved system-level performance in distributed environments.

**$C_4$: QoC-driven adaptive control of container resources.**

We introduce a QoC-driven resource management approach that tightly couples application-level performance with infrastructure-level resource allocation. The method dynamically adjusts both the execution characteristics of control applications (e.g., sampling rates) and the CPU reservations of their hosting containers based on real-time QoC feedback. This closed-loop adaptation enables the system to respond effectively to disturbances, improving control performance while avoiding resource overprovisioning. The approach demonstrates how control-theoretic concepts can be leveraged to guide resource management in real-time containerized systems.

Together, these contributions form a coherent framework addressing real-time container execution across three layers: (i) system-level orchestration, (ii) runtime resource adaptation, and (iii) application-level control. This integrated approach improves predictable execution and resource utilization in containerized environments despite workload variability and resource interference.

## 4.3 Mapping of contributions, research questions, and publications

This section presents the relationships between the research questions, contributions, and publications. Table 4.1 maps the contributions to the research questions. Table 4.2 provides a mapping between the publications and the contributions. $RQ_1$ is addressed through the systematic analysis of existing real-time container technologies, identifying the approaches to enable real-time containers and limitations affecting temporal predictability. $RQ_2$ is addressed through the development of adaptive resource management mechanisms capable of mitigating interference and workload variability at runtime. $RQ_3$ is

addressed through the design of orchestration mechanisms that integrate real-time requirements into deployment and scheduling decisions.

Table 4.1: Contributions $C_1$ through $C_4$ address research questions $RQ_1$ through $RQ_3$.

| | **$RQ_1$** | **$RQ_2$** | **$RQ_3$** |
|---|---|---|---|
| **$C_1$** | X | | |
| **$C_2$** | | X | |
| **$C_3$** | | | X |
| **$C_4$** | | X | |

Table 4.2: Papers A through F address contributions $C_1$ through $C_4$.

| | **$C_1$** | **$C_2$** | **$C_3$** | **$C_4$** |
|---|---|---|---|---|
| **Paper A** | X | | | |
| **Paper B** | | X | X | |
| **Paper C** | | X | | X |
| **Paper D** | | X | X | |
| **Paper E** | | | X | |
| **Paper F** | | X | X | |

## 4.4 Paper contributions

In this section, the contributions of each research paper included in the thesis are outlined. Furthermore, the individual contributions of the author to each paper are explicitly described.

### 4.4.1 Personal contributions

The personal contributions in the papers included in this thesis are summarized based on CRediT[1] (Contributor Roles Taxonomy) definition in Table 4.3.

[1] Available at `https://credit.niso.org/`

Table 4.3: Personal contributions using CRediT taxonomy.

| Role | Paper A | Paper B | Paper C | Paper D | Paper E | Paper F |
|---|---|---|---|---|---|---|
| Conceptualization | X | X | X | X | X | X |
| Methodology | X | X | X | X | X | X |
| Software | | X | X | X | X | X |
| Validation | | X | X | X | X | X |
| Formal analysis | | X | X | X | | X |
| Investigation | X | X | X | X | X | X |
| Resources | | | | | | |
| Data curation | X | X | X | X | X | X |
| Writing—review & editing | | | | | | |
| Visualization | X | X | X | X | X | X |
| Supervision | | | | | | |
| Project administration | | | | | | |
| Funding acquisition | | | | | | |

### 4.4.2 Included papers

In this section, we provide an overview of the research papers included in this thesis. Each paper is briefly summarized to highlight its main contributions, followed by information about the respective publication venue. The Figure 4.8 provides an overview of the included papers. Paper A identifies key limitations of container-based virtualization and orchestration. Building on these findings, the work then branches into two directions: container orchestration in paper B and paper E, and resource adaptation mechanisms in paper C and paper F. Finally, paper D connects these two paths by jointly considering resource adaptation and orchestration.

#### Paper A

**Title:** Real-Time Containers: A Survey [78]
**Authors:** *Václav Struhár*, Moris Behnam, Mohammad Ashjaei, Alessandro V. Papadopoulos
**Published in:** In the 2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020)

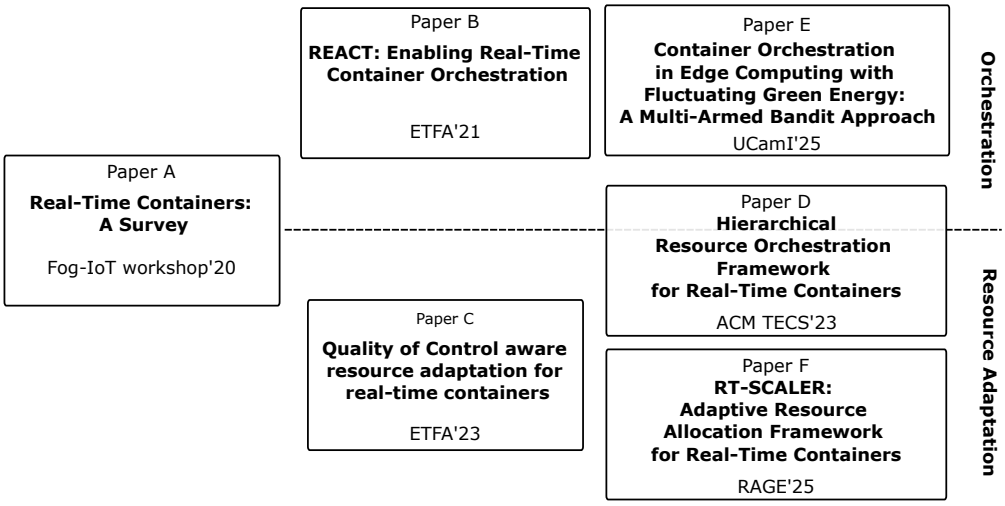



Figure 4.8: Overview of the included papers.

**Abstract:** Container-based virtualization has gained significant importance in the deployment of software applications in cloud-based environments. The technology fully relies on operating system features and does not require a virtualization layer (hypervisor) that introduces performance degradation. container-based virtualization allows co-locating multiple isolated containers on a single computation node as well as decomposing an application into multiple containers distributed among several hosts (e.g., in the fog computing layer). Such technology seems very promising in other domains as well, e.g., in industrial automation, automotive, and aviation industries, where mixed criticality containerized applications from various vendors can be co-located on shared resources. However, such industrial domains often require real-time behavior (i.e, a capability to meet predefined deadlines). These capabilities are not fully supported by container-based virtualization yet. In this work, we provide a systematic literature survey study that summarizes the effort of the research community on bringing real-time properties in container-based virtualization. We categorize existing work into main research areas and identify possible immature points of the technology.

**Paper contributions:**

- **Systematic literature survey:** The paper provides a structured review of research on real-time containers.
- **Classification of real-time approaches:** The paper categorizes existing solutions into key approaches, including real-time kernel patches, co-kernel solutions, hierarchical scheduling, and custom methods, offering an overview of the research field.
- **Identification of challenges and limitations:** The paper outlines key shortcomings of current solutions, such as a lack of real-time communication support, insufficient tooling, and limited industrial maturity. It points out open issues and areas that require further research to enable broader adoption of real-time containers.

**Paper B**

**Title:** REACT: Enabling real-time container orchestration [102]
**Authors:** *Václav Struhár*, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos
**Published in:** In the 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2021)
**Abstract:** Fog and edge computing offer the flexibility and decentralized architecture benefits of cloud computing without suffering from the latency issues inherent in the cloud. This makes fog computing very attractive in real-time and safety-critical applications, especially if combined with container-based technologies. Whereas different orchestration systems are available to manage the container placement based on their resource demand, no orchestration system considers real-time requirements for containerized applications. In this paper, we present the architecture and design of a real-time container orchestrator based on Kubernetes. Moreover, this paper defines metrics for the performance evaluation of real-time containers and describes an initial model for allocating a mixture of real-time and non-real-time containers. We present an initial implementation of our real-time container extension and evaluate its feasibility on Linux-based systems.

**Paper contributions:**

- **Real-time Container orchestration architecture:** The paper proposes a novel orchestration framework (based on Kubernetes) that explicitly considers real-time requirements for containerized applications.

- **Support for deployment and runtime adaptation:** The paper enables both placement and dynamic management of real-time and best-effort (non-real-time) containers while respecting timing constraints.
- **Definition of performance metrics:** The paper introduces container-level and system-level metrics to monitor timing behavior, such as deadline misses, lateness, and response time.
- **Real-time-aware scheduling and admission control:** The paper develops mechanisms to assign containers to nodes based on resource availability and schedulability analysis, preventing system overload.

**Paper C**

**Title:** Resource Adaptation for Real-Time Containers Considering Quality of Control containers [103]
**Authors:** *Václav Struhár*, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos, Silviu S. Craciunas
**Published in:** In the 28th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2023)
**Abstract:**

In industrial control systems, controllers are designed with fixed timing constraints, which limit their ability to adjust to changing system dynamics. This leads to suboptimal performance and response deviations. One approach to address this issue is to adjust the execution rate of the controller dynamically, depending on the system's current state. However, dynamically changing the execution rate of a virtualized controller is not straightforward, as the resource reservation for the real-time container hosting the controller must be updated accordingly. Therefore, this study aims to develop a dynamic execution rate control mechanism for real-time virtualized controllers that can adjust the controller's execution rate in response to changing system dynamics. This paper provides a method for the dynamic adjustment of controller properties (execution rates) and dynamic resource reservation of real-time containers (that utilizes Hierarchical Scheduling Framework).

**Paper contributions:**

- **Quality of Control aware resource adaptation mechanism:** The paper proposes a method that uses QoC metrics to dynamically guide resource allocation decisions for real-time containers.

- **Joint adaptation of timing and real-time container resource allocation:** The paper introduces a mechanism that simultaneously adjusts control application timing (e.g., sampling periods) and container CPU reservations of a real-time container to better reflect system needs.
- **Runtime adaptability to disturbances and workload changes:** The paper enables the system to dynamically react to disturbances by increasing or decreasing resource allocation based on observed QoC.
- **Experimental validation with control applications:** The paper validates the approach through experiments showing improved control performance QoC while optimizing resource utilization.

### Paper D

**Title:** Hierarchical Resource Orchestration Framework for Real-Time Containers [104]
**Authors:** *Václav Struhár*, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos
**Published in:** In the ACM Transactions on Embedded Computing Systems (TECS 2023)
**Abstract:** Container-based virtualization is a promising deployment model in fog and edge computing applications because it allows a seamless co-existence of virtualized applications in a heterogeneous environment without introducing significant overhead. Certain application domains (e.g., industrial automation, automotive, or aerospace) mandate that applications exhibit a certain degree of temporal predictability. Container-based virtualization cannot be easily used for such applications since the technology is not designed to support real-time properties and handle temporal disturbances. This paper proposes a framework consisting of a static offline and a dynamic online phase for resource allocation and adaptive re-dimensioning of real-time containers. In the offline phase, the optimal initial deployment and dimensioning of containers are decided based on ideal system models. Additionally, in order to adapt to dynamic variations caused by changing workloads or interference, the online phase adapts the CPU usage and limits of real-time containers at runtime in order to improve the real-time behavior of the real-time containerized applications while optimizing resource usage. We implement the framework in a real Linux-based system and show through a series of experiments that the proposed framework is able

to adjust and re-distribute computing resources between containers in order to improve the real-time behavior of containerized applications in the presence of temporal disturbances while optimizing resource usage.

**Paper contributions:**

- **Enables runtime resource adaptation**: Dynamically adjusts CPU budgets and execution periods of containers during operation to cope with changing workloads and system conditions.
- **Combines offline and online resource management:** Establishes an initial allocation and dimensioning of resources at design time, complemented by runtime adaptation to handle deviations from ideal assumptions.
- **Mitigates performance interference:** Continuously redistributes resources among co-located containers to reduce the impact of shared resource contention and timing disturbances
- **Uses a hierarchical control mechanism:** Introduces multi-level controllers (container-, node-, and cluster-level) to locally and globally coordinate resource allocation decisions.
- **Supports scalability through orchestration and migration:** Adapts not only resource allocations but also container placement across nodes when local resources are insufficient.

### Paper E

**Title:** Container orchestration in Edge Computing with Fluctuating Green Energy: A Multi-Armed Bandit Approach [105]
**Authors:** *Václav Struhár*, Alessandro V. Papadopoulos, Inmaculada Ayala, Mercedes Amor, Lidia Fuentes
**Published in:** In the 17th International Conference on Ubiquitous Computing and Ambient Intelligence (UCamI 2025)
**Abstract:** Sustainable edge computing demands intelligent scheduling of containerized workloads to exploit intermittently available renewable energy at geographically distributed sites. This work introduces a Contextual Multi-Armed Bandit (CMAB) framework for green-aware Container orchestration, leveraging real-time context, such as energy availability and resource utilization, via linear CMAB algorithms. A Python-based simulator models realistic solar and

wind dynamics across regions. Compared to optimal, random, and naïve baselines, our CMAB scheduler improves green-energy utilization by up to 30% and cuts brown-energy use by 20%, while maintaining application performance guarantees. These findings underscore the potential of learning-based methods for advancing energy-efficient and sustainable edge infrastructures.

**Paper contributions:**

- **Container scheduler formulation as Contextual Multi-Armed Bandit problem:** The paper models container placement in edge computing as a Contextual Multi-Armed Bandit problem, combining energy, QoS, and resource aspects (this may include real-time aspects) into a framework.
- **Adaptive Learning Scheduler:** Proposes a dynamic scheduling algorithm that learns over time to favour nodes with more green energy while still exploring alternatives.
- **Realistic Simulation Environment:** Develops a Python-based simulator with real-world solar and wind patterns and heterogeneous edge nodes for evaluation.
- **Quantified Sustainability Gains:** Achieves up to 30% higher green energy use and 20% reduction in brown energy consumption, while maintaining Quality of Service (QoS).

### Paper F

**Title:** RT-SCALER: Adaptive Resource Allocation Framework for Real-Time Containers [104]
**Authors:** *Václav Struhár*, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos
**Published in:** In the 1st International Workshop on Real-time and intelliGent Edge computing (RAGE 2022)
**Abstract:** Container-based virtualization has emerged as an advantageous deployment model in fog computing platforms since it enables the seamless colocation of applications in a heterogeneous environment with minimal overhead. For some application domains requiring a certain degree of predictability in the time domain (e.g., industrial automation), the adoption of container-based virtualization is not straightforward since the technology is not built to support real-time properties.

In this paper, we propose RT-SCALER, which is a framework for adaptive resource allocation and dimensioning for real-time containers. RT-SCALER dynamically adapts the resource reservation of real-time enabled containers in order to improve the temporal predictability of the real-time applications running within the containers. We discuss the high-level orchestration approach, relating the different control levels, and give some practical insights into node-level container adaptation.

**Paper contributions:**

- **Proposal of RT-SCALER framework:** The paper introduces RT-SCALER, a framework for adaptive resource allocation of real-time containers. Its goal is to improve the temporal predictability of real-time applications running inside containers by dynamically adapting their resource reservations.
- **Two-phase resource adaptation:** The paper defines a two-phase approach: an offline phase, where initial real-time parameters for containers are computed, and an online phase, where these parameters are adapted at runtime in response to workload changes, interference, or performance degradation.
- **Integration of real-time container resource reservation:** The paper builds on the HCBS [75] approach and considers real-time container interfaces defined as CPU budget and period parameters.
- **Runtime adaptation of CPU budget:** The paper demonstrates how the CPU budget of a real-time container can be adapted online based on measured response times of tasks inside containers. This allows the system to react to workload changes and maintain real-time performance closer to a desired target.

### 4.4.3 Not Included Papers

Additional papers not included in the doctoral thesis:

1. Shaik Mohammed Salman, *Václav Struhár*, Alessandro V. Papadopoulos, Moris Behnam, Thomas Nolte, "*Fogification of industrial robotic systems: research challenges*", In the 1st Workshop on Fog Computing and the IoT (Fog-IoT 2019) [106].

2. Shaik Mohammed Salman, *Václav Struhár*, Zeinab Bakhshi, Van-Lan Dao, Nitin Desai, Alessandro V. Papadopoulos, Thomas Nolte, "*Enabling Fog-based Industrial Robotics Systems*", In the 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2021) [107].

3. Mirgita Frasheri, *Václav Struhár*, Alessandro V. Papadopoulos, Aida Čaušević, "*Ethics of Autonomous Collective Decision-Making:the CAESAR Method*", In the Science and Engineering Ethics Journal (2022) [108].

4. *Václav Struhár*, Mohammad Ashjaei, Moris Behnam, Silviu S. Craciunas, Alessandro V. Papadopoulos, "*DART: Dynamic bandwidth distribution framework for virtualized software defined networks*", In the 45th Annual Conference of the IEEE Industrial Electronics Society (IECON 2019) [109].

# Chapter 5

# Conclusion

This section summarizes the main findings and contributions of the thesis. It briefly discusses the results and their implications, and outlines potential directions for future work and further improvements.

## 5.1 Conclusion and summary

This thesis investigated the challenges of real-time container-based virtualization. The findings show that performance isolation and shared-resource interference in container-based virtualization remain important obstacles to their adoption in real-time systems. To address these challenges, the thesis proposed a set of resource management and orchestration mechanisms spanning multiple levels of the system. These include a hierarchical resource orchestration framework for real-time containers, QoC-driven CPU management, real-time-aware orchestration, and learning-based deployment strategies for uncertain environments.

From a high-level perspective, this thesis presents methods for dynamically reserving CPU resources based on runtime monitoring of real-time tasks executing in containers. The thesis begins with a systematic literature survey that classifies existing approaches and identifies key limitations, open challenges, and research gaps in real-time container-based virtualization. Motivated by the survey findings, particularly the reported concerns regarding real-time pre-

dictability and performance isolation, the thesis develops a CPU resource-adaptation mechanism to improve the real-time behavior of container-based virtualization. It further extends the architecture and design of a container orchestrator to support explicit resource reservation and real-time-aware management.

In addition, the thesis proposes a QoC-driven resource adaptation mechanism for container-based virtualization, where application-level control performance is explicitly connected to dynamic resource management. Thus, ensuring that computational resources are adjusted to maintain the required level of QoC. Finally, we address uncertainty in deployment decisions using a multi-armed bandit approach, which is particularly useful when the (real-time) performance of a container cannot be accurately estimated before deployment.

## 5.2 Discussion and future work

We have published the systematic literature review on real-time containers in which we analyzed and categorized existing approaches for enabling real-time behavior in container-based virtualization environments. The study systematically classified techniques addressing timing predictability, resource isolation, and real-time scheduling support using different technologies. At the time of writing, this work was the first systematic study to consolidate and structure the knowledge on real-time containers, providing an overview of the state of the art and identifying key research directions and open challenges in the field. The work was extended and complemented in the journal paper [45] by another research group, who identified a similar categorization and research challenges. The presence of another research group working on a similar concept and reaching the same categorization further validates our approach. The survey has received more than 100 citations.

Based on both the systematic literature review and our experimental evaluation of container performance, we identified the noisy-neighbor problem as a significant challenge for real-time containers. Our experiment showed that a container's performance, particularly for real-time computation, can be significantly affected by the presence of co-located containers competing for shared resources (even in cases of simple matrix multiplication). To address this issue, we proposed resource adaptation mechanisms, focusing primarily on CPU resources. Specifically, we monitor the runtime performance of real-time tasks

executing inside the container (we introduced CLM metrics) and, based on these observations, dynamically adjust the allocated CPU resources. This adaptive approach aims to mitigate interference effects and stabilize the real-time behavior of real-time workloads despite co-location with other containers. We have enhanced the HCBS [75], which requires static CPU resource allocation for the containers, with dynamically adjusted resources based on the performance of the tasks within the container.

Container orchestration is a natural complement to container-based virtualization. We identified a gap in existing container orchestrators in their support for real-time container-based virtualization. Thus, we present REACT, a real-time Kubernetes extension that adds explicit support for real-time requirements. In REACT, the Kubelet is extended to manage real-time containers, while the Kubernetes Scheduler is enhanced to monitor and account for the real-time performance of worker nodes. Placement decisions are based on a utilization-driven schedulability analysis that considers container periods and runtimes, following the HCBS. The Kubernetes scheduler performs admission control to verify the schedulability of real-time containers co-located with best-effort workloads on the same nodes. Furthermore, the system incorporates a set of controllers that dynamically adjust resource allocations for real-time containers in order to maintain timing guarantees.

This thesis demonstrates that container-based virtualization can support predictable soft real-time workloads through a combination of runtime resource adaptation, control-aware feedback mechanisms, and real-time-aware orchestration, thereby bridging the gap between cloud-native technologies and real-time system requirements.

### 5.2.1 Future Work

The results presented in this thesis open several directions for future research. First, the current resource-adaptation mechanisms are primarily reactive. Resource reallocation is triggered after performance degradation has been observed. Although this approach is effective in mitigating disturbances, it allows degradation to become visible before corrective action is taken. Future work should therefore investigate proactive and predictive resource-management mechanisms. Such mechanisms could use workload trends,

performance counters, QoC indicators, or learned interference patterns to anticipate resource shortages before timing degradation occurs.

Second, future work should investigate interference-aware migration strategies. In the current approach, migration is used when local resources are insufficient to satisfy the requirements of real-time containers. However, it may often be effective to migrate the container causing interference rather than the container experiencing degraded performance. Identifying the best container to migrate is a non-trivial problem, especially when interference depends on (a combination of) shared hardware resources such as caches, memory bandwidth, and I/O operations.

Third, scalability remains an important challenge. Some of the proposed placement and allocation mechanisms (e.g., in Paper D) rely on SMT or OMT solvers, which can provide high-quality or optimal solutions but may not scale to large industrial environments with many containers and many compute nodes. Future work should therefore explore scalable heuristics and approximation algorithms.

Finally, further validation in industrial settings is needed. The thesis has demonstrated the feasibility and effectiveness of the proposed mechanisms through experimental evaluation, but real industrial automation environments introduce additional constraints, including legacy systems, certification requirements, heterogeneous hardware, mixed-criticality workloads, and long deployment lifecycles. Evaluating the proposed mechanisms in such environments would provide deeper insight into their practical applicability and limitations.

# Bibliography


[1] Sébastien Vaucher, Rafael Pires, Pascal Felber, Marcelo Pasin, Valerio Schiavoni, and Christof Fetzer. Sgx-aware container orchestration for heterogeneous clusters. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*. IEEE, 2018.

[2] Roberto Morabito. Power consumption of virtualization technologies: an empirical investigation. In *2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC)*. IEEE, 2015.

[3] Senecca Miller, Travis Siems, and Vidroha Debroy. Kubernetes for cloud container orchestration versus containers as a service (caas): Practical insights. In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, 2021.

[4] Matheus Stolet, Liam Arzola, Simon Peter, and Antoine Kaufmann. Virtuoso: High resource utilization and sub-micro s-scale performance isolation in a shared virtual machine tcp network stack. *arXiv preprint arXiv:2309.14016*, 2023.

[5] Sergio Moreschini, Fabiano Pecorelli, Xiaozhou Li, Sonia Naz, David Hästbacka, and Davide Taibi. Cloud continuum: The definition. *IEEE Access*, 2022.

[6] Sergio Rubio, Santiago Bogarra, Marco Nunes, and Xavier Gomez. Smart grid protection, automation and control: Challenges and opportunities. *Applied Sciences*, 2025.

[7] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & information systems engineering*, 2014.

[8] Mohammed Salman Shaik, Vaclav Struhar, Zeinab Bakhshi, Van-Lan Dao, Nitin Desai, Alessandro V. Papadopoulos, Thomas Nolte, Vasileios Karagiannis, Stefan Schulte, Alexandre Venito, and Gerhard Fohler. Enabling fog-based industrial robotics systems. In *25th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2020.

[9] Shaik Mohammed Salman, Václav Struhár, Alessandro Vittorio Papadopoulos, Moris Behnam, and Thomas Nolte. Fogification of industrial robotic systems: Research challenges. In *Proceedings of the Workshop on Fog Computing and the IoT (Fog-IoT)*, Apr. 2019.

[10] Christian Matt. Fog computing: Complementing cloud computing to facilitate industry 4.0. *Business & information systems engineering*, 2018.

[11] Hongyu Pei Breivold and Kristian Sandström. Internet of things for industrial automation–challenges and technical solutions. In *2015 IEEE International Conference on Data Science and Data Intensive Systems*. IEEE, 2015.

[12] Philip Axer, Rolf Ernst, Heiko Falk, Alain Girault, Daniel Grund, Nan Guan, Bengt Jonsson, Peter Marwedel, Jan Reineke, Christine Rochange, Maurice Sebastian, Reinhard Von Hanxleden, Reinhard Wilhelm, and Wang Yi. Building timing predictable embedded systems. *ACM Trans. Embed. Comput. Syst.*, mar 2014.

[13] Luca Abeni and Giorgio Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Systems*, jul 2004.

[14] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 3rd edition, 2011.

[15] Dakshina Dasari, Benny Akesson, Vincent Nelis, Muhammad Ali Awan, and Stefan M Petters. Identifying the sources of unpredictability in cots-based multicore systems. In *2013 8th IEEE international symposium on industrial embedded systems (SIES)*. IEEE, 2013.

[16] Guillem Bernat, Antoine Colin, and Stefan M Petters. Wcet analysis of probabilistic hard real-time systems. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002.*, pages 279–288. IEEE, 2002.

[17] Marisol García-Valls, Tommaso Cucinotta, and Chenyang Lu. Challenges in real-time virtualization and predictable cloud computing. *Journal of Systems Architecture*, 2014.

[18] Surya Kant Garg and J. Lakshmi. Workload performance and interference on containers. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing & Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, 2017.

[19] Wen-Yan Chen, Ke-Jiang Ye, Cheng-Zhi Lu, Dong-Dai Zhou, and Cheng-Zhong Xu. Interference analysis of co-located container workloads: a perspective from hardware performance counters. *Journal of Computer science and Technology*, 35(2):412–417, 2020.

[20] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems - EuroSys'10*, 2010.

[21] Francisco Muro, Eduardo Baena, Sergio Fortes, Lars Nielsen, and Raquel Barco. Noisy neighbour impact assessment and prevention in virtualized mobile networks. *IEEE Transactions on Network and Service Management*, 2022.

[22] Simon Volpert, Sascha Winkelhofer, Jörg Domaschka, and Stefan Wesner. Detecting noisy neighbors in cpu-isolated cgroups environments. In *Proceedings of the 16th ACM/SPEC International Conference on Performance Engineering*, 2025.

[23] Gaurav Chaudhary, Derssie Mebratu, Bryan Lewis, Rahul Khanna, Jun Jin, and Mohammad Hossain. Monitoring workload performance in noisy neighborhoods using performance monitoring units. In *2023 IEEE/ACM International Workshop on Cloud Intelligence AIOps (AIOps)*, 2023.

[24] Miryeong Kwon, Donghyun Gouk, Changrim Lee, Byounggeun Kim, Jooyoung Hwang, and Myoungsoo Jung. Deterministic i/o and resource isolation for os-level virtualization in server computing. In *Proc. 12th Annual Non-Volatile Memories Workshop*, 2021.

[25] M. G. Xavier, I. C. De Oliveira, F. D. Rossi, R. D. Dos Passos, K. J. Matteussi, and C. A. F. D. Rose. A performance isolation analysis of disk-intensive workloads on container-based clouds. In *Euromicro Int. Conf. on Par., Distr., and Netw. Proc. (PDP)*, 2015.

[26] Junaid Khalid, Eric Rozner, Wesley Felter, Cong Xu, Karthick Rajamani, Alexandre Ferreira, and Aditya Akella. Iron: Isolating network-based {CPU} in container environments. In *15th USENIX Symposium on Networked Systems Design and Implementation (NSDI 18)*, 2018.

[27] Mohammad Shahrad, Sameh Elnikety, and Ricardo Bianchini. Provisioning differentiated last-level cache allocations to vms in public clouds. In *Proceedings of the ACM Symposium on Cloud Computing*, 2021.

[28] Junwen Zhang, Weiling Yang, Jianbin Fang, Dezun Dong, and Xianzhang Chen. Flashgemm: Optimizing sequences of matrix multiplication by exploiting data reuse on cpus. *ACM Trans. Archit. Code Optim.*, 22(4), December 2025.

[29] Milo MK Martin, Mark D Hill, and Daniel J Sorin. Why on-chip cache coherence is here to stay. *Communications of the ACM*, 2012.

[30] Are noisy neighbors in your data center keeping you up at night? `https://kib.kiev.ua/x86docs/Intel/RDT/intel-rdt-infrastructure-paper.pdf`. Accessed: 2026-07-24.

[31] M.Reza HoseinyFarahabady and Albert Y. Zomaya. Controlling performance interference in multi-tenant containerized environments. In *2024 22nd International Symposium on Network Computing and Applications (NCA)*, 2024.

[32] Jakob Danielsson, Marcus Jägemar, Moris Behnam, Mikael Sjödin, and Tiberiu Seceleanu. Measurement-based evaluation of data-parallelism

for opencv feature-detection algorithms. In *2018 IEEE 42nd annual computer software and applications conference (COMPSAC)*, volume 1, pages 701–710. IEEE, 2018.

[33] D.C Liang. Hard real-time bus architecture and arbitration algorithm based on amba. In *2015 International Conference on Information and Communications Technologies (ICT 2015)*, pages 1–7, 2015.

[34] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Autonomic vertical elasticity of docker containers with elasticdocker. In *2017 IEEE 10th international conference on cloud computing (CLOUD)*. IEEE, 2017.

[35] Karl Johan Åström and Richard Murray. *Feedback systems: an introduction for scientists and engineers*. Princeton university press, 2021.

[36] Gordana Dodig-Crnkovic. Scientific methods in computer science. In *Proceedings of the Conference for the Promotion of Research in IT at New Universities and at University Colleges in Sweden, Skövde, Suecia*, 2002.

[37] Susanta Nanda Tzi-cker Chiueh and Stony Brook. A survey on virtualization technologies. *Rpe Report*, 142, 2005.

[38] Sharvari Tamane. A review on virtualization: A cloud technology. *International Journal on Recent and Innovation Trends in Computing and Communication*, 2015.

[39] Nancy Jain and Sakshi Choudhary. Overview of virtualization in cloud computing. In *2016 Symposium on Colossal Data Analysis and Networking (CDAN)*. IEEE, 2016.

[40] Gernot Heiser. The role of virtualization in embedded systems. In *Proceedings of the 1st workshop on Isolation and integration in embedded systems*, 2008.

[41] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th international middleware conference*, 2016.

[42] Luca Abeni. Virtualized real-time workloads in containers and virtual machines. *Journal of Systems Architecture*, 2024.

[43] Vivian Noronha, Ekkehard Lang, Maximilian Riegel, and Thomas Bauschert. Performance evaluation of container based virtualization on embedded microprocessors. In *2018 30th International Teletraffic Congress (ITC 30)*, 2018.

[44] Amir Varasteh, Farzad Tashtarian, and Maziar Goudarzi. On reliability-aware server consolidation in cloud datacenters. In *2017 16th International Symposium on Parallel and Distributed Computing (ISPDC)*. IEEE, 2017.

[45] Rui Queiroz, Tiago Cruz, Jérôme Mendes, Pedro Sousa, and Paulo Simões. Container-based virtualization for real-time industrial systems—a systematic review. *ACM Computing Surveys*, 2023.

[46] Claudio Scordino, Ida Maria Savino, Luca Cuomo, Luca Miccio, Andrea Tagliavini, Marko Bertogna, and Marco Solieri. Real-time virtualization for industrial automation. In *2020 25th IEEE international conference on emerging technologies and factory automation (ETFA)*, 2020.

[47] E Torres, P Eguia, O Abarrategui, M Larruskain, V Valverde, and G Buigues. Virtualisation in substations: Technologies and applications. *Renewable Energies and Power Quality Journal*, 2024.

[48] David J MacDonald, Mital Kanabar, and Ramu Gudimetla. Software and hardware considerations for virtualised protection and control. In *19th IET Conference on Developments in Power System Protection (DPSP Europe 2025)*. IET, 2025.

[49] Yehor Karpichev, Mahmoud Chick Zaouali, Todd Charter, and Homayoun Najjaran. A deployable and scalable ros-docker framework for multi-platform digital twin applications. In *2025 IEEE Canadian Conference on Electrical and Computer Engineering (CCECE)*, 2025.

[50] Jueun Jeon, Byeonghui Jeong, and Young-Sik Jeong. Intelligent resource scaling for container-based digital twin simulation of consumer electronics. *IEEE Transactions on Consumer Electronics*, 2024.

[51] Jani Valtari, Anna Kulmala, Sandro Schönborn, D Kozhaya, R Birke, and J Reikko. Real-life pilot of virtual protection and control-experiences and performance analysis. In *27th International Conference on Electricity Distribution (CIRED 2023)*. IET, 2023.

[52] Siprotec v — siemens. `https://www.siemens.com/en-us/products/siprotec/siprotec-v/`. Accessed: 2026-07-24.

[53] Michael Sollfrank, Frieder Loch, Steef Denteneer, and Birgit Vogel-Heuser. Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation. *IEEE Transactions on Industrial Informatics*, 2020.

[54] Zheng Li, Maria Kihl, Qinghua Lu, and Jens A Andersson. Performance overhead comparison between hypervisor and container based virtualization. In *2017 IEEE 31st International Conference on advanced information networking and applications (AINA)*. IEEE, 2017.

[55] F. Ramalho and A. Neto. Virtualization at the network edge: A performance comparison. In *IEEE Int. Symp. A World of Wirel., Mob. and Multim. Net. (WoWMoM)*, 2016.

[56] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. My vm is lighter (and safer) than your container. In *Proceedings of the 26th Symposium on Operating Systems Principles*, 2017.

[57] Cornelia Wulf, Michael Willig, and Diana Göhringer. A survey on hypervisor-based virtualization of embedded reconfigurable systems. In *2021 31st International Conference on Field-Programmable Logic and Applications (FPL)*. IEEE, 2021.

[58] Ankita Desai, Rachana Oza, Pratik Sharma, and Bhautik Patel. Hypervisor: A survey on concepts and taxonomy. *International Journal of Innovative Technology and Exploring Engineering*, 2013.

[59] Sachchidanand Singh and Nirmala Singh. Containers and docker: Emerging roles and future of cloud technology. In *2016 2nd International Conference on Applied and Theoretical Computing and Communication Technology (iCATccT)*, 2016.

[60] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 2014.

[61] Guoqing Li, Keichi Takahashi, Kohei Ichikawa, Hajimu Iida, Chawanat Nakasan, Pattara Leelaprute, Pree Thiengburanathum, and Passakorn Phannachitta. The convergence of container and traditional virtualization: strengths and limitations. *SN Computer Science*, 4(4):387, 2023.

[62] Miguel G Xavier, Marcelo V Neves, Fabio D Rossi, Tiago C Ferreto, Timoteo Lange, and Cesar AF De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *2013 21st Euromicro international conference on parallel, distributed, and network-based processing*. IEEE, 2013.

[63] Kata containers. `https://katacontainers.io/`. Accessed: 2026-07-24.

[64] Richard S. Canon and Andrew Younge. A case for portability and reproducibility of hpc containers. In *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, 2019.

[65] S. He, L. Guo, Y. Guo, C. Wu, M. Ghanem, and R. Han. Elastic application container: A lightweight approach for cloud resource provisioning. In *IEEE Int. Conf. on Adv. Inform. Netw. and Appl. (AINA)*, 2012.

[66] Thuy Linh Nguyen and Adrien Lebre. Conducting thousands of experiments to analyze vms, dockers and nested dockers boot time. Research report, INRIA, 2018.

[67] Miguel Gomes Xavier, Marcelo Veiga Neves, and Cesar Augusto Fonticielha De Rose. A performance comparison of container-based virtualization systems for mapreduce clusters. In *Euromicro Int. Conf. on Par., Distr., and Netw. Proc. (PDP)*, 2014.

[68] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and Linux containers. In *IEEE Int. Symp. on Perf. Analysis of Syst. and Soft. (ISPASS)*, 2015.

[69] Open Container Initiative. Open container initiative: Linux container configuration, 2026.

[70] The opencontainers runtime spec. `https://specs.opencontainers.org/runtime-spec`. Accessed: 2026-07-24.

[71] The opencontainers image spec. `https://specs.opencontainers.org/image-spec`. Accessed: 2026-07-24.

[72] The opencontainers distribution spec. `https://specs.opencontainers.org/distribution-spec`. Accessed: 2026-07-24.

[73] John A Stankovic et al. *Real-time computing systems: The next generation*. University of Massachusetts at Amherst. Computer and Information Science [COINS], 1988.

[74] Chung Laung Liu and James W Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 1973.

[75] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the Linux kernel. *ACM SIGBED Review*, 11 2019.

[76] Marco Barletta, Francesco Boccola, Marcello Cinque, Luigi De Simone, Raffaele Della Corte, and Daniele Ottaviano. Integrating containers and partitioning hypervisors for dependable real-time industrial clouds. 2024.

[77] Geoffrey Phi C Tran, Yu-An Chen, Dong-In Kang, John Paul Walters, and Stephen P Crago. Hypervisor performance analysis for real-time workloads. In *2016 IEEE High Performance Extreme Computing Conference (HPEC)*. IEEE, 2016.

[78] 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.

[79] Marcello Cinque, Raffaele Della Corte, Antonio Eliso, and Antonio Pecchia. Rt-cases: Container-based virtualization for temporally separated mixed-criticality task sets. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*. Schloss Dagstuhl–Leibniz-Zentrum für Informatik, 2019.

[80] Florian Hofer, Martin Sehr, Antonio Iannopollo, Ines Ugalde, Alberto Sangiovanni-Vincentelli, and Barbara Russo. Industrial control via application containers: Migrating from bare-metal to IAAS. In *IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom)*, 2019.

[81] Thomas Goldschmidt and Stefan Hauck-Stattelmann. Software containers for industrial control. In *Euromicro Conf. on Soft. Eng. and Adv. Appl. (SEAA)*, 2016.

[82] S. Schönborn, R. Birke, D. Kozhaya, and T. Sivanthi. Real-time performance of virtualised protection and control software. In *27th International Conference on Electricity Distribution (CIRED 2023)*, 2023.

[83] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *2008 Euromicro Conference on Real-Time Systems*. IEEE, 2008.

[84] Marco Barletta, Marcello Cinque, Luigi De Simone, Raffaele Della Corte, Giorgio Farina, and Daniele Ottaviano. Partitioned containers: Towards safe clouds for industrial applications. In *2023 53rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, 2023.

[85] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[86] Shashank Shekhar, Hamzah Abdel-Aziz, Anirban Bhattacharjee, Aniruddha Gokhale, and Xenofon Koutsoukos. Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018.

[87] Paul Veitch, Edel Curley, and Tomasz Kantecki. Performance evaluation of cache allocation technology for nfv noisy neighbor mitigation. In *2017 IEEE Conference on Network Softwarization (NetSoft)*. IEEE, 2017.

[88] Udi Margolin, Alberto Mozo, Bruno Ordozgoiti, Danny Raz, Elisha Rosensweig, and Itai Segall. Using machine learning to detect noisy neighbors in 5g networks. *arXiv preprint arXiv:1610.07419*, 2016.

[89] Ethem Alpaydin. *Introduction to machine learning*. MIT press, 2020.

[90] Hedi Bouattour, Yosra Ben Slimen, Marouane Mechteri, and Hanane Biallach. Root cause analysis of noisy neighbors in a virtualized infrastructure. In *2020 IEEE Wireless Communications and Networking Conference (WCNC)*. IEEE, 2020.

[91] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019.

[92] Maria A Rodriguez and Rajkumar Buyya. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, 2019.

[93] Anshita Malviya and Rajendra Kumar Dwivedi. A comparative analysis of container orchestration tools in cloud computing. In *2022 9th International Conference on Computing for Sustainable Global Development (INDIACom)*, 2022.

[94] Marco Barletta, Marcello Cinque, Luigi De Simone, and Raffaele Della Corte. Criticality-aware monitoring and orchestration for containerized industry 4.0 environments. *ACM Transactions on Embedded Computing Systems*, 2024.

[95] Marco Barletta, Marcello Cinque, Luigi De Simone, and Raffaele Della Corte. Introducing k4. 0s: a model for mixed-criticality container orchestration in industry 4.0. In *2022 IEEE Intl Conf on Dependable, Autonomic and Secure Computing, Intl Conf on Pervasive Intelligence*

*and Computing, Intl Conf on Cloud and Big Data Computing, Intl Conf on Cyber Science and Technology Congress (DASC/PiCom/CBDCom/-CyberSciTech)*. IEEE, 2022.

[96] 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*, 2022.

[97] Arvind Easwaran, Insik Shin, and Insup Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Systems*, 2009.

[98] Cecil Wöbker, Andreas Seitz, Harald Mueller, and Bernd Bruegge. Fogernetes: Deployment and management of fog computing applications. In *NOMS 2018-2018 IEEE/IFIP Network Operations and Management Symposium*. IEEE, 2018.

[99] D. Santoro, D. Zozin, D. Pizzolli, F. De Pellegrini, and S. Cretti. Foggy: A platform for workload orchestration in a fog computing environment. In *2017 IEEE International Conference on Cloud Computing Technology and Science (CloudCom)*, 2017.

[100] Marcus Jägemar, Andreas Ermedahl, Sigrid Eldh, and Moris Behnam. A scheduling architecture for enforcing quality of service in multi-process systems. In *2017 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, pages 1–8, 2017.

[101] Aleksandrs Slivkins. Introduction to multi-armed bandits. 2024.

[102] 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)*. IEEE, 2021.

[103] Václav Struhár, Mohammad Ashjaei, Moris Behnam, Alessandro V Papadopoulos, and Silviu S Craciunas. Resource adaptation for real-time containers considering quality of control. In *2023 IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2023.

[104] Václav Struhár, Silviu S Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V Papadopoulos. Hierarchical resource orchestration framework for real-time containers. *ACM Transactions on Embedded Computing Systems*, 23(1):1–24, 2024.

[105] Václav Struhár, Alessandro V. Papadopoulos, Inmaculada Ayala, Mercedes Amor, and Lidia Fuentes. Container orchestration in edge computing with fluctuating green energy: A multi-armed bandit approach. In *Proceedings of the International Conference on Ubiquitous Computing and Ambient Intelligence (UCAmI 2025), Volume 2*. Springer Nature Switzerland, 2026.

[106] Shaik Mohammed Salman, Vaclav Struhar, Alessandro V Papadopoulos, Moris Behnam, and Thomas Nolte. Fogification of industrial robotic systems: research challenges. In *1st Workshop on Fog Computing and the IoT (Fog-IoT 2019)*, 2019.

[107] Mohammed Salman Shaik, Václav Struhár, Zeinab Bakhshi, Van-Lan Dao, Nitin Desai, Alessandro V Papadopoulos, Thomas Nolte, Vasileios Karagiannis, Stefan Schulte, Alexandre Venito, et al. Enabling fog-based industrial robotics systems. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*. IEEE, 2020.

[108] Mirgita Frasheri, Vaclav Struhar, Alessandro Vittorio Papadopoulos, and Aida Causevic. Ethics of autonomous collective decision-making: The caesar framework. *Science and Engineering Ethics*, 2022.

[109] Václav Struhár, Mohammad Ashjaei, Moris Behnam, Silviu S Craciunas, and Alessandro V Papadopoulos. Dart: Dynamic bandwidth distribution framework for virtualized software defined networks. In *IECON 2019-45th Annual Conference of the IEEE Industrial Electronics Society*. IEEE, 2019.

# II

# Included Papers

# Chapter 6

# Paper A: Real-time containers: A survey

*Václav Struhár*, Moris Behnam, Mohammad Ashjaei, Alessandro V. Papadopoulos

In 2nd Workshop on Fog Computing and the IoT (Fog-IoT 2020).

**Abstract**

Container-based virtualization has gained importance in the deployment of software applications in cloud-based environments. The technology allows for hosting multiple isolated containers on a single computation node as well as decomposing an application into multiple containers distributed among several hosts. Such an approach can bring many benefits across different fields, e.g., in industrial automation, automotive, and aviation, where software functions from various vendors can be co-located on shared resources. On the other hand, containers do not yet fully support the real-time requirements of containerized applications.

In this work, we provide a systematic literature survey study that summarizes the effort of the research community on bringing real-time properties in container-based virtualization.

## 6.1 Introduction

Fog Computing, as well as cloud computing, relies extensively on resource virtualization. In this area, container-based virtualization is gaining importance as a lightweight alternative to hypervisor-based virtualization. The container technology allows executing applications and their software dependencies in a virtual environment independently of the software ecosystem of the hosts. A host can accommodate multiple containers at a time, providing means for container isolation and resource control for the containers. Container-based virtualization (OS level virtualization) does not require a hypervisor, and therefore it provides near-native performance [1, 2], rapid deployment times, and a low overhead while still retaining a certain level of resource isolation and resource control. The containers are a de facto standard for the development of large-scale web applications, adopted by a number of companies [3].

The benefits of container-based virtualization are aligned with the strive of the companies in other areas, such as in industrial and robot control, automotive, and aviation. In these industrial domains, there are strong requirements to (i) consolidate computational resources (Electronic Control Units, physical controllers) and (ii) provide a flexible environment for running (real-time) applications. Additionally, container-based virtualization can enable interruption-free hardware and software maintenance, dynamic system redundancy change and system redundancy healing [4], However, in such fields stringent real-time requirements are often needed. This means that an application inside a container should meet the predefined deadlines independently of other co-located containers.

In this survey, we summarize the research carried out in the area of real-time containers since the introduction of containers in the Linux kernel.

**The main contributions of this paper include**:

- Systematic literature survey of the real-time containers.
- Overview of the approaches and technology enabling real-time behavior of containers.
- Identification of pitfalls, challenges and future research directions for real-time containers.

## 6.2 The Review Process

The systematic literature survey is carried out with the guidance in [5]. The research questions are defined together with search queries and sources of the studies, and subsequently, we extract the data and answer the questions. We apply the snowballing [6] method to identify relevant papers outside the search query. Databases used: *Scopus*, *IEEE Xplore*, *ACM Digital Library*. Only full peer review papers published between 2008 and 2019 are considered. We search the databases using the following search queries:

| (***Real-time*** OR ***RT***) AND (***Containers*** OR ***Container***) |
|---|

### 6.2.1 Question formalization

In this work, we elaborate the following questions:

- **RQ1**: *Why and in which context have real-time containers been used?* Answering this question will give an overview of the motivation behind the use of real-time containers, expected benefits and areas where real-time containers are used.
- **RQ2**: *What approaches are used for enabling real-time guarantees of containers?* The answer will give an overview of the approaches and technologies, their combinations and their usages for the real-time container-based virtualization.
- **RQ3**: *What are the pitfalls and weak points of using real-time containers that prevent full adoption of such technology in industry?* The answer for this question will give list of research challenges and problems for real-time container computing.

## 6.3 Containers technology

From the runtime perspective, a container is a set of resource-limited processes that are isolated from the rest of the system and from other containers. This is achieved by utilizing two Linux kernel features: (i) Namespaces and (ii) Control groups (cgroups). Namespaces virtualize global resources (e.g., processes, network, inter-process communication) in a way that a group of processes can

see and use one set of resources while another group can use a different set of resources. Cgroups provide a mechanism for aggregating and partitioning sets of tasks, and all their future children, into hierarchical groups with specialized behavior[1]. It allows for organizing processes hierarchically and distributing system resources along the hierarchy.

### 6.3.1 Container platforms

There are several container solutions, all of them rely on the aforementioned Linux kernel features. Thus, all the platforms pose similar options and performance [7]. The philosophy of using the two most commonly used container platforms LXC and Docker differs. Docker containers are microservice-based (each container should contain a single application), whereas LXC allows to run a complex ecosystem of applications which is beneficial for emulation of legacy systems.

### 6.3.2 Real-Time Containers

The term real-time implies that the correctness of the system depends not only on the results of the computation but also on the time at which the results are produced [8]. Real-time systems can be categorized into three groups: hard, firm and soft real-time. Missing a deadline in a hard real-time system may cause catastrophic consequences, whereas missing deadline in a firm real-time system leads to the complete loss of the utility of the result. Missing a deadline in a soft real-time system just degrades the utility of the result.

A real-time container is a container that provides resource isolation, resource control, and additionally provides time-deterministic and predictable behavior for the containerized application.

### 6.3.3 Real-time support of Linux

To ensure the time-predictable behavior of the containers, the operating systems must provide such a capability. Default (Vanilla) Linux does not give any time guarantees on execution of tasks and therefore the predictability is low [9]. However, there are several approaches to improve the predictability:

[1] `https://www.man7.org/linux/man-pages/man7/cgroups.7.html`

the real-time patch that improves preemptability of the Linux kernel and co-kernel approaches that run a real-time micro kernel in parallel to the Linux kernel. Containerized applications are scheduled in the same way as native applications using the host's scheduler. The default Linux kernel provides three schedulers: *(i) Completely Fair Scheduler (CFS)*: Aims to maximize CPU utilization while also maximizing interactive performance. It provides no time guarantees. *(ii) Real-Time scheduler (RT)*: The scheduler allows scheduling tasks in a fixed priority manner using First In First Out or Round Robin policies. The tasks run till they yield or are preempted by higher-priority tasks. The Real-Time group scheduling[2] extension allows dividing and allocating CPU time between real-time and non-real-time tasks. *(iii) Earliest Deadline First Scheduler (EDF)* Uses Constant Bandwidth Server [10] and allows to associate with each task a budget and a period.

## 6.4 Survey Results

In this section, we conclude relevant papers. There are four main directions for enabling real-time behavior of containers: (i) real-time patch-based, (ii) co-kernel-based, (iii) hierarchical scheduling-based, and (iv) custom approach. A short summary is provided in Table 6.1.

### 6.4.1 Real-time patch based

The real-time patch (PREEMPT_RT) improves the kernel's locking primitives to maximize preemptible sections. The advantage of the patch is that there is no need for special libraries or API needed by the application developers.

Moga et al. [17] consider real-time containers in the context of industrial automation systems that work with real-time data and have real-time deadlines on detection and response to events. The paper emphasizes the need for OS-level virtualization in industrial automation and gives examples of timing requirements of industrial applications (e.g., motor drive typically requires a cycle time between 1ms to 250$\mu$s) and a need for synchronization between the containers. The evaluation of the effects of containers on the performance of industrial automation systems is provided in two cases: (i) Cyclic behavior of a

[2] `https://www.kernel.org/doc/Documentation/scheduler/sched-rt-group.txt`

| Study | Main focus | Approach & Technology | Communication aspects |
|---|---|---|---|
| Cinque et al. [11, 12] | · Architecture definition<br>· Faulty tasks monitoring<br>· Implementation details | · RTAI<br>· Docker<br>· Fixed priority scheduling | |
| Cucinotta et al. [13, 14, 15] | · Temporal Interference between containers | · Hierarchical Scheduling | - |
| Tasci et al. [16] | · Architecture definition<br>· Real-time communication between containers | · Combination of Real-Time patch and Xenomai<br>· Docker | Design of messaging system based on ZeroMQ. |
| Moga et al. [17] | · Feasability study<br>· Communication between containers<br>· Communication overheads | · Docker<br>· Real-Time patch | Network performance and overhead measurements between containers using default Docker Linux NAT Bridge. |
| Hofer et al. [18] | · Experimental comparison between Real-Time patch, Xenomai, Vanilla Linux | · Real-Time patch<br>· Xenomai<br>· Vanilla Linux | |
| Goldschmidt et al. [19, 4] | · Architecture definition<br>· Feasibility study | · Real-Time patch<br>· Legacy systems emulation in real-time containers | |
| Telschig et al. [20] | · Model-based architecture and analysis<br>· Dependable real-time container computing. | · LXC | |
| Mao et al. [21] | · Minimizing latencies in software-based Radio Access Networks | · Docker<br>· Real-Time patch | Application of fast packet processing using Intel Data Plane Development Kit. |
| Masek et al. [22] | · Systematic evaluation of sandboxed software | · Real-Time patch | |
| Wu et al. [23] | · Dynamic CPU allocation for mixed-criticality real-time systems | · Custom scheduling mechanism<br>· Docker | |

Table 6.1: Summary of studies elaborating on real-time containers.

containerized application, (ii) Virtual networking performance for communications between containers. Cyclic behavior test evaluates the ability to execute application logic at pre-defined intervals, measures accuracy, and jitter. Virtual networking test evaluates the ability to communicate between co-located containers in a time-bounded manner. The researchers see real-time container computing as a promising technology, communication mechanisms between containers are not clear.

The work in [4] (and previously [19]) addresses a container-based architecture for real-time controllers that allows a flexible function deployment and supports legacy control applications. Such architecture is needed to preserve the functionality of legacy control programs and to reduce the maintenance cost of legacy systems (in which the software is often bound to a specific hardware and software ecosystem). The researchers investigate the feasibility of building a real-time capable system (for legacy systems) based on real-time containers, they target PLCs and automation controllers with the cycle time between 100ms to 1s. They perform a set of tests under various load scenarios (i) using containerized applications inside of Docker and (ii) running a complete operating system (PowerPC) inside LXC. They conclude that a containerized execution of control applications can meet the requirements of PLCs and automation controllers.

From a latency perspective, Masek et al. [22] conducted a literature review of sandboxed real-time software, using self-driving vehicles as an example. The researchers were interested in the question: How does the execution environment influence the scheduling precision and input/output performance of a given application? The result shows that Docker does not impose additional overhead (as in [24]) on scheduling and I/O performance. However, selecting the correct kernel has a greater impact on the scheduling precision and input/output performance of containers.

Mao et al. [21] uses real-time containers to enable software-based RAN (Radio Access Network) in order to avoid high capital and operating expenditures during deployment of new standards. However, the software based RAN has strict deadlines to satisfy (1ms). The researchers use real-time patch to decrease the latency, interestingly they improve the latency 13.9 times by applying the patch in comparison to the vanilla Kernel.

### 6.4.2 Co-Kernel based

In this approach, a real-time micro-kernel runs in parallel to the Linux kernel. The real-time co-kernel handles time-critical activities (e.g., handling interrupts and scheduling real-time threads), and the standard Linux kernel runs only when the co-kernel is idle. Compared with the real-time patch, the co-kernel approach yields lower latency and jitter. On the other hand, it requires special APIs, tools, and libraries for application development. Additionally, there are impediments to scaling co-kernel solutions on large platforms (e.g., many-core platforms). There are two co-kernel alternatives: Real Time Application Interface (RTAI) and Xenomai.

RTAI aims to minimize latencies to the lowest technically possible values. Real-time tasks are compiled as kernel modules and run in the kernel space. Xenomai [25] is a fork of RTAI. Its mission is to enable real-time tasks in the user space. It consists of an emulation layer that is capable of reusing code from other RTOSes.

Tasci et al. [16] elaborates on modularization of real-time control applications into real-time containers. Such modular architecture needs two essential parts: (i) a computational part, enabled by a real-time operating system (combination of Xenomai and real-time patch), and (ii) a messaging part that allows passing messages between containers in a real-time manner. Traditional monolithic architectures communicate through function calls and shared memory; the containers do not make the assumption that they are running on the same host or in a distributed environment (they communicate through the standard OS networking stack), therefore, direct passing messages through shared memory is not directly supported. Hence, the researchers provide a design and implementation of a custom-made real-time messaging system for containers based on ZeroMQ[3].

Hofer et al. [18] use the real-time containers in the context of control applications. The paper presents comparison between type 1 hypervisor, Vanilla Linux, Xenomai co-kernel and Linux with real-time patch for various idle and stress scenarios.

[3] `https://zeromq.org/`

### 6.4.3 Hierarchical Scheduling based

Inspired by a similar concept in the hypervisor-based virtualization, where a global scheduler assigns CPU time for the virtual machines, the second-layer scheduler schedules the individual tasks of the VM.

Cucinotta, Abeni et al. [14, 15, 13] proposed the use of real-time containers in the field of Network Function Virtualization (NFV), where the functionality of traditional physical network devices (e.g., firewalls) is transformed into software components (in containers) that are consolidated in a single computing device. NFV has critical latency requirements imposed by the need for time-critical per-packet processing. The researchers modified the Linux scheduling mechanism to provide two-level hierarchical scheduling. First level Earliest Deadline First scheduler selects the container to be scheduled on each CPU. Subsequently, the second-level Fixed Priority scheduler selects a task in the container. CPU reservation (runtime quota and period) is assigned to each of the containers.

### 6.4.4 Custom real-time approaches

Wu et al. [23] proposed the Flexible Deferrable Scheduler for containerized mixed-criticality real-time systems that consist of real-time and non-real-time containers. The scheduler guarantees the allocated CPU capacity to real-time containers and dynamically distributes the unused capacity to non-real-time containers. The work supplements the Completely Fair Scheduler with a Workload Adjustment Module that collects CPU utilization by containers and a Dynamic Adjustment Module that allocates CPU to the containers.

Cinque et al. [12] (previously [11]) implemented real-time containers using a Linux kernel patched with the real-time co-kernel (RTAI) and custom monitoring and policy-enforcing modules. Their solution allows cohabiting containers with different criticality levels and prevents fixed-priority hard real-time periodic tasks inside the containers from affecting the temporal guarantees of other containers. The temporal guarantees are provided by two mechanisms: (i) proper task priority assignments to tasks inside the containers and (ii) monitoring and enforcing temporal protection policies. The former ensures that tasks inside the high-criticality containers are assigned higher priorities than tasks in the lower-criticality containers, and thus they are never preempted by tasks of lower criticality containers. The latter monitors the tasks and, in case

of overruns or overtime, it enforces one of the temporal protection policies (i.e., kill or suspend the faulty task, suspend the task until the next period).

## 6.5 Weaknesses of real-time containers

The reviewed papers identify shortcomings and immature aspects of real-time container virtualization that prevent the expansion of the technology. Below, we list them categorized in three groups: (i) tools support, (ii) real-time communication support, and (iii) miscellaneous.

**Lack of tools for real-time container management:** The reviewed papers emphasize the need for supporting tools for real-time containers. Tools that enable deployment on containers, taking into account the real-time requirements of containers and the properties of computational nodes.

- The need for an orchestration tool that can schedule real-time containers based on pre-configured capabilities [18].
- Middleware that is aware of both communication needs as well as runtime and performance isolation needs [17].
- Framework to expose the runtime requirements of real-time applications running inside containers and to enforce an optimal allocation of containers to resources [17].

**Communication between real-time containers:** Real-time communication between a container and its environment has to be further researched. Currently, the reviewed papers emphasize the following issues:

- Need for a real-time communication among containers [17].
- Further investigation on container security restricted container access and intra-container communication [18].
- Would it complicate data management if data needs to be shared across containers [19]?

**Miscellaneous:** In addition to generic issues that may harm the real-time behavior (i.e., shared caches, I/O), the studies reviewed highlight the following points and questions:

- Lack of safety, security analysis of real-time containers and vulnerability management for the acceptance in industry [20, 19].
- Lack of latency and performance tests of recent releases of a patched Linux Kernel. As well as a proper analysis of the configuration of the Linux kernel parameters that may improve overall task determinism. [18].

- The measurements of memory overhead of the container solution, and whether it is acceptable for real-world applications [19].
- Processes in different containers may use the same resources in the same way because of their independent views of the system (i.e., processes are not aware of a resource-limited isolated environment co-located with other containers). This results in poor resource utilization and a potential violation of the real-time execution assumptions [17].
- Container approaches are a new technology. Will this create problems due to its possible immaturity [19]?

## 6.6 Conclusion

Container-based virtualization has become popular as a lightweight alternative to hypervisor-based virtualization. The technology has proven its viability in large cloud-based systems, it has been adopted by a number of enterprise companies, and it is supported by a large scale of tools (e.g., container orchestration and monitoring tools).

However, in industrial domains where real-time behavior is required, container-based virtualization does not seem to be mature enough. In this paper, we summarize the research carried out in the field of real-time containers. We show in what contexts, what approaches and technologies are used, and what are the possible immaturity points of container virtualization.

# Bibliography


[1] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and linux containers. In *2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS)*, 2015.

[2] Cristian Spoiala, Alin Calinciuc, Cornel Turcu, and Constantin Filote. Performance comparison of a webrtc server on docker versus virtual machine. *13th International Conference on DEVELOPMENT AND APPLICATION SYSTEMS, Suceava, Romania, May 19-21, 2016*, pages 295–298, 05 2016.

[3] Thanh Bui. Analysis of docker security. *ArXiv*, abs/1501.02967, 2015.

[4] Thomas Goldschmidt, Stefan Hauck-Stattelmann, Somayeh Malakuti, and Sten Grüner. Container-based architecture for flexible industrial control applications. 2018.

[5] Jo Hannay, Dag Sjøberg, and Tore Dybå. A systematic review of theory use in software engineering experiments. *Software Engineering, IEEE Transactions on*, 2007.

[6] Claes Wohlin. Guidelines for snowballing in systematic literature studies and a replication in software engineering. *ACM International Conference Proceeding Series*, 2014.

[7] Roberto Morabito, Jimmy Kjällman, and Miika Komu. Hypervisors vs. lightweight virtualization: a performance comparison. In *2015 IEEE International Conference on Cloud Engineering*, pages 386–393. IEEE, 2015.

[8] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*, volume 24. Springer Science & Business Media, 2011.

[9] Claudio Scordino and Giuseppe Lipari. Linux and real-time: Current approaches and future opportunities. In *IEEE Internafional Congress ANIPLA*, 2006.

[10] Luca Abeni, Giuseppe Lipari, and Juri Lelli. Constant bandwidth server revisited. *Acm Sigbed Review*, 2015.

[11] Marcello Cinque and Domenico Cotroneo. Towards lightweight temporal and fault isolation in mixed-criticality systems with real-time containers. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*. IEEE, 2018.

[12] Marcello Cinque, Raffaele Della Corte, Antonio Eliso, and Antonio Pecchia. Rt-cases: Container-based virtualization for temporally separated mixed-criticality task sets. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, 2019.

[13] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Alessio Balsini, and Carlo Vitucci. Reducing temporal interference in private clouds through real-time containers. 2019.

[14] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Alessio Balsini, and Carlo Vitucci. Virtual network functions as real-time containers in private clouds. In *IEEE CLOUD*, 2018.

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

[16] 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)*. IEEE, 2018.

[17] Alexandru Moga, Thanikesavan Sivanthi, and Carsten Franke. Os-level virtualization for industrial automation systems: are we there yet? 2016.

[18] Florian Hofer, Martin Sehr, Antonio Iannopollo, Ines Ugalde, Alberto Sangiovanni-Vincentelli, and Barbara Russo. Industrial control via application containers: Migrating from bare-metal to iaas. 2019.

[19] Thomas Goldschmidt and Stefan Hauck-Stattelmann. Software containers for industrial control. In *2016 42th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. IEEE, 2016.

[20] Kilian Telschig, Andreas Schonberger, and Alexander Knapp. A real-time container architecture for dependable distributed embedded applications. *2018 IEEE 14th International Conference on Automation Science and Engineering (CASE)*, pages 1367–1374, 2018.

[21] C. Mao, M. Huang, S. Padhy, S. Wang, W. Chung, Y. Chung, and C. Hsu. Minimizing latency of real-time container cloud for software radio access networks. In *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, 2015.

[22] Philip Masek, Magnus Thulin, Hugo Andrade, Christian Berger, and Ola Benderius. Systematic evaluation of sandboxed software deployment for real-time software on the example of a self-driving heavy vehicle. In *2016 IEEE 19th International Conference on Intelligent Transportation Systems (ITSC)*, 2016.

[23] J. Wu and T. Yang. Dynamic cpu allocation for docker containerized mixed-criticality real-time systems. In *2018 IEEE International Conference on Applied System Invention (ICASI)*, 2018.

[24] A. Krylovskiy. Internet of things gateways meet linux containers: Performance evaluation and discussion. In *2015 IEEE 2nd World Forum on Internet of Things (WF-IoT)*, 2015.

[25] Philippe Gerum. Xenomai-implementing a rtos emulation framework on gnu/linux. *White Paper, Xenomai*, pages 1–12, 2004.

# Chapter 7

# Paper B: REACT: Enabling real-time container orchestration

*Václav Struhár*, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos

In 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA 2021).

**Abstract**

Fog and edge computing offer the flexibility and decentralized architecture of cloud computing without the latency issues inherent in the cloud. This makes fog computing very attractive in real-time and safety-critical applications, especially if combined with container-based technologies. Whereas different orchestration systems are available to manage the container placement based on their resource demand, no orchestration system considers real-time requirements for containerized applications. In this paper, we present the architecture and design of a real-time container orchestrator based on Kubernetes. Moreover, this paper defines metrics for the performance evaluation of real-time containers and describes an initial model for allocating a mixture of real-time and non-real-time containers. We present an initial implementation of our real-time container extension and evaluate its feasibility on Linux-based systems.

## 7.1 Introduction

Fog and Edge Computing (FEC) are designed to address key limitations of cloud computing, e.g., unbounded communication latency and variability in bandwidth availability [1]. FEC enables the benefits of offloading data collection and decision-making even in application domains that require RT guarantees, e.g., in industrial automation [2, 3]. FEC systems reduce communication latency for critical data, enabling its processing on decentralized computational devices at the network's edge. While guaranteeing deterministic communication behavior between nodes can be achieved through the use of, e.g., TTEthernet or TSN, guaranteeing the RT behavior of computation functions can be challenging. The RT execution of functions on nodes cannot always be maintained in the same way due to the complexity and non-determinism of hardware artifacts (e.g., caches), and software layers within the compute nodes (e.g., OS layers, network stack, interrupts). Hence, to maintain the RT behavior of functions, isolation and adaptation/re-allocation of functions at run-time is necessary. Typically, the reallocation and isolation of functions have been facilitated through the use of virtualization technologies.

Container-based virtualization is gaining importance in industrial domains as a lightweight alternative to full-blown virtualization [4, 5]. Containers are standalone self-containing software packages, comprising applications and their software dependencies, that simplify the deployment of software [6]. The technology enables the sharing of computation resources among several containers while preserving isolation. Recently, there has been an effort to enhance container-based virtualization with time predictability (i.e., RT containers), enabling container applications with RT requirements [7, 8, 9]. Through these enhancements, containers may be used for time-critical applications required by many industrial systems, such as robot control in fog computing [2, 3]. In parallel, due to the continuous change in resource usage by applications running in fog and the availability of the resources, there is a need to dynamically manage and deploy containers in a cluster of compute nodes necessitating container orchestrators that take into account both resource requirements and resource availability. Additionally, container orchestrators offer further benefits, e.g., scalability and availability, fault-tolerance, and efficient resource utilization [10].

RT systems require additional constraints regarding the reaction time to

environmental events [11]. So far, container orchestrators do not consider RT requirements of applications running inside containers, neither from theoretical nor practical points of view. There are no mechanisms for the distribution of containers based on their RT requirements without violating these requirements and how to manage the co-execution of RT and Best Effort (BE) containers on the same node. RT containers are a novel topic [9]. There are no strategies for dealing with dynamic changes in container workloads that may change interference between RT containers. Containers do not provide strict resource isolation as they share not only physical resources but also Operating System (OS) kernel [12], and hence, they are prone to performance degradation in the presence of other competing containers. For example, disk-intensive workloads can induce performance degradation up to 35% [13]. Therefore, if the benefits of containers are to be exploited for RT systems, run-time monitoring and container orchestration that continuously evaluates QoS metrics and uses them in container scheduling decisions is necessary.

This paper proposes a container orchestration architecture to support the co-existence of RT and BE containers with temporal isolation. Specifically, the contributions of this paper are as follows:

- The design of a container orchestration architecture enabling the deployment and online adaptation of both RT and BE containers considering the timing requirements defined for time-critical applications;
- The definition of a mathematical model for the RT components of the designed orchestration architecture, and of the performance metrics needed to implement run-time monitoring and orchestration of the RT containers;
- The implementation of the proposed orchestration over the existing open-source container orchestrator Kubernetes.

## 7.2 Background and Prior Work

RT systems are computing systems that require deterministic temporal behavior since the correctness of their functionality depends not only on the value but also on the timing of the computed result [11]. Many software applications utilize periodic tasks that are cyclically executed [14]. Typically, RT systems require that tasks finish their execution in every period instance to be considered correct (assuming an implicit deadline for the tasks).

Virtualization is a technology that allows multiple virtualized applications or systems to be co-located onto the same shared platform [15]. There are two prevalent virtualization technologies: hypervisor- and container-based virtualization. Hypervisor-based virtualization utilizes a hypervisor which is a software layer that creates the different partitions within which each virtualized instance of an OS runs. In contrast, container-based virtualization utilizes OS features to create an isolated environment for processes. Container-based virtualization does not impose any requirements on hardware support for virtualization. Hence, such technology can be utilized on a broad range of devices. Containers co-located on a single computing node run as user-space isolated tasks and share functions of the host's OS (e.g., scheduling, memory allocation). From the user's view, each container appears and executes like a standalone OS. In comparison to hypervisor-based virtualization, container-based virtualization has a negligible overhead, is more resource efficient (e.g., 29.4 times less RAM usage than a VM [16]), has higher flexibility, provides fast booting times [17], and enables a near-native performance [6, 4, 5]. However, container-based virtualization provides a low level of isolation for memory, disk, and network operations [13]; hence, such operations may lead to degrading QoS of the applications.

The container-based virtualization relies on *namespaces* and *control groups*, referred to as *cgroups*, implemented in the host OS [18]. *Namespaces* partition global resources, e.g., tasks, network, inter-process communication, into different sets, only visible by different task groups [9]. *cgroups* enable the organization of tasks into hierarchical subgroups with various configurable runtime properties, that allow a dynamic resources redistribution among the subgroups [19].

Looking at the RT support for container-based virtualization, three major directions are aiming to improve RT behavior of containers: hard RT co-kernel that co-exists with a standard Linux Kernel [20, 21, 22], solutions based on the *preempt_rt* patch for Linux that aims to minimize the latency in the Kernel [23, 24, 25], and solutions that employ hierarchical scheduling [7, 18, 8]. The RT properties of container-based virtualization depend on the underlying OS. In this paper, we consider the general-purpose OS Linux. While Linux was not designed for RT operation, there are considerable efforts to improve the time determinism on several levels, e.g., introducing RT scheduling as a standard kernel feature and providing time-predictable kernel behavior through

full preemption. On the task scheduling level, there are RT scheduling policies in the Linux kernel: First-In-First-Out and Round Robin combined with priority queues to provide different priority levels for tasks, and Earliest Deadline First/Constant-Bandwidth Server that prioritizes tasks dynamically according to their deadlines.

RT scheduling support for containers has been added in [18] through the implementation of hierarchical scheduling combining standard fixed priority scheduler (*SCHED_FIFO* or *SCHED_RR* scheduling policy) and *SCHED_DEADLINE*, consistent with the multiprocessor periodic resource analysis from [26]. The kernel is extended with a reservation-based scheduling policy in which each vCPU is assigned a quota and a period, bounding the execution of the vCPU to the respective quota in each time interval of length equal to the period. On the container level, the global *SCHED_DEADLINE* policy selects at each time instant the container with the earliest deadline, while at the task level, the *SCHED_FIFO*/*SCHED_RR* policy is used to schedule tasks within containers. Once there is no task to be scheduled by the *SCHED_FIFO*, other BE tasks, are executed via the default scheduling policy. This enables the co-existence of RT and BE containers. We use this hierarchical patch as the implementation basis for our environment to host containers in compute nodes.

Container orchestrators automate the deployment, management, and scaling of containers in clusters of heterogeneous computing nodes. The main functionality of container orchestrators is the placement of containers in a cluster of compute nodes following placement policies and user-supplied placement constraints. The goal is to choose an optimal compute node to start the requested containers on. The orchestrator matches the resource requirements with the resource capacity of the nodes, e.g., CPU, memory, and disk storage capacity, and applies strategies to maximize the performance (e.g., the highest spread of containers). Additionally, orchestrators address fault-tolerance of the deployed containers, scaling or removing containers based, load balancing, container health monitoring, and efficient resource utilization. There are several container orchestrators available: Kubernetes and Docker Swarm, None of them support the deployment of containers based on their timing requirements [10]. Closest to our work is [27], where Xi et al. utilizes OpenStack to orchestrate RT VMs.

## 7.3 Orchestration of real-time containers

This paper introduces a solution to enable the orchestration of RT containers so that the RT requirements are taken into account during the scheduling process. We define a set of scheduling policies, run-time monitoring mechanisms, performance metrics, and an implementation that supports the deployment and execution of RT containers. The RT container orchestrator provides the following functionalities:

- *Placement of RT and BE containers*: The orchestrator places the RT containers according to their RT requirements (i.e., quota and budget). The orchestrator prevents the over-reservation of the resources at the container scheduling level by performing a utilization-bound schedulability test.
- *Run-time monitoring of RT QoS of the containers*: The orchestrator continuously monitors resource usage and the delivered QoS expressed by a set of metrics of the containers deployed in the compute nodes administrated by the orchestrator. As the OS does not enforce strong container isolation, there may be an interference with other noise-perturbing containers that may influence the timing behavior of RT containers. The orchestrator takes the information into account in the next scheduling decisions.

### 7.3.1 System Model

In this section, we define our system model including infrastructure, compute node, container, and task elements.

**Infrastructure**: We consider a system $\mathcal{S}$ consisting of a container orchestrator $F$ and $n$ connected compute nodes denoted by $f_1, \ldots, f_n$.

**Compute node**: Each node $f_j$ offers memory and storage resources of a given capacity, denoted with $f_j^M$ and $f_j^S$, respectively. Each compute node $f_j$ is capable of hosting a mixture of RT containers and BE containers. We define the set of RT and BE containers on node $f_j$ with $\Pi_j^{rt}$ and $\Pi_j^{be}$, respectively. We introduce the OSLM property of a node $f_j$, denoted by $OSLM_j$, to quantify its performance.

**Container**: Each container $\pi_k$ has memory ($\pi_k^M$) and storage ($\pi_k^S$) demands. An RT container $\pi_k \in \Pi_j^{rt}$ has RT interface expressed as $(P_k, Q_k)$ where $Q_k$

is a CPU quota over period $P_k$. We introduce the metric of a container $\pi_k$, denoted by $CLM_k$, to capture the performance of RT containers.

**Task**: Each container $\pi_k$ accommodates a set of tasks denoted by $\mathcal{T}_k$. Each RT task $\tau_i$ is defined by the tuple $\langle a_i, C_i, T_i, D_i \rangle$, where $a_i$ represents the release or arrival time (i.e., the time relative to the period when the task becomes active), $C_i$ is the worst-case execution time bound, $T_i$ is the period, and $D_i$ is the relative deadline of the task. We use two functions $L_i(t_0, t_1)$, and $R_i(t_0, t_1)$ to capture the maximum lateness and maximum response time of the task $\tau_i$ in the interval $[t_0, t_1)$. The lateness represents the delay of the task's completion with respect to its deadline, while the response time measures the difference between the finishing and arrival time of a task. The functions return a set of lateness and response time values, respectively, corresponding to the task instances executed within the given time interval. We also use a counter defined as $\mu_i(t_0, t_1)$ to capture the number of deadlines misses for task $\tau_i$ in the time interval $[t_0, t_1)$. Tasks assigned to BE containers do not have any timing requirements.

There are methods [28] to abstract the timing requirements of a set of RT tasks under certain scheduling algorithms into a periodic resource model in a form similar to the RT interface $(Q_k, P_k)$ of the containers. If the appropriate scheduling mechanisms are in place, the abstracted resource guarantees that when the resource receives an allocation of $Q_k$ over a period $P_k$, all RT tasks within the resource will meet their RT requirements [28].

### 7.3.2 Performance Metrics

There are several metrics, collectively defined as OSLM [29], to evaluate the performance of a system in terms of task execution. Such metrics are useful to estimate the suitability of the system to run RT tasks. E.g., Interrupts with a non-preemptable section that can influence the RT performance of the system [30], CPU Utilization, number of handled interrupts per second, number of I/O requests per second, and amount of data read/written. There is a number of tools for collecting performance data [31]: *mpstat*, *iostat*. These tools collect the performance data of tasks from *proc* and *cgroups* directories to estimate the OSLM.

On the container level, there is a lack of measurement methodology, tools, and best practices, as well as a lack of metrics on the characterization of the container overhead [31]. Available tools, e.g., *docker stats* and *cAdvisor* allow

us to estimate the basic set of container-related metrics (e.g., CPU and memory utilization). However, when considering the RT QoS evaluation of containers, the available tools are lacking such capabilities. In this paper, we define CLM that helps to evaluate the performance of RT containers.

### 7.3.3 Container Level Metrics

Several metrics are used to evaluate the timeliness [11] of tasks running in the system, which we adapt to assess the RT performance of containers. The following properties characterize the RT performance of a task:

- *Number of deadline misses*: represents the number of times that a deadline of a task was exceeded.
- *Lateness*: represents the delay of a task with respect to its deadline [11]. If the task finishes before its deadline, the lateness is negative.
- *Response time*: represents the difference between the finishing time and the start time of a task. [11]

We define CLM to evaluate the RT performance of tasks running in the respective container. For each of the tasks in a container $\pi_k$, the CLM are:

- *Number of deadline misses*: characterizes the total number of deadline misses of tasks inside of the container $\pi_k$ between time $t_0$ and $t_1$:

$$\mathcal{M}_k(t_0, t_1) = \sum_{\tau_i \in \mathcal{T}_k} \mu_i(t_0, t_1).$$

- *Maximum lateness*: characterizes the maximum lateness encountered by a task inside the container between time $t_0$ and $t_1$:

$$\mathcal{L}_k^{\max}(t_0, t_1) = \max_{\tau_i \in \mathcal{T}_k} \{L_i(t_0, t_1)\}$$

- *Maximum response time*: characterizes the maximum response time encountered by a task inside the container between time $t_0$ and $t$:

$$\mathcal{R}_k^{\max}(t_0, t) = \max_{\tau_i \in \mathcal{T}_k} \{R_i(t_0, t_1)\}.$$

We express CLM within the observation interval $[t_0, t_1)$ (usually from system start at $t_0 = 0$ until the current time) as follows:

$$CLM_k(t_0, t_1) = [\mathcal{M}_k(t_0, t_1), \mathcal{L}_k^{\max}(t_0, t_1), \mathcal{R}_k^{\max}(t_0, t_1)]$$

## 7.4 Design of the RT Orchestrator

The orchestration system is based on the master-minion architecture that consists of a master node and a set of minion compute nodes connected in a cluster as described in [10]. The core of the system is the master node that makes global decisions about the cluster; it receives users' requests for container deployments, continuously monitors states of compute nodes in the cluster and schedules containers on computing nodes. The master node's functionality can be distributed across several physical machines to avoid a single point of failure [10].

The compute nodes, depicted in Fig. 7.1, provide an environment for hosting containers and run a node agent that communicates with the master node through defined APIs. The node agent takes container deployment specifications defining container requirements and deployment parameters.

### 7.4.1 RT Extension of the Master Node

The master node depicted in Fig. 7.2 is a central point in the architecture; it accepts user-defined container deployment specification enhanced with RT interface and task annotations. It provides mechanisms for admission control and scheduling of containers. Additionally, it continuously collects performance metrics of compute nodes and containers. In the following text, we elaborate on the proposed enhancements:

#### Container Deployment Specification

Each container deployment specification, which is supplied to the master node via a dedicated API, contains the specification of the RT interface and the container annotation. The RT interface specifies the CPU reservation $(P_k, Q_k)$ of the respective container following the periodic resource model of the hierarchical scheduling framework (c.f. [32]). The container specification contains the description of the tasks inside the container to compute the CLM during run-time.

### RT Resource Monitor

RT Resource Monitor stores the OSLM and CLM from individual compute nodes. The data are accessible to RT Admission Controller and RT Container Scheduler to support the scheduling decisions.

### RT Admission Control

The admission control determines if there are nodes in the cluster with enough available resources to accommodate the resource demands of the new container. Moreover, it performs necessary utilization-bound schedulability tests that reject those nodes on which the RT timing requirements cannot be met.

We perform the checks as defined below to decide if a new container, denoted by $\pi_{new}$, can be allocated to a certain node.
1) We check if the available resources (memory, storage), when considering both RT and BE containers, are enough to fit the resource demands of the new container:

$$\forall f_i \in \mathcal{S} : \begin{cases} \pi^M_{new} + \sum\limits_{\pi_k \in \Pi^{rt}_i \cup \Pi^{be}_i} \pi^M_k \leq f^M_i \\ \pi^S_{new} + \sum\limits_{\pi_k \in \Pi^{rt}_i \cup \Pi^{be}_i} \pi^S_k \leq f^S_i \end{cases} \tag{7.1}$$

2) We check that the new container will not make the existing RT containers unschedulable by performing a necessary utilization-based test [11]:

$$\forall f_i \in \mathcal{S} : \frac{Q_{new}}{P_{new}} + \sum_{\pi_k \in \Pi^{rt}_i} \frac{Q_k}{P_k} \leq 1 - \delta_i \tag{7.2}$$

where, $\delta_i$ refers to the system overhead of node $f_i$ (discussed in detail below), which reduces the amount of CPU bandwidth available to the containers/tasks. We remind the reader that $(P_k, Q_k)$ is the RT interface of a RT container $\pi_k$, which defines that the container will be scheduled for at most $Q_k$ time units over a period of $P_k$ time units. Hence, this utilization-based test (c.f. [14]) defines that if the utilization of the RT containers (including the utilization of the new container) exceeds the bandwidth of the CPU, then the RT requirements of the new container cannot be guaranteed. Please note that the check is not sufficient, i.e., if the check is passed, it does not mean that the RT behavior will always be guaranteed since the actual execution of the RT tasks diverges from

the ideal RT schedule due to e.g., timer resolution, scheduling jitter, locking mechanisms (c.f. [30]) or the overhead introduced by the container mechanism.

### RT Container Scheduler

The Container Scheduler decides which of the feasible nodes in the cluster, the feasibility being determined through the admission control tests defined above, to assign the new container to. The decision is based on the CLM and OSLM introduced above, which are constantly monitored at run-time. Additionally, the scheduler might decide to change some properties of already running containers (e.g., the RT interface) to be able to fit the new container on a node. Hence, the problem of where to place new containers according to their RT requirements or which containers to modify can be viewed as a multi-objective optimization problem. While the exact mechanism on how to assign (and re-dimension) containers is out of the scope of this paper, we give a brief discussion on the important aspects of this orchestration problem and leave the definition and solution of this optimization problem for future work. Furthermore, selecting the best mix of metrics and optimization objectives to use is also not in the scope of this paper. However, we will briefly discuss several strategies below.

In terms of the observed OSLM properties, we identify several important aspects and strategies which we briefly discuss below.

The *system overhead* ($\delta$) reduces the amount of CPU bandwidth that the containers can use. We show in the experiment section (Sect. 11.3) that the overhead when co-locating RT and BE containers influences only the BE containers while the RT containers are guaranteed their allotted budget (c.f. Fig. 7.4). We see that the overhead remains constant and jitters around the constant value both when changing the RT container period and when increasing the number of RT containers. Please note that the overhead analysis needs to be extended in future work to create a more accurate overhead model that produces a safe upper bound for admission control. However, even though the overhead model is deduced empirically and not analytically, we can still use it in the admission control since any overhead impact on RT containers will be corrected by the online reconfiguration algorithm.

Another objective may be to perform load balancing on nodes to not over-utilize some of the nodes. This decreases the probability of deadline misses

and lateness for tasks and increases the probability of fitting future container requests.

The optimization function may also select nodes with a low number of context switches as this also influences the system utilization, leaving more CPU bandwidth available for container execution. Furthermore, nodes with a high number of interrupts/sec are more likely to lead to deadline misses due to the irregular nature of interrupt arrivals. Hence, they might not be the best selection for placing RT containers.

In terms of CLM properties, nodes with few (or zero) deadline misses are better candidates for new RT containers. However, adding additional RT containers might increase the number of deadline misses or the lateness/response times of tasks. The container scheduler needs to consider if an increased rate of timing failures is acceptable, depending on the nature of the running RT containers.

When using EDF to schedule containers, the admission test above also becomes necessary, i.e., if the test is passed, the new container can, in theory, be scheduled on the respective node. However, the divergence from the real schedule (described in [30]) can lead to deadline misses and/or negative effects on the lateness and response times of tasks. Hence, the run-time monitoring of the CLM properties can give hints about which containers to re-assign or re-dimension. The observation interval in the CLM properties can be dynamically adjusted to identify which of the new containers has a negative impact on the RT properties of the already running containers.

Feedback-based resource reservation mechanisms can be used to adapt the CLM properties at run-time while keeping hard RT guarantees for all the RT containers [33]. Such mechanisms can be implemented at the orchestration level to compensate for potential unforeseen over-or under-runs, providing a limited impact on the other RT containers. The overhead for feedback-based resource reservation is minimal since it requires implementing an integral control strategy for each container. Such approaches have proven successful also in other domains, such as mixed-critical systems [34].

### 7.4.2 RT Extension of Compute Nodes

We enable RT containers on node-level through the use of the *preempt_rt* patch, hierarchical scheduling of containers [18], hard RT co-kernel, or a combination

of these technologies. The overview of the compute node extension is depicted in Fig. 7.1. A module responsible for the RT related functionality is denoted as *RT manager*. The process of deployment of RT containers is as follows: Upon receiving a deployment request from the orchestrator, the RT manager locate a requested image (either locally or in a remote repository), the container image is fetched, and instantiated with the requested parameters that set-up resources for the container. Concerning the RT functionality, the *RT Manager* offers the following:

- Deployment of RT containers: The containers have to be instantiated in RT mode and must be assigned the requested quota and period.
- Monitoring of RT performance: The monitoring functionality assesses the wellness and performance of the compute node and the containers deployed. There are two monitoring parts: the OSLM monitor and the CLM monitor. The OSLM monitor collects data through the *proc* and *cgroup* file-system, which contain information related to interrupts, memory usage, and CPU utilization. The CLM monitor evaluates the timelines of the containerized tasks in the RT containers.
- Reporting of the performance metrics to the orchestrator: The compute nodes report the OSLM and CLM to the master node, which uses the collected data for the admission control and container scheduling and, additionally, it can take decisions on whether to re-allocate and/or re-assign resources of the containers.

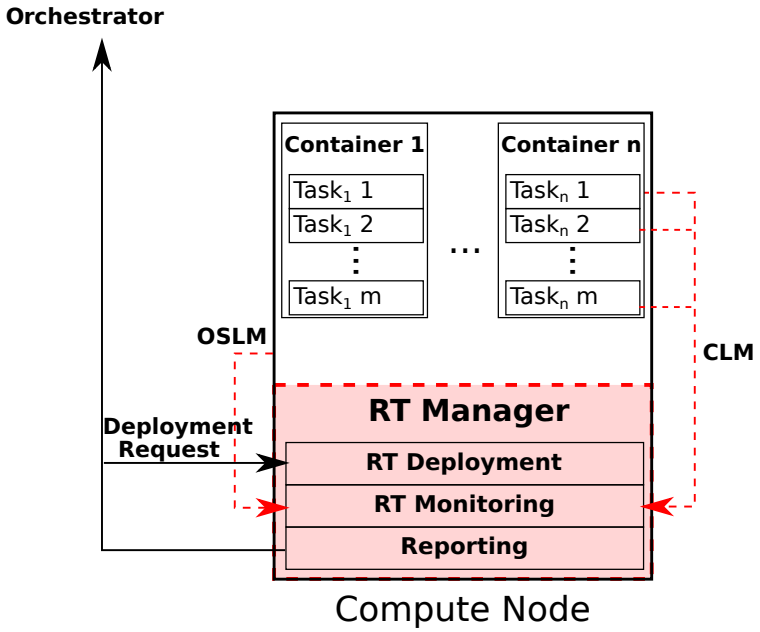



Figure 7.1: Compute node.

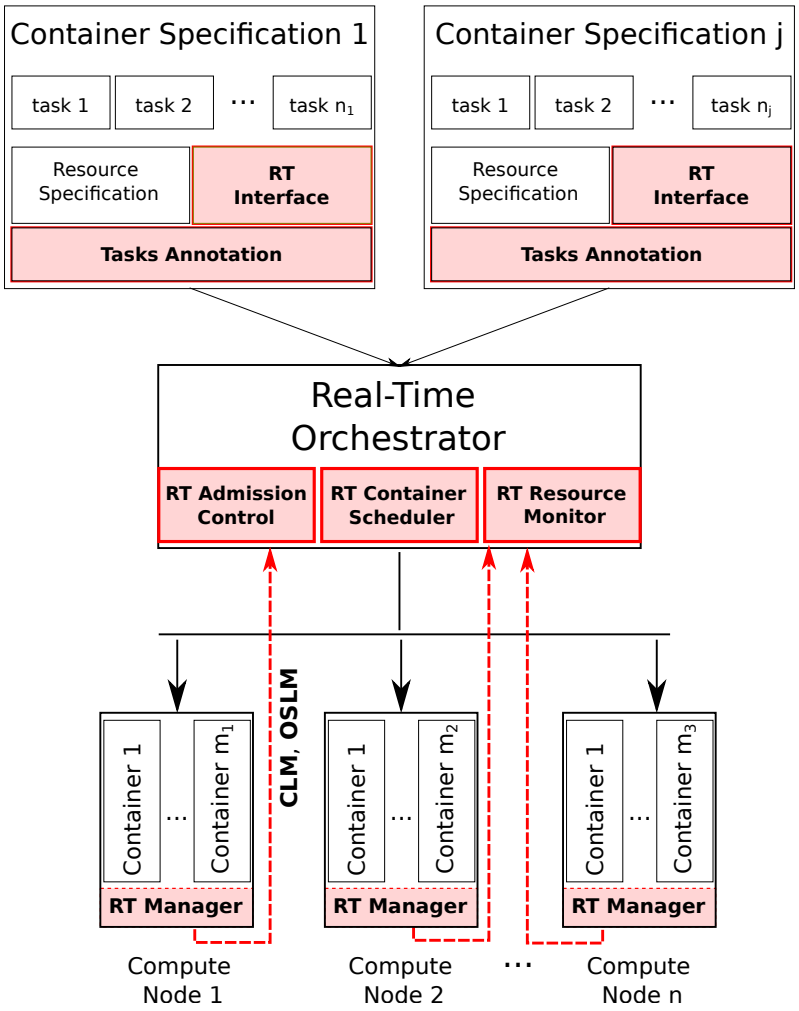



Figure 7.2: A high-level container orchestrator architecture enhanced with RT capabilities.

## 7.5 Implementation

In order to show the feasibility of the proposed system, we have extended the existing open-source orchestrator Kubernetes with the ability to schedule containers onto compute nodes while taking into account their RT requirements. The RT behavior of containers on compute nodes is enabled by using the hierarchical patch[1] presented in [18], which we have extended with the monitoring capabilities. The system allows users to define RT requirements of the containers which need to be deployed (i.e., quota and period) and ensures that at the container scheduling level, the allocation to compute nodes respects the given requirements and does not lead to an overload in the respective node. Additionally, as the containers do not provide strong resource isolation, the system provides run-time monitoring of the containers' performance. The implemented extension consists of the following components:

- *The RT Scheduler Extender* on the Master node is an extension of the Kubernetes control plane, which provides admission control and scheduling of RT containers onto compute nodes in the cluster.
- *The RT Manager* on compute nodes provides functionality to deploy RT containers and periodically evaluate and report the RT performance to the Master node.

The architecture of the Kubernetes extension is described in Fig. 7.3. The Kubernetes Master receives a deployment request to deploy a set of containers (denoted as a Pod in the context of Kubernetes). A new Pod is placed in a queue with other unscheduled pods, and the Kubernetes scheduler (*kube-scheduler*) periodically checks the queue. If there is an unassigned pod, the Kubernetes scheduler attempts to place the Pod in a suitable node. First, the scheduler filters out infeasible nodes with insufficient available resources (e.g., insufficient memory or storage) as described in Section 7.4.1. Subsequently, so-called custom webhooks are triggered during each schedule polling cycle. The webhooks permit to attach custom actions to scheduling events (e.g., filtering and prioritizing events). We have implemented a custom *rt-filter* webhook that takes into account the utilization of the node as well as the RT interface as requested in the container deployment specification. If the utilization, including the new RT container, is above a certain level, the node is filtered out as infeasible and not used for hosting the container. In this way, we avoid overloading the host.

[1] Available at `https://github.com/lucabe72/LinuxPatches/tree/HCBS`

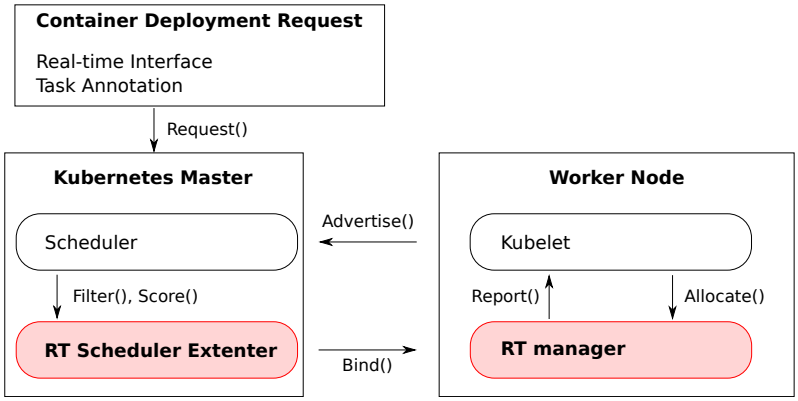



Figure 7.3: Scheduling process of Kubernetes.

The *RT Scheduler Extender* performs secondary filtering as described in Section 10.6 and ranks the nodes with the custom *rt-scoring* webhook. The *rt-scoring* ranks the nodes according to their suitability to host the RT container. For this work, the scoring is computed as the remaining unreserved CPU capacity. However, the rt-scoring step can incorporate additional properties of the compute node given in the OSLM and CLM to minimize the number of, e.g., deadline misses on compute nodes. As the orchestration is not part of the current paper, we leave this extension to future work.

The Pod is then assigned to the node with the highest score. The node agent on the compute node identifies that the Pod is assigned to its node and deploys it with the given RT parameters. The *RT Manager* continuously monitors the node's state beyond the understanding of the default Kubernetes monitoring metrics and reports them back to the master node. The RT manager monitors the previously described CLM and OSLM: i.e., the number of context switches, interrupts per second, I/O access, as well as task deadline misses, response time, and lateness.

As an input, the Kubernetes master accepts a Pod deployment request that contains the deployment specification of a group of one or more containers. If there are multiple containers in a pod, these are assigned and scheduled on the same node and run in a shared context (i.e., sharing memory and network). As a simplification, we assume that each Pod contains at most one RT container. We amend the deployment configuration (stored in the annotation part of the

deployment file) to contain the RT interface $(Q_k, P_k)$ as well as the list of tasks and their $\langle C_i, T_i, D_i \rangle$ parameters.

The *RT manager* runs as a Kubernetes daemon set and provides functionality for run-time monitoring and reporting of the performance of the RT containers and the system. A daemon runs on every compute node in the cluster. The monitoring part continuously monitors OSLM and CLM. The OSLM is computed by utilizing the Linux special *procfs* file system (*/proc* and */cgroups*) that contain information about tasks, similar to [31]. The CLM is derived from the custom tracepoints that we injected into the schedulers implementing the *SCHED_FIFO* and *SCHED_DEADLINE* policies in the Linux Kernel. The following events are recorded and periodically evaluated by the daemon:

- Container Started: The container is in a running state, the quota/period has been allocated, and the container's tasks are ready to run.
- Container Throttled: The container used more CPU quota than the allocated one and therefore, the scheduler throttled the container.
- Task Instance started: The start of $j^{th}$ instance of task $\tau_i$.
- Task Instance finished: The end of $j^{th}$ instance of task $\tau_i$.

From these tracepoints, we compute deadline misses, lateness, and response times of tasks within the RT containers.

## 7.6 Evaluation

In this section, we show the system's behavior for co-located RT and BE containers on a single compute node. The set of experiments illustrates the distribution of the CPU time amongst co-located containers. Tables 7.1, 7.2 and 7.3 show the feasibility of having a mixture of RT and BE containers on a single compute node. We change the reservation period and budget from values 0.1ms to 1000ms and measure the overhead (described below). We show the behavior of low utilization containers (10%) and high utilization (90%). We investigate if RT containers with very short periods introduce significantly more overhead than those with large periods. In the experiments, we instantiate an RT container and a heavy load BE container. The experiments for Table 7.1, 7.2 use containers that run CPU-intensive operations. We measure the actual time that the containers spend on the CPU. The rest of the CPU that is not used by any container is considered as an overhead. Table 7.3 shows a similar experiment

Table 7.1: RT containers (10% utilization) on a single core with noise generated by 10 CPU-intensive containers.

| | **(1ms, 0.1ms)** | **(10ms, 1ms)** | **(100ms, 10ms)** | **(1000ms, 100ms)** |
|---|---|---|---|---|
| **RT** | 10.0607% ± 0.00193% | 10.0021% ± 0.00002% | 9.9983% ± 0.00002% | 9.9978% ± 0.00002% |
| **Non-RT** | 88.7870% ± 0.00406% | 88.8963% ± 0.00280% | 88.8713% ± 0.00222% | 88.8906% ± 0.00264% |
| **overhead** | 1.1522% ± 0.0035% | 1.1016% ± 0.0028% | 1.1304% ± 0.0022% | 1.1115% ± 0.0025% |

where the BE container runs *stress-ng* to generate an excessive workload aiming to affect the assigned CPU time of the RT containers. The stress-generating BE containers execute 10 CPU-intensive threads, 10 HDD-intensive threads, 10 threads generating I/O stress, and 10 threads generating context switches.

We use an Intel i5 machine with 8GB RAM running Debian Linux, and Kernel 5.2. patched with Hierarchical Scheduling Patch [18], and Docker v20.10.

We consider the overhead to be part of the CPU capacity that is not used for any computation of the containerized tasks. It is caused by system-related tasks, context switches or docker-related processes, etc. In the experiments, we execute RT and BE containers simultaneously. Each of the containers executes a loop with a CPU-heavy computation. In theory, the containerized processes should fully utilize the processor; however, the full CPU capacity is not used exclusively for these processes. We can see that the overhead stays the same even when using an RT container for a very short period.

To investigate the overhead, we utilize *systemTap*, which is an instrumentation framework for Linux-based kernels. SystemTap allows instrumenting Linux events with user-defined code in form of loadable kernel modules. We monitor the following events in the Linux kernel:

- *scheduler.cpu_on*: The process is beginning execution on a CPU.
- *scheduler.cpu_off*: The process leaving the CPU.

From the recorded events, we are getting the total measurement time (from the first event to the last one) and each containerized task's total time.

The utilization test in Fig. 7.4 shows the distribution of CPU time on a single core amongst multiple RT and BE containers when increasing the number of RT containers from 1 up to 7. Each RT container has a RT demand of 10% of the CPU bandwidth (RT period = 100ms, RT budget = 10ms). The experiments indicate that hierarchically scheduled containers using the Hierarchical Scheduling Patch [18] can keep the allocated CPU resource even when com-

Table 7.2: RT containers (90% utilization) on a single core with noise generated by 10 CPU-intensive containers.

| | **(1ms, 0.9ms)** | **(10ms, 9ms)** | **(100ms, 90ms)** | **(1000ms, 900ms)** |
|---|---|---|---|---|
| **RT** | 89.9037% ± 0.00031 | 89.8597% ± 0.00042% | 89.8818% ± 0.00066% | 89.7780% ± 0.00316% |
| **Non-RT** | 9.0345% ± 0.00212 | 9.0368% ± 0.00193% | 9.0349% ± 0.00045% | 9.0658% ± 0.00164% |
| **overhead** | 1.0618% ± 0.0022 | 1.1035% ± 0.0019% | 1.0833% ± 0.0005% | 1.1563% ± 0.0018% |

Table 7.3: RT containers (10% utilization) on one core with noise generated by BE containers executing *stress-ng*.

| | **(1ms, 0.1ms)** | **(10ms, 1ms)** | **(100ms, 10ms)** | **(1000ms, 100ms)** |
|---|---|---|---|---|
| **RT** | 10.0508% ± 0.00045% | 10.0065% ± 0.00005% | 9.9956% ± 0.00006% | 9.9956% ± 0.00002% |
| **Non-RT** | 89.0251% ± 0.00091% | 88.9333% ± 0.00034% | 88.8779% ± 0.00296% | 88.9798% ± 0.00142% |
| **overhead** | 0.9241% ±0.0037% | 1.0602% ±0.0031% | 1.1265% ±0.0024% | 1.0246% ± 0.0029% |

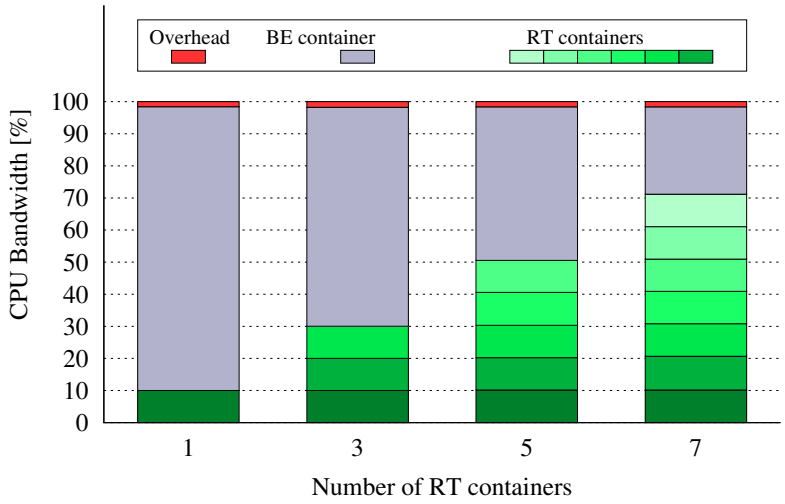



Figure 7.4: Distribution of CPU in a multi-container environment.

peting with BE containers under heavy load. Moreover, the RT containers keep their reserved resource allocation (CPU budget over the periods) with very low run-time jitter on a single core. The system overhead does not influence the RT containers but reduces the remaining CPU utilization used by BE containers.

The experiments indicate that RT containers maintain the target resource reservation even in the presence of heavy RT and BE load. The overhead (indicated in red in Fig. 7.4) remains relatively constant when increasing the number of containers scheduled on the same node.

## 7.7 Conclusion

In this paper, we have introduced a container orchestration for RT systems designed to prevent over-reservation of the CPU at the container scheduling level. Additionally, we considered the weak isolation inherent in container-based virtualization (e.g., the effects of the use of shared resources or context switches), which can lead to an interference and thereby harm temporal guarantees of RT containers. We have proposed metrics for measuring the RT performance at the container and node levels, which can be used for both the admission control and the online re-configuration of container deployment in order to guarantee timely behavior. We have implemented a scheduler extension and node monitoring system on top of an orchestrating system Kubernetes and have shown the feasibility of co-locating RT and BE containers on the same node in a series of experiments.

We aim to address the optimization problem arising from the orchestration needs in RT containerized systems in future work. This orchestration includes both the admission/allocation of new container requests as well as the online resource re-dimensioning of already deployed RT containers in case of runtime performance drops.

# Bibliography


[1] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *W. on Mob. cloud comp.*, 2012.

[2] Shaik Mohammed Salman, Václav Struhár, Alessandro V. Papadopoulos, Moris Behnam, and Thomas Nolte. Fogification of industrial robotic systems: Research challenges. In *W. on Fog Comp. and IoT (Fog-IoT)*, 2019.

[3] M. S. Shaik, V. Struhár, Z. Bakhshi, V. L. Dao, N. Desai, A. V. Papadopoulos, T. Nolte, V. Karagiannis, S. Schulte, A. Venito, and G. Fohler. Enabling fog-based industrial robotics systems. In *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, volume 1, pages 61–68, 2020.

[4] F. Ramalho and A. Neto. Virtualization at the network edge: A performance comparison. In *IEEE Int. Symp. A World of Wirel., Mob. and Multim. Net. (WoWMoM)*, 2016.

[5] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and Linux containers. In *IEEE Int. Symp. on Perf. Analysis of Syst. and Soft. (ISPASS)*, 2015.

[6] Miguel Gomes Xavier, Marcelo Veiga Neves, and Cesar Augusto Fonticielha De Rose. A performance comparison of container-based virtualization systems for mapreduce clusters. In *Euromicro Int. Conf. on Par., Distr., and Netw. Proc. (PDP)*, 2014.

[7] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Alessio Balsini, and Carlo Vitucci. Virtual network functions as real-time containers in private clouds. In *IEEE Int. Conf. on Cloud Comp. (CLOUD)*, 2018.

[8] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Alessio Balsini, and Carlo Vitucci. Reducing temporal interference in private clouds through real-time containers. In *IEEE Int. Conf. on Edge Comp. (EDGE)*, 2019.

[9] Václav Struhár, Moris Behnam, Mohammad Ashjaei, and Alessandro Vittorio Papadopoulos. Real-time containers: A survey. In *W. on Fog Comp. and IoT (Fog-IoT)*, 2020.

[10] Maria A Rodriguez and Rajkumar Buyya. Container-based cluster orchestration systems: A taxonomy and future directions. *Software: Practice and Experience*, 2019.

[11] Giorgio C. Buttazzo. *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2011.

[12] Chang Zhao, Yusen Wu, Zujie Ren, Weisong Shi, Yongjian Ren, and Jian Wan. Quantifying the isolation characteristics in container environments. In *Net. and Par. Comp. (NPC)*, 2017.

[13] M. G. Xavier, I. C. De Oliveira, F. D. Rossi, R. D. Dos Passos, K. J. Matteussi, and C. A. F. D. Rose. A performance isolation analysis of disk-intensive workloads on container-based clouds. In *Euromicro Int. Conf. on Par., Distr., and Netw. Proc. (PDP)*, 2015.

[14] C. L. Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM*, 1973.

[15] S. Xi, J. Wilson, C. Lu, and C. Gill. RT-Xen: towards real-time hypervisor scheduling in Xen. In *ACM Int. Conf. on Emb. Soft. (EMSOFT)*, 2011.

[16] S. He, L. Guo, Y. Guo, C. Wu, M. Ghanem, and R. Han. Elastic application container: A lightweight approach for cloud resource provisioning. In *IEEE Int. Conf. on Adv. Inform. Netw. and Appl. (AINA)*, 2012.

[17] Thuy Linh Nguyen and Adrien Lebre. Conducting thousands of experiments to analyze vms, dockers and nested dockers boot time. Research Report RR-9221, INRIA, 2018.

[18] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the Linux kernel. *ACM SIGBED Review*, 11 2019.

[19] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Benchmarking and resource measurement. In *Model Checking Software*, 2015.

[20] Philippe Gerum. Xenomai - implementing a RTOS emulation framework on GNU/Linux. *White Paper, Xenomai*, 2004.

[21] Timur Tasci, Jan Melcher, and Alexander Verl. A container-based architecture for real-time control applications. In *IEEE Int. Conf. on Eng., Tech. and Innov. (ICE/ITMC)*, 2018.

[22] Florian Hofer, Martin Sehr, Antonio Iannopollo, Ines Ugalde, Alberto Sangiovanni-Vincentelli, and Barbara Russo. Industrial control via application containers: Migrating from bare-metal to IAAS. In *IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom)*, 2019.

[23] Alexandru Moga, Thanikesavan Sivanthi, and Carsten Franke. OS-level virtualization for industrial automation systems: are we there yet? In *ACM Symp. on Applied Computing (SAC)*, 2016.

[24] Thomas Goldschmidt, Stefan Hauck-Stattelmann, Somayeh Malakuti, and Sten Grüner. Container-based architecture for flexible industrial control applications. *J. of Syst. Architecture*, 2018.

[25] Thomas Goldschmidt and Stefan Hauck-Stattelmann. Software containers for industrial control. In *Euromicro Conf. on Soft. Eng. and Adv. Appl. (SEAA)*, 2016.

[26] Arvind Easwaran, Insik Shin, and Insup Lee. Optimal virtual cluster-based multiprocessor scheduling. *Real-Time Syst.*, 2009.

[27] Sisu Xi, Chong Li, Chenyang Lu, Christopher D. Gill, Meng Xu, Linh T.X. Phan, Insup Lee, and Oleg Sokolsky. Rt-open stack: Cpu resource management for real-time cloud computing. In *2015 IEEE 8th International Conference on Cloud Computing*, 2015.

[28] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *IEEE Real-Time Syst. Symp. (RTSS)*, 2003.

[29] M. Jägemar, A. Ermedahl, S. Eldh, and M. Behnam. A scheduling architecture for enforcing quality of service in multi-process systems. In *IEEE Int. Conf. on Emerging Tech. and Factory Aut. (ETFA)*, 2017.

[30] L. Abeni, A. Goel, C. Krasic, J. Snow, and J. Walpole. A measurement-based analysis of the real-time performance of Linux. In *IEEE Real-Time and Emb. Tech. and Appl. Symp. (RTAS)*, 2002.

[31] Emiliano Casalicchio and Vanessa Perciballi. Measuring docker performance: What a mess!!! In *ACM/SPEC on Int. Conf. on Perf. Eng. (ICPE)*, 2017.

[32] Insik Shin, Arvind Easwaran, and Insup Lee. Hierarchical scheduling framework for virtual clustering of multiprocessors. In *Euromicro Conf. on Real-Time Syst. (ECRTS)*, 2008.

[33] Alessandro Vittorio Papadopoulos, Martina Maggio, Alberto Leva, and Enrico Bini. Hard real-time guarantees in feedback-based resource reservations. *Real-Time Syst.*, 2015.

[34] Alessandro Vittorio Papadopoulos, Enrico Bini, Sanjoy Baruah, and Alan Burns. AdaptMC: A control-theoretic approach for achieving resilience in mixed-criticality systems. In *Euromicro Conf. on Real-Time Syst. (ECRTS)*, 2018.

# Chapter 8

# Paper C: Resource Adaptation for Real-Time Containers Considering Quality of Control

*Václav Struhár*, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos, Silviu S. Craciunas

In IEEE 28th International Conference on Emerging Technologies and Factory Automation (ETFA 2023).

**Abstract**

Container-based virtualization has become a promising deployment model for industrial applications mainly due to its benefits, such as providing support for co-located applications in heterogeneous environments. However, such facilitation brings challenges, including full temporal isolation among real-time applications and support for time-critical applications. In this paper, we tackle such challenges, in particular when the applications are time-sensitive Control Applications. The literature suggests that flexible timing constraints for Control Applications are beneficial in responding to disturbances and minimizing response deviation. Therefore, we propose a mechanism to support such a runtime adaptation in container-based virtualization. To show the performance of the proposed mechanism, we implement our approach on a Linux-based hierarchical scheduling platform, and we evaluate it for a Control application.

## 8.1 Introduction

The deployment of virtualized applications in heterogeneous environments with minimal overhead and near-native performance is made possible by container-based virtualization, which shows promise in industrial settings [1, 2, 3, 4]. Container-based virtualization is a lightweight virtualization technology that allows applications to run in isolated environments, sharing the host operating system's kernel, enabling efficient resource utilization and easy deployment. Container-based virtualization brings several benefits, including easier application deployment and management, optimized hardware utilization, and increased application flexibility and scalability. This technology also enables the development of microservices-based applications, facilitating agile development, efficient resource utilization, and improved fault isolation. However, despite its advantages, container-based virtualization can pose challenges for applications that require stringent temporal behavior and predictability. This shortcoming can hinder the utilization of containers for time-sensitive controllers that have timing requirements.

Industrial domains often leverage controllers to control the behavior of various systems, such as industrial robots and vehicles. Traditionally, controllers are designed as digital controllers with fixed timing behavior (e.g., fixed periods and deadlines for the control) [5]. Several works [6, 7] show that flexible timing constraints are beneficial in responding to disturbances and minimizing response deviations. Moreover, flexibility can lead to better resource utilization, both in computation and communication. Similarly, in the control community, event-based controllers [8] have also been proposed with the idea of computing new control actions only when necessary, i.e., triggered by specific events, leading to lower utilization of the computing and communication resources.

The implementation of controllers with flexible timing constraints is not straightforward in container-based virtualization that uses static resource reservation. The change in timing constraints must be reflected in the resource reservation for the real-time container hosting the virtualized controller. Additionally, container-based virtualization is known for its imperfect temporal isolation, which leads to timing disturbances in virtualized applications. These challenges in the current container-based virtualization technologies hinder a full-fledged utilization of them in time-sensitive control systems where the sys-

tem dynamics have to be considered for a better Quality of Control (QoC).

This paper tackles the challenges mentioned above by proposing a resource adaptation mechanism for real-time containers to improve the QoC [9] for virtualized controllers while minimizing the resource utilization for computing new control actions. The main goal of the proposed mechanism is to dynamically adjust the timing constraints of a flexible controller together with the adaptation of the resource reservation for real-time containers.

The resource reservation has to reflect the change in the timing constraints in the sense that the shorter control period requires a higher resource reservation and vice versa for the longer control period. Adjustment of the timing constraints and resource reservation is done jointly to obtain improved resource utilization and, at the same time, to improve the QoC. In this setup, each control function resides in a real-time container. The proposed mechanism is based on existing work on flexible control, e.g., [6, 7], to be used together with dynamic resource reservation for real-time containers. The concrete contributions of this paper are as follows:

- We propose a mechanism to dynamically adapt the Control Application's timing constraints, including the control sampling rates, together with the resource reservation of real-time containers. Adjustment of timing constraints may lead to improved QoC [6].
- We implement the proposed mechanism in the hierarchical control group scheduler patch (HCBS) [10] on Linux. This enables Docker containers to host controllers with flexible timing constraints.
- We experimentally evaluate the performance of the proposed mechanism on an example of a Control Application. We demonstrate that such an adaptation, together with the resource reservation in real-time containers, will lead to better responses to disturbances in the Control Application, as well as minimize response deviations.

This paper is structured as follows. Section 10.3 reviews the related literature. Section 8.3 presents the technical background on resource adaptation for real-time containers and on control using flexible constraints. Section 8.4 proposed an adaptive virtualized controller, while Section 11.3 evaluates the feasibility of the adaptive virtualized controller. Finally, in Section 11.4 we conclude the paper and present an outlook on future work.

## 8.2 Related Work

There have been several proposals related to support for real-time applications in container-based virtualization that can be a foundation for the virtualization of control applications. Moreover, several works have addressed the challenges of dynamic resource reservation for container-based virtualization. In addition, the field of control theory focuses on flexible timing constraints. In this section, we briefly present these proposals to position the contribution of this paper.

**Resource adaptation for containers:** Several works focus on resource adaptation (also known as vertical scaling) for containers in the presence of performance losses. For instance, the work in [11] proposes a framework that, through a controller hierarchy, dynamically adjusts the resources of real-time containers to match the required performance levels. In this work, the HCBS patch [10] is utilized to implement the adaptation mechanisms. Moreover, the work in [12] proposes ElasticDocker, a system that automatically scales Docker containers vertically based on the current workload of containerized applications. ElasticDocker monitors the performance of containers at run-time and dynamically changes resource reservations to adapt to the dynamic changes via a threshold-based algorithm. The work presented in [13] proposes metrics to measure the performance of real-time containers. The paper describes a container orchestration framework based on Kubernetes that uses real-time metrics for container placement decisions. Rossi et al. [14] uses reinforcement learning to enable adaptive runtime deployment without manually tuning container-based applications using horizontal and vertical elasticity. The work of Shekhar et al. [15] proposes a data-driven approach by machine learning to create predictive system performance runtime models that can be adapted to workload changes.

**Controllers with flexible timing constraints:** The work in [6] argues that static timing constraints hinder controllers from adapting to changing system dynamics, resulting in sub-optimal results since higher execution rates that can more quickly react to disturbances waste resources when the system is at an equilibrium. Furthermore, the work in [16] argues that there is a need to add flexibility in real-time control loops, since, previously, systems have been based on static analysis and design under the assumption that the controllers execute in a predictable (hardware and software) environment. However, the authors in [16] argue that hardware platforms and operating systems tend to

be commercial-off-the-shelf (COTS) products with poor real-time specifications that are typically tailored to improve average-case performance rather than predictable worst-case performance.

Our work aims to bridge the gap between these two distinguished research areas of dynamic resource adaptation for container-based virtualization and control with flexible timing constraints to better utilize computational resources.

## 8.3 Technical Background

This section aims to provide a comprehensive overview of technical concepts related to the control with flexible timing constraints and resource reservation for real-time containers.

### 8.3.1 Control with flexible timing constraints

Classical closed-loop control applications that have a static execution rate assume that the sampling and actuation rates are constant over the lifetime of the system. The chain of (i) measuring the controlled variable, (ii) computing the control signal, and (iii) applying the computed value to the actuators is triggered by a single periodic source of events [17]. Not only is the choice of the rate of sensing, computing, and actuating crucial for the correct operation of the system but this rate must also be maintained between successive execution instances of the control loop in order to ensure system stability and meet the QoC objectives [5]. In principle, it may be possible to use different rates of execution for the control, but typically control system designers adhere to one value that has been computed at design time. Translated into real-time requirements, the sensing, control, and actuation actions are expressed as periodic tasks (with the task period being equal to the period of the control) with implicit or constrained deadlines that are mainly used to restrict the completion times of tasks [6]. Marti et al. [6] demonstrates that being able to adapt the rate of execution and, therefore, the timing constraints for control tasks can lead to better control by allowing faster reactions to transient disturbances. Fig. 8.1 illustrates control period instances and the error (represented by the shaded area) in the closed-loop system that was induced by a perturbation. The figure shows

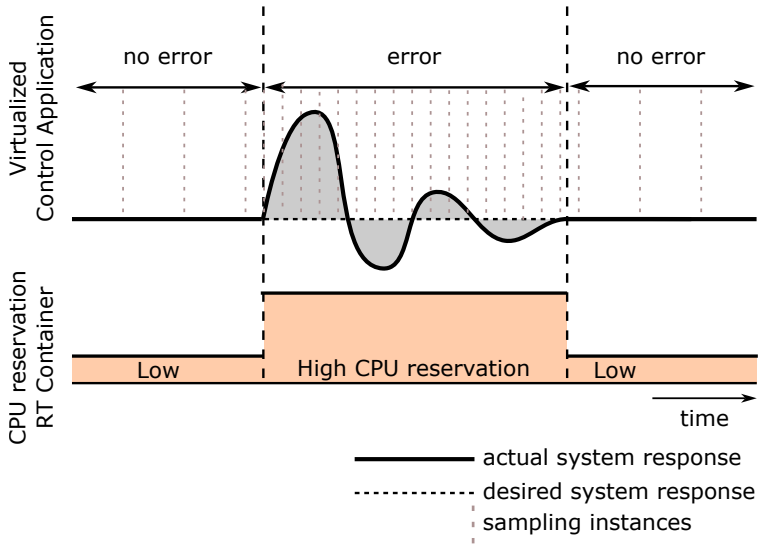



Figure 8.1: The control period and resource reservation change in relation to the entity of the error (adapted from [6]).

that the controller executes a control algorithm more frequently at instants with a high quantity of perturbation.

### 8.3.2 Resource reservation for real-time containers

Techniques such as container-based virtualization have emerged as interesting mechanisms to be used in complex embedded systems [10, 18, 19]. Container-based virtualization uses mechanisms of the host operating system (e.g., cgroups in Linux) to provide spatial isolation of tasks and control their bandwidth access to, e.g., CPU and memory. Improving the timing predictability of container-based virtualization has been attempted, e.g., via PREEMPT_RT patch [20, 21], real-time co-kernel such as Xenomai [22], or RTAI [23]. Several real-time scheduling mechanisms can be employed to ensure temporal isolation amongst virtualized components sharing the same physical resource. One such approach is the Compositional Scheduling Framework [24], which is implemented in Linux as the Hierarchical Control Group Scheduler (HCBS) [10].

HCBS uses a reservation-based algorithm in which at each replenishment period ($P_k$), each container ($k$) is assigned a maximum budget (or runtime) $Q_k$. The scheduler guarantees that the container will be able to execute the reserved budget within the period but not exceed it, thus not interfering with the resource allocation of other co-located containers. HCBS uses static resource reservation that a system designer typically determines before the system starts. Therefore, the system cannot respond to dynamic changes such as changes in workload or the need to execute the controller at a higher (or lower) rate. Additionally, container-based virtualization is prone to timing disturbances due to shared platforms of co-located containers [25, 15, 26, 27]. In instances of changing workloads or interference from other components, static reservation may lead to timing violations. Conversely, not being able to adapt the rate of execution may lead to either insufficient resource usage or a degrading of QoC.

## 8.4 QoC Controller

In this section, we present the design of a QoC controller targeting time-critical Control Applications. We design the QoC controller to automatically change the timing constraint of the virtualized application and, at the same time, adjust the container's resource reservation to the varying QoC metrics expressing the performance of the virtualized Control Application. The overall concept is presented in Fig. 8.2, which consists of four elements. These elements include (i) a real-time (RT) container that encompasses the Control Application, (ii) the Control Application itself that is responsible for controlling the system under control, (iii) the Plant, which is the target system to be controlled by the Control Application, and (iv) the QoC Controller which is responsible for adaptation in the resources and timing constraints for improving the performance of the Control Applications. Following, we describe the elements in more detail.

**Real-time container (RT container)**: Real-time containers provide a separate and secure environment for running software applications, and they are designed to ensure the temporal predictability of the system. We use the HCBS patch [10] for the implementation of real-time containers that allow for a static reservation of computing resources. We use Docker, which is a platform that enables the packaging, configuration, and execution of applications via container-based virtualization technology (e.g., utilizing Linux features cgroups and names-

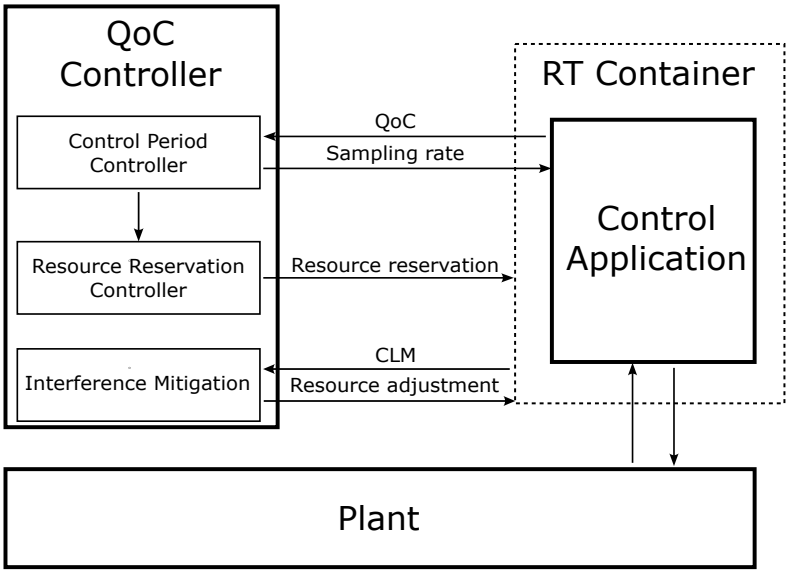



Figure 8.2: System overview containing QoC Controller and its modules, real-time container hosting a Control Application, and a plant.

paces). Developers can bundle applications and their dependencies into containers that can be ported and run on any machine that has Docker installed. This makes it easy to deploy in various environments. In terms of real-time behavior, each of the real-time containers ($k$) has a CPU reservation $Q_k$ that is replenished every period $P_k$, as described above. Once the budget is exhausted, the container will be suspended until the next replenishment period.

In order to ensure the schedulability of the system, the following condition must be fulfilled $\sum_{k=1}^{n} P_k/Q_k \leq 1$, where $n$ is a number of co-located containers in a physical platform. In our case, schedulability refers to all containers being able to execute until their deadlines, which are assumed to be implicit, i.e., they are equal to the container period.

**The Control Application:** The Control Application is implemented as a periodic real-time task that performs computations at predefined intervals (not only a single interval) determined by the system designer [17]. The shorter the interval, the better the QoC, and the more computational power required.

This can be a limiting factor for embedded systems with limited resources. Nevertheless, this is a typical trade-off in real-time systems in which more resources will lead to better performance; however, the available resources are typically limited. Therefore, resource management becomes an important issue to handle, in particular dynamically, to support the system's dynamicity.

**The Plant:** The term *plant* refers to the physical instance or mathematical model of the system being controlled. The continuous-time behavior of the plant is sampled at regular intervals through sensors.

**QoC Controller** The primary objective of the QoC controller, as depicted in Fig. 8.2, is to optimize the performance of the Control Application through the execution of four distinct functions: (i) a function to adjust timing constraints of the virtualized Control Application based on its QoC metrics, (ii) a container's resource reservation, (iii) interference mitigation, and (iv) communication with co-located QoC controllers.

Timing constraints adjustment refers to the adaptation of control periods for control applications based on QoC metrics. It refers to the adjustment of the control period. In particular, QoC can be improved by sampling the state of the plant more frequently by decreasing the period of control, resulting in more precise control actions. This is particularly useful for handling transients. However, if the QoC is fairly good, the frequency at which the controller takes action may be decreased, which in turn releases computing resources for other co-located controllers.

The resource allocation function is responsible for adjusting the time constraints of the container resource reservation based on current requirements and real-time performance. It can either increase or decrease the resource allocation accordingly. This function dynamically adjusts the allocation of CPU resources to meet the changing demand of the Control Application. The interference mitigation function aims to compensate for performance losses caused by co-located containers, which can result in unpredictable interference, and to optimize the provisioned CPU resources. In the next section, we provide a detailed analysis of the performance of these functions within the QoC controller. Lastly, the communication function ensures fair distribution of CPU resources among co-located QoC controllers and prevents the over-reservation of available resources.

Table 8.1: Threshold of QoC controller.

| QoC | Control Period | Container Budget |
|---|---|---|
| **Low** | Very High | Very High |
| **Normal** | Normal | Normal |
| **High** | Low | Low |
| **Very High** | Very Low | Very Low |

### 8.4.1 Control Period Controller

The Control Period Controller continuously observes the Control Application's QoC metric and, based on this metric, the controller determines the control period.

**QoC metric measurement:** The performance of the Control Application can be expressed as a QoC cost function computed as follows:

$$\mathrm{QoC}(t) = \int_{t-h}^{t} \|y_{\mathrm{des}}(t) - y_{\mathrm{act}}(t)\|^2 \tag{8.1}$$

where, $y_{\mathrm{des}}(t)$ is the setpoint, i.e., the desired behavior of the controlled variable $y_{\mathrm{act}}(t)$, and $t - h$ is the time the last control action has been taken. Note that high values of QoC correspond to poor controller performance and low values of QoC correspond to good performance.

The controller determines the difference between the current ($y_{\mathrm{act}}$) and desired ($y_{\mathrm{dest}}$) states of the controller system, then generates a control signal with the aim of bringing the difference to zero. The instantaneous value of QoC can be used to evaluate the current performance of the controller, and it can be used to adjust the control period of the controller.

**Control period calculation:** To implement the controller, we chose a threshold controller as our control system. A threshold controller functions by comparing a measured quantity, specifically the QoC, to a predefined threshold or setpoint value. When the measured quantity exceeds the threshold value, the controller initiates appropriate actions to modify the system and bring the measured quantity back within the desired range. The thresholds for the QoC controller are described in Table 8.1. We assume that the thresholds are defined by a system designer.

### 8.4.2 Resource Reservation Controller

Based on the decision of the Control Period Controller, the resource reservation controller adjusts the resource reservations for the real-time container. It adjusts the resource reservation by modifying the period and budget of the containers.

Similarly to the Control Period Controller, the Resource reservation controller employs threshold-based rules to provision and de-provision resources for the RT containers, as shown in Table 8.1.

### 8.4.3 Interference Mitigation

The resource adjustment function acts as a bridge between the ideal container resource reservation, which is based on simplified models, and the actual dynamic behavior of the containers. The performance of containers can be negatively impacted by the presence of co-located containers. This issue becomes particularly significant in systems like the one examined in this article, where co-located containers experience significant variations in their workload. When containers are colocated on shared platforms, they may compete for shared resources that may introduce timing disturbances, such as cache misses, memory and bus contention, and translation look-aside buffers [15], which are discussed in [11].

Hence, the resource adaptation function in the system dynamically modifies the CPU resources allocated to real-time containers by continuously assessing their real-time performance. Its primary objective is to counteract the unforeseen interference caused by co-located containers. To evaluate real-time performance, the system uses container-level metrics (CLM) introduced in a previous work [13]. These metrics include response time and missed deadlines. By measuring and readjusting the CPU bandwidth reservation for containers, this function compensates for performance losses within a computing node. Its purpose is to ensure that the control application meets the specified response time in the presence of interference that may cause the execution time to exceed the worst-case assumption that was used to check the system schedulability.

### 8.4.4 Communication with other controllers

Due to the limited CPU resources on physical platforms, it is essential for the system to monitor the available CPU resources that can be allocated among real-time (RT) containers based on predefined policies. The problem is the fact that there are limited resources if multiple co-located Control Applications face an increase in QoC values, and the controller can not independently increase the control period. As depicted in Fig. 8.3, there are multiple QoC controllers that adjust the control period and CPU resources for its corresponding real-time containers.

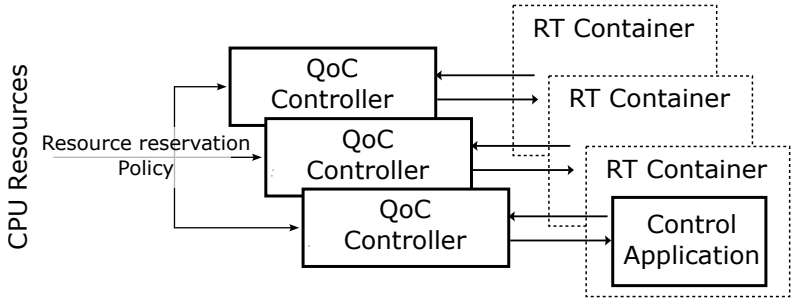



Figure 8.3: Multiple controllers.

## 8.5 Experimental Evaluation

In this section, we present an experiment that evaluates the feasibility of controlling the control periods and CPU resources based on the QoC values. We conducted an experimental demonstration of the adaptation process utilizing an Intel i5 computer equipped with 8GB of RAM, running Debian Linux (Kernel version 5.2.8) that was patched with the HCBS, and running Docker v20.10.

In this experiment, we used the inverted pendulum use case to show the performance of the designed control system. An inverted pendulum is a widely studied use-case in control theory, which involves a pendulum anchored on a platform (cart) that can move along a track in one dimension (along the x-axis). The governing equations for the inverted pendulum given in the literature

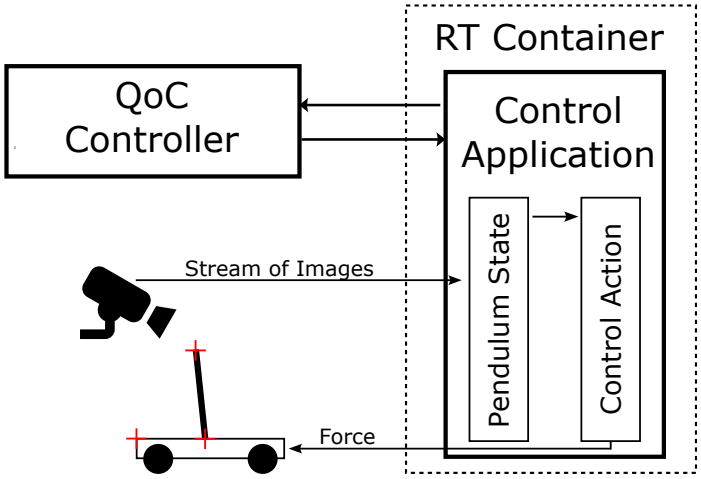



Figure 8.4: Inverted pendulum experiment.

(e.g. [28, p. 43]) are:

$$\begin{cases} (M+m)\ddot{x} + b\dot{x} + ml\ddot{\theta}\cos\theta - ml\dot{\theta}^2\sin\theta = F \\ (I+ml^2)\ddot{\theta} + mgl\sin\theta = -ml\ddot{x}\cos\theta \end{cases} \tag{8.2}$$

where $M$ are the masses of the cart and pendulum, respectively, $b$ represents the friction coefficient, $l$ is the length of the pendulum, $I$ represents the moment of inertia of the pendulum, $F$ is the force that is applied to the entire cart, $x$ is the position of the cart, and $\theta$ is the angle of the pendulum beam. The pendulum is balanced in an inverted position, which means that its center of mass is directly above the pivot point. The control loop tries to maintain the inverted pendulum in balance while also maintaining a given position of the cart by actuating the force applied to the cart along the x-axis. We use a virtualized PID controller as a Control Application. Fig. 8.5 illustrates a system consisting of an uncontrolled inverted pendulum.

The setting of the inverted pendulum experiment is depicted in Fig. 8.4. We developed a pendulum simulator that is equipped with a (virtual) camera that produces images of the pendulum. The camera streams the pendulum images into the controlled system. The controller utilizes image processing algorithms to analyze the image stream and infer the state of the pendulum. The stream

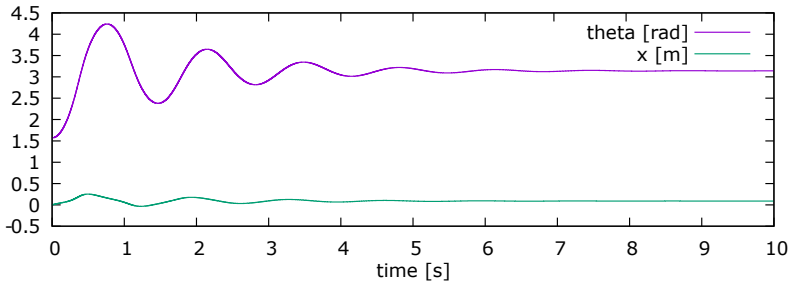



Figure 8.5: Inverted pendulum use-case.

of images is processed by the OpenCV library, which is a computer vision toolkit that offers a range of functions and algorithms specifically designed for real-time object detection and tracking. Using the OpenCV library, the controller captures the image feed from a camera and applies image-processing techniques to detect the position and angle of the pendulum. This information is used to calculate the appropriate control actions to keep the pendulum upright. The algorithm uses a template matching to locate the tip of the pendulum, its base, and the left corner of the cart as indicated in Fig. 8.4. The processing of the images is a non-trivial action that requires properly allocated CPU resources.

The results of the inverted pendulum experiment are illustrated in Fig. 8.6. In all sub-figures, the y-axis presents the measured values, while the x-axis is the time instance we measured each value. In this experiment, we have the pendulum not in the upright position at the beginning. Then, after equilibrium, at time instance 300, we intentionally make the pendulum unbalanced again to see how the QoC value develops again.

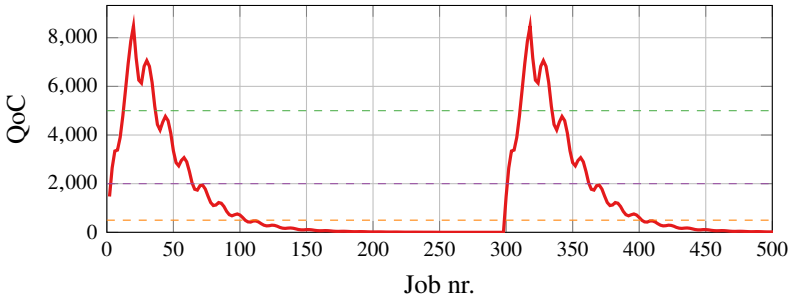
Based on this experiment, Fig. 8.6.a shows the QoC values and instant errors of the control. Initially, the pendulum is not situated in the desired upright and balanced position, leading to a significant error between the desired and actual state. As time progresses, the pendulum converges towards its setpoint, resulting in the reduction of the error and QoC. At time instance 300 the pendulum is perturbed, leading to an increase in error and QoC again. We divided the QoC range into four thresholds that classify the level of QoC: Low (for QoC lower than 500), Normal (when it is 2000), High (when it is 5000), and Very High (QoC higher than 5000). Note that these values are selected based on the

use-case which can vary depending on the system under control. The development of QoC values shows that the quality can reach the Low range when the pendulum becomes stable. In general, Fig. 8.6.a shows that the Control Application within the container performs as expected.

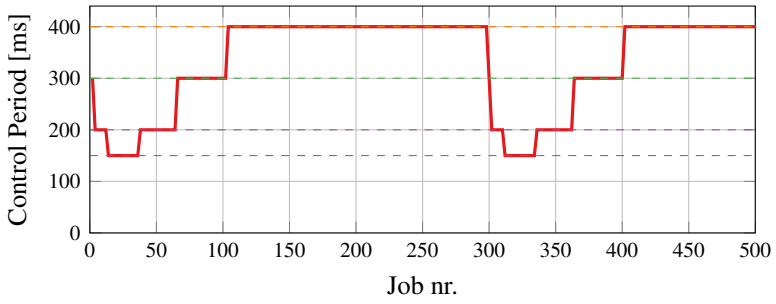
Fig. 8.6.b shows the corresponding control period. The value for the control period is based on the QoC value. With the decreasing QoC value, while the pendulum is converging to equilibrium, the control period increases. This will reduce computing resource utilization as there is no need for frequent control of the system. At time instance 300, the control period decreases again to perform a better control on the pendulum, which is not in equilibrium again. This is necessary because there is a need for more frequent control of the system.

Fig. 8.6.c shows the CPU budget. Similarly to the control period, the CPU budget value is based on the QoC value. With decreasing QoC value, the CPU budget reservation decreases. The CPU budget decreases to decrease the computing resource utilization when the pendulum becomes stable, i.e., there is no need for frequent control. Note that increasing the period and decreasing the budget significantly reduces computing resource utilization. Finally, Fig. 8.6.d shows the response time of the container that consists of the image processing part and the control action. The value depends on the CPU budget reservation. Allocating more CPU budget decreases the response time.

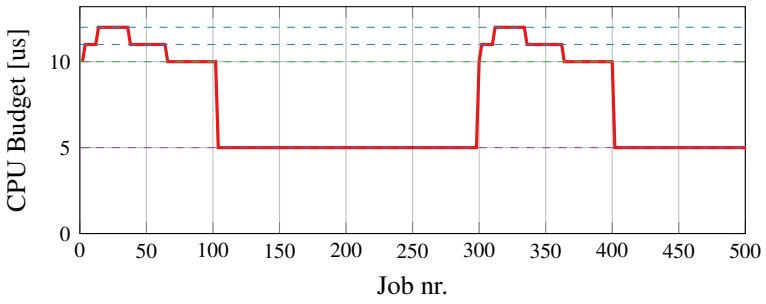
(a) QoC



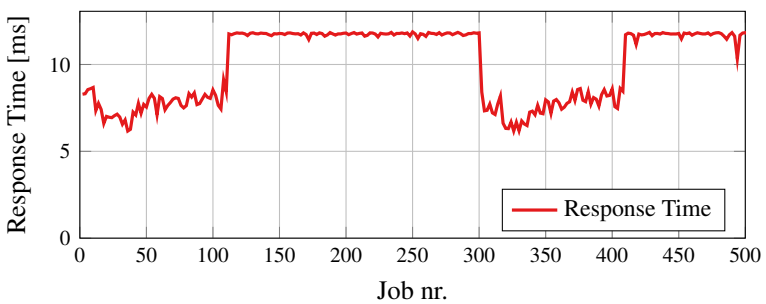

(b) Control Period




(c) CPU Budget




(d) Response time



## 8.6 Implementation on Linux

In order to demonstrate the feasibility and behavior of the system, we have integrated the suggested QoC-aware controller into the Debian GNU/Linux 10 (buster) operating system patched with the HCBS patch. This patch enables the management of resources for containers during runtime without the need for modifications of the container runtime. We have enhanced the Linux Kernel by incorporating the adaptability of CPU resources for RT containers based on the QoC values. While the HCBS patch allows static configuration of CPU budget and CPU period for containers, our supplementary modifications enable dynamic adjustment of CPU resources for RT containers during runtime. The cornerstone of the Kernel extension is a set of custom syscalls that are being called from a containerized Control Application. Based on the syscalls, the Kernel adjusts the control period and resource reservation. The containerized application is a C application that generates an infinite sequence of jobs. Each job measures controlled variables and computes the value of the control signal. After the completion of the job, the application is suspended until the time of the next job activation (using *clock_nanosleep()*). The application communicates with the Linux kernel via syscalls, which computes error and QoC then actuates the CPU budget for the real-time containers and control period. An overview of custom syscalls is as follows:

- **Application Initialization**: It is called after the application is started and sets initializes variables and constants for the control mechanism in Linux Kernel: QoC values, the QoC thresholds, periods, and CPU budgets.
- **Job Activation**: This syscall is called right after a job is activated (waked up). It captures the job activation time, that is used for computing job response time.
- **Job Finished**: It is called on the completion of the job. It computes error as the difference between the desired and actual states. And computes current QoC as shown in Equation 8.1.

## 8.7 Conclusions and future work

In this paper, we presented a resource adaptation approach that takes into account the QoC for real-time containers. Container-based virtualization offers significant benefits for the deployment of industrial applications. However,

addressing challenges such as ensuring complete temporal isolation among real-time applications and supporting time-critical applications becomes crucial, particularly in scenarios with varying workloads. This paper proposes a mechanism to dynamically adapt timing constraints and resource reservations in order to improve the QoC for time-sensitive control applications. The proposed mechanism is implemented and evaluated on a Linux-based hierarchical scheduling platform. The results obtained from the evaluation demonstrate the feasibility of our approach, where the controller modifies the timing constraints of the virtualized control application and updates the resource reservation for the hosting container accordingly. With an experiment using an inverted pendulum, we show that it is possible to control the pendulum while simultaneously adjusting the utilization of computing resources during the control process. This capability has a direct impact on optimizing computing resource usage.

There are several interesting directions for future work. In this paper, we have presented a resource adaptation mechanism for real-time containers considering the QoC. However, more complex control, decision, and predictive algorithms are needed to capture more complex scenarios. An ongoing work targets systems that have multiple virtualized controllers.

# Bibliography


[1] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. De Rose. Performance evaluation of container-based virtualization for high performance computing environments. In *Euromicro Int. Conf. on Par., Distr., and Netw. Proc. (PDP)*, 2013.

[2] Miguel Gomes Xavier, Marcelo Veiga Neves, and Cesar Augusto Fonticielha De Rose. A performance comparison of container-based virtualization systems for mapreduce clusters. In *Euromicro Int. Conf. on Par., Distr., and Netw. Proc. (PDP)*, 2014.

[3] F. Ramalho and A. Neto. Virtualization at the network edge: A performance comparison. In *IEEE Int. Symp. A World of Wirel., Mob. and Multim. Net. (WoWMoM)*, 2016.

[4] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio. An updated performance comparison of virtual machines and Linux containers. In *IEEE Int. Symp. on Perf. Analysis of Syst. and Soft. (ISPASS)*, 2015.

[5] Björn Wittenmark, Karl Johan Åström, and Karl-Erik Årzén. Computer control: An overview. *IFAC Professional Brief*, 2002.

[6] Pau Marti, Josep M Fuertes, Gerhard Fohler, and Krithi Ramamritham. Improving quality-of-control using flexible timing constraints: metric and scheduling. In *23rd IEEE Real-Time Systems Symposium, 2002. RTSS 2002*. IEEE, 2002.

[7] Gerhard Fohler. Dynamic timing constraints - relaxing overconstraining specifications of real-time systems. 2001.

[8] Karl J. Åström. *Event Based Control*. Springer Berlin Heidelberg, 2008.

[9] Mohammadreza Barzegaran, Anton Cervin, and Paul Pop. Towards quality-of-control-aware scheduling of industrial applications on fog computing platforms. In *Proceedings of the Workshop on Fog Computing and the IoT*, IoT-Fog '19. Association for Computing Machinery, 2019.

[10] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the linux kernel. *ACM SIGBED Review*, 2019.

[11] Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V. Papadopoulos. Hierarchical resource orchestration framework for real-time containers. 2023.

[12] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Autonomic vertical elasticity of docker containers with elasticdocker. In *2017 IEEE 10th international conference on cloud computing (CLOUD)*. IEEE, 2017.

[13] 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)*. IEEE, 2021.

[14] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019.

[15] Shashank Shekhar, Hamzah Abdel-Aziz, Anirban Bhattacharjee, Aniruddha Gokhale, and Xenofon Koutsoukos. Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*. IEEE, 2018.

[16] Anton Cervin. *Integrated control and real-time scheduling*. PhD thesis, Univ., 2003.

[17] Alberto Leva and Alessandro Vittorio Papadopoulos. Tuning of event-based industrial controllers with simple stability guarantees. *Journal of Process Control*, 2013.

[18] Antonio Celesti, Davide Mulfari, Maria Fazio, Massimo Villari, and Antonio Puliafito. Exploring container virtualization in iot clouds. In *2016 IEEE international conference on Smart Computing (SMARTCOMP)*. IEEE, 2016.

[19] Alessandro Vittorio Papadopoulos, Martina Maggio, Alberto Leva, and Enrico Bini. Hard real-time guarantees in feedback-based resource reservations. *Real-Time Systems*, 2015.

[20] Alexandru Moga, Thanikesavan Sivanthi, and Carsten Franke. Os-level virtualization for industrial automation systems: are we there yet? In *SAC '16*, 2016.

[21] Thomas Goldschmidt, Stefan Hauck-Stattelmann, Somayeh Malakuti, and Sten Grüner. Container-based architecture for flexible industrial control applications. 2018.

[22] 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)*. IEEE, 2018.

[23] Marcello Cinque, Raffaele Della Corte, Antonio Eliso, and Antonio Pecchia. Rt-cases: Container-based virtualization for temporally separated mixed-criticality task sets. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, 2019.

[24] Insik Shin and Insup Lee. Compositional real-time scheduling framework. In *25th IEEE International Real-Time Systems Symposium*. IEEE, 2004.

[25] David Lo, Liqun Cheng, Rama Govindaraju, Parthasarathy Ranganathan, and Christos Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture*, 2015.

[26] Surya Kant Garg and J. Lakshmi. Workload performance and interference on containers. In *2017 IEEE SmartWorld, Ubiquitous Intelligence & Computing, Advanced & Trusted Computed, Scalable Computing &*

*Communications, Cloud & Big Data Computing, Internet of People and Smart City Innovation*, 2017.

[27] Prateek Sharma, Lucas Chaufournier, Prashant Shenoy, and YC Tay. Containers and virtual machines at scale: A comparative study. In *Proceedings of the 17th international middleware conference*, 2016.

[28] G. Conte, C.H. Moog, and A.M. Perdon. *Algebraic Methods for Nonlinear Control Systems*. Communications and Control Engineering. Springer London.

# Chapter 9

# Paper D: Hierarchical resource orchestration framework for real-time containers

*Václav Struhár*, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos

In ACM Transactions on Embedded Computing Systems 23.

**Abstract**

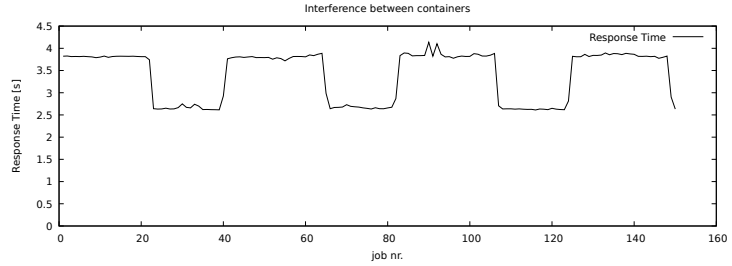
Container-based virtualization is a promising deployment model in fog and edge computing applications because it allows a seamless co-existence of virtualized applications in a heterogeneous environment without introducing significant overhead. Certain application domains (e.g., industrial automation, automotive, or aerospace) mandate that applications exhibit a certain degree of temporal predictability. Container-based virtualization cannot be easily used for such applications since the technology is not designed to support real-time properties and handle temporal disturbances. This paper proposes a framework consisting of a static offline and a dynamic online phase for resource allocation and adaptive re-dimensioning of real-time containers. In the offline phase, the optimal initial deployment and dimensioning of containers are decided based on ideal system models. Additionally, in order to adapt to dynamic variations caused by changing workloads or interferences, the online phase adapts the CPU usage and limits of real-time containers at runtime in order to improve the real-time behavior of the real-time containerized applications while optimizing resource usage. We implement the framework in a real Linux-based system and show through a series of experiments that the proposed framework is able to adjust and re-distribute computing resources between containers in order to improve the real-time behavior of containerized applications in the presence of temporal disturbances while optimizing resource usage.

## 9.1 Introduction

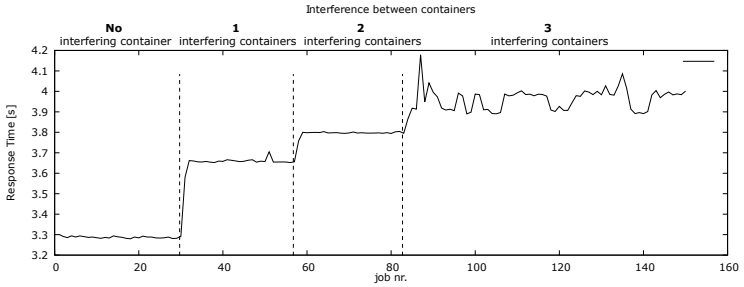
Container-based virtualization and container orchestration have become *de facto* standards for packaging and deploying applications in cloud environments [1]. Virtualization provides functional isolation, enabling the co-location of applications on a shared physical machine, while orchestration manages containerized applications, automates deployment, and maintains scalability across a cluster of physical machines. Such technologies are paving the way for recent trends such as fog and edge computing, in which an application spans multiple computing layers. In fog computing, the edge layers host low-latency functions, while the cloud layers host computationally demanding functions. Currently, industrial domains, e.g., robot control, automotive, aviation, and mixed-criticality-systems in general [2] seek to adopt container-based virtualization and orchestration in their ecosystems in order to fulfill Industry 4.0 needs [3, 4, 5, 6]. That is the flexibility of the manufacturing process, decentralization of functions, increasing automation, and resource efficiency.

However, industrial domains have stringent requirements for applications, especially in real-time domains [7, 2]. Industrial applications require both functional correctness and timeliness to produce computation results [8, 5]. In the presence of timing violations, the usefulness of computed results decreases (i.e., soft real-time), or it may even result in catastrophic consequences (i.e., hard real-time). Unfortunately, real-time behavior is challenging to achieve in systems using container-based virtualization and orchestration since timing predictability is not a major concern of these technologies. As noted in [9], remotely-hosted virtual environments suffer from variances in processing times that are changing with the executed workloads [9, 10].

Due to the limited performance isolation of container-based virtualization, co-located applications may affect each other's performance in unpredictable ways [11]. For instance, co-located containers may share the Last Level Cache (LLC), i.e., a cache that is accessed by the cores prior to fetching from memory. Intensive simultaneous use of the cache by multiple containers may increase the response time of the containerized applications, and time predictability (i.e., real-time behavior) may be negatively impacted. The effect of performance interference is depicted in the experiment (Figure 9.1), which shows a performance loss due to the simultaneous use of a shared LLC cache. Figure 9.1a shows how the response time of a container is impacted by a co-located




(a) The response time of a co-located container may vary over time due to changes in the workload of the interfering container. In this example the co-located container changes its workload periodically.




(b) An example of changing response times of a container caused by the interfering containers. We successively deploy 1 to 3 interfering containers on different CPU cores. The response time is negatively affected by the number of co-located containers.

Figure 9.1: Two examples of interference between containers utilizing LLC caches. The examples show extended response times for a containerized application caused by extensive LLC cache usage among co-located containers.

container running on another core. The co-located container periodically alternates between a cache-intensive and a non-cache-intensive workload. As a result, the response times increase during the cache-intensive periods. Similarly, Figure 9.1b shows the change in response time when an increasing number of containers are co-located with the measured containerized application.

The problem of timing interference between containers is further complicated by the fact that it varies widely across platforms. This fact is crucial in fog and edge computing, where heterogeneous physical machines are present and the effect of interference may not be known beforehand. An additional factor that hinders the real-time behavior of container-based computing (and computing in general) is the unpredictable changing workload. Containers may be performing updates or serving increased user-generated workloads that could affect the performance of other co-located containers, and these effects may be unknown in advance.

Given the aforementioned factors that influence the real-time behavior of containers, this paper proposes a joint hierarchical framework for container virtualization and orchestration to improve their real-time performance in a multi-container environment. The proposed framework consists of two phases: an offline phase and an online phase. The offline phase is in charge of solving the deployment problem, i.e., it decides where to place containers and how to dimension their real-time interfaces on the basis of design-time models. This phase is related to the server design problem in hierarchical scheduling [12, 13]. On the other hand, the online phase attempts to bridge the gap between the assumptions that are needed for the deployment, in which idealized models are used, and the actual dynamic behavior of the containers that may negatively impact the runtime performance of real-time containers, as described above. The online phase hence adjusts and distributes the CPU resources among real-time and non-real-time best-effort (BE) containers based on a continuous runtime evaluation of the real-time performance of containers. In this work, we emphasize the problem of resource interference and performance loss due to the changing workload of co-located containers, a problem that is especially significant in heterogeneous platforms (e.g., fog computing). We focus on the computational part of mitigating the resource interference (i.e., adjusting computational resources) and keep other parts (e.g., location awareness, networking) inherent to fog computing for future work.

In this paper, we define and solve both the offline and the online phases

of the real-time container orchestration problem. For the offline phase, we use well-known approaches for the server design problem and extend them to the problem at hand, formulating the allocation and dimensioning of containers as an optimization problem that can be solved via, e.g., integer linear programming (ILP) and satisfiability modulo theories (SMT) solvers or using well-known heuristics if the problem size exceeds the scalability of optimal solutions. In the online phase, the Hierarchical Framework dynamically compensates for the real-time performance variability in a multi-node container-based environment using a three-level control hierarchy. The first- and second-level controllers compensate for the performance losses in a computing node by continuously measuring and re-adjusting the CPU bandwidth reservation for containers. The third-level controller is in charge of the migration of containers amongst computing nodes in case the required real-time performance cannot be achieved within the scope of the individual node. Finally, we describe the implementation in Linux of our Hierarchical Framework and a series of experiments that shows the viability of our solution.

Contributions of this paper are as follows:

- We propose a framework that combines offline placement and online redimensioning of resources for real-time containers. In the offline phase, we formulate the initial deployment of containers as the optimization problem. In the online phase, we enable the self-adaptation of resources for real-time containers based on their timing requirements.
- We present an architectural design and a detailed explanation of the three-level hierarchy of controllers utilized for adapting resources and migrating real-time containers.
- We implement the online adaptation of real-time containers in Linux and provide an evaluation that shows the feasibility of the solution. We utilize and enhance the HCBS (Hierarchical Constant Bandwidth Server) patch [14] with the ability to adjust resources at runtime.

The paper is organized as follows: Section 10.3 presents the background and surveys the related work, followed by the system architecture and system model in Section 9.3. We present the details of our orchestration framework, including the offline and online adaptation phases in Section 9.4. Section 11.3 shows experimental results, and finally, Section 10.8 concludes this work.

## 9.2 Background and Related Work

Providing real-time performance for container-based computation is outgoing research, the main research directions are summarized in [15]. The major research directions focus on the mitigation of scheduling latencies and scheduling jitters for containers by using PREEMPT_RT patch [16, 17], applying real-time co-kernels for container-based virtualization [18, 19, 20, 21], and employing the hierarchical scheduling framework for real-time containers to enable temporal isolation of containerized applications [14]. However, none of the studies addresses performance interference between containers nor mitigating performance interference.

Abeni et al. [14] present the Linux Kernel enhanced with hierarchical scheduling that utilizes the Hierarchical Scheduling Framework. The authors hierarchically connect two existing Linux scheduling policies (SCHED_DEADLINE and SCHED_RT). The former policy implements the EDF (Earliest Deadline First) algorithm combined with CBS (Constant Bandwidth Server) [22], serving as a selector of the container with the earliest deadline. The latter local policy schedules the highest-priority runnable task from the particular container. Thus, a user can reserve a CPU quota over a given period. We base our resource adaptation framework on this approach which allows us to continuously adapt container resources by changing the CPU quota and the period. Our work utilizes the proposed Kernel patch and adds a mechanism for resource reservation adjustments at runtime.

Container orchestrators are tools that manage containerized applications, automate deployment, and enable scalability in a cluster of physical machines. However, the development of real-time container orchestration is in the early stage. Several studies attempt to enhance existing orchestrators (i.e, Kubernetes[1], OpenStack[2]) with awareness of real-time requirements. For instance, Struhar et al. [23] and Fiori et al. [9] leverage Kubernetes for the deployment of real-time containers. Similarly, Cucinotta et al. [24] use OpenStack, which is able to deploy real-time containers. The solutions conduct a utilization-based admission test upon a container deployment request. Barletta et al. [25] introduce an orchestration approach for mixed-criticality systems with real-time requirements. The before-mentioned papers extend the orchestrators with the

[1] Available at `https://kubernetes.io/`

[2] Available at `https://openstack.org/`

ability to perform schedulability tests and place containers in suitable nodes. However, our work adds additional dimensions for the orchestration: real-time migration (when there are not enough resources) and online adjustment of resources to maintain soft real-time performance.

Containerized applications can be scaled either horizontally or vertically. Horizontal scaling (also known as replication) involves adding computational resources linked with the application [26]. Vertical scaling implies changing functional or non-functional resources such as CPU time, memory, etc. In the study by Al-Dhuraibi et al. [26], the authors provide a system for elastic scaling of containers that adjusts memory, vCPU cores, and CPU fractions according to the workload. The resource adjustment strategy is based on static thresholds of response times. The study provides a simple strategy to vertically control resources. In our work, we focus on real-time containers containing periodic tasks. We monitor and take control actions continuously, and thus, eliminate long breath-down/break-up periods needed to reach a stable system state. Additionally, we consider the offline phase for computing the parameters of containers and computing the initial placement of the containers.

The study by Rossi et al. [27] proposes reinforcement learning techniques for adapting at runtime the deployment of container-based applications by means of horizontal and vertical elasticity. The system adapts without the need to tune the system manually. Furthermore, both vertical and horizontal scaling (that can introduce performance penalties) is explored. The paper [28] presents a data-driven, machine-learning technique based on Gaussian Processes to build a predictive runtime model of the system's performance, which can adapt itself to the variability in workload changes.

## 9.3 System architecture and System model

In this section, we present the architecture and system model of the proposed framework. The architecture of the system is depicted in Figure 9.2. The framework consists of the offline phase and a three-level control architecture for the online phase encompassing container-level, node-level, and cluster-level controllers. The offline phase is (a) responsible for the computation of ideal real-time interfaces (an initial value of the resource reservation) for a given set of containers and their real-time requirements and (b) responsible for selecting a node for an incoming container. The three-level controller hierar-

Table 9.1: System model notations.

| Model | Notation | Definition |
|---|---|---|
| **Node** | $n$ | total number of nodes |
| | $\Pi_j^{be}$ | Set of BE containers on node $j$ |
| | $\Pi_j^{\text{rt}}$ | Set of RT containers on node $j$ |
| | $f_j^M$ | Memory capacity |
| | $f_j^S$ | Storage capacity |
| **Task** | $\tau_i \in \mathcal{T}$ | $\tau_i = (T_i, \{C_i^1, \ldots, C_i^n\}, D_i, p_i)$ |
| | $T_i$ | Period |
| | $C_i^1, \ldots, C_i^n$ | Node-dependent Worst-Case Execution Times |
| | $D_i$ | Deadline |
| | $p_i$ | Task priority |
| **Container** | $\pi$ | $\pi = (\mathcal{T}^k, P_k, Q_k, CLM_k, \pi_k^M, \pi_k^S)$ |
| | $m$ | total number of containers |
| | $Q_k$ | CPU quota |
| | $P_k$ | Replenishment period |
| | $CLM_k$ | Container Level Metrics |
| | $\pi_k^M$ | Memory demand |
| | $\pi_k^S$ | Storage demand |

chy mitigates temporal disturbances by a continuous redistribution of system resources and migration of containers in case of a shortage of resources in a computing node.

The system model is based on the React Orchestrator presented in [23] that consists of the following elements: the *Task model*, the *Container model*, the computer *Node model*, the *Cluster model*, and the *Hierarchical scheduler*. The notation of the model is summarized in Table 9.1 and described as follows.

For the computing node model, we assume that there are $n$ nodes in the system, where each node $f_j$, $j = 1, \ldots, n$ has a certain amount of memory and storage resources, denoted with $f_j^M$ and $f_j^S$, respectively. We assume that each node has a single-core CPU with a given CPU frequency which will affect the WCET of a task assigned to it (see below). Multi-core CPUs can be modeled as separate single-core nodes that are part of the cluster. Each computing node $f_j$ is capable of running both *RT* containers and best effort (*BE*) containers. We define the set of *RT* and *BE* containers on node $f_j$ with $\Pi_j^{rt}$ and $\Pi_j^{be}$, respectively. The cluster $C$ consists of several nodes, $f_1, \ldots, f_n$, that are connected with the orchestrator.

For the task model, we assume a periodic task model [29] that describes a

periodically executed stream of jobs. A task ($\tau_i \in \mathcal{T}$) consists of a stream of jobs. Each job of a task $\tau_i$ is periodically activated at the beginning of each period ($T_i$) and has a prescribed relative deadline $D_i$. The execution time of each job is specified by Worst-Case Execution Time (WCET), denoted as $C_i$. Additionally, each task has a relative priority $p_i$ that is used to determine the processing sequence within the container. For heterogeneous systems with a significant difference in hardware capabilities the execution time of tasks will have a high variance depending on where the container is placed. Hence, the execution time $C_i$ of tasks will depend on the CPU frequency of the respective node. For capturing tasks and containers that are allocated on heterogeneous systems, we introduce instead of $C_i$ an array $C_i^1, \ldots, C_i^n$, representing the worst-case execution time of the task $\tau_i$ on every node.

For the container model, we assume that there are $m$ containers in the system, each container $\pi_k$, $k = 1, \ldots, m$ having a memory ($\pi_k^M$) and storage ($\pi_k^S$) demand resulting from the individual memory and storage demands of the tasks running in the respective container that has to be satisfied by the node's resources, similar to [23]. A real-time *RT* container $\pi_k \in \Pi_j^{\text{rt}}$ has an *RT* interface expressed as $(P_k, Q_k)$ where $Q_k$ is a CPU quota that is required to be provided over a period $P_k$. We introduce the metric of a container $\pi_k$, denoted by $CLM_k$, to capture the performance of *RT* containers. Each container is running a pre-defined subset of the tasks $\mathcal{T}^k \in \mathcal{T}$.

In each node, there may be multiple sources of non-determinism that may affect the real-time behavior of containerized tasks, e.g., kernel latencies, interrupts, context switches, caches, etc. However, we argue that it is difficult to accurately model and capture all these sources of non-determinism in a real system. Hence, we keep our node model simple and do not attempt to include different overhead and interference sources. Our online phase is tailored to address this very problem and not to mandate that all sources of non-determinism in the system are known since these latencies will be compensated for in the online adaptation mechanism if they negatively affect the real-time behavior of the tasks within containers (see below).

For the scheduler/dispatcher model, we assume that the system uses a two-level hierarchical scheduler. The global scheduler uses the EDF algorithm to select the container with the earliest deadline (the deadline of the container is aligned with its period). The EDF scheduler is combined with the Constant Bandwidth Server with hard allocation. The local scheduler is a fixed

priority scheduler that chooses the highest priority runnable task inside of the selected container. This model can be achieved with the modified Linux distribution (e.g., Abeni's HCBS implementation [14]) where a CBS scheduler (`SCHED_DEADLINE`) has a higher priority than the fixed-priority scheduler (`SCHED_RT`) in the kernel scheduling hierarchy. We assume that other tasks (i.e., BE container tasks) are scheduled by the Completely Fair Scheduler (CFS), having the lowest priority.

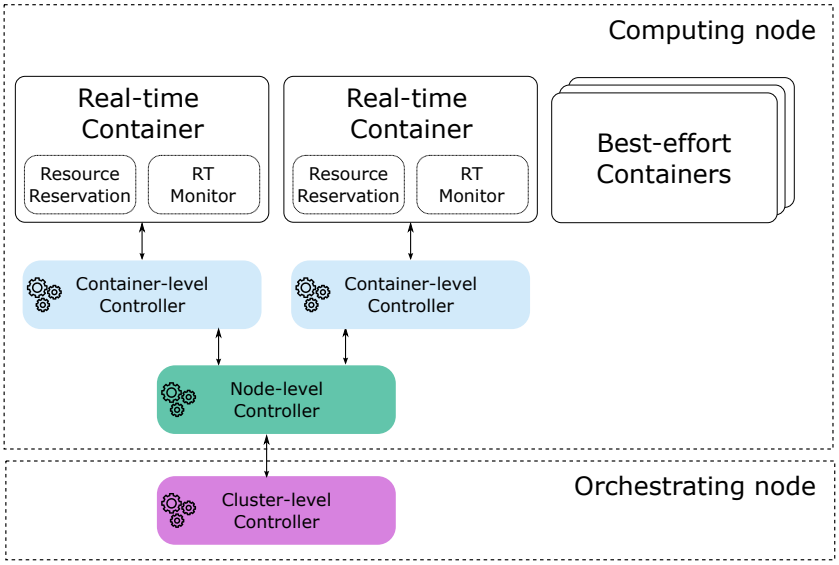



Figure 9.2: Architecture of the system.

## 9.4 Hierarchical Resource Orchestration Framework

We present the Hierarchical Resource Orchestration Framework, which is a general orchestration framework for static and dynamic allocation and dimensioning of real-time containers. The framework leverages real-time containers, i.e., containers that provide both spatial and temporal isolation of containerized applications. Adding real-time properties to containers has been addressed in [14] by introducing a hierarchical scheduling patch for Linux-based systems. The patch ensures that the container would not violate the CPU resource reservation of other containers (however, it does not solve the possible interference between containers due to sharing of resources other than CPU). Our framework manages the deployment and adaptation of real-time containers in distributed applications featuring heterogeneous computing nodes so that individual real-time task requirements are met. These requirements are not only related to real-time behavior but also resource usages such as memory, I/O, and storage requirements and non-functional requirements such as fault-tolerance, power consumption, or resource efficiency.

The input to the Hierarchical Resource Orchestration Framework is a set of real-time tasks and containers. The containers include either real-time tasks with constrained deadlines assigned to real-time (RT) containers or non-real-time tasks which are assigned to best-effort (BE) containers. Real-time tasks are additionally defined using a worst-case execution time (WCET) and a period specifying an upper bound on the computation of the task in each period and the rate at which the task is activated. Both RT and BE containers can coexist on the same core, but they are spatially and temporally isolated. The real-time tasks are pre-allocated to containers, but the containers are not pre-allocated to computing nodes (and cores), although they can have a certain affinity set, constraining the set of nodes to which they can be allocated. As defined in [23], each RT container $\pi_k$ has additionally an RT interface consisting of $(P_k, Q_k)$ where $Q_k$ is the CPU quota within an interval (period) $P_k$, defining that the container $\pi_k$ cannot use more than $Q_k$ time units over an interval of $P_k$ time units.

The system, proposed in this paper, consists of two phases: offline and online phase. The former ensures the computation of the initial real-time interfaces, and the latter phase deals with the online adaptation of resources to the

containers. Both phases are described in the following sections.

### 9.4.1 Offline Phase

For the offline phase, the tasks are pre-assigned to containers, but containers (scheduled using the CBS `SCHED_DEADLINE` scheduler class of Linux) are not assigned to nodes, although they may have a certain affinity to a subset of nodes based on, e.g., ASIL level or node capabilities. The tasks inside the containers are scheduled via a fixed-priority scheduling class (usually the `SCHED_RT` scheduler class in Linux). We assume that the task priorities are given (usually via a deadline-monotonic assignment). In line with the model from [23], a *RT* container $\pi_k$ has an *RT* interface expressed as $(P_k, Q_k)$ where $P_k$ is a period and $Q_k$ is a CPU quota.

The main objective of the offline phase is to assign to each container an ideal *RT* interface by selecting $(P_k, Q_k)$ such that the fixed-priority tasks are schedulable. This problem is generally referred to as the server design problem in hierarchical scheduling [12, 13]. We use the lower linear approximation for the supply bound function of the periodic resource defined in [30, 13]. From [13] we have $\forall \pi_k \in \Pi_j^{\text{rt}}$, using $\alpha_k = Q_k/P_k$ and $\Delta_k = 2 \cdot (P_k - Q_k)$, the linear supply lower bound function $lslbf(t)_k$ is defined as

$$lslbf_k(t) = \max\{0, (t - \Delta_k) \cdot \alpha_k\}. \tag{9.1}$$

With the fixed-priority scheduler within the container, we can use the analysis from [13] to relate the $(P_k, Q_k)$ values to the schedulability of the tasks within the respective container $\pi_k$. Theorem 3 of [13] states that the task set $\mathcal{T}^k$ is schedulable under the resource abstraction $(P_k, Q_k)$ using the linear supply lower bound function $lslbf_k(t)$ if

$$\Delta_k \leq \min_{\tau_i \in \mathcal{T}^k} \max_{t \in \mathcal{P}_{i-1}(D_i)} t - \frac{1}{\alpha_k}\left(C_i + \sum_{\substack{\tau_j \in \mathcal{T}^k \\ p_j \geq p_i}} \left\lceil \frac{t}{T_j} \right\rceil \cdot C_j\right), \tag{9.2}$$

where:

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

Any values $P_k = \frac{\Delta_k}{2(1-\alpha_k)}$ and $Q_k = \alpha_k P_k$, for which $\alpha_k$ and $\Delta_k$ conform to Eq. (9.2) are valid. However, as described in [13], there is a trade-off between the wasted bandwidth and the context switch cost, as bigger periods result in fewer context switches, but the allocated container utilization has to be kept low to avoid wasting CPU bandwidth. Therefore, a cost function is proposed in [13] that leverages the two objectives via a weighted sum tuned by two design-time constants. The cost function proposed in [13] contains two design-time constants, $c_1$ and $c_2$, which enable the system designer to parameterize the trade-off depending on the design goals of the system. We adapt the cost function $J$ for a container from [13] as follows:

$$J = c_1 \frac{\delta_j}{P_k} + c_2 \alpha_k \tag{9.3}$$

where $\delta_j$ is the context switch overhead of the node $f_j$ where the container is placed. As can be seen, the cost function assumes that the container has already been assigned and the context switch overhead is known, which is not the case here since the containers have not been allocated yet.

In homogeneous systems or in heterogeneous systems where there is not a significant variance in the hardware capabilities, we can solve the server design problem immediately by choosing the lowest CPU speed as a reference. Therefore, the second stage allocation of containers to nodes becomes a bin-packing problem (with complexity NP-hard) that can be solved using exact methods or, for larger problem sizes, by heuristic approximations [31].

For heterogeneous systems with a significant difference in hardware capabilities, not only the system overhead, but also the execution time of tasks will have a high variance depending on where the container is placed. Hence, both the execution time $C_i$ of tasks, as well as the context switch overhead $\delta_j$, become variables that depend on the allocation. Moreover, both Eq. (9.2) and the cost function from Eq. (9.3) have to be considered as part of the allocation problem. For the purposes of the optimization problem, we remind the reader that instead of $C_i$ representing the single WCET in the standard model (c.f. [13]), we use an array $C_i^1, \ldots, C_i^n$, representing the worst-case execution time of the task $\tau_i$ on every node.

We define for each container $\pi_k, k = 1, \ldots, m$ the following variables:

- a set of Boolean variables $v_1^k, \ldots, v_n^k$ specifying whether the container is allocated to the respective node or not,

- the container period $P_k$ and budget $Q_k$.

The optimization problem can therefore be expressed as:

$$\underset{\{Q_k, P_k, v_1^k, \ldots, v_n^k | k=1,\ldots,m\}}{\text{minimize}} \; c_1 \cdot \sum_{k=1}^{m} \frac{\sum_{j=1}^{n} v_j^k \cdot \delta_j}{P_k} + c_2 \cdot \sum_{k=1}^{m} \frac{Q_k}{P_k} \tag{9.4}$$

subject to:

$$\forall j = 1, \ldots, n : \sum_{k=1}^{m} v_j^k \cdot \pi_k^M \leq f_j^M, \tag{9.5}$$

$$\forall j = 1, \ldots, n : \sum_{k=1}^{m} v_j^k \cdot \pi_k^S \leq f_j^S, \tag{9.6}$$

$$\forall j = 1, \ldots, n : \sum_{k=1}^{m} \frac{v_j^k \cdot Q_k}{P_k} \leq 1, \tag{9.7}$$

$$\forall k = 1, \ldots, m, \forall j = 1, \ldots, n : v_j^k = \{0, 1\}, \tag{9.8}$$

$$\forall k = 1, \ldots, m : \sum_{j=1}^{n} v_j^k = 1, \tag{9.9}$$

$$\forall k = 1, \ldots, m : \frac{Q_k}{P_k} \geq \sum_{\tau_i \in \mathcal{T}^k} \frac{\sum_{j=1}^{n} v_k^j \cdot C_i^j}{T_i}, \tag{9.10}$$

$$\forall k = 1, \ldots, m : \bigwedge_{\tau_i \in \mathcal{T}^k} \bigvee_{t \in \mathcal{P}_{i-1}(D_i)} t - 2 \cdot (P_k - Q_k) - \frac{P_k}{Q_k} \left( \sum_{j=1}^{n} v_k^j \cdot C_i^j + \sum_{\substack{\tau_l \in \mathcal{T}^k \\ p_l \geq p_i}} \sum_{j=1}^{n} \left\lceil \frac{t}{T_l} \right\rceil \cdot v_k^j \cdot C_l^j \right) \geq 0. \tag{9.11}$$

The first two constraints (Eqs. (9.5) and (9.6)) check the memory availability on each node and the constraint defined in Eq. (9.7) enforces that each node is not overutilized. The next two conditions (Eqs. (9.8) and (9.9)) constrain the Boolean assignment variables to be correct, i.e., one container can only be assigned to one node. The condition in Eq. (9.10) ensures that the bandwidth of the container is sufficient to execute all tasks within the container. Please note that we need the sum over the list of possible WCETs for each task multiplied by the binary assignment variable for the container to node assignment in order to select the correctly scaled WCET for each task. The last condition

(Eq. (9.11)) is the schedulability test from [13], reformulated and extended in the context of the RT container offline allocation problem. We have extended the test from [13] by including the allocation problem via the binary variables $v_1^k, \ldots, v_n^k$, which select the correct WCET for the given node allocation. If a container $k$ is allocated to a specific node $j$, all the Boolean allocation variables $v_1^k, \ldots, v_n^k$ will have a value of 0 except for the Boolean variable $v_j^k$ which will have a value of 1. Hence, equation Eq. (9.11) reduces to the schedulability test from [13] but additionally selects the correct WCET ($C_i^j$) for the respective node allocation. We also note that the condition in Eq. (9.10) is not necessary for the schedulability test since it is implied by Eq. (9.11). However, since Eq. (9.11) is quite complex and results in a large number of assertions, we add the necessary condition in Eq. (9.10) to prune our invalid parameter values for the containers and therefore speed up the search when a solver is used (see below).

Constrained satisfiability or optimization problems such as the one defined above can be solved efficiently using Satisfiability Modulo Theory (SMT) solvers or Optimization Modulo Theory (OMT) solvers [32]. SMT solvers address more generalized instances of Boolean SAT problems which benefit from being formulated in more expressive languages such as first-order logic that include non-Boolean variables and constants, quantifiers, functions, and predicate symbols [33]. In order to be more efficient and make the satisfiability problem decidable, SMT solvers can restrict the interpretations of specific symbols a specific background theory like linear integer arithmetic over natural numbers, arrays, real-numbers, or bit-vectors [33]. OMT solvers add optimization objectives on top of the underlying first-order logic [32]. There are many efficient state-of-the-art SMT and OMT solvers (e.g., CVC4 [34], MathSAT5 [35], openSMT [36], Z3 [37], Yices [38, 39]) that compete each year against each other in the *SMT-COMP* (c.f. [40]) and have been used successfully in application areas ranging from model checking [41], static analysis [42], automated test case generation [43], scheduling [44], and optimization [33]. Hence, the optimization problem defined above is a suitable candidate to be solved using, e.g., Optimization Modulo Theory (OMT) solvers [32] like Z3 [37] under the quantifier-free fragment of non-linear integer arithmetic (QF_NIA).

Tuning the weights $c_1$ and $c_2$ can be done based on the overhead impact when running RT containers. For example, in [23], we see that the system overhead is not highly impacted by the container period or the number of RT

containers; hence we can aim to optimize the bandwidth usage to a higher degree.

We note that it may not always be necessary to obtain the best solution with respect to the optimization objective from [13] (Eq. 9.3), since the online phase will compensate for the system overhead and dynamic behavior, and also free up CPU bandwidth that is not used by a container (see below). In this case, we can simplify and speed up the offline phase by retrieving any feasible solution for the RT interfaces. We can thus simply encode the offline dimensioning problem into first-order logic, leaving out the optimization objective, and solve it using high-performance Satisfiability Modulo Theory (SMT) solvers like Yices [39] or Z3 [37].

The offline phase only guarantees the schedulability and correct temporal behavior of tasks in an ideal scenario. At runtime, there may be any number of interference sources that affect the temporal behavior. For example, it is impossible to include a complete model of the underlying system (including the runtime interactions between tasks and containers) in the timing analysis. Additionally, certain runtime artifacts (e.g., cache misses, SW/HW interrupts, the overhead of the underlying operating system), as well as unforeseen changes in task workloads (that have not been considered in the WCET calculation), also influence the runtime temporal behavior. Hence, both the dimensioning and the allocation of the containers from the offline stage may not guarantee that task deadlines are always maintained at runtime. Moreover, new containers may be added during runtime, requiring a dynamic adaptation and allocation of the new container entities to nodes. Hence, we next describe the online orchestration phase that is tailored to adapt to these runtime changes and interferences.

### 9.4.2 Online Phase

The online phase complements the initial deployment of the containers aiming to mitigate possible performance disturbances by continuously re-adjusting the system resources of the containers. The online phase takes action on three levels: on the container level, on the node level, and on the cluster level. We envision two components (online monitoring component and online resources adjusting component) interacting with each other to be able to respond to unforeseen changes in the temporal behavior of runtime tasks. The online monitoring component is in charge of monitoring the real-time and health aspects

of applications. This aspect is described along with the general framework for deploying and implementing an online container orchestrator module in [23]. The authors introduce Container Level Metrics (CLM) to capture and continuously evaluate: (i) the number of *deadline misses*, (ii) the *lateness*, and (iii) the response-time of real-time tasks. Additionally, they also monitor Operating System-Level Metrics (OSLM) that convey a picture of the health of the underlying system and the containers. In [23], special attention is given to the system overhead, which can affect the temporal isolation property and system utilization. These aspects can be used to optimize the response time of tasks as well as to detect overload scenarios or starvation in BE containers.

### Runtime Actions

At runtime, there are several actions that we can take based on the CLM and OSLM measurements. The most straightforward parameter to change is the container budget. For example, when detecting a deadline miss of a task, one can increase the budget of the corresponding container, thus giving the tasks more CPU bandwidth to resume their correct behavior. The decision of when and how much to increase the budget is non-trivial as it depends on multiple aspects like the overall system utilization, the effects on other RT and non-RT containers, and the implications on the system overhead. One can also change the period of a container, e.g., to let it run more frequently. This also has complex implications on the overall system behavior and may also affect other tasks running in the same container. Additionally, the implications of changing the period at runtime on preempted but not finished tasks need to be considered.

A third dimension where one can enact changes is container allocation. If we detect at runtime that the system is becoming over-utilized or that it cannot be guaranteed the temporal correctness of all RT tasks and containers, the system may move one or multiple containers onto another (less congested) node. Here, the complexity comes from identifying which containers to move and to which node(s) to move them to [45]. Additionally, while the container budget and period are more continuous in nature since we have a whole range of possible values to choose from, the reallocation decision is inherently a binary one. Thus, at runtime, we need to be very careful when switching from slightly adjusting container parameters to deciding that one or multiple containers need to be moved.

The node-level view is depicted in Figure 9.2 where RT- and BE-containers coexist in a node, and, additionally, there is one container-level controller per RT container. The container-level controller is responsible for adapting local container-level parameters. While the controllers here are local to the container-level containers (and hence apply changes to local container values), they need to synchronize and orchestrate to node-level and cluster-level controllers. By having this controller hierarchy, we can ensure that the holistic view of the distributed system is maintained and the correct overall decisions are made. We envision simple but fast controllers that interact locally with the budget of a container (within some predefined bounds) and can react quickly to runtime violations. Moreover, a node-level controller orchestrates between the container-level controllers, e.g., to modify the allowed bounds for local budget changes and compute correct dimensioning of, e.g., container periods depending on the overall node-level system state. On the next hierarchical level, a centralized controller orchestrates the migration of containers to other nodes.

Another aspect of online redimensioning or migration is resource and task optimization. Even when no real-time requirements are violated, the controllers may decide that there are enough free resources in a node to redimension a particular container (e.g., increasing its budget) to reduce the task response times. Alternatively, a controller may detect that tasks within an RT-Container finish well before their deadlines and decide to reduce the budget in order to, e.g., optimize non-functional properties such as power consumption or give more bandwidth to BE containers.

### Control Hierarchy

The overall control is divided into three levels of hierarchy. Thus, this control structure allows to maintain the separation of concerns and allows for the implementation of different policies for each of the control levels. For instance, cluster-level control can only focus on defining the strategies for migrating containers without a need for reasoning about resource allocation for the containers (that is, the controllers on other levels). Additionally, such an architecture enables the co-existence of various policies, e.g., there can be various per-container container-level controllers. In the following section, we describe the individual controllers.

### Container-level Controller

A primary goal of the *container-level controller* is to free up resources whenever possible by lowering the budget. In this way, the utilization of the given container is reduced, releasing computing resources to be used by the node-level controller for other co-located containers. The container-level controller implements a free-up resource policy that decides the time instant of the free-up action and the number of resources to be released. The decision to reduce the budget is taken based on the measured response time of the tasks within the container ($CLM.rt^{\text{Measured}}$). If the container-level controller notices that the tasks finish with enough laxity to meet their deadlines (in a certain span of time), then the tasks must finish with enough laxity. In that case, it can decide that the tasks do not need as many computing resources as they currently have. Consequently, it may decrease their resource usage by reducing their budget. For real-time tasks, the target is to let tasks response times to be as close as possible to their deadlines to decrease the required budget and hence the deadline can be used as the target CML ($CLM.rt^{\text{Target}}$).

The container-level controller that we use in this work frees up resources proportionally to the deviation between the required response time and the measured response (denoted as $\varepsilon$). The more the budget is overshot (resulting in lower measured response time), the more it releases the overshot budget. The controller follows an Integral Controller strategy, similar to the one presented in [46], to adjust the budget $Q(t)$ allocated to the container at a point in time $t$:

$$\varepsilon(t) = CLM.rt^{\text{Target}}(t) - CLM.rt^{\text{Measured}}(t) \tag{9.12}$$

$$Q^{\text{new}}(t) = Q(t-1) - K^{\text{clc}} \cdot \max(\varepsilon(t), 0) \tag{9.13}$$

$$Q(t) = \max(Q^{\text{new}}(t), Q_{\min}) \tag{9.14}$$

where, the variable $\varepsilon(t)$ denotes the disturbances between the target and measured performance of a given container, the variable $Q^{\text{new}}(t)$ is an intermediate value of the budget calculated by the integral controller, and $Q(t)$ denotes the new value of the budget after release from the container $\Pi^{\text{rt}}$. The parameter $K^{\text{clc}} \in (0, 1]$ is the container-level controller gain, and it is a design parameter for the controller that defines how aggressively the budget should be decreased. Notice that the allocated budget is saturated to a minimum value to guarantee a minimal allocation of time. This strategy is computationally extremely simple

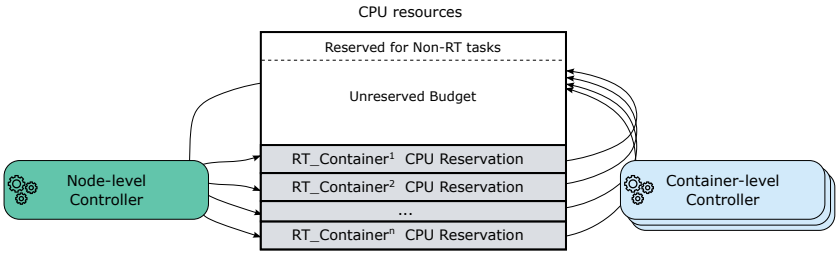



Figure 9.3: CPU resources distribution.

(thus, low overhead), and it is implemented for every RT container running in a given node.

### Node-level Controller

The node-level container has complete information about the state of the node $j \in \{1, \ldots, n\}$ (e.g., overall RT-container utilization, Operating System-Level Metrics), the state of the containers $i \in \Pi_j^{\text{rt}}$ (e.g., their budgets and periods, Container Level Metrics, etc.), and the history of the state changes. While the container-level controller has the main aim of freeing up resources whenever possible, the node-level container attempts to maintain the real-time properties of tasks by using unused computing resources and distributing them to containers that need them. This is done by increasing the budget. Both decisions need to include system-level aspects like overall utilization, effect on other RT and non-RT containers, system overhead, etc.

The node-level control aims to distribute the system resources in order to keep the total utilization of real-time containers under a predefined threshold, so even applications outside real-time containers get access to system resources. However, the controller can temporarily exceed the predefined threshold and oversubscribe resources for real-time containers, e.g., in cases of exceptional disturbances or when a migration of a container to another node is initialized. Node-level controllers keep track of the available resources in the system and define resource re-distribution policies that allocate a part of the resources to containers. The controller can also utilize predictive or probabilistic methods to adjust resources for a particular container before the container suffers from interference or a changing workload.

The node-level controller keeps track of the available CPU resources that can be distributed between RT containers as shown in Figure 9.3. By default, the controller allocates 80% of the CPU bandwidth that can be used by the RT containers. An unused part of it, which is at least 20% of CPU bandwidth, is used by non-real-time BE containers and other (system) tasks. The node-level controller, computes, for every single container $i \in \Pi_j^{\text{rt}}$, an Integral Control strategy:

$$\epsilon_i(t) = CLM.rt_i^{\text{Target}}(t) - CLM.rt_i^{\text{Measured}}(t) \tag{9.15}$$

$$Q_i^{\text{new}}(t) = Q_i(t-1) + K_i^{\text{nlc}} \cdot \min(\epsilon_i(t), 0) \tag{9.16}$$

$$Q_i(t) = \min(Q_i^{\text{new}}(t), Q_i^{\text{av}}(t)) \tag{9.17}$$

where $K_i^{\text{nlc}} \in (0, 1]$ is the node-level controller gain, and it defines how aggressively the budget should be increased while the response time of the container does not meet the target response time. Moreover, $Q_i^{\text{av}}(t)$ is the currently available budget in the node, as seen by the container $i \in \Pi_j^{\text{rt}}$, and it is computed as

$$Q_i^{\text{av}}(t) = Q_j^{\text{tot}} - \sum_{k \in \Pi^{\text{rt}} \setminus i} Q_k(t)$$

where $Q_j^{\text{tot}}$ is the 80% total budget on the computing node.

The controller continuously monitors the disturbances between the target and measured performance ($\epsilon(t)$) and attempts to increase the resources. The control mechanism is similar to the container-level controller, however, this controller can only increase resources for the controller based on the disturbance and available budget. If a container has multiple tasks then the control action calculation will be based on the worst-behaving tasks with respect to their response times and deadlines, i.e., the task that generates the longest deadline miss.

In case tasks inside a container miss $N$ subsequent deadlines ($\epsilon_i < 0$) and the budget has not increased compared with the previous instances of the controller then the cluster-level controller will be triggered to migrate this container to another node.

#### Cluster-level Controller

The cluster-level controller monitors individual nodes in the cluster and initializes the migration of containers that are not able to meet demanded real-time performance. The cluster-level controller has a holistic view of the system and can decide to migrate containers to less congested nodes. The node-level controller signals to the cluster-level controller that it cannot meet the real-time requirements of tasks by local changes only. Therefore, the decision to move one or more containers needs to be made. Additionally, the cluster-level container may decide on its own to move certain containers based on, e.g., optimization objectives. However, the controller needs to be very careful not to transform a functioning system into a non-functioning one.

The node-level container selects which container(s) to move, and the cluster-level container decides where to transfer them to. In this case, the cluster-level controller does not need to have some of the local information about the health of the containers since it does not decide which container to move.

The decision of where to move containers can be made based on more high-level information (e.g., node utilization) and not on more specific data like individual container health, so the container-level controller does not need to have a detailed view of which containers are where and their history (or, e.g., the deadline miss history of a node). As a result, the container-level controller relies on the node-level controller to dimension the container in the right way once it is assigned depending on local information. An alternative is to decide where to move/place a container based on more detailed information and dimension it with a knowledge of the parameters/health/ history of other containers on that node. The downside is that the cluster-level controller will need to be updated more frequently and have a more detailed view of each node.

### 9.4.3 Implementation

To show the feasibility and the behavior of the system, we implement the proposed Hierarchical Framework in Debian GNU/Linux 10 (buster) patched with the HCBS (Hierarchical Constant Bandwidth Server) patch[3] introduced in [14]. This patch allows controlling containers' resources at runtime without any need to modify the container runtime. The resources can be controlled by setting the

[3] Available at `https://github.com/lucabe72/LinuxPatches/tree/HCBS`

values in cgroups or by setting the values directly in Linux Kernel (e.g., in the task scheduler or in syscall handlers). We enhance the Linux Kernel to adapt CPU resources for RT containers dynamically. While the HCBS patch allows us to statically set the CPU budget and CPU period for the containers, our additional modifications allow us to dynamically adapt the CPU resources for the RT containers at runtime. We implemented container- and node-controller within the Linux Kernel (specifically, a task scheduler to compute CLM metrics and custom syscalls to trigger control actions). The controllers directly access the container, tasks, and resource reservation entities without causing additional overhead. The cluster-level controller is implemented as a user-space application that periodically observes the states of RT containers and performs migration of a deficient container(s) to another node. The migration process is described in Section 9.4.3.

The workflow and implementation overview is as follows:

- A container is executed with the initial parameters computed in the offline phase (e.g., *docker run –cpu-rt-runtime=40000 –cpu-rt-period=50000 rt_container). The initial values are stored in cgroups as cpu.rt_quota and cpu.rt_period. The values can be changed at runtime from the kernel space.*
- The scheduler selects the highest priority container with available CPU quota, i.e., containers with the earliest deadline following the SCHED_DEADLINE policy. After that, the highest priority runnable tasks are selected from each of the scheduled containers and granted a CPU.
- Each task, implemented in *struct task_struct*, is annotated with a task period and activation start. The values are used to compute the performance metrics.
- On each job activation, the Linux Kernel is notified that the job has started via a custom syscall.
- After each job is completed, the Kernel is notified about the job competition via a custom syscall, and the task is suspended until the time of the next job activation.
- Upon the finish-job syscall, the CLM metrics are updated. The CLM metrics are computed based on the activation time, finishing time, and period of the job. We compute response time, and lateness, and update the deadline miss counter. The CLM metrics are saved in */proc* filesystem and are accessible to all controllers.

- Additionally, upon the finish-job syscall, the custom syscall executes the control hierarchy as follows (as described in Section 9.4.2):
  - Container-level controller attempts to free up reserved resources if possible.
  - Node-level controller increases resources to the containers with unsatisfactory performance. If there are not enough free resources, the containers are marked as to-be-migrated (in */proc* filesystem).
- Cluster-level controller periodically checks for to-be-migrated containers and performs the migration process.

### Migration of containers

The migration of containers is performed by a custom Python-based Cluster-level controller application based on a client-server architecture. The server-side application keeps track of connected computed nodes, deployed containers, and performance metrics of the nodes and individual containers. The client side periodically reports performance metrics to the server and performs the migration of containers as depicted in Figure 9.4.

Once a container fails to obtain the necessary resources to meet the desired performance, the Node-level controller records it in /proc filesystem. This triggers the Cluster-level controller on the client side to inform the server of the container that needs to be relocated. The server then finds a host that meets the requirements mentioned in Section 9.4.1 and requests the new host to create the container. When the container is ready, the new host notifies the old host, prompting it to stop the container.

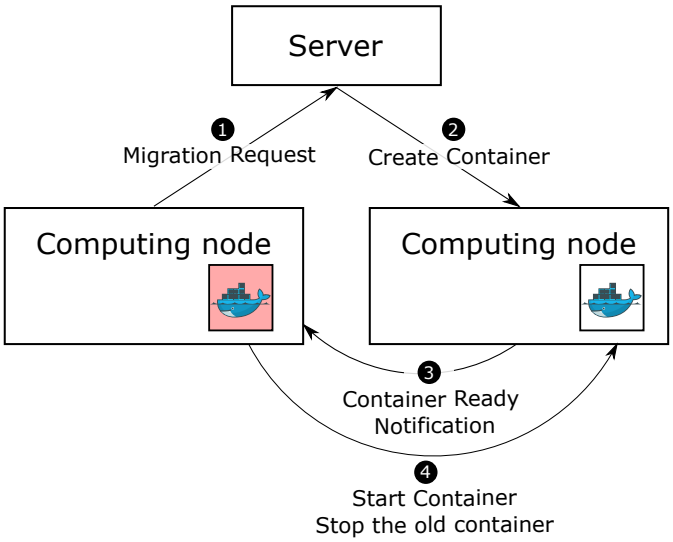



Figure 9.4: Migration process of containers.

## 9.5 Evaluation of the framework

In this section, we evaluate the proposed Hierarchical Resource Orchestration Framework. We perform the experiments on two COTS platforms: Intel Core i5, 4 GB RAM, Debian Linux 5.2.8 patched with HCBS patch [14] and Intel Core i7, 16 GB RAM, Debian Linux 5.2.8 patched with HCBS patch. Moreover, we run some scalability experiments for the offline phase using the z3 solver in version 4.11.2.

### 9.5.1 Demonstration of performance interference between containers

We illustrate the interference caused by the use of shared resources between co-located containers executed on a single computing node as shown in Figure 9.1. We perform two experiments that run a single RT container collocated with a) one non-real-time container and b) multiple non-real-time containers. In the first case (depicted in 9.1a), the co-located non-real-time container is periodically changing its workload between cache-intensive workload (matrix multiplication) and simple workload (simple busy loop). As seen in the subfigure, the response time of an RT container is aligned with the period of cache-intensive (e.g., between 0s and 20s, 40s and 62s, etc.) workload in the co-located non-real-time container. The response time ranges between 2.7s and 3.7s. In the second case (depicted in 9.1b), in addition to the RT container, we gradually execute non-real-time containers performing a cache-intensive workload. We can see that with the increased number of non-real-time containers, the response time of the RT container exhibits increasing response time. The results of the experiment show the performance interference of the container.

In the following sections, we first evaluate the offline phase, and then we show the dynamic adaptation of the CPU resources that can deal with performance disturbances of the executed containers by the proposed hierarchical framework.

### 9.5.2 Evaluation of the offline phase

In the offline phase, we study the scalability of solving the constrained optimization problem as defined in Section 9.4.1. Generally, the allocation problem

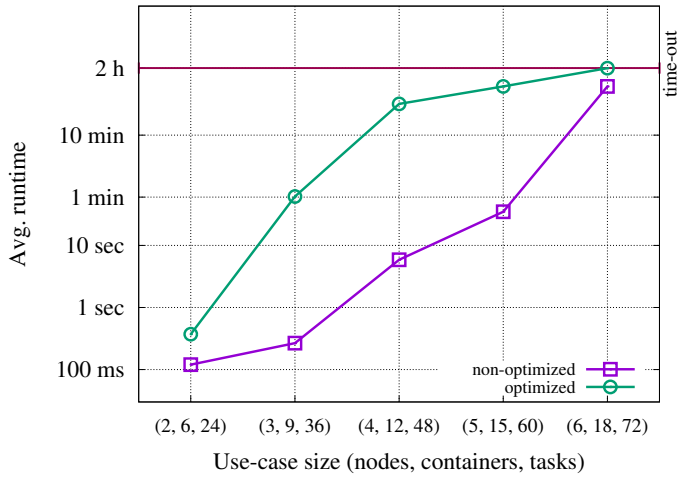



Figure 9.5: Scalability results for the offline dimensioning and allocation step with and without optimization.

in hierarchical scheduling reduces to the bin-packing problem [47] and is thus contained in the NP-hard complexity class. We implemented the constrained optimization problem using the z3 solver in version 4.11.2 [4].

We generate 5 different problem sizes in terms of the number of nodes, the number of containers, and the number of tasks, and for each problem size, we generate 10 test cases. Each node has a memory and storage availability chosen randomly between $10MB$ and $20MB$ and between $1GB$ and $2GB$, respectively. The tasks are defined by a period chosen randomly from the set $\{200, 300, 400, 500, 600, 700, 800, 900, 1000\}ms$, a WCET selected randomly between 1 and 10 ms, a storage usage randomly chosen between 2 and 8 MB, and a memory usage chosen randomly between 16 and 64 KB.

In Figure 9.5, we show the scalability of the offline assignment and dimensioning with the following system configurations (x-axis) in terms of (nodes, containers, tasks): $(2, 6, 24)$, $(3, 9, 36)$, $(4, 12, 48)$, $(5, 15, 60)$, $(6, 18, 72)$. On the logarithmic y-axis, we show the average runtime when solving the offline phase with and without optimization constraints and for each size configura-

[4] Available at `https://github.com/Z3Prover/z3`

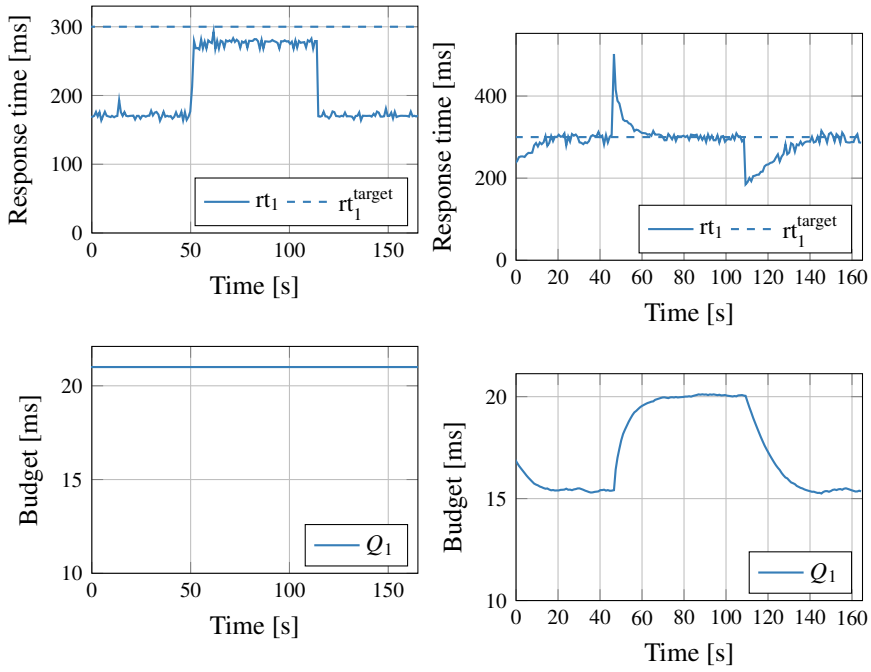
tion defined above. We set the timeout to 2 hours, after which we deem the test case to be infeasible.

As can be seen, the average runtime increases exponentially, both with and without optimization, as the size of the problem increases. Without optimization criteria, we see that also relatively large use cases can be solved in a reasonable amount of time with a system consisting of 5 nodes, 15 containers, and 60 tasks needing, on average, 34 seconds. For a system consisting of 6 nodes, 18 containers, and 72 tasks, 5 out of the 10 test cases reached the timeout, while the average runtime for the other 5 "solvable" test cases was around 1.6 minutes. As expected, when the optimization criteria of minimizing the utilization of the containers ($c_1 = 0$ and $c_2 = 1$) were enabled, the average runtime grew more quickly, leading to worse scalability. We can see that small to medium system configurations can be solved in a reasonable time, especially when optimization is not enabled, which is usually the preferred approach since the online phase will take care of freeing up unused resources and adapting to fluctuations due to system and context switch overheads. For larger systems, using SMT or OMT solvers that deliver optimal solutions will not scale anymore, making it necessary to use heuristics that sacrifice optimality and completeness for faster runtimes.

### 9.5.3 Evaluation of container-level and node-level controllers

This section discusses two types of experiments, depicted in Figure 9.6 and Figure 9.7, respectively. Both experiments focus on the online CPU budget adaptation using the joint container-level and node-level controllers that decrease/increase the container CPU budget to reach the target performance level. The first experiment (Figure 9.6) shows the response time and budget of a single container, in the case of a fixed budget (Figure 9.6a), and of a variable budget allocation (Figure 9.6b). The workload is varied within the container.

In Figure 9.6a, we statically set the replenishment quota to 21ms over the period of 25ms. This scenario does not employ any adaptation mechanism. The response time changes periodically between 170ms and 285ms, while the target response time is set to 300ms. We can see the changing response time of the container at times 45s, 116s. The allocated budget remains constant. This leads to the over-allocation of computing resources, part of the unused reserved




(a) A baseline scenario without the budget adaptation.

(b) A scenario with online budget adaptation.

Figure 9.6: An experiment showing an effect of budget adaptation.

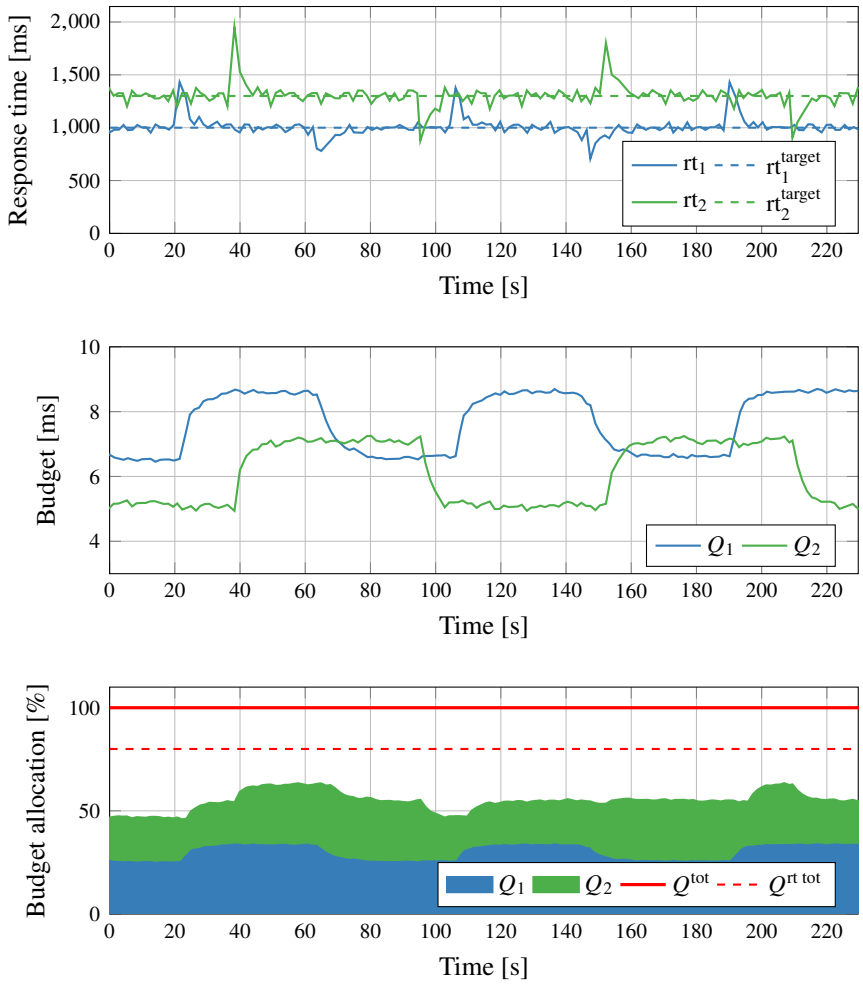



Figure 9.7: Distribution of budget amongst containers. Response times and budget allocation per container.

budget could be reserved for another RT container (that may have insufficient resources). Figure 9.6b shows the same scenario with the enabled online CPU

budget adaptation. In this case, the allocated budget is automatically adjusted to match the target response time. As there is no control nor measurement of the time-varying workload, the control mechanism dynamically reacts to its variation without any information on how long such a variation can last. The controller, therefore, serves as an iterative learning mechanism to identify the right allocation of the budget over time. In this scenario, we used two different $K$ parameters in the container- ($K^{\text{clc}} = 0.3$) and node-level ($K^{\text{nlc}} = 0.6$) controllers to have a more aggressive strategy to increase the budget and a more relaxed strategy to release the budget. When increasing the budget of a container it is better to do it as fast as possible to reduce the quality degradation in terms of deadline misses. In contrast, it is better to release the budget gradually to avoid sudden increases in response times.

The second experiment (Figure 9.7) shows two RT containers deployed on a single computing node. The figure shows a) the measured response time and target response time, b) per-container budget allocation and c) the percentage of the entire CPU budget allocation.

The containers are experiencing performance disturbances. For instance, RT container 1 experiences an increased workload between 21s and 63s, and between 108s and 148s. The RT container 2 experiences an increased workload between 39s and 95s, and between 152s and 210s. The per-container budget allocation shows that the budget is increased by the node-level controller and decreased by the container-level controller in line with the change in workload that is experienced.

The figure shows how the two controllers are able to dynamically adjust the allocated budgets to match the desired target responses of the respective containers. The bottom graph in Figure 9.7 shows how the overall budget is partitioned over the two containers. In this specific example, we do not consider the case in which the overall allocated budget exceeds the maximum share allowed for the real-time containers (indicated as a dashed red line in the graph). The case in which such a threshold is exceeded would trigger a container migration, and it is analyzed in the next section.

### 9.5.4 Evaluation of cluster-level controller

The experiment depicted in Figure 9.8 shows the collaboration between the 3-level control hierarchy proposed in this paper. On the x-axis, we depict

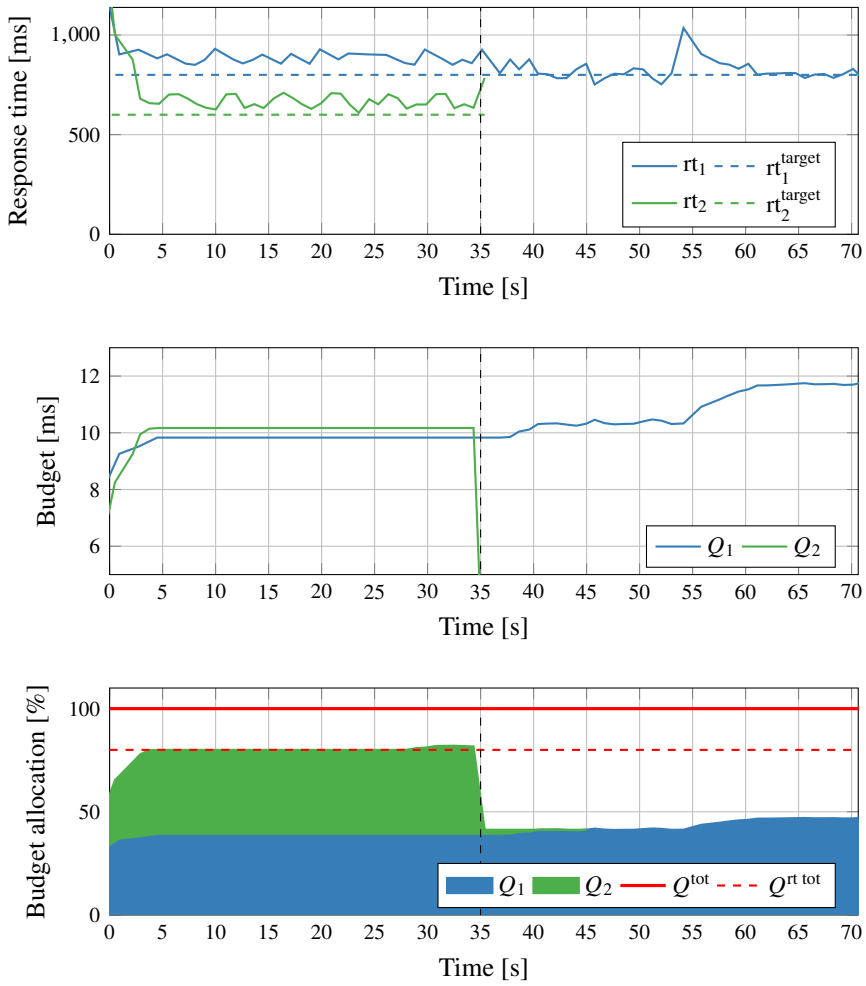



Figure 9.8: A demonstration of a cluster-level controller that migrates Container 2 on a less congested computing node.

the timeline of the system and its trace. Container- and node-level controllers are designed to balance CPU resources amongst two RT containers, while the

cluster-level controller is designed to monitor the performance of each of the RT containers, and if the performance is not as desired for a period of time, then one of the containers will be stopped and redeployed to another computing (less loaded) node.

The experiment shows two RT containers (Initial $Q_1 = 8ms$ and $Q_2 = 7.5ms$ with periods $25ms$) that are unable to reach the target response times ($800ms$ and $600ms$) due to insufficient CPU budget (top figure). On the y-axis, we depict the response time in milliseconds, showing both the desired and measured response times of the two containerized tasks with dotted and solid lines, respectively. In the middle part of the figure, we see the budget allocation for both containers (the budget being depicted on the y-axis in milliseconds). Moreover, in the bottom part of the figure, we see that the overall budget allocation is saturating at the maximum set value of 80% of total CPU bandwidth. Hence, since the node- and container-level controllers cannot change the budget allocation in such a way that both containers meet their respective response times, the cluster-level controller performs migration of one of the containers in order to relieve the computing node.

At the time 35 seconds, the cluster-level controller detects that the system can not satisfy the response-times requirements of the deployed containers (neither of the containers can reach the target response time). Thus, the controller chooses container 2 to be redeployed on another node. The container is stopped, and the allocated budget ($Q_2$) is fully released. Then, container 1 can instantly use the newly allocated budget, thus being able to fulfill its response time requirement. We can see in the top part of the figure that the response time of the remaining containerized application reaches the desired level a short time after the migration has been finished. Moreover, at time 54 seconds, we can see that there is some interference on the remaining container, causing a spike in the response time and the node-level controller being able to allocate even more of the freed-up budget to the remaining container.

## 9.6 Discussion

The experiments, that we conducted, show that the proposed online adaptation mechanism can adapt the CPU bandwidth of real-time containers and match the required performance. On the other hand, the control mechanism can also

decrease the resource for containers with over-reserved resources and these resources can be used by containers with poor performance.

The implementation in the Linux Kernel enables a more rapid control loop as well as the ability for runtime adaptation of resources without the need for so-called breath-duration periods, which are break intervals that allow a system to reach a steady state after scaling [26].

The system takes control decisions at the end of each job. However, it is possible to adjust resources during the execution of the job, not only at the end of the job. For example, if a system detects that a job may not finish before the deadline, the system may allocate more resources at the time instant of the detection of a possible delay. However, we keep this topic for future work.

The limitation of the experiment is the use of synthetic tasks; we are seeking a use-case to demonstrate our control mechanism for real-world application (e.g., autonomous driving, robot control). Also, previously we defined CLM and OSLM metrics that contain a multitude of values that describe the real-time performance of containers and the system; however, in our experiments, we use only the basic response time metric.

## 9.7 Conclusion

In this paper, we propose a Hierarchical Resource Orchestration Framework for Real-Time Containers that aims to mitigate the timing disturbances present in container-based virtualization due to, e.g., the effects of the use of shared resources, which can affect the temporal behavior of real-time containers. The proposed framework contains two phases: an offline phase that decides the initial deployment and dimensioning of the real-time containers and an online phase that complements the initial deployment to re-adjust the resources when unexpected changes occur.

We implement the framework on a real Linux-based system patched with the HCBS patch to show the feasibility of the proposed framework. In our experiments, we demonstrate a) the effect of performance interference between containers, b) the control of resources for the containers, c) the re-distribution of resources between the containers, and d) a migration of containers when the overall resources are not sufficient to meet the real-time needs of the containerized applications. Moreover, we also perform a scalability evaluation of an SMT-based implementation of the offline phase. Our experiments show

that the proposed framework can mitigate timing disturbances of containerized applications in Linux-based systems and is able to adjust and re-distribute computing resources between containers when needed.

There are several interesting directions for future work. First, the proposed framework utilizes a simple control mechanism to reserve and distribute CPU resources for the RT containers. However, more complex control, decision, and predictive algorithms are needed to capture more complex scenarios. We aim to extend the cluster-level controller with the ability to identify the optimal set of containers to migrate since it may be more beneficial to migrate the containers that are causing performance interference instead of those with poor real-time performance. Additionally, the offline phase utilizes SMT (or OMT) solvers that deliver optimal solutions but does not scale for large systems. Hence, we aim to investigate efficient heuristics that can scale for large systems, like those found in some industrial automation applications.

# Bibliography

[1] Sébastien Vaucher, Rafael Pires, Pascal Felber, Marcelo Pasin, Valerio Schiavoni, and Christof Fetzer. Sgx-aware container orchestration for heterogeneous clusters. In *2018 IEEE 38th International Conference on Distributed Computing Systems (ICDCS)*, jul 2018.

[2] Marcello Cinque, Domenico Cotroneo, Luigi De Simone, and Stefano Rosiello. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Future Generation Computer Systems*, 129:315–330, 2022.

[3] Theodosios Gkamas, Vasileios Karaiskos, and Sotirios Kontogiannis. Performance evaluation of distributed database strategies using docker as a service for industrial iot data: Application to industry 4.0. *Information*, 13(4):190, 2022.

[4] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. Industry 4.0. *Business & information systems engineering*, 6(4):239–242, jun 2014.

[5] Mohammed Salman Shaik, Vaclav Struhar, Zeinab Bakhshi, Van-Lan Dao, Nitin Desai, Alessandro V. Papadopoulos, Thomas Nolte, Vasileios Karagiannis, Stefan Schulte, Alexandre Venito, and Gerhard Fohler. Enabling fog-based industrial robotics systems. In *25th IEEE Conference on Emerging Technologies and Factory Automation (ETFA)*, Sep. 2020.

[6] Shaik Mohammed Salman, Václav Struhár, Alessandro Vittorio Papadopoulos, Moris Behnam, and Thomas Nolte. Fogification of industrial

robotic systems: Research challenges. In *Proceedings of the Workshop on Fog Computing and the IoT (Fog-IoT)*, Apr. 2019.

[7] Michael Sollfrank, Frieder Loch, Steef Denteneer, and Birgit Vogel-Heuser. Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation. *IEEE Transactions on Industrial Informatics*, 17(5):3566–3576, 2020.

[8] Luca Abeni and Giorgio Buttazzo. Resource reservation in dynamic real-time systems. *Real-Time Systems*, 27(2):123–167, jul 2004.

[9] 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*. ACM, apr 2022.

[10] Luca Abeni, Alessandro Biondi, and Enrico Bini. Partitioning real-time workloads on multi-core virtual machines. *Journal of Systems Architecture*, 131:102733, 2022.

[11] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. Q-clouds: managing performance interference effects for qos-aware clouds. In *Proceedings of the 5th European conference on Computer systems - EuroSys'10*, 2010.

[12] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Proceedings. 2003 International Symposium on System-on-Chip (IEEE Cat. No.03EX748)*. IEEE Comput. Soc, 2003.

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

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

[15] 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, Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[16] Alexandru Moga, s Thanikesavan Sivanthi, and Carsten Franke. Os-level virtualization for industrial automation systems: are we there yet? In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, apr 2016.

[17] Thomas Goldschmidt, Stefan Hauck-Stattelmann, Somayeh Malakuti, and Sten Grüner. Container-based architecture for flexible industrial control applications. *Journal of Systems Architecture*, 84:28–36, mar 2018.

[18] Marcello Cinque and Domenico Cotroneo. Towards lightweight temporal and fault isolation in mixed-criticality systems with real-time containers. In *2018 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)*, jun 2018.

[19] Marcello Cinque, Raffaele Della Corte, Antonio Eliso, and Antonio Pecchia. Rt-cases: Container-based virtualization for temporally separated mixed-criticality task sets. In *31st Euromicro Conference on Real-Time Systems (ECRTS 2019)*, 2019.

[20] Marco Barletta, Marcello Cinque, and Raffaele Della Corte. Hierarchical scheduling for real-time containers in mixed-criticality systems. In *2021 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*, oct 2021.

[21] Marco Barletta, Marcello Cinque, Luigi De Simone, and Raffaele Della Corte. Achieving isolation in mixed-criticality industrial edge systems with real-time containers. In *34th Euromicro Conference on Real-Time Systems (ECRTS 2022)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2022.

[22] Luca Abeni, Giuseppe Lipari, and Juri Lelli. Constant bandwidth server revisited. *ACM SIGBED Review*, 11(4):19–24, jan 2015.

[23] 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)*, sep 2021.

[24] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Riccardo Mancini, and Carlo Vitucci. Strong temporal isolation among containers in OpenStack for NFV services. *IEEE Transactions on Cloud Computing*, 2021.

[25] Marco Barletta, Marcello Cinque, Luigi De Simone, and Raffaele Della Corte. Introducing k4.0s: a model for mixed-criticality container orchestration in industry 4.0, 2022.

[26] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Autonomic vertical elasticity of docker containers with elasticdocker. In *2017 IEEE 10th international conference on cloud computing (CLOUD)*, jun 2017.

[27] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, jul 2019.

[28] Shashank Shekhar, Hamzah Abdel-Aziz, Anirban Bhattacharjee, Aniruddha Gokhale, and Xenofon Koutsoukos. Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications. In *2018 IEEE 11th International Conference on Cloud Computing (CLOUD)*, jul 2018.

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

[30] Luis Almeida and Paulo Pedreiras. Scheduling within temporal partitions: Response-time analysis and server design. In *Proceedings of the fourth ACM international conference on Embedded software - EMSOFT'04*, 2004.

[31] E. G. Coffman, M. R. Garey, and D. S. Johnson. *Approximation Algorithms for Bin Packing: A Survey*. 1996.

[32] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. *νz* - an optimizing smt solver. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 194–199, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[33] Clark Barrett and Cesare Tinelli. *Satisfiability Modulo Theories*, pages 305–343. Springer International Publishing, Cham, 2018.

[34] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. Cvc4. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification*, pages 171–177, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[35] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 smt solver. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 93–107, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[36] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. The opensmt solver. In Javier Esparza and Rupak Majumdar, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 150–153, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[37] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Proc. TACAS*, pages 337–340, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[38] Computer Science Laboratory – SRI International. The Yices SMT Solver. `http://yices.csl.sri.com/`. retrieved 4-Jan-2023.

[39] Bruno Dutertre. Yices 2.2. In Armin Biere and Roderick Bloem, editors, *Computer-Aided Verification (CAV'2014)*, Lecture Notes in Computer Science, pages 737–744. Springer, July 2014.

[40] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. The SMT competition 2015-2018. *J. Satisf. Boolean Model. Comput.*, 11(1):221–259, 2019.

[41] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. Smt-based model checking for recursive programs. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 17–34, Cham, 2014. Springer International Publishing.

[42] Sanjit A. Seshia and Jonathan Kotker. Gametime: A toolkit for timing analysis of software. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 388–392, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[43] Jan Peleska, Elena Vorobev, and Florian Lapschies. Automated test case generation with smt-solving and abstract interpretation. In Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods*, pages 298–312, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[44] Silviu S. Craciunas and Ramon Serna Oliver. Combined task- and network-level scheduling for distributed time-triggered systems. *Journal of Real-Time Systems*, 52(2):161–200, 2016.

[45] Alessandro Vittorio Papadopoulos and Martina Maggio. Virtual machine migration in cloud infrastructures: Problem formalization and policies proposal. In *IEEE 54th Annual Conference on Decision and Control (CDC)*, New York, NY, USA, Dec. 2015. IEEE.

[46] Alessandro Vittorio Papadopoulos, Martina Maggio, Alberto Leva, and Enrico Bini. Hard real-time guarantees in feedback-based resource reservations. *Real-Time Systems*, 51(3):221–246, Jun. 2015.

[47] Brandon Woolley, Susan Mengel, and Atila Ertas. An evolutionary approach for the hierarchical scheduling of safety- and security-critical multicore architectures. *Computers*, 9(3), 2020.

# Chapter 10

# Paper E: Container Orchestration in Edge Computing with Fluctuating Green Energy: A Multi-Armed Bandit Approach

*Václav Struhár*, Alessandro V. Papadopoulos, Inmaculada Ayala, Mercedes Amor, Lidia Fuentes

In 17th International Conference on Ubiquitous Computing and Ambient Intelligence (UCamI 2025).

**Abstract**

Sustainable edge computing demands intelligent scheduling of containerized workloads to exploit intermittently available renewable energy at geographically distributed sites. this work introduces a Contextual Multi-Armed Bandit (CMAB) framework for green-aware container orchestration, leveraging real-time context, such as energy availability, and resource utilization, via linear CMAB algorithms. A Python-based simulator models realistic solar and wind dynamics across regions. Compared to optimal, random, and naïve baselines, our CMAB scheduler improves green-energy utilization by up to 30% and cuts brown-energy use by 20%, while maintaining application performance guarantees. These findings underscore the potential of learning-based methods for advancing energy-efficient and sustainable edge infrastructures.

## 10.1 Introduction

Container orchestration platforms automate the deployment, scaling, and management of containerized applications, providing a flexible abstraction layer on diverse compute resources. In cloud data centers, orchestrators such as Kubernetes and Docker Swarm can rely on relatively homogeneous infrastructure and predictable workload patterns to achieve high resource utilization and service reliability. However, emerging applications (e.g., augmented reality, autonomous vehicles, and environmental monitoring) require ultra-low latency and location-aware processing, which can not be met by cloud computing. The Edge Computing (EC) [1] paradigm takes advantage of the edge nodes placed at the Internet's frontier, proposing a more sustainable solution, while reducing the latency and resources demanded by the cloud. EC proposes to offload containerized services from the cloud onto nearby edge nodes/servers located in a range of one or two hops. This means that the data processing and storage of the IoT services and emerging applications can be moved to nodes located closer to where data and services are produced and consumed.

However, EC environments exhibit several unique characteristics that complicate orchestrators' placement decisions [2]: (a) Heterogeneity (Edge nodes vary widely in CPU architecture, memory capacity, and software stack), (b) Workload Uncertainty (Application demands at the edge are highly dynamic, driven by user mobility, and event-driven spikes), and (c) Resource Contention (applications co-located on the same node compete for scarce resources).

Meanwhile, the growing environmental impact of large-scale cloud computing has spurred interest in integrating renewable energy into infrastructure planning. Data centers alone are projected to consume more than 10% of global electricity by 2025 [3, 4]. EC nodes offer new opportunities to utilize locally available green energy sources, such as solar panels or small-scale wind turbines [5], to power edge infrastructure. Orchestrator decisions should take into account green energy consumption, promote its use, and adopt differentiated scheduling strategies based on containers' resource demands and delay sensitivity, while prioritizing the use of green energy whenever possible. However, the intermittent and inherently unpredictable nature of renewable energy introduces additional scheduling complexity. Traditional heuristic or optimization-based schedulers struggle to adapt in real time to these dual uncertainties of workload and power supply. Forecasting inaccuracies can lead to missed op-

portunities for green energy utilization or, conversely, to degrade QoS when energy falls short of predictions. To navigate this trade-off, we turn to online learning methods that naturally balance exploration of underutilized resources against exploitation of high-yield opportunities.

This work formulates the container placement problem in edge environments as a Contextual Multi-Armed Bandit (CMAB) problem. Each edge node is modeled as an "arm", and at each decision epoch, the orchestrator observes a context vector comprising: (a) Real-time green energy fraction (ratio of renewable to total available power), (b) Current CPU, memory and I/O utilization, (c) Recent workload trends and QoS metrics (e.g., request latency, error rates). A CMAB algorithm uses these contexts to learn a scheduling policy that maximizes cumulative green energy usage, while ensuring that application performance remains within acceptable bounds. We validate our approach using a realistic simulation platform and demonstrate that our approach achieves up to 30% higher cumulative green energy utilization and 20% lower brown-energy consumption compared to state-of-the-art heuristics and non-contextual bandit baselines, without compromising QoS guarantees.

The main contributions of this paper are as follows:

1. **Formulation as Contextual Multi-Armed Bandit.** We rigorously formulate an energy-aware container placement in edge computing as an CMAB problem, capturing both resource states and renewable energy contexts in a unified framework.

2. **Adaptive Scheduling Algorithm.** We design and implement a scalable CMAB-based scheduling algorithm that dynamically prioritizes nodes with high renewable availability, yet continues to explore under-sampled nodes to adapt to shifting conditions.

3. **Realistic Simulation Platform.** We develop a Python-based simulator that integrates multiregion solar and wind trace datasets, heterogeneous node capacities, and mixed-workload generation models, enabling reproducible evaluations under diverse scenarios.

4. **Comprehensive Evaluation.** Through extensive experiments, we demonstrate that our approach achieves up to 30% higher cumulative utilization of green energy and 20% lower brown energy consumption compared to

state-of-the-art heuristics and non-contextual bandit baselines, without compromising QoS guarantees.

The remainder of the paper is organized as follows. Section 10.3 compares with some related work. Section 10.2 provides the necessary background. Section 10.4 formalizes our system and the problem model. Sections 10.5 and 10.6 detail the formal modeling and design of our scheduler. Section 10.7 describes the simulation environment and presents our experimental results. Finally, Section 10.8 concludes and outlines directions for future work.

## 10.2 Background: Multi-Armed Bandits

Multi-Armed Bandit (MAB) constitutes a straightforward but highly effective framework for algorithms that make sequential decisions in applications involving a degree of uncertainty [6]. Examples of applications of MAB are online advertising, clinical trials, website optimization, and finance.

An MAB algorithm has $K$ possible actions to choose from, also known as arms, and $T$ rounds. In each round, the algorithm chooses an arm and collects a reward for this arm. The reward is drawn independently from a fixed distribution (i.e., depends only on the chosen arm) but is unknown to the algorithm [6]. The algorithm receives feedback only for the selected arm after each round, while the rewards for the non-chosen arms remain unknown. Therefore, the algorithm must experiment with various arms through exploration to acquire new information. Thus, the algorithm needs a trade-off between exploration and exploitation: making optimal near-term decisions based on the available information. This trade-off arises in numerous application scenarios and is essential in MAB. The algorithm aims to learn which arms are best while minimizing the time spent exploring. There are several extensions of the MAB framework, e.g., the MAB has been extended in various directions to enable more complex decision-making scenarios. For example, the Contextual Multi-Armed Bandit bandit, which adds observable contextual information into the arm-selection process, and the combinatorial MAB, which enables the simultaneous selection of multiple arms.

## 10.3 Related Work

Sustainability in orchestration involves various metrics, such as brown and green energy consumption, as well as other environment-related indicators. Several works address this issue [7, 8, 9, 10]. In this work, we compare our proposed approach with existing studies that consider cloud-edge environments and aim to increase green energy usage or reduce carbon emissions. We have excluded works focusing on Internet Energy or those centered on increasing the availability of green-powered nodes, as the complexity they introduce makes them not directly comparable to our approach.

KEIDS [11] is a scheduler with three objectives: maximize the use of green energy to minimize carbon emissions, optimize performance with minimal interference, and reduce overall energy consumption. This Multi-Objective Optimization problem is formulated as an Integer Linear Programming (ILP) problem that provides the optimal solution to map containers to pods. KEIDS shares with the approach presented here the multi-objective perspective (i.e., maximization of green energy, while maximizing performance) and energy model that considers edge nodes with different percentages of green energy. However, using ILP, which seeks the optimal solution, poses scalability problems.

Aldossary et al. [12] propose a multi-level approach using a mixed-integer linear programming (MILP) model to optimize application placement based on carbon intensity and traffic demands, aiming to minimize carbon emissions. The MILP optimization is combined with an offline heuristic that classifies nodes by their carbon emissions. Specific equations are used to calculate emissions for different network layers and fog/cloud nodes, integrating factors such as traffic demand and energy consumption. So, this work does not consider the contribution of green energy to reduce carbon emissions. In addition, it is a single-objective optimization process.

DECA [13] employs A* search algorithm and fuzzy sets to minimize energy costs and carbon emissions while ensuring sufficient capacity for application components. Its problem formulation includes resource availability constraints and considers intra- and inter-edge cloud communication costs and emissions. Like our approach, DECA must address conflicting goals. A key difference with the work presented here is that its energy model does not explicitly consider real-time fluctuations in energy source availability, such as those caused by intermittent renewable energy sources.

GreenFog [14] is a framework that operates in a Fog environment powered by on-site renewable energy and supplemented by grid energy when necessary. The framework dynamically adjusts QoS of IoT applications by scaling software containers and controlling tasks in IoT devices. GreenFog uses three strategies to optimize green energy utilization: exact optimal solutions using ILP, a heuristic approach based on linear regression, and an MAB approach. The paper concludes that the ILP approach achieves better results in terms of energy saving but requires precise knowledge of energy requirements and is computationally intensive, making it less practical for real-time applications. On the other hand, if properly trained, the MAB approach can have similar results. GreenFog seeks a trade-off between QoS (similar to performance in this approach) and using green energy as we do. However, its most effective optimization strategy (i.e., the one that uses Integer programming) introduces latency in scaling operations, such as shutting down nodes, which can take 10–12 minutes, and the MAB approach used that does not consider context requires extensive training to learn system patterns effectively.

The paper [15] proposes a novel scheduling approach that balances energy efficiency with QoS requirements, considering intermittent green energy sources and available batteries to supply energy. The optimization technique is a stable matching–based heuristic that utilizes distributed controllers that interact with a global controller in the context of a federated continuum. The adaptation action is to allocate serverless functions to nodes based on green energy availability and QoS constraints. The approach is compared with three baselines, random, first-fit, and local deployment, showing a potential to reduce energy consumption while maintaining system performance. So, the approach is not compared with other solutions with energy concerns.

As we can see, a majority of the approaches rely on exact methods like ILP, which have scalability problems and are impractical for real scenarios. Approaches like GreenFog [14] demonstrate that online learning methods have potential in this context. However, they require extensive training. In this context, we explore the use of Contextual Multi-Armed Bandits that can use contextual information to take more informed decisions.

## 10.4 System Model

This section presents a system model of edge computing with fluctuating green energy, as specified below.

*Computing nodes:* Edge computing consists of geographically distributed, heterogeneous nodes ($n$) that allocate limited resources (e.g., CPU ($f_j^C$), memory ($f_j^M$), and storage ($f_j^S$)) to containerized workloads. These nodes are networked and can communicate, but their energy sources vary by location. As a result, the availability of green energy differs across nodes based on local weather conditions. Each node is aware of the instant level of green energy.

*Containers:* A container (i-th container on node j $\pi_j, i$) is a lightweight, standalone, and executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings. Each container possesses specific resource requirements (i.e., CPU ($\pi_k^C$), memory ($\pi_k^M$), storage ($\pi_k^S$)). We assume that all nodes have pre-fetched copies of the container images; thus, the container image fetching time can be disregarded.

*Green Energy:* Each computing node operates on green energy (green energy at time $t$ is denoted as $E(t)$), with the proportion of green to non-green energy fluctuating over time due to varying wind and solar intensities. We assume that it can be measured at any given time and expressed as a percentage range between 0% (no green energy) and 100% (powered purely by green energy).

## 10.5 Modeling

In CMAB, decision making is guided by additional contextual information that allows for more informed selections. A crucial aspect of modeling CMAB involves defining three fundamental elements: context representation, action space, and reward function. The context representation captures relevant features of the system that influence the choice of actions. Action space defines options that the algorithm can take. The reward function quantifies the outcomes of actions.

**Context Representation:** Each context vector $x_t$ captures the state of the system at time $t$ and includes the following metrics for each node:

- Green Energy Availability ($e_{t,n}$): Measures the proportion or absolute value of green energy available at node $n$ at time $t$.
- Schedulability Conditions ($s_{t,n}$): Reflects the ability of node $n$ to meet the timing requirements of the container, possibly indicated by metrics like current task queue lengths, average response times, and missed deadlines.
- Computational Resources ($r_{t,n}$): Includes data on CPU utilization, memory usage, disk I/O, and network bandwidth for node $n$.

Therefore, the context for each node can be represented as a concatenated vector: $x_{t,n} = [e_{t,n}, s_{t,n}, r_{t,n}]$.

**Action Space:** The action at any decision point $t$ involves selecting a node $n$ from the set of all possible nodes $N$ where the container could potentially be migrated. Thus, each action $a_t$ directly corresponds to one specific node $a_t \in N$.

**Reward Function:** The reward function $R_t(a_t, x_t)$ evaluates the outcome of migrating a container to node $n$ under the current context $x_t$. It is defined as a weighted sum of multiple criteria: $R_t(a_t, x_t) = \alpha f(e_{t,n}) + \beta g(s_{t,n}) + \gamma h(r_{t,n})$, where:

- $f(e_{t,n})$ is the energy efficiency reward that increases with higher green energy availability,
- $g(s_{t,n})$ is the schedulability reward that reflects the node's ability to meet timing guarantees,
- $h(r_{t,n})$ is the resource utilization reward that promotes efficient usage of computational resources,

and $\alpha$, $\beta$, and $\gamma$ are weights balancing the contribution of each term.

**Contextual Multi-Armed Bandit Formulation:** We formulate the container migration problem as a Contextual Multi-Armed Bandit problem. Each computing node $n \in N$ is modeled as an arm in the bandit framework. At each decision epoch $t$, the system observes a context vector $\phi(x_{t,n})$ for node $n$, which is a feature representation of $x_{t,n}$ (potentially including an intercept term). The expected reward from selecting node $n$ is assumed to be a linear function of the context $r_{t,n} = \phi(x_{t,n})^\top \theta^* + \epsilon_{t,n}$, where $\theta^* \in \mathbb{R}^d$ is an unknown parameter vector and $\epsilon_{t,n}$ is a zero-mean noise term. The objective is to maximize the cumulative reward over a time horizon $T$: $R_T = \sum_{t=1}^{T} r_{t,a_t}$.

**Feature Mapping $\phi$ for Container Migration:** In the context of container migration, the raw context vector for each node is given by:

$$x_{t,n} = [e_{t,n},\, s_{t,n},\, r_{t,n}], \tag{10.1}$$

where:

- $e_{t,n}$ is the green energy availability at node $n$ at time $t$,
- $s_{t,n}$ represents the schedulability conditions,
- $r_{t,n}$ denotes the computational resource metrics.

A feature mapping $\phi : \mathbb{R}^3 \to \mathbb{R}^d$ is applied to transform this raw context (containing green energy availability, schedulability conditions, and resource metrics) into a higher-dimensional feature space that is amenable to linear modeling. An effective choice is to include an intercept term and pairwise interaction terms among the metrics.

$$\begin{aligned} &\phi(x_{t,n}) = \\ &\begin{bmatrix} 1 & e_{t,n} & s_{t,n} & r_{t,n} & e_{t,n} \cdot s_{t,n} & e_{t,n} \cdot r_{t,n} & s_{t,n} \cdot r_{t,n} \end{bmatrix}^\top . \end{aligned} \tag{10.2}$$

This mapping produces a 7-dimensional feature vector, enabling the model to capture both the individual contributions and the interactions between the metrics, which can significantly influence the effectiveness of container migration decisions. The feature representation $\phi$ is used in Algorithm 1 in Upper Confidence Bound Calculation and Reward Observation and Update.

### 10.5.1 Learning algorithm for optimal policy: LinUCB

Our approach employs the LinUCB algorithm [16] to balance exploration and exploitation by leveraging the linear relationship between the context and the reward. For each node $n$, the algorithm maintains $A_n = \lambda I \quad$ and $\quad b_n = \mathbf{0}$, where $\lambda > 0$ is a regularization parameter and $I$ is the identity matrix. At each decision time $t$, for each node $n$, the following steps are executed:

1. **Context Observation:** Observe the context vector $x_{t,n}$ and construct its feature representation $\phi(x_{t,n})$.
2. **Parameter Estimation:** Compute the estimate of the parameter vector $\theta_n = A_n^{-1} b_n$.
3. **Upper Confidence Bound Calculation:** Compute the upper confidence bound (UCB) for node $n$:

$$p_t(n) = \phi(x_{t,n})^\top \theta_n + \alpha \sqrt{\phi(x_{t,n})^\top A_n^{-1} \phi(x_{t,n})}, \tag{10.3}$$

where $\alpha > 0$ is a tunable parameter controlling the exploration-exploitation trade-off.

4. **Node Selection:** Select the node with the highest UCB:

$$n_t = \arg\max_{n \in N} \; p_t(n). \tag{10.4}$$

5. **Reward Observation and Update:** Migrate the container to node $n_t$ and observe the reward $r_t$. Update the parameters for node $n_t$ as follows:

$$A_{n_t} \leftarrow A_{n_t} + \phi(x_{t,n_t})\phi(x_{t,n_t})^\top, \quad b_{n_t} \leftarrow b_{n_t} + r_t\,\phi(x_{t,n_t}). \tag{10.5}$$

**Algorithm 1** LinUCB for container migration decision

**Initialize:**
**for** each node $n$ **do**
    $A_n \leftarrow \lambda I$, $b_n \leftarrow \mathbf{0}$
**end for**
**for** each decision point $t$ **do**
    Select container to migrate
    Observe context $x_t$ for each node $n$
    Construct feature vector $\phi(x_{t,n})$ for each node $n$
    **for** each node $n$ **do**
        $\theta_n \leftarrow A_n^{-1} b_n$
        $p_t(n) \leftarrow \theta_n^\top \phi(x_{t,n}) + \alpha\sqrt{\phi(x_{t,n})^\top A_n^{-1} \phi(x_{t,n})}$
    **end for**
    Execute migration to node $n_t$, Observe reward $r_t$
    Update: $A_{n_t} \leftarrow A_{n_t} + \phi(x_{t,n_t})\phi(x_{t,n_t})^\top$, $b_{n_t} \leftarrow b_{n_t} + r_t\phi(x_{t,n_t})$
**end for**

The LinUCB algorithm's implementation for container migration in a fog-edge architecture is presented in Algorithm 1.

## 10.6 Orchestrator Architecture

In this section, we present the architecture of the orchestrator employing a CMAB. The orchestrator is the central component of the edge computing archi-

tecture as it drives overall system functionality and adaptability. Continuously gathers contextual information and uses this data to decide CMAB to migrate containers. The architecture of the orchestrator (see Figure 10.1) contains the following components:

- *Contextual Data Acquisition Module*to continuously gather contextual data from individual edge nodes.
- *Migration Evaluation Module* to continuously evaluate contextual data (such as green energy availability, schedulability, and resource status) to select candidate containers for migration.
- *CMAB* to perform the CMAB algorithm for each of the candidate containers for the migration (it decides where to migrate the container).
- *Constraints Checking Module* to ensure the feasibility of container migrations, compensating for the lack of constraint checks within the CMAB module.
- *Container Migration Manager* to trigger and manage the migration process in the edge computing environment.

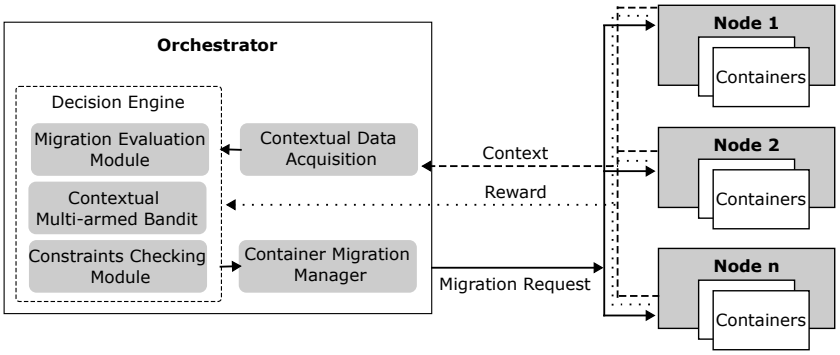



Figure 10.1: Overview of the orchestrator.

## 10.7 Experimental evaluation

We developed a Python-based simulator for experimentation with CMAB-based and alternative orchestration strategies [17]. It supports infrastructure modeling, green energy dynamics, and policy tuning, enabling analysis of reward/re-

gret behavior and scalable deployment scenarios.

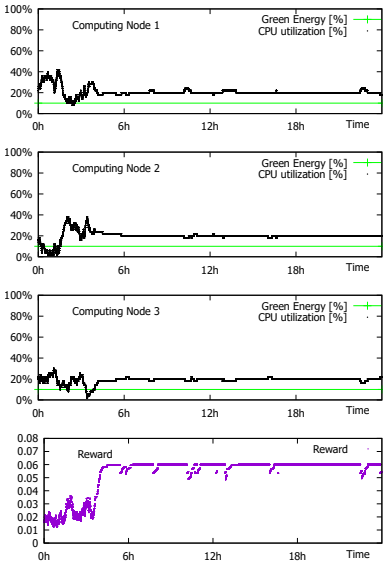
This section evaluates the system's simulated performance under diverse scenarios, each defined by unique infrastructure setups and green energy patterns. During a 24-hour period, nodes powered by varying renewable sources host container deployments, where placement decisions directly influence the cumulative reward of the system. Figures from 10.2a to 10.2c show the results of these experiments. In the figures, the top plots (denoted as Node 1, Node 2, ..., Node x) show the CPU utilization (shown as black lines or dots) and green energy levels (green lines). The higher the amount of containers deployed (thus the higher CPU utilization), the higher the lines (more extended deployment) or the dots (shorter deployment) are. The CPU utilization is expressed with values between 0% to 100%. Similarly, the green lines show the levels of green energy powering the respective nodes. After the node plots, the instant ($R_t$) reward is shown.

The first scenario (Figure 10.2a) shows three nodes with 30 containers in total. The green energy is constant and at the same level in all three nodes. After the initial exploration, the distribution of the containers evenly between all nodes, as the reward of CMAB depends on both green energy utilization and resource utilization given by the context.

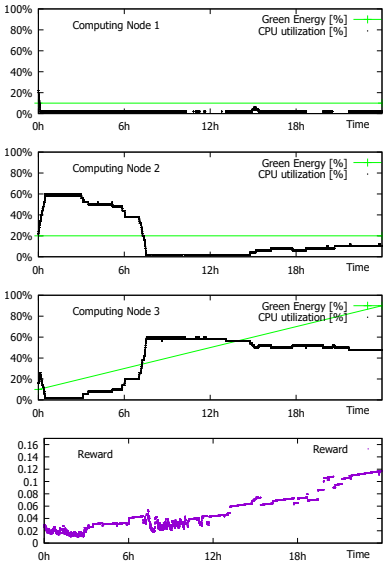
In the second scenario (Figure 10.2b), the green energy is constant in the first two nodes at 10% and 20%, respectively. The green energy at the third node increases linearly over time from 10% to 80%. So, we confirm the adaptability of the proposed method. At the beginning, the CMAB favors the second edge node as it offers the highest proportion of green energy. As the green energy in the third node surpasses the green energy in the second node (20%), the CMAB starts preferring the third node to migrate the containers to this node.

The scenario depicted in Figure 10.2c explores a more variable fluctuation in green energy. There are five nodes and 30 containers in total. There are spikes in green energy in nodes 1, 2, and 3. Node 4 has decreasing green energy, and node 5 has increasing green energy. We can see that the containers' deployment is aligned with the green energy levels. The most favorable is the first node, as it offers the highest proportion of green energy in the spikes. The system prefers the other available nodes when green energy ceases in the first node. Initially, node four is favored. Over time, as green energy availability increases in node 5, the preference gradually shifts toward node 5.

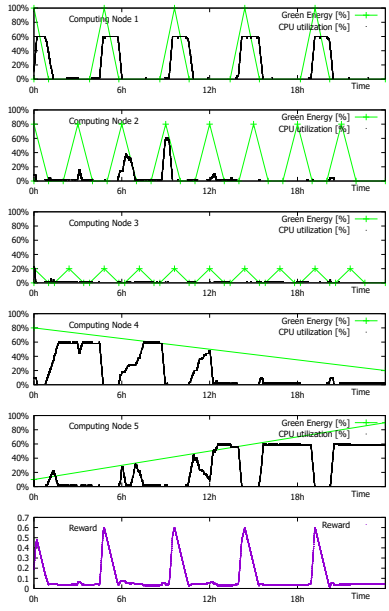
The previous experiments show artificially defined green energy levels to



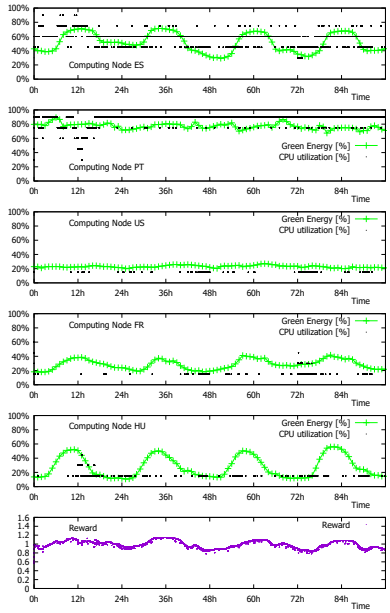

(a) Scenario 1: constant green energy.




(b) Scenario 2: linear green energy increase.




(c) Scenario 3: dynamic green energy.




(d) Scenario 4: green energy from Electricity Maps.

Figure 10.2: Scenarios 1 to 4.

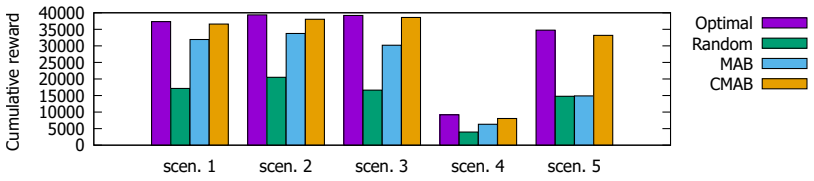



Figure 10.3: Comparison of different decision-making policies.

illustrate the behavior of CMAB. Here, we utilize the actual green energy distribution as recorded by Electricity Maps (`http://electricitymaps.com`). The Electricity Maps provides dataset records of hourly Carbon-free energy percentage. The setup consists of five nodes located in the United States, Portugal, Spain, France, and Hungary. The simulation spans over four days (August 20th-24th, 2024), during which four distinct increases in green energy occur due to solar availability in Spain and Hungary's edge nodes. The experiment shows the preference for deployment in the second node as it offers significantly more green energy. In this experiment, there are 10 containers, and each container requires 15% of CPU, thus they can not be deployed on a single node and need to be dispersed between more nodes. In this scenario, the first node is the second preferred one.

Lastly, we evaluated five different scenarios, comparing different deployment methods as shown in Figure 10.3. Each of the bars shows total reward; thus, the higher the bar, the better the deployment method in the specific scenario. The results show that CMAB performs comparably to the best deployment. We conducted this comparison in relatively small simulated infrastructures (ranging from 3 to 5 nodes with 5 containers) since an exhaustive search may not be feasible for larger infrastructures.

The conclusion of these experiments is that the CMAB approach is able to adapt to fluctuating green energy levels. Even for extremely fluctuating scenarios like Figure 10.2c, it is able to select the most beneficial node. For small scenarios, their results are nearly optimal.

## 10.8 Conclusion

This paper highlights the potential of Contextual Multi-Armed Bandit for optimizing energy-efficient scheduling in geographically distributed edge-computing systems. By prioritizing computing nodes with higher availability of fluctuating green energy, our approach enhances sustainability in resource allocation. Through a series of simulated experiments, we demonstrate the feasibility and effectiveness of this method compared to alternative policies, including the optimal policy, random selection, and a naive multi-armed bandit approach. The results underscore the advantages of our strategy in advancing energy-aware scheduling, contributing to the broader vision of greener infrastructures.

# Bibliography


[1] G. Premsankar, M. Di Francesco, and T. Taleb. Edge computing for the Internet of Things: A case study. *IEEE Internet of Things Journal*, 2018.

[2] Yuqiu Zhang, Tongkun Zhang, Gengrui Zhang, and Hans-Arno Jacobsen. Lifting the fog of uncertainties: Dynamic resource orchestration for the containerized cloud. In *Proc. of ACM Symposium on Cloud Computing*, 2023.

[3] Wei He, Qing Xu, Shengchun Liu, Tieying Wang, Fang Wang, Xiaohui Wu, Yulin Wang, and Hailong Li. Analysis on data center power supply system based on multiple renewable power configurations and multi-objective optimization. *Renewable Energy*, 2024.

[4] Sahar Ahmadisakha and Vasilios Andrikopoulos. Architecting for sustainability of and in the cloud: A systematic literature review. *Information and Software Technology*, 171:107459, 2024.

[5] Xiaojing Chen, Hanfei Wen, Wei Ni, Shunqing Zhang, Xin Wang, Shugong Xu, and Qingqi Pei. Distributed online optimization of edge computing with mixed power supply of renewable energy and smart grid. *IEEE Transactions on Communications*, 2022.

[6] Aleksandrs Slivkins. Introduction to multi-armed bandits. 2024.

[7] Marco Gaglianese, Jacopo Soldani, Stefano Forti, and Antonio Brogi. Green orchestration of cloud-edge applications: State of the art and open challenges. In *IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2023.

[8] Belen Bermejo and Carlos Juiz. Improving cloud/edge sustainability through artificial intelligence: A systematic review. *Journal of Parallel and Distributed Computing*, 2023.

[9] Yongsheng Hao, Jie Cao, Qi Wang, and Jinglin Du. Energy-aware scheduling in edge computing with a clustering method. *Future Generation Computer Systems*, 2021.

[10] Giovanni Perin, Francesca Meneghello, Ruggero Carli, Luca Schenato, and Michele Rossi. Ease: Energy-aware job scheduling for vehicular edge networks with renewable energy resources. *IEEE Transactions on Green Communications and Networking*, 2023.

[11] Kuljeet Kaur, Sahil Garg, Georges Kaddoum, Syed Hassan Ahmed, and Mohammed Atiquzzaman. Keids: Kubernetes-based energy and interference driven scheduler for industrial iot in edge-cloud ecosystem. *IEEE Internet of Things Journal*, 2020.

[12] Mohammad Aldossary and Hatem A. Alharbi. Towards a green approach for minimizing carbon emissions in fog-cloud architecture. *IEEE Access*, 2021.

[13] Ehsan Ahvar, Shohreh Ahvar, Zoltan-Adam Mann, Noel Crespi, Roch Glitho, and Joaquin Garcia-Alfaro. Deca: A dynamic energy cost and carbon emission-efficient application placement method for edge clouds. *IEEE Access*, 2021.

[14] Adel N. Toosi, Chayan Agarwal, Lena Mashayekhy, Sara K. Moghaddam, Redowan Mahmud, and Zahir Tari. Greenfog: A framework for sustainable fog computing. In *Service-Oriented Computing*. Springer Nature Switzerland, 2022.

[15] Yashwant Singh Patel and Paul Townend. A stable matching approach to energy efficient and sustainable serverless scheduling for the green cloud continuum. In *IEEE International Conference on Service-Oriented System Engineering (SOSE)*, 2024.

[16] Lihong Li, Wei Chu, John Langford, and Robert E. Schapire. A contextual-bandit approach to personalized news article recommendation. New York, NY, USA, 2010. ACM.

[17] Bandit simulation. https://github.com/struharv/BanditsSimulation. Accessed: 2025-07-14.

# Chapter 11

# Paper F: RT-SCALER: Adaptive Resource Allocation Framework for Real-Time Containers

*Václav Struhár*, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, Alessandro V. Papadopoulos

1st International Workshop on Real-time And intelliGent Edge computing (RAGE 2022).

**Abstract**

Container-based virtualization has emerged as an advantageous deployment model in fog computing platforms since it enables the seamless co-location of applications in a heterogeneous environment with minimal overhead. For some application domains requiring a certain degree of predictability in the time domain (e.g., industrial automation), the adoption of container-based virtualization is not straightforward since the technology is not built to support real-time properties.

In this paper, we propose RT-SCALER, which is a framework for adaptive resource allocation and dimensioning for real-time containers. RT-SCALER dynamically adapts the resource reservation of real-time enabled containers in order to improve the temporal predictability of the real-time applications running within the containers. We discuss the high-level orchestration approach, relating the different control levels, and give some practical insights into node-level container adaptation.

## 11.1 Introduction

Container-based virtualization has emerged as a suitable deployment model in fog computing [1] and edge platforms [2, 3, 4], as it provides near-native performance with low memory footprints and rapid start-up times. Additionally, containers provide portability, ensuring that applications will work the same way regardless of the environment where they are deployed. Moreover, containers eliminate the additional overhead of dedicated virtualization layers and, thus, use the host's available resources more efficiently. The properties mentioned above make container-based virtualization a suitable technology for hosting applications in heterogeneous distributed environments, such as fog and cloud computing platforms. However, using containers in domains imposing safety and real-time requirements (e.g., industrial automation), requires that containers are spatially and temporally isolated and can offer some form of timing predictability for the underlying applications. While spatial isolation is a fundamental property of containers, support for real-time is still under ongoing research.

In general, real-time properties relate to the ability of applications to produce not only correct logical results but also to uphold temporal limits (deadlines) when computing the results [5]. Such temporal requirements are challenging, especially in heterogeneous environments with a dynamically changing number of containers with unpredictable workloads and access patterns to shared resources. However, only limited attempts have been made to enable real-time behavior in container-based virtualization within this research area. For instance, the Hierarchical Constant Bandwidth Server (HCBS) solution by Abeni et al. [6] provides temporal isolation of containers that share the same physical host. The HCBS consists of two-level schedulers, the global scheduling policy for the containers, while the second level is based on fixed-priority scheduling for software tasks.

Nevertheless, at runtime, container-based virtualization is prone to resource interference. As the containers share the operating system kernel of the host, the performance interference, that is, the performance isolation problem will occur between the containers due to the resource competition [7]. Resource interference may influence the execution times of the containerized applications and introduce timing unpredictability. Within this context, the HCBS solution [6] specifies CPU time reservation for RT containers in the form of a cer-

tain percentage of the CPU bandwidth over a specified period. The reservation is commonly done via reserving a CPU *budget* over the *period*. Choosing the correct amount of the budget and period is a non-trivial task, as it directly affects the timing properties of the containers. The reservation problem becomes even more challenging in heterogeneous and dynamic systems due to (i) the worst-case execution time (WCET) on the specific platform being unknown beforehand, (ii) due to the unpredictable performance interference originating in other co-located containers, and (iii) due to possible dynamic workload changes in the RT containers. Therefore, it is not sufficient to compute the resource reservation in terms of the budget and period at design time (offline phase), but it is also necessary to adapt it at runtime to improve the utilization of resources and the time-predictability of services (online phase).

In order to achieve the goal of time predictability of container-based virtualization, we propose an orchestration framework named RT-SCALER in this paper that considers both the offline and online aspects of the adaptation problem for real-time containers. Besides an offline phase, in which the container dimensioning can be done based on an static system model without considering runtime overhead, we propose an online phase that is able to respond to changes in the performance of real-time tasks as well as to changes in the required workload (e.g., when new applications or containers join the system), in order to both preserve the timeliness of applications and to utilize the available resources more efficiently. We describe the overall design of RT-SCALER along with its components and highlight the hierarchical nature of the control problem for runtime container adaptation (Sec. 11.2). Additionally, we discuss some practical insights and experimentally show the benefits of controlling the resources during runtime at the local node level (Sec. 11.3) and draw some conclusions in Sec. 11.4.

The nature of container-based virtualization provides a ground for resource adaptation. Due to rapid start-up times, it is easy to horizontally scale up the number of containers and balance the workload between them as shown in [8]. Additionally, it is simple to vertically scale the container's resources via cgroups. It is shown in [9] where authors scale container resources based on CPU utilization of containers.

## 11.2 Container orchestration

We present the high-level idea of RT-SCALER, which is a general orchestration framework for static and dynamic allocation and dimensioning of real-time containers. Real-time containers supplement the spatial isolation properties of containers with real-time capabilities relating to temporal isolation and deadline fulfillment. Adding real-time properties to containers has been addressed in [6] by introducing a hierarchical scheduling patch for Linux-based systems.

The main aim of our container orchestration, called *RT-SCALER*, is to manage the deployment and adaptation of real-time containers in distributed applications featuring heterogeneous computing nodes such that individual real-time task requirements are met. These requirements are not only related to real-time behavior but also resource usages such as memory, I/O, and disk requirements and non-functional requirements such as fault-tolerance, power consumption, or resource efficiency.

The input to our RT-SCALER orchestrator is a set of real-time tasks and containers. The containers can include either self-suspending real-time tasks with implicit (or constrained) deadlines, in which case they are labeled as real-time (RT) containers or non-real-time tasks, in which case we talk about best-effort (BE) containers. Real-time tasks are additionally defined using a worst-case execution time (WCET) and a period specifying an upper bound on the computation of the task in each period and the rate at which the task is activated. Both RT and BE containers can coexist on the same core, but they are spatially and temporally isolated. The real-time tasks are pre-allocated to containers, but the containers are not pre-allocated to computing nodes (and cores), although they can have a certain affinity set constraining the set of nodes to which they can be allocated to. As defined in [10], each RT container $\pi_k$ has additionally an RT interface consisting of $(P_k, Q_k)$ where $Q_k$ is the CPU quota within an interval (period) $P_k$, defining that the container $\pi_k$ cannot use more than $P_k$ time units over an interval of time of $Q_k$ time units.

We envision our RT-SCALER orchestrator to consist of two phases, an offline and an online one.

### 11.2.1 Offline phase

Given a set of containers and real-time applications as defined above, in the offline phase, RT-SCALER decides where to place the containers such that the real-time requirements of tasks are fulfilled. This decision will be based on two steps.

The first step is to calculate a set of ideal RT interfaces for the RT containers, i.e., compute $(P_k, Q_k)$ for every RT container $\pi_k$. This is similar to the non-trivial server design problem [11, 12]. In this case, the server design problem is to compute the optimal budget and period of all containers given the set of real-time tasks and their assignment to containers. There are several (computationally intensive) methods to compute the optimal server parameters (e.g. [13, 14]) for both fixed- and dynamic priority schedulers, which rely on a worst-case schedulability analysis of the tasks within the server. Since we know the underlying scheduling mechanism (fixed priority) used to dispatch tasks within a container, we can employ a response time analysis (e.g. [15]) under the worst-case service pattern assumption for the given RT interface to compute the optimal RT interface $(P_k, Q_k)$.

After solving the server design, the allocation of containers to cores becomes an optimization problem similar to the bin-packing problem, and thus NP-hard. There are, however, efficient offline heuristics that can offer near-optimal solutions in a reasonable time even for larger problem sizes [16].

Finally, BE containers need to be assigned, and this can be done either in a random or more balanced fashion. A balanced assignment of BE containers could take into account, e.g., the number of BE tasks in the container and the remaining CPU bandwidth of each node. Once assigned, the remaining bandwidth on each node can be distributed (uniformly or non-uniformly) to the respective BE containers.

The offline phase is not sufficient to guarantee the desired real-time behavior of applications at runtime. It is too complex (and unrealistic) to build an exact model of the underlying system and the runtime interactions between the containers in order to be able to rely solely on the resulting offline dimensioning of containers. Runtime effects may lead to a variance in the temporal behavior since the temporal isolation is not perfect and influenced by runtime artifacts (e.g., cache effects, interrupts) and system overhead. Hence, the ideal server dimensioning done at the offline stage may not lead to the desired behav-

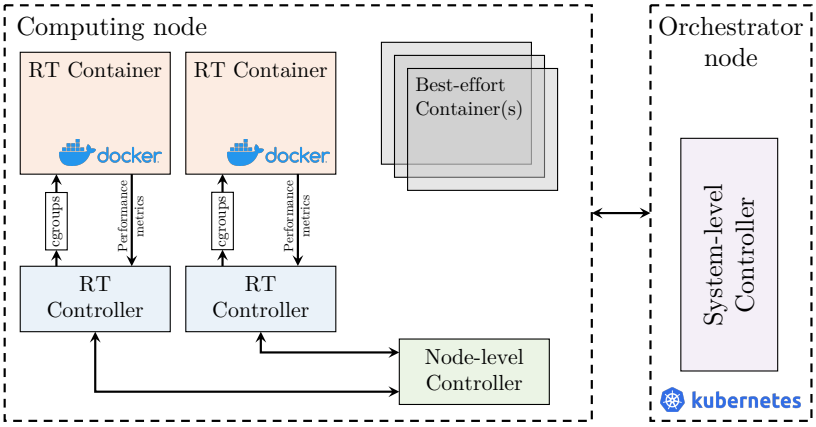



Figure 11.1: Overview of the system.

ior in practical deployments. Additionally, new applications or containers can be released in the system, requiring a runtime adaptation of existing containers or assignment of new containers to the available (and possible) cores.

### 11.2.2 Online phase

In the online phase, we envision two components interacting with each other to be able to respond to unforeseen changes in the temporal behavior of runtime tasks. One component is online monitoring of the real-time and health aspects of applications. This aspect is described along with the general framework for deploying and implementing an online container orchestrator module in [10]. In [10], the authors introduced Container Level Metrics (CLM) to capture and continuously evaluate: (i) the number of *deadline misses*, (ii) the *lateness*, and (iii) the response-time of real-time tasks. Additionally, we also monitor Operating System-Level Metrics (OSLM) that give us a picture of the health of the underlying system and the containers. In [10] the authors give special attention to the system overhead, which can affect the temporal isolation property and system utilization which can be used to optimize the response time of tasks as well as to detect overload scenarios or starvation in BE containers.

At runtime, there are several actions that we are able to take based on the CLM and OSLM measurements. The most straightforward parameter to

change is the container budget. For example, when detecting a deadline miss of a task, we can increase the budget of the corresponding container, thus giving the tasks within more CPU bandwidth to resume their correct behavior. Naturally, the decision of when and how much to increase the budget is non-trivial as it depends on multiple aspects like the overall system utilization, the effects on other RT and non-RT containers, and the implications on the system overhead. We can also change the period of a container, e.g., to let it run more frequently. This also has complex implications on the overall system behavior and may also affect other tasks running in the same container. Additionally, the implications of changing the period at runtime on preempted but not finished tasks need to be considered. A third dimension where we can enact changes is the container allocation. If we detect at runtime that the system is becoming overutilized or that we cannot guarantee the temporal correctness of all RT tasks and containers, the system may move one or multiple containers onto another (less congested) node. Here, the complexity comes from identifying which containers to move and to which node(s) to move them. Additionally, while the first two parameters were (somewhat) more continuous in nature since we have a whole range of possible values to choose from, the reallocation decision is inherently a binary one. Thus, at runtime, we need to be very careful when switching from slightly adjusting container parameters to deciding that one or multiple containers need to be moved.

The node-level view is depicted in Fig 11.1 where RT- and BE-Containers coexist in a node, and, additionally, there is one RT-Controller per RT container. The RT-Controller is responsible for adapting local container-level parameters. We do not detail the specifics of what type of controller to use since this is ongoing work, but we envision a runtime adaptation based on control theory. While the RT-Controllers here are local to the RT containers (and hence apply changes to local container values), they need to synchronize and orchestrate to node-level and system-level controllers. By having this controller hierarchy, we can ensure that the holistic view of the distributed system is maintained and the correct overall decisions are made. We envision simple but fast controllers that interact locally with the budget of a container (within some predefined bounds) and can react quickly to runtime violations. Moreover, a node-level controller needs to orchestrate between the RT-Controllers, e.g., to modify the allowed bounds for local budget changes and compute correct dimensioning of, e.g., container periods depending on the overall node-level system state.

On the next hierarchical level, a centralized controller needs to orchestrate the migration and reallocation of containers to other nodes.

Another aspect of online redimensioning/reallocation is resource and task optimization. Even when no real-time requirements are violated, the RT-Controllers may decide that there are enough free resources in a node to redimension a particular container (e.g., increasing its budget) to reduce the task response times. Alternatively, an RT-Controller may detect that tasks within an RT-Container finish well before their deadlines and decide to reduce the budget in order to, e.g., optimize non-functional properties such as power consumption or give more bandwidth to BE containers.

## 11.3 Practical Insights

This section provides a brief insight into the local control of RT-Containers via a simple PID loop to underscore the potential of runtime adaptation in real-time containers.

The underlying container system in our work is based on the HCBS patch[1] by Abeni et al. [6] that provides temporal isolation of containers sharing the same physical host. The patch is consistent with existing real-time analysis (some results show that it is compatible with the Multiple Periodic Resource Model analysis). The patch hierarchically chains two existing scheduling policies (SCHED_DEADLINE and SCHED_FIFO). The SCHED_DEADLINE scheduling policy implements the Constant Bandwidth Server (CBS) algorithm as the root scheduling policy of the scheduling hierarchy. The second level scheduling policy is fixed-priority scheduling policy SCHED_FIFO. The scheduler provides an interface to control the parameters of the scheduling policies via cgroups. In a cgroups virtual file system, 'cpu.rt_runtime_us' serves to control CPU time reservation for each RT container.

We enhance the Linux Kernel with an online RT task monitoring module and an adaptation module for adapting local container-level parameters. The task monitoring module recognizes containerized real-time tasks that run within the context of HCBS patch, and it continuously collects RT-related performance metrics [5, 10]. In our experiments, we consider the response time of the self-suspending periodic task. However, the module also collects other

[1] Available at: `https://github.com/lucabe72/LinuxPatches/tree/HCBS`

data consistent with CLM metrics defined in [10]. The module timestamps scheduler-related events. The adaptation module aims to continuously adjust the budget for the real-time containers in a reactive manner in order to keep the real-time performance stable. The adaptation module interacts with the monitoring module to obtain monitoring data (CLM, OSLM) and utilizes a PID controller to adapt the budget of the RT container.

We perform an experiment to demonstrate the adaptation process on the Intel i5 computer with 8GB RAM using Debian Linux (Kernel 5.2.) patched with the Hierarchical Scheduling Patch, and running Docker v20.10.

An experiment showing the feasibility of our idea is depicted in Fig. 11.2 and Fig. 11.3. Fig. 11.2 shows a possible scenario when the response-time of an RT container is affected by unforeseen circumstances. We simulated such a change by changing the workload in the container. At the $60^{th}$ and $180^{th}$ job instances, the workload increases by 30%. At the $120^{th}$ and $240^{th}$ job instance the workload returns to its original level. The RT budget of the RT container is reserved to fulfill the target response time ($200ms$). However, any workload change or system interference may affect the Quality of Service of the RT container at runtime such that it is not able to deliver demanded performance.

Fig. 11.3 shows the same scenario that employs RT-SCALER. The monitoring module continuously keeps track of the response times of the containerized tasks. The adaptation module changes the RT budget based on the measured response time. As can be seen, the online adaptation quickly responds to the changing workload and keeps the response time closer to the desired value represented by the blue horizontal line.

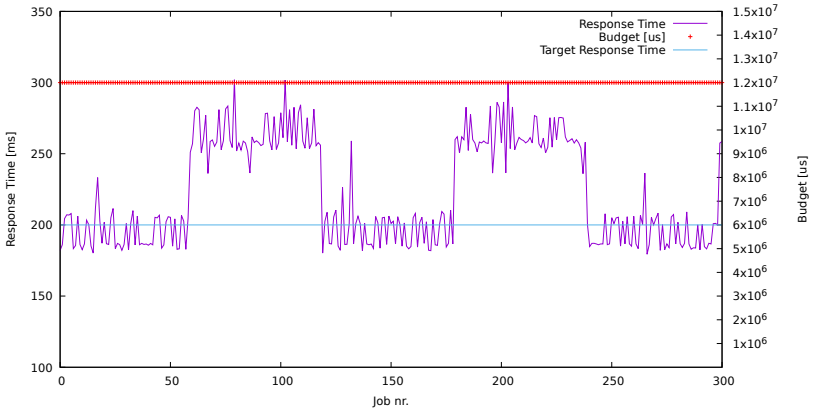



Figure 11.2: Response time of a dynamically changing containerized task without online adaptation.

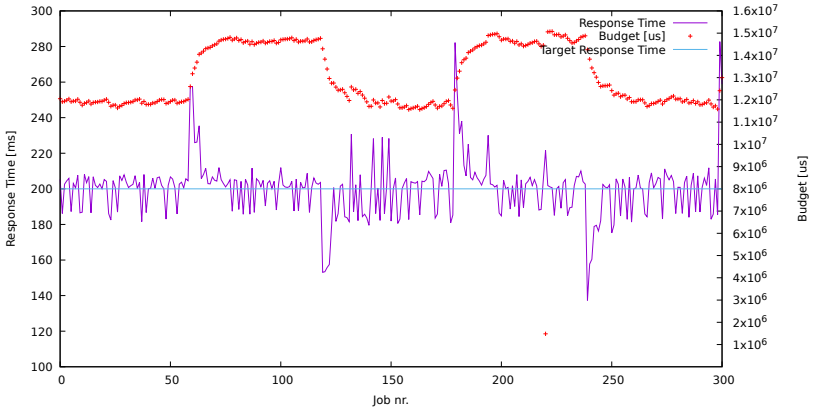



Figure 11.3: Response time of a dynamically changing containerized task with online adaptation.

## 11.4 Conclusion

We presented RT-SCALER, a container orchestration framework that introduces a two-phased orchestration approach, aiming to preserve RT performance in a multi-container environment. The initial offline phase of the system attempts to compute the theoretically optimal set of RT parameters for the RT containers. This phase is based on the theoretical background of optimal server dimensioning. However, the computed values may not be ideal in real-world heterogeneous systems that may suffer from interference artifacts or workload changes requiring some form of online adaptation. Thus, we introduced an online phase that adapts the RT container values at runtime based on a hierarchical approach. We presented the general RT orchestrator design, proposed the system's architecture, and showed an experiment that demonstrates the feasibility of the idea at the local RT container level.

In future work, we want to investigate different adaptation strategies of real-time containers. For example, the adaptation mechanism could predict workload from previous historical data and proactively dimension the resources. Moreover, we want to address the control challenge of deciding which server parameter to change (including the decision to relocate an RT container to another node) and experimentally evaluate the complex control loop across different hierarchical levels in distributed edge computing applications.

# Bibliography


[1] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proc. MCC*, 2012.

[2] Walid A Hanafy, Amr E Mohamed, and Sameh A Salem. A new infrastructure elasticity control algorithm for containerized cloud. *IEEE Access*, 2019.

[3] Corentin Dupont, Raffaele Giaffreda, and Luca Capra. Edge computing in iot context: Horizontal and vertical linux container migration. In *Proc. GIoTS*, 2017.

[4] Roberto Morabito. Virtualization on internet of things edge devices with container technologies: A performance evaluation. *IEEE Access*, 2017.

[5] Giorgio C Buttazzo. *Hard real-time computing systems: predictable scheduling algorithms and applications*. 2011.

[6] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. Container-based real-time scheduling in the Linux kernel. *SIGBED Rev.*, 2019.

[7] Cao Jiqing. I/O performance optimization analysis of container on cloud platform. In *Proc. ICPICS*, pages 84–86, 2020.

[8] Henning Titi Ciptaningtyas, Bagus Jati Santoso, and Muhammad Fahrur Razi. Resource elasticity controller for docker-based web applications. In *Proc. ICTS*, 2017.

[9] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. Autonomic vertical elasticity of docker containers with elasticdocker. In *Proc. CLOUD*, 2017.

[10] Václav Struhár, Silviu S Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V Papadopoulos. REACT: enabling real-time container orchestration. In *Proc. ETFA*, 2021.

[11] Insik Shin and Insup Lee. Periodic resource model for compositional real-time guarantees. In *Proc. RTSS*. IEEE, 2003.

[12] Insik Shin and Insup Lee. Compositional real-time scheduling framework. In *Proc. RTSS*, 2004.

[13] Giuseppe Lipari and Enrico Bini. Resource partitioning among real-time applications. In *Proc. ECRTS*, 2003.

[14] Arvind Easwaran, Madhukar Anand, and Insup Lee. Compositional analysis framework using edp resource models. In *Proc. RTSS*, 2007.

[15] Luis Almeida and Paulo Pedreiras. Scheduling within temporal partitions: Response-time analysis and server design. In *Proc. EMSOFT*. ACM, 2004.

[16] E. G. Coffman, M. R. Garey, and D. S. Johnson. *Approximation Algorithms for Bin Packing: A Survey*. 1996.