University of Verona
Department of Computer Science

Graduate School of Natural Sciences and Engineering
Doctoral Program in Computer Science
Cycle XXXVI°

Software Optimization and Orchestration for Heterogeneous and Distributed Architectures

S.S.D. ING-INF/05

Coordinator: [illegible]
Prof. Ferdinando Cicalese

Tutor: [illegible]
Prof. Nicola Bombieri

Doctoral Student: [illegible]
Dott. Francesco Lumpp

Software Optimization and Orchestration for Heterogeneous and Distributed Architectures — Francesco Lumpp
PhD Thesis
Verona, April 17, 2024

Abstract

In the context of the Edge-Cloud computing continuum, containerization and orchestration have become two key requirements in software development best practices. Containerization allows for better resource utilization, platform-independent development, and secure software deployment. Orchestration automates the deployment, networking, scaling, and availability of containerized workloads and services. However, there are still several open challenges. First, the optimization of software tailored for edge computing, with the aim of enhancing software portability in real-time distributed environments and containerized applications within the realms of robotics and Industry 4.0/5.0 technologies. Second, the orchestration of real-time containers within mixed-criticality systems. Third, the expansion of the Edge-Cloud computing continuum through innovative runtime scheduling techniques geared towards enhancing throughput and reducing response times. This thesis tackles these challenges with a software methodology that targets various aspects of the Edge-Cloud computing continuum. The aforementioned objectives are divided into five categories that are subsequently analyzed, expanded, and optimized with techniques that allow for improved performance, safety, and reliability. It also addresses the growing demand for faster data processing and more responsive computing in the contemporary technological landscapes of industrial automation. The methodology is analyzed on a multitude of synthetic benchmarks and scenarios, but is also always verified through real-case studies, such as the software implementing the mission of a Robotnik RB-Kairos mobile robot interacting with an industrial agile production chain. The experimental results demonstrate that these objectives were achieved. First, the methodology allows for better performance on heterogeneous embedded edge devices that make use of unified memory architectures in vision-based applications. In addition, the introduction of containers into industrial automation and robotic contexts facilitates flexible software development. Furthermore, mixed-criticality environments benefit from the introduction of orchestration and the real-time plugin that allows for runtime monitoring of software. Finally, the verification and migration of assertions guarantee the reliability and safety of modern robots.

Abstract (Italian)

Nel contesto dell'Edge-Cloud computing continuum, la containerizzazione e l'orchestrazione sono diventati due pilastri delle migliori pratiche di sviluppo del software. La containerizzazione consente un migliore utilizzo delle risorse, lo sviluppo di componenti software indipendenti, e la distribuzione sicura del software. L'orchestrazione automatizza la distribuzione, la scalabilità e la disponibilità dei carichi di lavoro e dei servizi containerizzati. Tuttavia, ci sono ancora diverse sfide aperte. In primo luogo, l'ottimizzazione del software progettato per il calcolo Edge, con l'obiettivo di migliorare la portabilità del software in ambienti distribuiti real-time e nelle applicazioni containerizzate nei settori della robotica e delle tecnologie dell'industria 4.0/5.0. In secondo luogo, l'orchestrazione dei container in real-time all'interno di sistemi mixed criticality. Terzo, l'espansione dell'Edge-Cloud computing continuum attraverso tecniche di programmazione innovative, che sfruttino il real-time, volte a migliorare il throughput e a migliorare i tempi di risposta. Questa tesi affronta queste sfide con una metodologia che mira ad ottimizzare vari aspetti dell'Edge-Cloud computing continuum. Gli obiettivi menzionati sono raggruppati in cinque categorie che vengono successivamente analizzate, approfondite e ottimizzate con tecniche che consentono una migliore performance, sicurezza e affidabilità. Inoltre, affronta la crescente domanda di elaborazioni di dati più veloci e di calcolo più reattivo nei contesti tecnologici industriali. La metodologia viene verificata tramite una moltitudine di benchmark sintetici, ma viene anche sempre verificata tramite casi di studio reali, per esempio, programmando la missione di un robot mobile Robotnik RB-Kairos che interagisce con una catena di produzione industriale agile. I risultati sperimentali ottenuti dimostrano come questi obiettivi siano stati raggiunti. La metodologia migliora le prestazioni su dispositivi embedded eterogenei che utilizzano memoria unificata nelle applicazioni basate su computer vision. Inoltre, l'introduzione dei container nei contesti di automazione industriale e robotica consentono uno sviluppo più flessibile del software. Per di più, gli ambienti mixed-criticality traggono vantaggio dall'introduzione dell'orchestrazione real-time, la quale consente inoltre il monitoraggio delle violazione temporali del software. Infine, le asserzioni garantiscono l'affidabilità e la sicurezza dei robot moderni tramite verifica formale.

Acknowledgements

Devo iniziare ringraziando la mia famiglia: mia mamma, mio papá e mio fratello. Marisa, Fabrizio e Andrea. Il loro amore e supporto incondizionato nel mio viaggio nel mondo accademico hanno contribuito in modo sostanziale al mio successo. Mi avete supportato con la vostra esperienza nei momenti più difficili, e avete festeggiato con me nei momenti di successo. Grazie.

Voglio ringraziare anche la mia compagna, Martina. Mi ha instancabilmente supportato, e sopportato, attraverso tutte le attività del dottorato. Un'impresa davvero titanica. Grazie.

Vorrei inoltre esprimere la mia gratitudine al mio tutor accademico, il Prof. Nicola Bombieri. Mi ha dato l'opportunità di perseguire il dottorato di ricerca in un momento di incertezza, quando l'intero mondo stava attraversando una crisi senza precedenti, la pandemia. Una persona avrebbe potuto guardare al futuro e vedere solo incertezze. Tuttavia, il dottorato ha dato una direzione e scopo al mio tempo, riempendolo di duro lavoro, esperienze e soddisfazioni, le quali non avrei mai potuto avere non avessi perseguito questa carriera.

Vorrei includere nei ringraziamenti anche i miei collaboratori e gli altri studenti del dottorato, che grazie alle loro conoscenze ed esperianza hanno reso possibile questo lavoro di tesi. La loro collaborazione ha permesso la pubblicazione di molti lavori scientifici a riviste e conferenze di primissimo livello.

Infine, vorrei esprimere un pensiero di gratitude anche verso tutti i miei amici, i quali potrebbero non avermi supportato direttamente durante il mio percorso, ma che sono sempre stati presenti.

Grazie a tutti.

[illegible]

Contents

List of Figures

List of Tables

List of Listings

List of Acronyms

ABV	Assertion-Based Verification
AI	Artificial Intelligence
API	Application Programming Interface
ASIC	Application-Specific Integrated Circuit
CE	Computing Elements
CGRA	Coarse-Grained Reconfigurable Array
CNN	Convolutional Neural Network
CPS	Cyber-Physical System
CPU	Central Processing Unit
CRD	Custom Resource Definition
DAG	Directed Acyclic Graph
DSP	Digital Signal Processor
EDF	Earliest Deadline First
FPGA	Field-Programmable Gate Array
GPU	Graphic Processing Unit
HEFT	Heterogeneous Early Finish Time
HPC	High Performance Computing
HW	Hardware
IaaS	Infastructure as a Service
iGPU	Integrated Graphic Processing Unit
IoT	Internet-of-Things
IPC	Inter-Process Communication
ISA	Instruction Set Architecture
LLC	Last Level Cache
LTL	Linear Temporal Logic
MCB	Main Control Board
MCS	Mixed Criticality System
MILP	Mixed Integer Linear Programming
NAT	Network Address Translation
NP	Nondeterministic Polynomial Time
OS	Operating System
QoS	Quality of Service
ROS	Robot Operating System
RPC	Remote Procedure Call
RT	Real-time
SC	Standard Copy
SLAM	Simultaneous Localization And Mapping
SoC	System-on-Chip
STL	Signal Temporal Logic
SUV	System Under Verification
SW	Software

UM	Unified Memory
UMA	Unified Memory Architecture
WCET	Worst Case Execution Time
XO	Exclusive Overlapping
ZC	Zero Copy

1

Introduction

While the principles of Industry 4.0 are now reaching a certain level of maturity across Europe, the technological transition continues to look towards the future. With the concept of Industry 5.0, the vision of the industry goes beyond efficiency and productivity as the only goals and strengthens the role and contribution of the industry to society [1]. With the fifth industrial revolution, the well-being of the worker is placed at the center of the production process, and new technologies are used to provide prosperity beyond employment and growth, while respecting the limits of the planet. Research and innovation are put at the service of the transition to a sustainable, human-centered, and resilient industry [2].

In this context, human-centered robotics and intelligent manufacturing play a fundamental role. Although robots originating in large-scale manufacturing plants work very efficiently behind fences (i.e., they do not interact with human operators), they are now spreading to more and more application areas due to their potential efficiency, the reduction of their costs, and their always increasing level of autonomy. These robots are tasked with understanding the world around them, planning their actions in it, and interacting with robots and humans also working in the same environments [3].

Such a high level of autonomy is based on a combination of artificial intelligence, cognition and human-robot interaction and has two immediate correlates, which form the motivations of this thesis. First, programming robots' missions and behaviors is increasingly complex and goes well beyond traditional and restricted objectives (e.g., the control of pre-defined movements, or static environments, or very low-level scene perception for simple tasks). Programming modern robots requires the integration of a multitude of *software components* from *different domains* like low-level control, dynamic planning, computer vision, monitoring, and inference applications based on neural networks [4] (see top of Fig. 1.1). This high degree of heterogeneity in software applications requires a high degree of optimization for the computing hardware located on the robots, which are often made of embedded boards with Unified Memory Architectures or heterogeneous hardware (e.g., GPU).

Second, because these robots operate at human scale (e.g., physical human-robot-interaction in factories, warehouses, and at homes) and perform safety-critical missions (like driving or surgery), the correctness requirements are stringent and include, beside functional requirements (payload, workspace, accuracy, speed), also *extra-functional constraints* such as real-time, reliability, safety, scalability, data privacy and integrity, and energy efficiency [5]. To address all these requirements, software for robotic applications has to be configured to run on *heterogeneous computing architectures*, by which the different software components have to be properly mapped and orchestrated across heterogeneous hierarchical Edge-Cloud computing platforms [6, 7]. Furthermore, embedded platforms are often used to create the edge component of the edge-cloud computing paradigm because they require low power to perform close-to-the-data computations. In these environments there has been a growing interest in open hardware architectures and RISC-V architectures play a dominant role in this context [8, 9].

Recently, some open-source and several proprietary model-based platforms have been proposed to ease the design of robotic systems (e.g., NVIDIA Isaac [10], Amazon AWS Robomaker [11]). However, they do not address modularity, interoperability, and Software

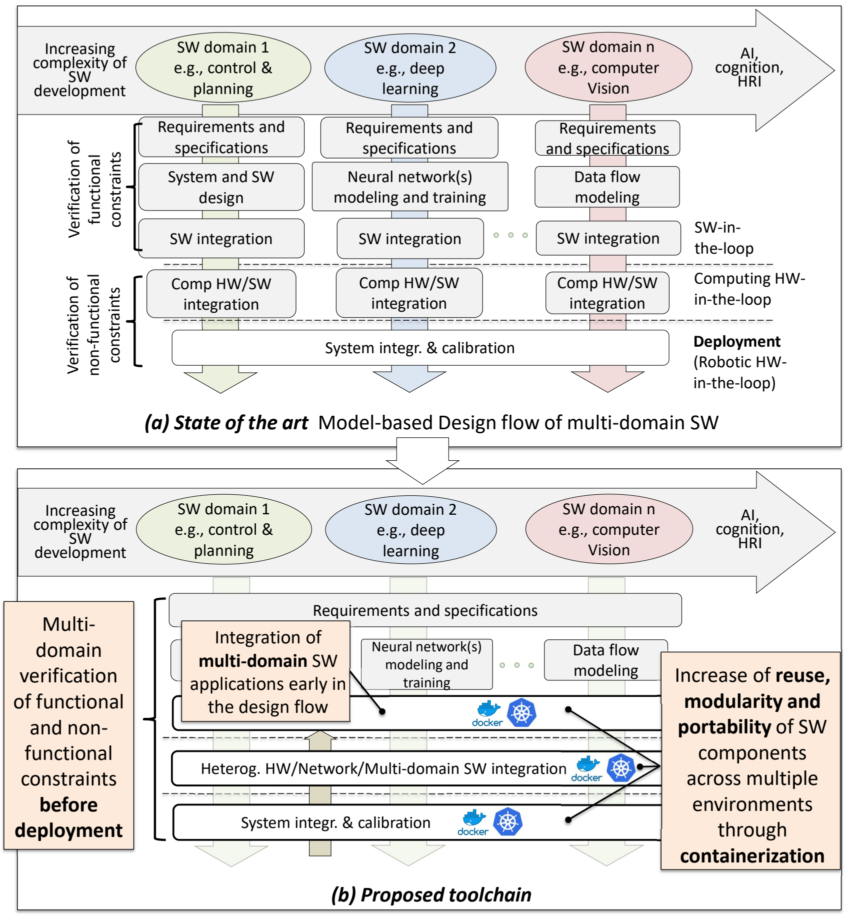

Increasing complexity of SW development
SW domain 1 e.g., control & planning
SW domain 2 e.g., deep learning
SW domain n e.g., computer Vision
AI, cognition, HRI
Verification of functional constraints
Requirements and specifications
System and SW design
Neural network(s) modeling and training
Data flow modeling
SW integration
SW-in-the-loop
Verification of non-functional constraints
Comp HW/SW integration
Computing HW-in-the-loop
System integr. & calibration
Deployment (Robotic HW-in-the-loop)
(a) State of the art Model-based Design flow of multi-domain SW
Multi-domain verification of functional and non-functional constraints before deployment
Integration of multi-domain SW applications early in the design flow
Heterog. HW/Network/Multi-domain SW integration
Increase of reuse, modularity and portability of SW components across multiple environments through containerization
docker
(b) Proposed toolchain

Fig. 1.1: Overview of the design flow.

(SW) block reuse among different SW development environments. As a consequence, these self-contained environments do not address the heterogeneity of the Hardware (HW)/SW domains by considering extra-functional constraints. Taking into account functional and extra-functional constraints in a seamless way is a key feature to design reliable robotic applications from the specifications to the system deployment.

In robotic and industrial environments, the Robot Operating System (ROS) [12] has become the de-facto standard for developing robotic applications. It has been proposed as a flexible framework for developing robot software through a collection of Application Programming Interface (API), libraries, and conventions that aim to simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms. Compliance with ROS is nowadays a key aspect for application re-use and easy integration of software blocks in complex systems.

Therefore, developing robotic applications for industrial scenarios requires addressing specific extra-functional characteristics, including the deployment of safe and isolated software applications, deployment automation, networking, scaling, reconfigurability, and availability of workloads and services.

In this context, *containerization* has emerged as a viable solution [13]. It offers advantages such as improved resource utilization, platform-independent development, and secure software deployment. However, as software for autonomous and intelligent robots becomes more complex, traditional containerization approaches may no longer suffice as they lack the means to scale to the more complex computing architectures. To address this complexity, it becomes necessary to partition services and tasks into distinct containers. This approach helps manage the increasing size of container images, adapt container mapping to different cluster nodes, and enhance system resilience against node failures [14].

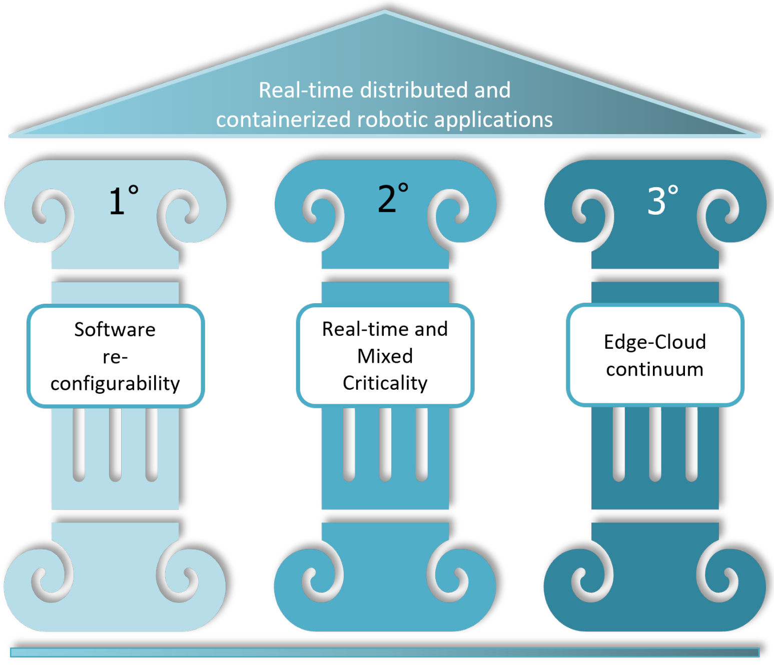

Real-time distributed and containerized robotic applications
1°
Software re-configurability
2°
Real-time and Mixed Criticality
3°
Edge-Cloud continuum

Fig. 1.2: The objectives of this PhD thesis.

Within the context of *multi-container* deployments, a significant challenge is ensuring continuous robot functionality even in the face of disruptions. As a result, many robotics companies are exploring platforms like Kubernetes, which is the de-facto standard for container *orchestration*, for automatic software deployment to address this issue [15, 16].

However, there is also a growing need for software standards that support *mixed-criticality* applications, which can be found in various domains such as industrial automation [17], automotive [18], and avionics [19]. A Mixed Criticality System (MCS) combines software components (e.g., ROS nodes) with different levels of criticality within a shared computing platform [20]. One of the primary research challenges in MCS is ensuring the correct execution of high-criticality tasks while sharing computing resources with lower- or non-critical tasks [21] in a user-transparent manner. Within this context, the introduction of software components and layers through containerization can complicate meeting Real-time (RT) requirements [22]. Although some research efforts have explored integrating real-time properties into container-based virtualization [23–26], supporting mixed-criticality requirements remains an open challenge.

Another emerging challenge for MCS is *integrating* real-time containers with orchestration. Orchestration has demonstrated its effectiveness in automating the deployment, networking, scaling, and availability of containerized workloads and services in cloud-native applications [27]. Nevertheless, current state-of-the-art orchestrators do not yet support *mixed-criticality* containers, limiting their adoption for robotic software.

This thesis addresses the issues related to the adoption of these paradigms to develop robotic software on edge-cloud architectures for industrial applications by presenting a design methodology to address the aforementioned challenges by automating the software synthesis, optimization, containerization, and orchestration on the Edge-Cloud computing continuum. The methodology sets objectives that can be categorized into three key areas, as illustrated in Fig. 1.2. Firstly, the focus is on the optimization of software tailored for edge computing, with the aim of enhancing software portability in real-time distributed environments and containerized applications within the realms of robotics and Industry 4.0/5.0 technologies. This optimization not only enables seamless functionality but also ensures a more efficient deployment process, facilitating the integration of heterogeneous software systems.

The research also delves into the orchestration of real-time containers within mixed-criticality systems. This involves the coordination of containers, each with its specific level of criticality, ensuring that they function without interference in real-time. This facilitates the development of robust systems where real-time responsiveness is paramount, especially in environments where safety, reliability, and precision are critical factors.

Finally, the expansion of the Edge-Cloud computing continuum through innovative run-time scheduling techniques is geared towards enhancing throughput and reducing response times. By implementing cutting-edge scheduling methods, the research aims to optimize the flow of data and processes in the continuum, thus significantly improving the overall efficiency of edge-cloud computing systems. These endeavors represent crucial advancements in the field, addressing the growing demand for faster data processing and more responsive computing in the contemporary technological landscapes of industrial automation.

These objectives have been realized through a series of incremental steps, each contributing significantly to the optimization and orchestration of software in heterogeneous and distributed edge-cloud architectures. The initial step involved improving the software performance on devices at the edge, in a *device-level heterogeneity* optimization. Subsequently, the focus shifted to the challenge of porting containerization technologies from traditional cloud environments to edge computing devices, requiring a careful analysis of the constraints of cluster-level heterogeneity, which is not found in traditional cloud environments.

A critical requirement emerged in the form of software reconfiguration, an essential prerequisite for deploying applications in a distributed edge-cloud computing platform. Addressing this requirement is another step to ensure adaptability and efficiency in the domain of edge-cloud computing.

The distinctive reality of robotic software brought forth unique challenges, notably stringent real-time deadlines. Addressing these constraints became imperative, leading to a thoughtful integration of these additional requirements into the orchestration process. This tailored approach ensured the seamless functioning of robotic applications within the orchestration framework.

The final crucial step entailed a paradigm shift: the adaptation of conventional static deployment techniques found in standard orchestration software to the dynamic environments prevalent in robotics and Industry 5.0 applications. This modification was vital to accommodate the rapidly changing scenarios, enabling the system to respond to the dynamic nature of modern robotic and industrial environments.

We can summarize the proposed design methodology with the following key steps and concepts:

1. Optimizing performance on heterogeneous devices at the edge for the first-level heterogeneity (Section 3.1):
   - *Improving the scheduling on Directed Acyclic Graph (DAG)-based embedded vision applications*: an implementation and analysis of the Heterogeneous Early Finish Time (HEFT) heuristic on heterogeneous embedded devices. The new rank-based static scheduling algorithm XEFT and static scheduling for pipelined execution of DAG instances (Section 3.1.1).
   - *Improving performance on Edge computing embedded boards with Unified Memory Architecture (UMA)*: a set of micro-benchmarks to characterize the Central Processing Unit (CPU)-Integrated Graphic Processing Unit (iGPU) communication and a zero-copy communication pattern to enhance performance by taking advantage of a synchronized and overlapped execution of CPU and iGPU tasks (Section 3.1.2).
   - *Improving performance for Cyber-Physical System (CPS) on Edge computing embedded boards with UMA*: different techniques to efficiently implement CPU–iGPU communication on UMA that comply with the ROS standard (Section 3.1.3).
2. Containerization and orchestration on heterogeneous Edge-Cloud computing architectures (Section 3.2):
   - *Extending Docker and Kubernetes for ROS-compliant containerized robotic applications*: a software methodology to develop robotic applications and classify ROS nodes and cluster them into containers to reduce memory overhead and improve

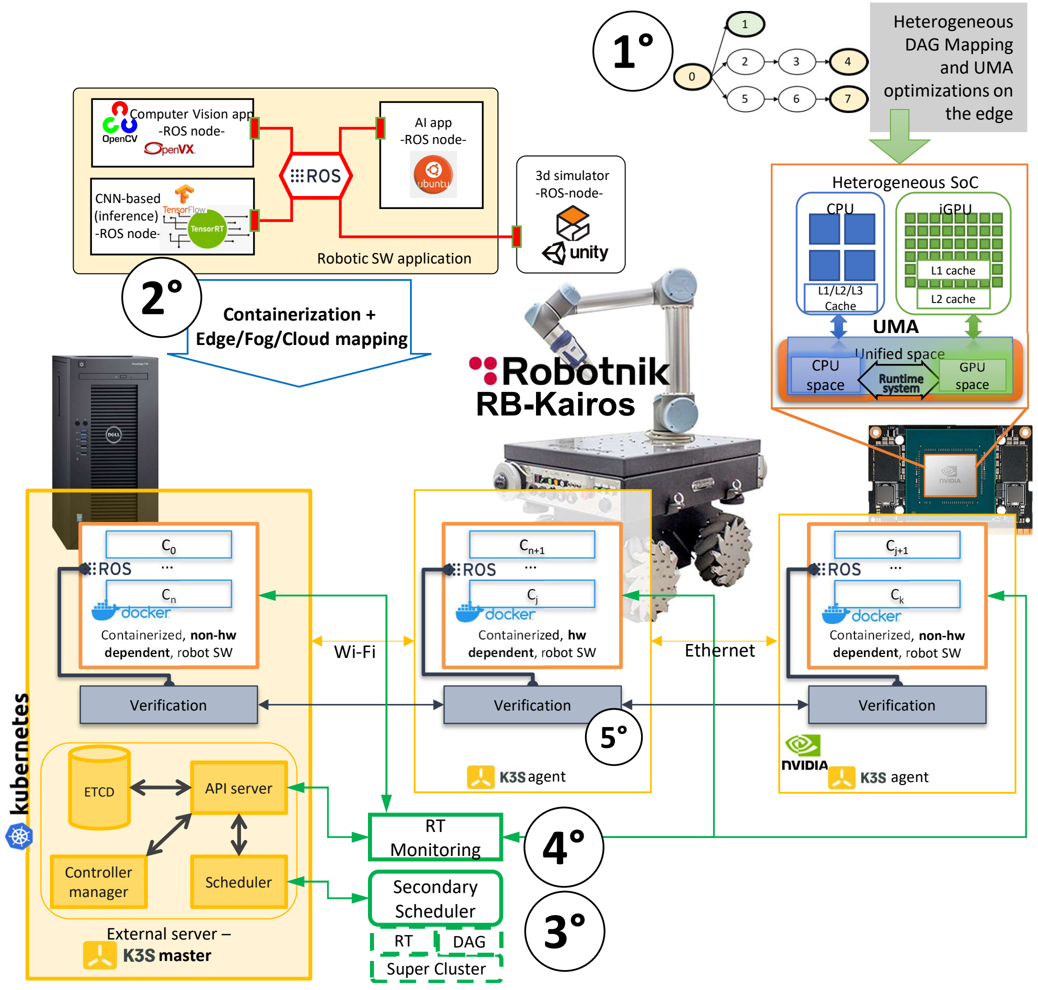


Fig. 1.3: Summary of this thesis methodology.

quality of service. Also, a technique to reduce storage overhead of containers by exploiting the control flow graph of the ROS nodes and package inheritance (Section 3.2.1).
- *Expanding the Edge-Cloud computing continuum to the RISC-V open hardware architectures*: porting and analysis of an orchestration platform to a RISC-V cluster prototype (Section 3.2.2).

3. Re-configurability of software for Edge-Cloud computing continuum (Section 3.3):
   - *Improving Kubernetes schedule's efficiency for ROS-based applications: Network:* a super clustering technique to efficiently map containers into Kubernetes Edge-Cloud computing nodes to reduce the utilization of the communication network (Section 3.3.1).
   - *Improving Kubernetes schedule's efficiency for ROS-based applications: Makespan:* adaptation and analysis of the most efficient Quality of Service (QoS)-oriented scheduling approach (HEFT) in Kubernetes. HEFT4K, a new scheduler for Kubernetes that, starting from the HEFT task ranking, takes advantage of the Operating System (OS) niceness to reduce priority inversions and preemptions of tasks. An event-driven remapping strategy that supports Kubernetes' scaling and recovery capabilities while minimizing the interruption of robot functionality (Section 3.3.2).
4. RT-Kube: Real-Time Kubernetes in the Edge-Cloud continuum (Section 3.4):
   - An orchestration platform for MCS that extends the Kubernetes scheduling with container sorting, node filtering, scoring, and container-node binding through criticality-aware algorithms and policies.

   - A monitoring mechanism that checks the status of each RT container across the Edge-Cloud and efficiently notifies temporal violations. An RT scheduler that implements the runtime migration of state-less containers across the cluster nodes to avoid system performance degradation.
5. Assertion-based verification and workload migration in Kubernetes for robotic systems (Section 3.5):
   - A workload-aware orchestration mechanism to migrate monitors synthesized from Signal Temporal Logic (STL) assertions by considering a strategy to handle overload scenarios where the available computational resources are insufficient to execute both the robot's functional tasks and the verification environment.

Fig. 1.3 summarizes the methodology of this thesis. The first step is represented on the right-hand side. The top of the figure shows the second step. The bottom shows the third and fourth steps. Finally, the dark boxes show the fifth step.

The methodology is analyzed on a multitude of synthetic benchmarks and scenarios. To guarantee the reliability and performance in high-complexity industrial scenarios, the methodology is also verified through real-case studies, such as the software implementing the mission of a Robotnik RB-Kairos mobile robot interacting with an industrial agile production chain.

1.1 Thesis outline

Chapter 2 explores the state-of-the-art and background required in the following chapters. Chapter 3 analyzes each methodological step presented in this Introduction to achieve the goals that have been set (see Fig. 1.2). Chapter 4 presents the results obtained on both a synthetic and a real case of study. Finally, Chapter 5 presents the conclusions, considerations on the results obtained, and how this methodology could be expanded in future work.

2

Background and Related Work

This chapter summarizes the background and related work for each of the five steps required to reach the objectives of Fig. 1.2. Section 2.1 details the background and related work for edge optimizations. Section 2.2 introduces the background related to containerization and orchestration, as well as the related work. Section 2.3 shows the background for containerization and orchestration in the context of software reconfigurability. Section 2.4 introduces the real-time background required to understand the notions used in the methodology and the state-of-the-art capabilities for orchestration software applied in MCSs. Finally, Section 2.5 shows the related work for the adaptations proposed to allow for runtime migration of tasks and containers.

2.1 Optimizing performance on heterogeneous devices at the edge

Edge computing and embedded boards offer new capabilities that allow to reduce the latency between data generation and data computation. They often do so with stringent requirements, such as small form factor, low power consumption, and low heat dissipation capabilities. Because of this, they use heterogeneous hardware accelerators such as CPU cores, GPUs, and Digital Signal Processor (DSP)s. This variety requires additional care in the configuration and implementation of software applications that take advantage of these heterogeneous embedded boards.

This section analyzes the background and related work regarding several important aspects required to optimize computations on edge-computing platforms:

- OpenVX and DAG-based embedded vision applications.
- Static scheduling for embedded vision applications.
- Efficient UMA programming.

OpenVX and the DAG-based embedded vision applications

OpenVX is a framework to develop embedded vision applications and optimize them by taking into account several extra-functional constraints such as performance and energy efficiency. It is constructed on top of a graph-based system to define a high-level and architecture independent representation of the application. Such system is modular and, by utilizing a set of already existing primitives, the user can build the application using the most commonly utilized functionalities and information objects in computer vision applications, such as scalars, arrays, matrices and images, as well as high-level data objects like histograms, image pyramids, and look-up tables.

Thanks to the libraries of architecture-oriented implementation of the primitives, which are provided by the board vendors, and to the data-structures available in the framework, the high-level representation (i.e., the *graph*) is then automatically optimized and synthesized into the target device.

The user can build a computer vision application by instantiating kernels as nodes and data objects as parameters (see the example in Fig. 2.1). Each graph node is identified as

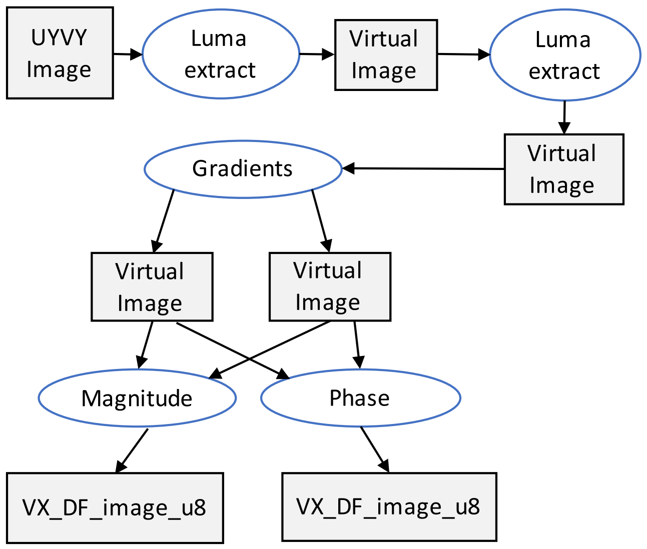


Fig. 2.1: OpenVX sample application (graph diagram).

a function kernel that can run on a Computing Elements (CE) of the target heterogeneous architecture. The application can then be executed across different hardware accelerators (e.g., CPU cores, GPUs, DSPs) thanks to the partitioning of the whole application into blocks given by the graph representation.

The programming flow starts with the creation of an OpenVX *context*, which manages the references to all objects used. The code builds the graph and generates all required data objects based on this context. It then creates the kernels as *graph nodes* and generates their connections. OpenVX checks the graph for integrity and correctness (e.g., checking for data type coherence between graph nodes and absence of cycles) and then, as the final step, it processes the graph. All created data objects, the graph, and the context are released at the end of the execution.

Fig. 2.1 shows an example of an application that computes the gradient magnitude and gradient phase from a blurred input image. The *Magnitude* and *Phase* nodes are computed independently, because they do not need the output of the other. OpenVX does not require a simultaneous or parallel execution, but leaves the decision of mapping and execution strategy to the runtime manager of the board vendor.

Vendor libraries that implement the graph nodes as Computer Vision primitives can be used by OpenVX to create several mapping strategies between nodes and processing elements of the heterogeneous board that can be used to target different design constraints (e.g., performance, power, energy efficiency).

The Neural Networks extension of the framework enables execution and integration of Deep Neural Networks in OpenVX processing graphs. Deep Neural Network topologies can be represented by OpenVX graph, where the layers are represented as OpenVX nodes (vx_node) and *vx_ tensor* objects as the data objects connecting the nodes (layers) of the OpenVX graph (Deep Neural Network). An OpenVX graph can be defined to represent a mix of Deep Neural Network layers and Vision nodes.

Static scheduling for embedded vision applications

There has been extensive prior research in task scheduling for multi/many cores at different levels of abstractions over the past decade. We refer the reader to [28] for an extensive overview of mapping and scheduling techniques. Considering the applications addressed in this section (i.e., computer vision at the edge on embedded heterogeneous devices), the focus is limited to the class of static scheduling for heterogeneous architectures, for which the most recent and related works is summarized in the following.

Algorithm	Intra cluster mapping	Inter cluster mapping	Heterogeneous architectures	Multiple node implementations	Optimization for partial multiple implementations	Global-best mapping
Tetris [29]	✓(*)	✓	✗	✗	✗	✓(*)
VisionWorks [30]	✗	✗	✓	✓	✗	✗
G-FL [31]	✓(**)	✓	✓	✗	✗	✗
Proposed HEFT impl.	✓	✓	✓	✓	✗	✓
XEFT	✓	✓	✓	✓	✓	✓

Table 2.1: Comparison of representative heuristics and static scheduling methods for heterogeneous platforms. (* application-level mapping)(** mapping implicitly performed by the OS).

In [29], Goens introduces TETRiS, a runtime system for static mapping of *multiple applications* on heterogeneous devices. It relies on compile-time information to extrapolate the best application mapping and to migrate such applications still preserving predictability of system performance. However, it is not tailored for DAG-based applications, and it does not support *multiple* (i.e., one implementation of a graph node for each CE) and *exclusive* implementations (i.e., single implementation of a node) of nodes for heterogeneous CEs (e.g., both CPUs and GPUs).

In [32] and in [31], the authors defined an approach to schedule DAG-based OpenVX applications for multi-core CPUs and GPU architectures. They introduce the concept of *pipelined scheduling* by overlapping sequential executions of the (DAG) application. They do not consider multiple implementations of nodes, and they do not implement any mapping heuristics. The mapping algorithm targets the best *local* solution, whereby if there exists a GPU kernel for a DAG node then that node is offloaded onto the GPU.

To support the node mapping over the different heterogeneous possibilities, we consider the HEFT algorithm [33]. Let us represent an application through a DAG, $G = (V, E)$, where V is the set of v tasks and E is the set of e edges between tasks. An edge $(t, q) \in E$ represents the constraint of precedence such that task t has to complete its execution before task q starts. HEFT maps and schedules tasks in two phases. The first is the *task prioritizing phase*, in which the tasks are ranked according to each task's priority as follows[1]:

$$rank(t) = \overline{w_t} + \max_{q \in succ(t)} (\overline{c_{t,q}} + rank(q)) \tag{2.1}$$

where $succ(t)$ is the set of immediate successors of task t, $\overline{c_{t,q}}$ is the average communication cost of edge (t, q), and $\overline{w_t}$ is the average computation cost of task t.

The second is the *processor selection phase*, in which each task t on the rank list is mapped onto the CE that minimizes the finish time of t. HEFT has shown to be the best static scheduling approach for a bounded number of heterogeneous processors [34,35], and its optimizations show its portability and offer encouraging potential for embedded multi/many-core architectures [36–38].

Table 2.1 summarizes and compares the features of the most representative scheduling algorithms and heuristics for computer vision applications on programmable embedded devices. The first column represents the intra-cluster mapping, which is the ability of the approach to choose on which core inside of a given CE (e.g., CPU) to map each single DAG task. Differently from HEFT and XEFT in which it is explicitly implemented, such a mapping is done by the operating system in multi-threaded applications. The second column

[1] We consider the *upword* ranking [33] since it has shown to provide the best results for our graph characteristics. However, the optimization based on the exclusive overlap is independent of any ranking methodology.

represents the capability of deciding on which CE, of the same type (e.g., either CPU or GPU) to map the graph node. This particularly applies in those architectures that have multiple CPU clusters. As an example, in the Nvidia Jetson TX2, once it has been decided that the node will run on the CPU, it is necessary to specify which of the two different CPU clusters (Dual-Core Denver or Quad-Core A57). The third column represents the ability of the algorithm to support the mapping and scheduling on heterogeneous architectures. The fourth column shows if the algorithms support multiple implementations of DAG nodes, that is, if they consider more than one execution times of each node during the scheduling. The fifth column shows which algorithms are optimized to handle nodes that have partial multiple implementations, meaning that the number of available implementations for each node may differ from one to many. Finally, the table shows if the scheduler targets system-level performance and provides near-optimal solutions rather than providing local-best solutions.

Efficient UMA programming

Programming UMA is a difficult task due to the complex nature of memory architectures. Since all processors are now using layered memory-access topologies through on-chip caches, sharing a unified memory space between multiple processors can become problematic for cache coherency.

Cache coherency for GPU accelerators has been investigated in many research works. In [39], the authors propose a push-based, coherence mechanism that explicitly exploits the CPU and GPU's producer-consumer relationship by automatically moving data from CPU to GPU's last-level cache. In [40], the authors propose a cache coherence protocol designed for forward-looking multi-GPU systems. HALCONE [41] is a timestamp-based coherence protocol for multi-GPU systems. It replaces the compute unit level logical time counters with cache level logical time counters to reduce coherence traffic. In [42], the authors propose selective caching, by which they disallow GPU caching of any memory that would require coherence updates to propagate between CPU and GPU. A survey of additional techniques for managing and leveraging caches in GPUs proposed more in the past is presented in [43].

In contrast to the above-mentioned works that propose cache coherency protocols for CPU-iGPU or multi-node GPUs, this thesis analyzes a framework to accurately estimate the potential speedup a CPU-iGPU application may have on a given device by considering different communication models.

A performance model for tuning GPU applications has been proposed in [44]. The model relies on a suite of micro-benchmarks to extrapolate the characteristics of specific GPU device components (e.g., arithmetic instruction units, memories, etc.) in terms of throughput, power, and energy consumption. GPUPerfML [45] combines decision trees and theoretical analytical models to locate performance bottlenecks in GPU applications and guide the optimization of the application. A comprehensive review of previous works addressing performance models for GPUs is presented in [46]. All the analysed contributions focus on GPU computation and memory access patterns over different platforms.

Unlike these previous works that target the tuning of GPU applications (i.e., kernels), the framework proposed in this step targets the tuning of CPU-iGPU communication on physically shared memory.

2.2 Containerization and orchestration on heterogeneous Edge-Cloud computing architectures

One of the major issues in programming and configuring robots in real contexts and different application areas, like agile production, agriculture, healthcare, and smart manufacturing, is that the system must satisfy in addition to functional constraints, also extra-functional constraints [47–50]. Heterogeneous (Internet-of-Things (IoT)/Edge/Cloud) architectures combine computation, storage, and network resources to solve such a problem [51]. Even though the heterogeneous nodes can provide resources to various devices, each device requires its own application to either process the received sensor data, or specific software to make use

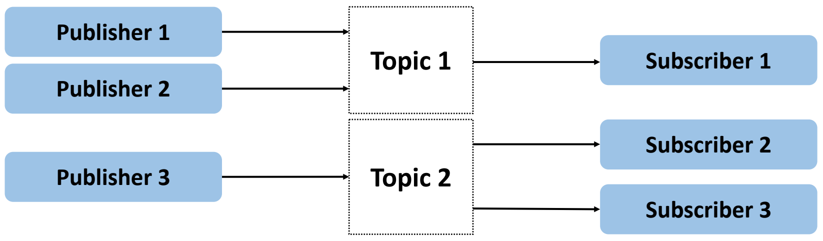


Fig. 2.2: ROS publisher-subscriber communication approach.

of the onboard sensors and forward the readings on the network, depending on where it is located on the architecture stack.

In this context, containerization is increasingly being adopted as a virtualization mechanism to solve such modularity and portability issues [52]. Software containers have emerged from the field of cloud computing and the need to manage large-scale server clusters [53]. Some generic architectures for CPSs, based on containers (e.g., Docker) and using ROS as a middleware, have been recently explored [54]. The modular architecture revealed the potential for better information flow among different network levels as well as increased modularity in the use of software components. Such a decoupling can ease the concurrent development of the subsystems as well as their runtime control at operation time. Moreover, Edge-Cloud computing is increasingly used to provide more advanced services, such as intelligent and adaptive control, fault detection, and state analysis [55].

Open-hardware architectures are also seeing an increased adoption rate with new consumer-level hardware becoming available. Nevertheless, the potential for Edge-Cloud computing continuum remains untapped due to a lack of software support, especially regarding containerization and orchestration. It is also unknown how much of a performance degradation such a new architecture might incur with the adoption of containerization and orchestration.

This section analyzes the background and related work regarding containerization and orchestration on Edge-Cloud computing platforms:

- ROS-based communication for modular CPSs.
- Container orchestration in Edge-Fog-Cloud architectures.
- Container orchestration in emerging open-hardware architectures and its overhead.

ROS-based communication for modular CPSs

ROS has been proposed as a flexible framework to develop software for robotic and cyber-physical systems. It is a collection of APIs, libraries, and conventions that aim at simplifying the task of creating robust and complex robot behavior on a wide variety of robotic platforms. It has become a de facto reference standard for developing robotic and cyber-physical system applications.

In ROS, the functionality of the system is implemented through nodes that communicate and interact. The nodes exchange data using two mechanisms: the *publish-subscribe* paradigm and the *service* model (i.e., Remote Procedure Call (RPC)) [56].

The publish-subscribe paradigm is based on *topics*, which are communication buses identified by a name [57]. A *publisher* node sends the data asynchronously and a *subscriber* node receives the data simultaneously.

The topic-based model relies on socket-based communication, which also allows collective communication. The communication channel is instantiated at the system start-up and never closed. In this type of communication, there can be many publishers and many subscribers on the same topic, as in the example of Figure 2.2.

With the service model, a node provides an on-demand service (see Fig. 2.3). Any client can query the service synchronously or asynchronously. In case of synchronous communication, the querying node waits for a response from the service. There can be multiple clients, but only one server is allowed.

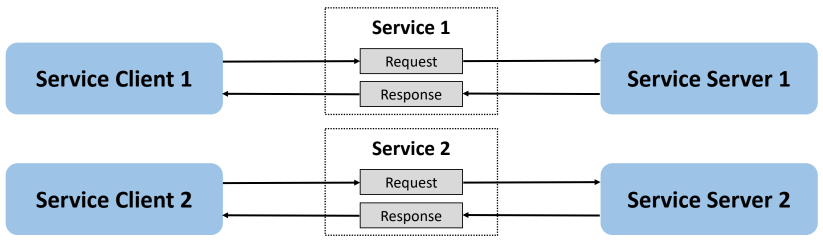


Fig. 2.3: ROS service communication approach.

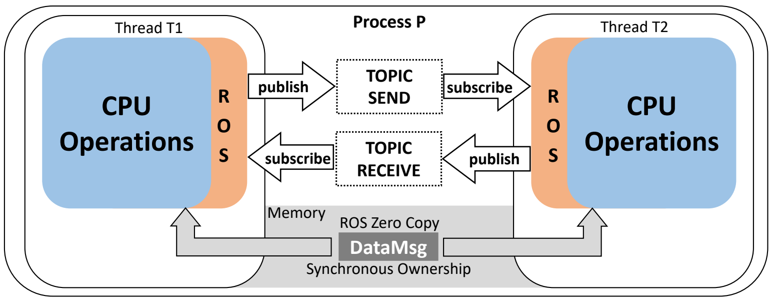


Fig. 2.4: ROS2 intra-process communication.

In general, the service model relies on point-to-point communication (i.e., socket-based) between client and server, and, for each client request, the server creates a dedicated new thread to serve the response. After receiving the response, the communication channel is closed.

The publish-subscribe communication paradigm is implemented through physical copy of data. In case of nodes running on the same device and sharing the same resources, the copy of large size messages may slow down the entire computation. *ROS2* introduced *ROS zero copy* (*ROS-ZC*), a new method to perform efficient intra-process communication [58].

Figure 2.4 shows the overview of this communication method. With ROS-ZC, two nodes exchange the pointer to the data through topics, while the data is shared (not copied) in the common physical space.

Even though such a zero-copy model guarantees high communication bandwidth between nodes instantiated on the same process, it has several limitations that prevent its applicability. First, it does not apply to service-based communication. Second, it does not apply to inter-process communication. Then, it does not support multiple concurrent subscribers. Finally, it does not allow for computation-communication overlapping.

Container orchestration with Kubernetes

Kubernetes is an open-source container orchestration platform that simplifies the management of containerized applications [59]. It works by coordinating and distributing workloads across a cluster of devices, i.e., the *computing nodes*, ensuring efficient resource utilization and high availability.

The standard Kubernetes architecture is composed of a single *master* and one *kubelet* unit per cluster node. Each kubelet manages one or more *pods*. Pods are logical units used to cluster related containers to share resources. Each pod can have one or more containers, each containing one application (or more). The master serves as the central control plane, overseeing the state of the cluster through a controller manager, a database of cluster information (ETCD), and a scheduler unit. The scheduler manages the container deployments across the cluster nodes. The functional units (i.e., master and kubelets) communicate through the HTTP REST protocol. The master manages the kubelets requests through an API server [60].

Containerization in Edge-Fog-Cloud architectures

Containerization in Edge-Fog-Cloud architectures has been investigated in several works (e.g., [61–63]). The experimental results show that container-based virtualization combined with the *Edge-Cloud computing continuum* provides better scalability, resource utilization, and performance. Containers have little to no performance overhead, although care is needed when multiple containers access shared resources [64].

Containerization combined to orchestration has been investigated for Fog and Edge computing [65–67]. Different microservice deployment strategies can be adopted to improve, in addition to performance, energy efficiency and carbon footprint [65] as well as quality of service [66, 67].

Distributed control of independently operating devices, providing local data storage, can be achieved. However, those resources are limited, making their effective management and utilization imperative to enable advanced flexibility in robotic systems. A few approaches seek to improve upon more monolithic control and production paradigms via virtualization and modern software approaches (e.g., [68]), which also considers the connection to the legacy systems. The results demonstrate merit for combining edge and server computing solutions, with Cloud applications and scheduling algorithms. In [69], an approach to workflow management and task modeling is discussed, in which low-level robotic operations are executed by the ROS framework to reduce code time and increase the reconfigurability of components. Containerizing ROS environments allows more reproducible and stable deployments across a distributed architecture of complex software [70, 71]. Efforts have been made to migrate ROS-based applications to the cloud to be used as containerized services [72]. In [73], the authors show an approach to generate container images from a configuration file for packages ROS, allowing easier and automatic deployment on the target HW.

RISC-V CPU

RISC-V is an open source Instruction Set Architecture (ISA) developed by researchers at the University of California, Berkeley, in 2010 [74]. The acronym RISC stands for Reduced Instruction Set Computer, which means the architecture has a smaller set of simple and standardized instructions. The simplicity and modularity of the RISC-V ISA make it ideal for various computing devices, such as smartphones, tablets, embedded systems, but also High Performance Computing (HPC) systems. Additionally, the open source nature of RISC-V allows anyone to design and manufacture chips based on the architecture, encouraging innovation and competition in the market.

The RISC-V architecture has evolved over the years, and various versions have been released, including RV32I, RV32E, and their 64 bits counterparts. The different letters in the naming scheme indicate variations in the ISA features. For example, "I" stands for the base integer instructions, "E" stands for the embedded profile, "F" includes instructions for single-precision floating-point arithmetic, and "D" for double-precision floating-point arithmetic. The RISC-V architecture also supports extensions that can be added to the base ISA to enhance functionality, such as the vector "V", bit manipulation "B", and compressed-instructions "C" extensions.

RISC-V technologies have gained popularity in recent years, with many companies adopting the architecture in their products. For instance, SiFive, a leading RISC-V chip designer, provides a range of processors for various applications, including embedded systems, IoT devices, and HPC systems. Other companies, such as Western Digital, NVIDIA, and Qualcomm, use RISC-V in their products.

The future of RISC-V technologies looks promising, with ongoing developments and collaborations among industry players and academia. The European Commission, for instance, in the context of the chip sovereignty strategy, is pushing open-source architectures and RISC-V, in particular, as a future seed for HPC systems [75]. This leads to a possible future scenario of edge and cloud systems composed of heterogeneous architectures with specific features (enabled by the RISC-V customizability), where computing continuum technologies play a crucial role in achieving global optimization thanks to orchestration.

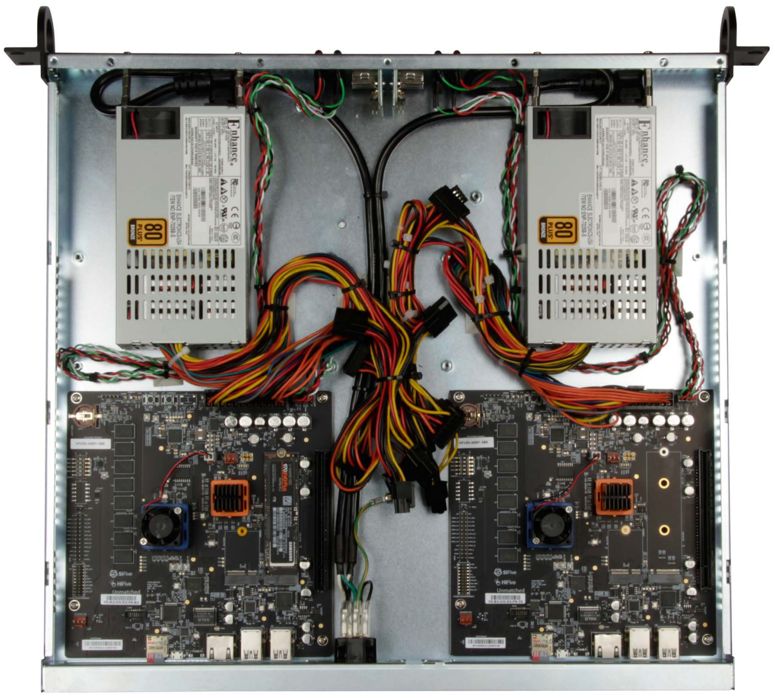


Fig. 2.5: The Monte Cimone Server Blade hosts two SiFive Freedom U740 SoCs and has a form factor of 4.44 cm (1 RackUnit) in height, 42.50 cm in width, and 40 cm in depth. Each RISC-V board has dimensions of 170 mm by 170 mm.

The Monte Cimone cluster

Monte Cimone represents a prototype and experimental platform of a comprehensive RISC-V (RV64) computing cluster, enclosing all the essential hardware components apart from processors, such as primary memory, non-volatile storage, and interconnect. It is composed of eight computing nodes; each one is based on the U740 System-on-Chip (SoC) from SiFive and integrates four U74 RV64GCB application cores (the U74 processor is a dual issue in-order execution pipeline, with a peak sustainable execution rate of two instructions per clock cycle), running up to 1.2 GHz and 16 GiB of DDR4, 1 TiB node-local NVME storage, and PCIe expansion cards. The U740 is Linux-capable SoC with a consumption of 6 Watt, placing the system in a low-power category, making it suitable for an edge-computing role.

As the target is to move towards high performance edge/fog computing, with a final goal of scaling toward HPC systems based on RISC-V, an entire HPC software ecosystem and a complete system monitoring infrastructure have been ported to Monte Cimone. In particular, it runs a software stack composed of a job scheduler (SLURM), an LDAP server, the Spack package manager with compiler toolchains and scientific libraries, and a monitoring framework based on ExaMon. It is established that actual HPC applications can be executed on Monte Cimone.

Impact of containerization and orchestration in Edge-Cloud architectures

Many works in literature have analyzed the impact of containerization, with several benchmarks emerging as standard, such as CPU compression/decompression of files, system memory and storage latency, as well as network bandwidth and latency. Other works have also analyzed specific applications, such as MySQL for cloud environments and REST applications for IoT.

The authors at IBM have analyzed the impacts of containerization with many different benchmarks, such as `pxz` for decompression, `linpack` for floating point performance, `Stream` for memory and `netperf` for the network. They found little performance overhead and some network latency degradation due to Network Address Translation (NAT) [76]. In an HPC-focused work [77], the authors have used different benchmarks, such as `sysbench`, `Stream` and `HPCG`, to measure the performance impact of containerization, finding no significant CPU overhead, but discovering a higher memory usage for containerized applications. In [78], the authors used the `Phoronix test suite` to benchmark the containerization overhead and analyze the impact of orchestration. They found minimal overhead in containerization and a worst-case scenario of 8% reduced performance when using an orchestration platform such as Kubernetes.

IoT architectures have also been analyzed with several different benchmarks. In [79], the authors used `7zip` for compression/decompression, `OpenSSL` for cryptography, `RAMspeed` for system memory latency, `Tio` for disk performance and `sockperf` for the network. In [80], the authors tested edge architectures based on AWS Greengrass and Microsoft Azure IoT Edge with custom-made benchmarks for speech and image recognition and found some limitations in system throughput when using containerization.

Other works, such as [81] and [82], have also compared the performance of containerization on different architectures like x86 and ARM. In the first work, many benchmarks were used, such as `lmbench`, `netperf`, `sysbench` and `linpack`, showing negligible overhead in all types of tests. In the second work, the authors analyzed the impact of containerization on REST services, finding that while the overhead of containerization is negligible, there are latency spikes when measuring end-to-end latency for the service.

When analyzing containers, the authors of [83] focused on the delay between a container being scheduled for execution and the container starting and found that delay very low and predictable, correlating the delay to the number of containers active on the device.

In [84], the authors built a complex architecture for data analysis, with edge nodes collecting data to compute and a suite of applications to analyze and store the data. These applications are distributed either on the edge nodes themselves or on fog or cloud nodes. They found that edge computing is a valid and robust alternative to fog or cloud computing while the amount of data to compute is lower than a threshold, thanks to much lower average latency.

2.3 Re-configurability of software for Edge-Cloud computing continuum

Improving adaptability and performance of containers on an Edge-Cloud computing architecture is of paramount importance to improve the reliability and safety of modern robotics systems found in industrial automation production plants. As these robotic applications are becoming more and more reliant on containerization and orchestration, there is a need to deploy them in a more efficient manner, while maintaining transparency for the containerized application. This translates in a need to optimize the orchestrating software to allow it to take more specialized decisions.

Some works have been proposed to better use the software and functionalities already included in Kubernetes. In [85], the authors use a modular architecture that employs a continuous probing of the deployed services (i.e., cloud native) to test their response time. They collect resource usage statistics and when a container is experiencing high load, they scale it up by one by using the Kubernetes Auto Scaler. In [86], the authors manually add data to deployments through Kubernetes labels, such as a compute domain tag (i.e., cloud, fog, edge) and geographical location. They then use affinities to match labels and nodes. They monitor the network topology and node failures, and if there are events, they modify the deployments to trigger the Kubernetes scheduler.

Cloud deployments require the users to define the containers' resource usage manually. The authors in [87] improve upon the Kubernetes vertical Autoscaler by analyzing data coming from the metrics server and updating deployments with better defined resources.

They obtain improved performance when the resource information on the initial deployment is not accurate.

Various works have been done for the 5G network computing infrastructure. Some work has been done to address the problem that, when a node dies, the containers in the node are not immediately restarted but take several minutes. This creates downtime for services that need to be restarted [88]. The authors also improve node filtering and scoring thanks to more flexible resource selection through the use of weights. In subsequent work [89], there has also been a proposal for an extended and more lightweight monitoring tool for edge devices. The scheduler can make decisions based on the collected data and the system administrator can adapt the scheduling with weights on the different data points, including CPU temperature to avoid throttling/crashing of edge nodes.

In [90], the authors propose a framework to trade between performance and energy consumption by using a stateless migration policy based on premade energy and performance models. They use a set of benchmarks to create the performance models. The benchmarks need to be run on all nodes of the target platform beforehand. They then use a function to balance performance and energy consumption based on a target ratio defined by the user.

There has also been a proposal for a framework that can analyze data from previous executions and decide which node is the best candidate by focusing on accelerators, such as GPUs and code with multiple implementations. It evaluates the possibility of waiting for a faster node or starting the container on a slower node. To achieve this, the platform makes use of external components, such as a secondary scheduler and a database to archive the performance data. They use the Earliest Finish Time to decide on which node to run the container [91]. In another work [92], Particle Swarm Optimization has also been proposed to improve the Kubernetes scheduler and better allocate containers to nodes. In [93], the authors proposed a framework that schedules and monitors repetitive and short-lived tasks, such as deep learning training or batch operations. They use progress as a metric to improve scheduling and reduce makespan. The application progress is obtained by modifying the application's source code and sending its progress to the monitoring framework.

HEFT is an offline list scheduler that produces high quality scheduling with low computational cost [33]. Effort has been made to improve this algorithm with the look ahead variation, which obtains up to 40% improved makespan compared to its original counterpart [94] and specifically in cloud-based virtual machine management, where HEFT has been modified to create a more accurate ranking and select the best possible CPU [95]. In the context of a microservice-oriented containerized edge cloud environment, the authors of [96] create an alternative algorithm to HEFT that computes the optimal data flow between edge nodes to obtain the lowest possible makespan when scheduling tasks, showing improvements over the original HEFT algorithm but having several limitations. Firstly, the requirement of direct control of the device's scheduling, and secondly, high complexity and overhead. Finally, there are no dynamic rescheduling capabilities.

Efforts from industry leading companies have also been made with the Telemetry Aware Scheduler (TAS) from Intel [97] and Trimaran from IBM [98]. TAS can use many node characteristics to decide where to deploy a container and then monitor its health. Trimaran monitors standard Kubernetes metrics providers (i.e., Kubernetes Metrics Server, Prometheus Server, and SignalFx) and extends the scoring phase of the Kubernetes scheduler, without interfering with the other native plugins, accounting for both average load and load spikes.

Table 2.2 summarizes the state of the art in this context. The first column specifies whether the proposed work makes use of standard Kubernetes features or requires source code modifications. The second column shows if the framework can take action after the deployment is made, e.g., if a node becomes unavailable and some containers need to be rescheduled. The third column is for edge-cloud specific optimizations, such as assuming that the computing nodes in the cluster might have very different computational capabilities. The fourth column shows if the proposed work takes into account the total makespan of the application deployed and if it optimizes for it. The task scheduling column shows if the tasks deployed inside the containers are being controlled and scheduled on the target device by the best effort scheduler of the operating system or by an optimized software. The last column

Work	No source code modifications	Dynamic Riadaptation	Edge-Cloud	Makespan Oriented	Task scheduling	Resource optimization
[85]	✗	✓	✗	✗	✗	✗
[86]	✓	✓	✓	✗	✗	✗
[87]	✗	✓	✗	✗	✗	✓
[88, 89]	✗	✓	✗	✗	✗	✗
[90]	✓	✓	✗	✗	✗	✗
[91]	✓	✗	✗	✓	✗	✓
[92]	✗	✗	✗	✗	✗	✓
[93]	✓	✗	✓	✓	✗	✓
[96]	✓	✗	✓	✓	✓*	✗
[97]	✗	✓	✓	✗	✗	✓
[98]	✗	✗	✓	✗	✗	✓
This thesis	✓	✓	✓	✓	✓**	✓

Table 2.2: Summary of the scheduling features proposed in the state of the art. *Optimizes the device's task scheduling, but requires direct control of each device's scheduling. **Optimizes the device's task scheduling, but does not control it directly.

shows whether the proposed frameworks can optimize the resource utilization beyond the simple resource limits/requests of Kubernetes.

While efforts have been made by both the research community and industry to tackle the problem of scheduling for DAG-based applications, they did so for cloud systems based on virtual machines or by using expensive exact algorithms that do not offer online scheduling or orchestration capabilities due to their high execution time. Some, like [33] and [96], require direct control over the devices' scheduling, which may cause overhead or invasive kernel modifications.

2.4 RT-Kube: real-time Kubernetes in the Edge-Cloud continuum

Real-time software applications are often crucial for mission-critical tasks, and their timing requirements must be precise and reliable. These applications can have varying degrees of requirements on their timing, typically falling into two categories: *soft* real-time and *hard* real-time.

In the case of a *soft* real-time application, consider a scenario like Simultaneous Localization And Mapping (SLAM) software. This software periodically processes sensor data to create a map. While it is essential that this computation is timely, some degree of delay may be tolerable. In this context, a delayed result could lead to a temporarily less accurate map, which might result in navigation errors. However, such errors are acceptable as long as they are within certain limits.

In contrast, a *hard* real-time application has no tolerance for delayed results. For example, in an antilock braking system in cars, any delay in applying the braking correction can be detrimental. A late correction could adversely affect the handling of the car, rather than improving it. In such cases, a task is considered late if it exceeds its *deadline*, which represents the absolute time by which its execution must be completed.

Both hard and soft real-time tasks typically consist of repetitive computation phases that are triggered periodically or sporadically. In *periodic* tasks, these computation phases are activated at fixed and regular intervals. Conversely, *sporadic* tasks are activated with a minimum time interval between each activation, but the actual intervals may vary, allowing for more flexibility in their timing.

During execution, tasks can vary in the amount of time they take to complete, but this variation must have an upper limit. This upper limit is known as the *Worst Case Execution Time (WCET)*, which represents the maximum amount of time a task can take to finish under any possible system condition. These conditions are influenced by the state of the system, including factors such as resource availability (e.g., CPU, memory, network) and resource characteristics (e.g., CPU/memory clock frequency, network bandwidth).

A real-time task is defined as follows:

$$\mathrm{RT\_task} = (P,\ D,\ WCET) \tag{2.2}$$

where P represents the task's period, indicating how often new instances of the task arrive. D is the deadline, specifying the time by which the task must be completed. $WCET$ is the worst-case execution time. These times need to abide the following:

$$WCET \leq D \leq P \tag{2.3}$$

Determining deadlines and periods is a crucial aspect of system design and involves specifying system requirements and task characteristics during the design phase. Estimating the WCET is accomplished through various techniques. These methods include static analysis, which carefully examines the code to identify the longest possible execution path, and dynamic analysis, where real-time tasks are executed on the target platform to measure their observed worst-case execution times. Advanced tools are already available for conducting this analysis and are considered state-of-the-art. An emerging technique uses probabilistic analysis instead of deterministic approaches to estimate task execution times, which should allow for enhanced resource utilization [99]. However, it is not considered since the Linux kernel that this thesis relies on does not implement it.

Linux scheduling

The Linux kernel has several scheduling policies, some for real-time tasks and others for standard processes. Real-time sporadic and periodic tasks can be scheduled using the `SCHED_DEADLINE` policy. This policy is an implementation of the Earliest Deadline First (EDF) scheduling algorithm, augmented with a Constant Bandwidth Server to allow for better timing control (the reader can refer to [100] to dive deeper into the topic).

The `SCHED_DEADLINE` policy requires an admission test to check if there is room to guarantee the task WCET within the deadline, every period. This means that the task (t_{new}) has to pass the multiprocessor global EDF *task admission test* to be placed in the scheduling pool of RT tasks:

$$\sum_{i}^{RT \cup \{t_{new}\}} \frac{\mathrm{WCET}_i}{P_i} \leq M * \frac{\mathrm{sched\_rt\_runtime\_us}}{\mathrm{sched\_rt\_period\_us}} \tag{2.4}$$

where RT is the set of running RT tasks, WCET_i and P_i are the worst-case execution time and period of the tasks, M is the number of CPU cores, and sched_rt_runtime_us/ sched_rt_period_us represents the maximum allowed utilization of the CPU for RT tasks (user-defined Linux kernel variable equal to 95% by default)[2].

Related work

Container-based Edge-Fog-Cloud systems have been investigated in several works (e.g., [61–63,101]). The experimental findings demonstrate that the *Edge-Cloud computing continuum* and container-based virtualization, when combined together, improve scalability, resource usage, and performance. Containers have minimal to no performance overhead, but caution

[2] In Equation (2.4), runtime is used instead of WCET due to Linux documentation terminology.

is required when many containers access the same shared resources [64]. Recently, various architectures for CPSs based on containers (e.g., Docker) and using ROS as middleware have been investigated [54]. These works have shown the potential to improve information flow among various network levels and increase software modularity.

Containerization combined with orchestration has been investigated in fog and edge computing [65–67, 102]. Different microservice deployment strategies can be adopted to improve performance, energy efficiency and carbon footprint [65, 103], as well as QoS [66, 67].

Several research works have been done to improve the performance of robotic applications. In this direction, SeART [104] is a framework that can intelligently schedule real-time tasks by taking into account the current context to activate the minimum-cost tasks. Other real-time robotic applications also show the need for improving performance to guarantee real-time constraints [105]. The issues tied to containerization in a Fog-based system for robotic applications have also been explored. In [15], the authors highlight the lack of Fog-based frameworks to satisfy the real-time demands of robotic applications.

Containers with real-time constraints (i.e., RT containers) have been the focus of several recent studies due to the increasing adoption of container-based virtualization. The review in [106] explores existing solutions that guarantee real-time constraints when working with containerized applications. The authors underline the lack of tools for real-time container management and analysis on communication between real-time containers.

Legacy applications with real-time constraints have been successfully emulated using containers [22] and the performance overhead is low enough to run containerized real-time applications for industrial automation applications [17]. Real-time technologies (real-time OS ResinOs, Ubuntu Core, Co-kernel Xenomai 3, and Ubuntu with Preempt_rt software patch) have been tested together with containerization to demonstrate that container isolation is a new, competitive paradigm that allows for better resource usage when combined with real-time [107]. The Linux scheduler `SCHED_DEADLINE` was compared to the *cgroups* policy (i.e., the main technology used to enable containers) to analyze which technology better handles real-time application resources [25]. The scheduler is consistently more reliable and achieves better results than *cgroups*. This is also true in resource-constrained situations caused by high system load. Extensions of the Linux kernel have been proposed to improve the efficiency of real-time scheduling of *cgroups* [24]. The same authors also proposed a modification of the Kubernetes source code to include some real-time constraints [108].

HPC makes wide use of containers. A framework has been proposed to efficiently schedule real-time containers in many-CPU systems [109] but additional research is needed to determine the effects of the operating system and I/O on deadlines. The feasibility of migrating real-time applications from bare-metal servers to virtualized Infastructure as a Service (IaaS) configurations for Industry 4.0 has also been explored [26].

A framework for applications in mixed criticality systems has been proposed in [23] to reduce interference between tasks when an application exceeds its WCET. The framework shows low efficiency due to the adopted static priority scheduler and it supports only one computing node. In subsequent work [110], it has been extended to use dynamic priority scheduling with a bandwidth server to improve performance.

To fulfill the need of safety-critical real-time systems and streamline the certification process, the authors in [111] examined the benefits of utilizing both SGX isolation and unikernel features. Another proposed framework for real-time orchestration introduces extensive modifications of the Kubernetes code-base and uses a unique patch for the Linux kernel to deploy best-effort and real-time tasks [112].

Several works [113–115] show how, in Edge-Cloud computing architectures, live container migration can improve performance when resource usage can be monitored and utilization spikes mitigated by migrating services to a different computing node.

2.5 Assertion-based verification and workload migration in Kubernetes for robotic systems

Several solutions have been proposed to automatically synthesize monitors from their high-level specifications and integrate them into ROS-based designs for both single robots [116, 117] and robot swarms [118]. In these solutions, temporal logic and regular expressions are the main languages adopted to describe system properties and temporal patterns. Such formal languages provide powerful environments to define temporal order and concurrency among states and events. Variants and extensions have also been proposed to address the complexity of such monitoring tasks [117, 119]. A common strategy is to design monitors that reproduce the original system specification according to a set of rules and current inputs. After such a transformation, each monitor alerts if a certain form has been obtained [120, 121]. As an alternative, the monitors are designed as automata implemented through a large look-up table that maps all possible transformations for all possible inputs [122]. The static definition of the table provides better performance at runtime than rewriting-based monitors. Nevertheless, these approaches do not scale well in size as the automata could potentially become very large, non-compositional, and non-extensible. A compositional and extensible approach relies on the design of monitors as a network of small computation nodes generated from temporal logic specifications [123]. This approach has been extended to timed specifications [124], quantitative [125], and parametric [126]. Similar solutions have been applied to fault detection and condition verification of production facilities [127–129]. In recent years, STL has become the accepted solution for monitoring robotic and distributed applications [130]; recent works propose efficient and reliable ways of synthesizing monitors from STL specifications [131–133]. Concerning the use of ROS, [134] and [135] propose a runtime verification framework for robotic applications. Finally, a complete survey of runtime verification of distributed systems is proposed in [136].

The state-of-the-art approaches assume no limit from the point of view of the availability of computational resources for the execution of monitors. As a consequence, they are not effective in scenarios where strict limits must be respected, especially concerning the overhead caused by the runtime verification of the System Under Verification (SUV).

3 Methodology

This chapter summarizes the five steps required to reach the objectives of Fig. 1.2. Section 3.1 analyzes the edge-related optimizations. Section 3.2 shows the porting of containerization and orchestration to robotic environments. Section 3.3 adapts containerization and orchestration to allow for the reconfigurability of software. Section 3.4 introduces the real-time capabilities for an orchestration software to be applied in a MCS context. Finally, Section 3.5 shows the adaptations proposed to allow for runtime migration of tasks and containers.

3.1 Optimizing performance on heterogeneous devices at the edge

The need to deploy complex applications such as low-level control, dynamic planning, computer vision, monitoring, and inference applications based on neural networks at the edge on programmable low-power embedded devices is increasingly spreading in many different fields ranging from robotics, autonomous driving, and security [137].

We define device-level heterogeneity as computing systems with multiple processing units available for computation. In this context, heterogeneous parallel computing implemented in modern embedded boards in the form of multi-core CPUs, accelerators, and GPUs is recognized as the best solution to simultaneously target different extra-functional constraints like processing performance, power consumption, and energy efficiency [138]. Nevertheless, as the complexity of such Artificial Intelligence (AI) applications increases, performance limitations due to slow memory access, workload balancing, synchronization, and communication between multi-cores, many-cores, and heterogeneous cores are expected to worsen [139]. This makes it difficult for developers to fully exploit the theoretically available computing power.

We apply the performance optimizations required to improve embedded boards' response time and throughput when used as the target of an edge-computing deployment. The proposed optimizations cover three very critical aspects:

- Section 3.1.1: improving the performance of DAG-based applications for embedded vision applications.
- Section 3.1.2: improving the performance of Edge-computing embedded boards that make use of UMA.
- Section 3.1.3: analyzing the effects of UMA-specialized techniques on the performance of ROS-based CPS applications.

3.1.1 Improving the scheduling on DAG-based embedded vision applications

OpenVX [140] has been proposed as a solution to assist in the development and deployment of computer vision applications for embedded and real-time use cases, and it has rapidly spread in the edge computing community as a de-facto standard for the design and optimization of embedded vision applications. The OpenVX design flow begins with a DAG representation of the application where the nodes use implementations (called primitives) from libraries provided by the target architecture vendors (e.g., NVIDIA VisionWorks [30], AMD OVX [141], Intel CV SDK [142]) or developed by users. OpenVX also supports the mapping and scheduling of the DAG and its automatic synthesis onto the target heterogeneous platform. OpenVX offers the benefits of functional and performance portability across different hardware platforms by providing a framework that supports different hardware architectures without requiring significant modifications to the OpenVX model.

Several research works have been done to optimize the performance of the code generated through OpenVX frameworks [143–149]. They implement advanced data access patterns such as DAG *node merge*, *data tiling*, and through heterogeneous parallel programming. Other research contributions proposed OpenVX task scheduling algorithms to provide real-time guarantees through more fine grained handling of the DAGs and by using pipelining to provide better task parallelism [31, 32].

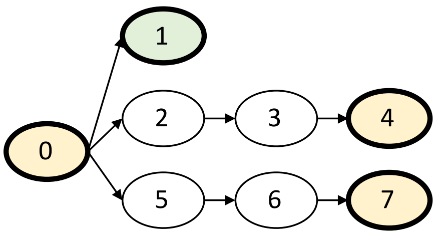
No work in literature addresses efficient *mapping and scheduling* strategies of OpenVX (DAG-based) applications for heterogeneous architectures. All state-of-the-art mapping approaches for OpenVX applications implement *local best* algorithms, by which if a DAG node has multiple implementations (e.g., both for GPU and CPU), the node is mapped on the computing element that provides the best node-level performance. In the NVIDIA VisionWorks framework, if a node has a GPU implementation then the node is mapped onto the GPU, as it is assumed that the GPU accelerator always provides better performance than the CPU [30].

What is missing is a mapping strategy that targets the system throughput (i.e., *global best*) rather than node throughput (i.e., *local best*). The fact that there can be multiple implementations for nodes (e.g., one that is executable on a GPU and another on CPU), which is pervasive in the OpenVX libraries [30,141,142,150], can allow for additional mapping flexibility, and as a consequence, for better load balancing at system level (e.g., a CPU implementation that is slower at kernel level can lead to a faster application at system level). However, such a combined mapping and scheduling problem is similar to the *Quadratic Assignment Problem* [151], a well-known Nondeterministic Polynomial Time (NP)-hard problem. Finding an optimal solution satisfying all the given DAG constraints is difficult. Thus, heuristics based on the application domain knowledge need to be employed to find a near-optimal solution.

To take into consideration the heterogeneity of the target architectures, the possible multiple implementations of DAG nodes, and the problem complexity, in this thesis we first present an implementation of the HEFT heuristic [33] for *static* mapping and scheduling of OpenVX applications. We present results obtained on a large set of benchmarks, in which we found that the HEFT implementation sensibly outperforms (i.e., up to 70% of performance gain) the state-of-the-art solution currently adopted in one of the most widespread smart system for moving computer vision at the edge (i.e., the NVIDIA VisionWorks framework on the NVIDIA Jetson TX2 device). Then, we show that such a heuristic, when applied to DAG graphs for which *not every* node has multiple implementations, can lead to idle periods for the CE. Since multiple implementations do not exist for all nodes in a majority of real embedded vision contexts, this thesis proposes an algorithm, which we call XEFT, that reorganizes the HEFT ranking to improve load balancing. XEFT aims at generating sequences of nodes with the single implementation (which we call *clusters* of *exclusive nodes*) in the ranking with the objective of reducing idle times caused by the combination of DAG constraints and exclusive implementations.

Finally, since scheduling approaches for pipelined DAG executions have been recently proposed to deliver real time guarantee in CPU-GPU devices [31], we present an extensive evaluation and comparison in terms of performance and memory footprint between those solutions [31], XEFT, and a pipelined version of XEFT. We present the results on a large set of benchmarks, including a real-world localization and mapping application (ORB-SLAM) combined with an NVIDIA Convolutional Neural Network (CNN) application for object detection.

This section first presents an implementation of the HEFT heuristic and its application for static scheduling of OpenVX applications. We show that it sensibly improves the system performance w.r.t. the native OpenVX scheduler. Then, an analysis of the main limitations of HEFT in the context of embedded vision applications, which can cause idle periods and



node #	t_{CPU} (ms)	t_{GPU} (ms)	HEFT rank (max)
0	1	-	15
1	-	10	10
2	4	2	14
3	4	1	10
4	6	-	6
5	4	2	14
6	4	1	10
7	6	-	6

Fig. 3.1: Example of DAG-based application, execution time of tasks mapped on CPU/GPU, and the corresponding HEFT ranking.

workload imbalance among heterogeneous computing elements. This section then focuses on XEFT, a rank-based static scheduling algorithm tailored for DAG-based embedded vision applications. We introduce the concept of *exclusive overlap* between nodes and show how XEFT implements such a concept to improve the load balancing. Finally, the static scheduling of pipelined execution of DAG instances is considered, including advantages and limitations of such an approach by considering different scheduling algorithms.

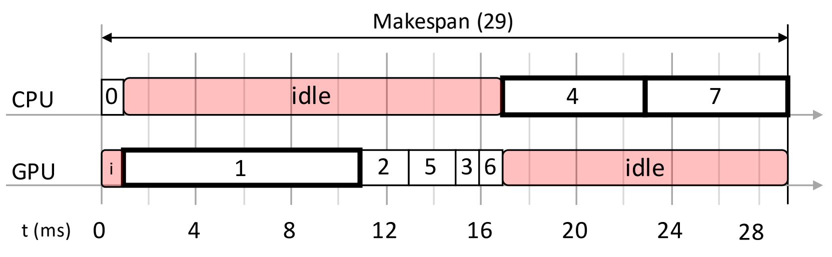
Embedded HEFT, exclusive overlapping and XEFT

Figure 3.1 depicts an example DAG model of a SW application to be deployed on a programmable heterogeneous CPU/GPU device. We assume there exists a library of primitives that implement the DAG nodes. In particular, the library includes the *exclusive* implementation for CPUs of node #0 (which is the starting point of the application), of node #4 and of node #7. The library also includes the exclusive implementation for GPUs (i.e., GPU kernel) of node #1, while it provides multiple implementations (CPU implementation and an equivalent GPU kernel) of nodes #2, #3, #5, and #6. Figure 3.1 also reports the execution time of each primitive when executed in isolation on the corresponding CE.

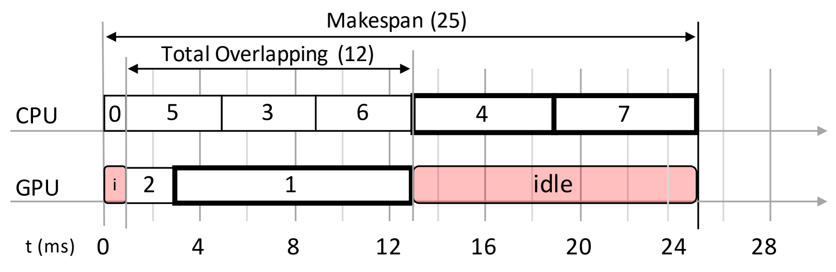
For the sake of brevity, in this example we assume the data transfer time between CPU and GPU to be negligible (we considered such an information both in the proposed scheduling formulation, implementation, and in the experimental analysis). We consider the heterogeneous target device to consist of one CPU core and one GPU accelerator. We also assume task executions to be non-preemptive.

Figure 3.2(a) shows the mapping and scheduling of the DAG nodes implemented by the runtime system of NVIDIA VisionWorks. A very similar approach is implemented by the runtime system of AMD OpenVX (AMDOVX). The mapping implements the *local best* optimization, i.e., a node is mapped on the GPU accelerator if there exists the corresponding GPU kernel in the library. The scheduling algorithm considers the topological order of nodes in the DAG and honors the topological order constraints among nodes. VisionWorks does not implement overlapped execution of tasks across the different CEs. In any case, implementing such an overlapping would reduce the makespan by 1 unit.

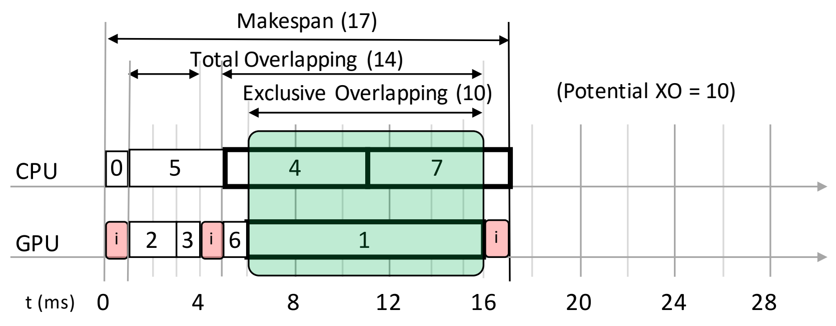
Figure 3.2(b) depicts, for the same running example of Figure 3.1, the mapping and scheduling of the proposed HEFT implementation for OpenVX. Such an implementation takes advantage of both task overlapping and the mapping implements a heuristic that targets the *global best* optimization. Starting from a task ranking generated as for equation (2.1), the algorithm maps one node at a time onto the CE that involves a better *application* execution time. A node can be mapped on a CE that leads to a higher execution time at the task level (see tasks of nodes #5, #3, and #6 in the example of Figure 3.2(b)). If we assume that the nodes have the multiple implementations and not necessarily all the GPU kernels are faster then the corresponding CPU primitives, the HEFT algorithm heuristically



(a): NVIDIA VisionWorks **standard** task scheduling order: 0, **1**, 2, 5, 3, 6, **4**, **7**



(b): **HEFT** task scheduling order: 0, 2, 5, **1**, 3, 6, **4**, **7**



(c): **Optimized HEFT** task scheduling order: 0, 2, 5, 3, 6, **1**, **4**, **7**

Fig. 3.2: Task scheduling algorithms of the DAG of Fig. 3.1: native NVIDIA VisionWorks (a), HEFT (b), and the proposed optimized HEFT (c).

provides workload balance over the CEs by *overlapping* the task execution. As confirmed by our experimental analysis, this leads to performance improvements at system level (i.e., reduction to the application makespan). However, the library includes nodes that do not have multiple implementations. In this case, the iterative nature of node mapping and balancing of HEFT may lead to (even long) idle periods. A meaningful example is the idle period on the GPU in Fig. 3.2(b), which could be reduced (or even fully avoided) by an implementation of node #7 for GPU.

In general, in the context of heterogeneous architectures, the main limitation of HEFT is that it iteratively maps one task at a time by following the rank order and by targeting the best load balancing at each iteration. It does not consider the single or multiple implementations of the nodes.

The proposed idea is that the load balancing can be improved by prioritizing the overlap between exclusive nodes, which we call *exclusive overlapping*. Considering the standard definition of *overlapping* between two tasks t and q as follows:

$$O(t,q) = \max\left(0, \min\left(t_{end}, q_{end}\right) - \max\left(t_{start}, q_{start}\right)\right) \tag{3.1}$$

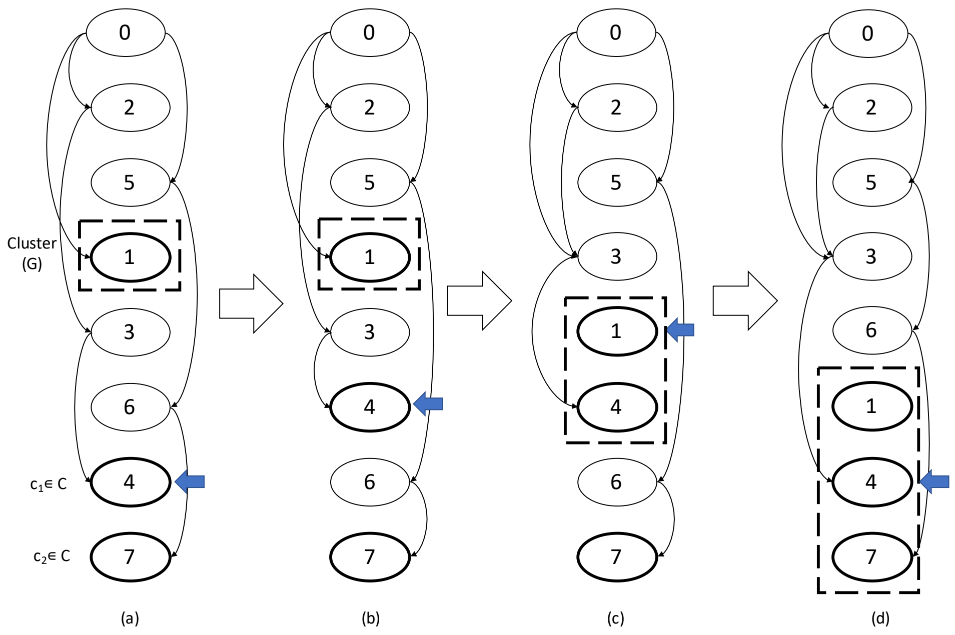


Fig. 3.3: Cluster generation step (`APPLY(rank, cluster)`) for the example in Fig. 3.1.

where t_{start} and t_{end} are the starting and ending times of task t, respectively. We define *Exclusive Overlapping (XO)* between two tasks t and q running on different CEs as follows:

$$XO(t,q) = \begin{cases} O(t,q), & \text{if } (\nexists\, t_{CPU} \wedge \nexists\, q_{GPU}) \\ & \vee(\nexists\, t_{GPU} \wedge \nexists\, q_{CPU}) \\ 0, & \text{otherwise} \end{cases} \tag{3.2}$$

where t_{CPU} and t_{GPU} represent the CPU implementation and GPU implementation, respectively, of task t.

The exclusive overlapping applies to nodes that cannot compete for the same CE due to exclusive implementations. It is a subset of the standard overlapping. We define the total overlapping and the total exclusive overlapping between tasks *of an application* A as follows:

$$O(A) = \sum_{t,q \in A} O(t,q) \tag{3.3}$$

$$XO(A) = \sum_{t,q \in A} XO(t,q) \tag{3.4}$$

Figure 3.2(c) shows the exclusive overlapping between the tasks of our running example. The main idea is that increasing XO can lead to shorter idle periods when these periods are caused by the combination of DAG constraints and exclusive implementation (see for example the idle time on the GPU between instants 13 and 25 in Figure 3.2(b)). In our experimental analysis, we found that increasing XO corresponds to an increase of the standard overlapping and, as a consequence, to a performance improvement.

To increase XO, we propose an algorithm (*Algorithm* 1) that, starting from a given ranking list, it reorganizes the list to identify and generate *clusters* of exclusive nodes, i.e., sequences of exclusive nodes that are strictly consecutive in the ranking (see nodes #1, #4, and #7 in Figure 3.2(c)).

The algorithm starts from the standard task ranking obtained by applying HEFT to the application graph (row 2). One node at a time, and for every node of the list (row 4), the algorithm identifies a new cluster starting from the next exclusive node of the list (row 6). It searches among all the next nodes in the ranking that are exclusive and that do not have topological constraints in the DAG with the current cluster nodes (i.e., that are not in the

Algorithm 1: Cluster identification and generation

```
1  Procedure BuildCluster(graph):
2      $rank \leftarrow build\_rank(graph)$
3      $i \leftarrow 0$
4      while $i < size(rank)$ do
5          if $rank[i]$ is exclusive then
6              $candidates \leftarrow rank[i]$
7              $j \leftarrow i + 1$
8              while $j < size(rank)$ do
9                  if $rank[j]$ is exclusive $\wedge \forall p \in candidates, p \not\twoheadrightarrow rank[j]$ then
10                     $candidates \leftarrow candidates \cup rank[j]$
11                 end
12                 $j \leftarrow j + 1$
13             end
14             $total_{cpu} \leftarrow reduce_{sum}(candidates, t_{cpu})$
15             $total_{gpu} \leftarrow reduce_{sum}(candidates, t_{gpu})$
16             $C \leftarrow$ exclusive CPU nodes in $candidates$
17             $G \leftarrow$ exclusive GPU nodes in $candidates$
18             if $total_{gpu} < \frac{total_{cpu}}{n_{cores}}$ then
19                 $cluster \leftarrow G$
20                 for $\forall c \in C$ do
21                     $t \leftarrow reduce_{sum}(cluster, t_{cpu})$
22                     if $|total_{gpu} - \frac{t + c_{cpu}}{n_{cores}}| < |total_{gpu} - \frac{t}{n_{cores}}|$ then
23                         $cluster \leftarrow cluster \cup c$
24                     end
25                 end
26             else
27                 $cluster \leftarrow C$
28                 for $\forall g \in G$ do
29                     $t \leftarrow reduce_{sum}(cluster, t_{gpu})$
30                     if $|\frac{total_{cpu}}{n_{cores}} - t + g_{gpu}| < |\frac{total_{cpu}}{n_{cores}} - t|$ then
31                         $cluster \leftarrow cluster \cup g$
32                     end
33                 end
34             end
35             Apply$(rank, cluster)$
36             $i \leftarrow cluster_{end} + 1$
37         else
38             $i \leftarrow i + 1$
39         end
40     end
41 return rank
```

same DAG path). The second condition is necessary to avoid serialization among the cluster nodes. The algorithm then extrapolates the total makespan of the cluster nodes for each CE (rows 12 and 13). The shortest among the calculated makespans characterizes the maximum XO of the cluster under generation. For the sake of clarity, in *Algorithm* 1, we considered two possible cluster makespans ($total_{cpu}$ and $total_{gpu}$). The algorithm concludes the cluster identification by including all nodes (for the same CE) that give the shortest makespan and, incrementally, with the exclusive nodes that lead to a comparable makespan on the other CEs (rows 16-26). The first node of any other CE that causes makespan unbalancing starts a new cluster in the following iteration of the algorithm.

The algorithm generates clusters of exclusive nodes by moving either the identified nodes up on the ranking or the whole cluster down on the ranking. All the identified nodes (i.e., *candidates* in *Algorithm* 1) and the cluster can be moved and made adjacent since, for the condition in row 9, they cannot have topological constraints against each other.

The `APPLY(rank,cluster)` procedure implements such a node shift in the ranking by considering the identified cluster. Figure 3.3 shows, as an example, the two shift types for the running example. Given that nodes #1, #4, #7 have not topological constraints, $G = \{\#1\}$, $C = \{\#4, \#7\}$, $total_{cgu} = 10$, and $total_{cpu} = 12$, the cluster starts by node #1 (Figure 3.3(a)). The algorithm merges the CPU node #4 to the cluster in two steps. First (Figure 3.3(b)) by moving node #4 upword in the ranking since it has not topological constraints with the switching nodes (node #6 in the example). Then (Figure 3.3(c)) by moving downword the cluster since node #4 has topological constraints with node #3. For the same concept, the algorithm merges node #7 to the cluster by moving downword the cluster (Figure 3.3(d)), since node #7 has a topological constraint with node #6.

Reducing idle time with pipelined DAG executions

In embedded vision applications, the whole DAG model has to be applied to each image (i.e., *frame*) of the input stream. In this context, a different technique to reduce the idle periods generated by the scheduler on the CEs is to overlap DAG executions. The idea is that multiple instances of DAG nodes can create a larger pool of tasks that satisfy their temporal constraints and that can be selected by the scheduler along the makespan. Such a pipelined DAG execution involves an additional double temporal constraint among nodes. Considering a pipeline of n overlapping frames, where $0 \leq i < n$:

$$\begin{aligned} &if\ initNode^{i} \in DAG^{i}\ and\ initNode^{i+j} \in DAG^{i+j} \\ &\Rightarrow initNode^{i}_{start} < initNode^{i+j}_{start}\quad with\ 1 \leq j < n \end{aligned} \tag{3.5}$$

$$\begin{aligned} &if\ termNode^{i} \in DAG^{i}\ and\ termNode^{i+j} \in DAG^{i+j} \\ &\Rightarrow termNode^{i}_{end} < termNode^{i+j}_{end}\quad with\ 1 \leq j < n \end{aligned} \tag{3.6}$$

whereby the initial task of any DAG instance always has to be scheduled before the corresponding initial tasks of the later DAG instances (Eq. (3.5)) and any DAG last task cannot be scheduled (and thus terminated) before the last task of an earlier DAG (Eq. (3.6))[1]. This is because not all applications allow input frames to be computed out of order due to inter-frame dependencies, so this restriction forces an *in order* execution that works for all visual computing applications. To avoid starvation of tasks, those from earlier DAG instances always have higher priority compared to the ones from later DAGs.

In addition, the scheduling time of any initial DAG task should consider the frame rate of the input camera, so that:

$$initNode^{i+1}_{start} - initNode^{i}_{start} \geq 1/CameraFPS \tag{3.7}$$

Considering, for example, an input sensor at 30 FPS, the distance between the initial nodes of two consecutive DAG instances has to be greater than $33.\bar{3}$ ms.

We started from the approach proposed in [31], in which the authors propose a pipelined version of the G-FL scheduling algorithm. Although the pipeline mechanism leads to makespan reductions w.r.t. the non-pipelined version on several benchmarks, it does not give any advantage on the majority real-case study we tested. This is due to the fact that the scheduling strongly depends on the constraints of Eq. (3.7).

Figures 3.4(a) and (b) show, for example, the comparison between the G-FL scheduling and a 2-frame pipelined G-FL scheduling, respectively, of our running example (Figure 3.1) by considering an input sensor at 45 FPS (22ms between consecutive instances of initial DAG node). In this example, we have several idle times but they cannot be filled with nodes from the subsequent DAG as the data for the nodes does not arrive until the 22ms mark, and therefore no overlapping can be forced.

[1] The same constraint applies between nodes of different DAG instances that have a dependency constraint (see OpenVX nodes with *delayed* output [140]).

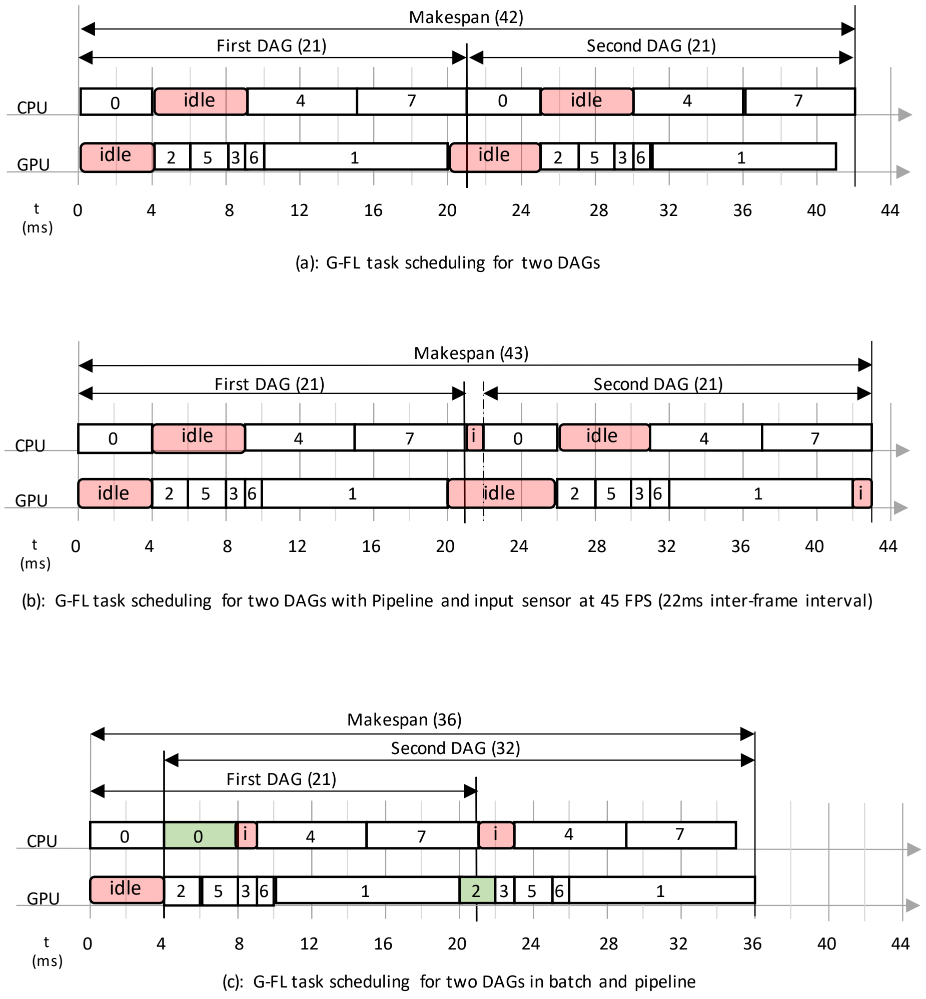

Makespan (42)
First DAG (21)
Second DAG (21)
CPU
GPU
t (ms)
(a): G-FL task scheduling for two DAGs
Makespan (43)
(b): G-FL task scheduling for two DAGs with Pipeline and input sensor at 45 FPS (22ms inter-frame interval)
Makespan (36)
Second DAG (32)
First DAG (21)
(c): G-FL task scheduling for two DAGs in batch and pipeline

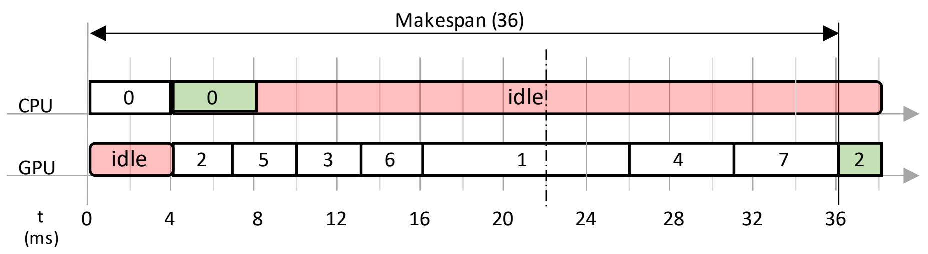
Fig. 3.4: G-FL task scheduling of example of Figure 3.1 without pipeline (a), with 2-frame pipeline and input sensor at 45 FPS (b), and with 2-frame batched pipeline (c).

To relax the constraints of Eq. (3.7), we implemented a *batched* approach for the DAG pipelining, by which a number n of input frames are stored simultaneously in a buffer structure in memory. We evaluated the overhead needed to handle the pipeline structure, the additional memory required to keep multiple frames in system memory, and the additional scheduling time required due to additional frames being in queue in Section 4.1.1.

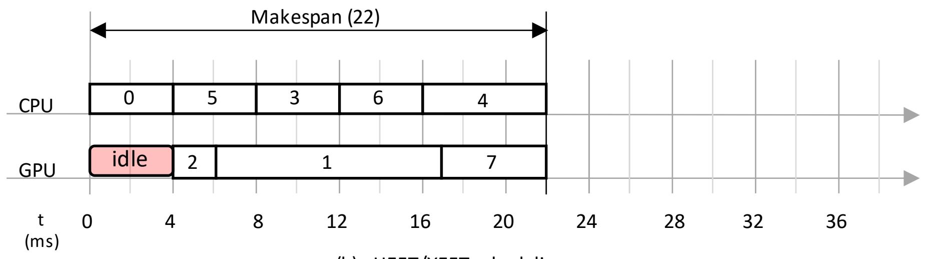
Figure 3.4(c) shows the result of such a batched solution. In the example, a batch of two frames allows the second instance of node #0 (green) to be executed right after the first instance of node #0, which then allows the second instance of node #2 to start earlier, while the last node of the first graph is still executing.

The batching mechanism improves the performance of any pipelined scheduling. Nevertheless, as confirmed by our experimental analysis, the combination of the pipeline with the batching mechanisms strongly amplifies the dependency of the scheduling performance with the mapping algorithm. This is particularly relevant in the context of OpenVX frameworks, in which the multiple implementation of many nodes makes the mapping strategy critical.

In particular, the most important factor to consider when using the pipeline is that the mapping has to be balanced over the different CEs. We found that the pipelined G-FL scheduling often causes bottlenecks on the GPU and idle periods on the CPU cores. This is due to the mapping approach implemented in [31], which emulates the runtime system of NVIDIA VisionWorks. The mapping process gives priority to the local best node implementations, i.e., the GPU kernels for almost every node. Because of this, the GPU is



(a): GPU bottleneck when using G-FL task scheduling with the native mapping



(b): HEFT/XEFT scheduling

Fig. 3.5: Task scheduling of batched pipelined G-FL for the example of Figure 3.1, in which all nodes, except node #zero, have been considered with the GPU implementation one time unit faster then the corresponding CPU implementation (a), and the result of rank-based approaches like HEFT and XEFT (b).

often loaded with a disproportionate number of nodes compared to the CPU cores, creating a bottleneck where the pipeline cannot do anything as almost all nodes are forced in a sequential execution.

Figure 3.5(a) shows the scheduling provided by the batched pipelined G-FL on a modified version of our running example, in which we consider all nodes except node #0 to have a faster GPU implementation by one time unit.

The result is that all nodes have been allocated to the GPU, and due to the graph dependencies, the overlapping is limited due to the intra-DAG time constraints between nodes and the GPU bottleneck. In these cases, any rank-based scheduling approach such as HEFT and XEFT does a much better job at balancing the workload (see Figure 3.5(b)).

3.1.2 Improving performance on Edge computing embedded boards with Unified Memory Architecture

Unlike traditional CPU-GPU communication models, which require copying data from the CPU memory to the GPU memory and back, *Zero Copy (ZC)* allows CPU and GPU to access concurrently to a shared memory space inside the UMA.

Communication based on ZC consists of passing data through pointers to the *pinned* shared address space. When CPU and the iGPU physically share the memory space, their communication does not rely on the PCIe bus; thus, communication can be made more efficient through ZC. Such a communication model is instrumental for performance and energy efficiency in many real-time applications. Examples are camera- or sensor-based applications, in which the CPU offloads streams of data to the GPU for high-performance and high-efficiency processing [152, 153].

Zero-copy through shared address space requires the system to guarantee cache coherency across the memory hierarchy of the two processing elements. Since the overhead involved by SW coherency protocols applied to CPU-iGPU devices may elude the benefit of the zero-copy communication, many systems address the problem by disabling the Last Level Cache (LLC) completely (see Fig. 3.6(a)) [154].

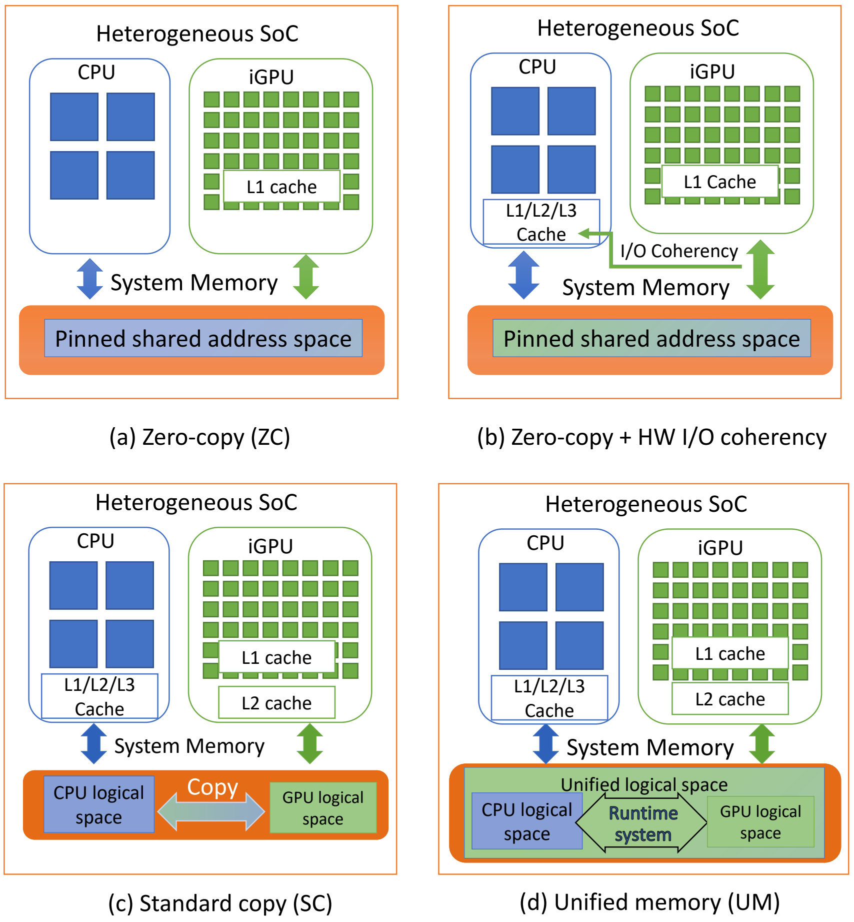


Fig. 3.6: CPU-iGPU communication models.

Although zero-copy best applies to many SW applications (e.g., applications in which CPU is the data producer and the GPU is the consumer), it often leads to strong performance degradation when the applications make intensive use of the GPU cache (i.e., *cache-dependent applications*). To reduce such a limitation, more recent embedded devices include hardware-implemented I/O coherence, by which the iGPU directly accesses the CPU cache, while the GPU cache is still disabled (see Fig. 3.6(b)). Even though these solutions limit the performance loss, traditional communication models based on data copy, which we call *Standard Copy (SC)*, are often the best solution with cache-dependent SW applications. With SC between CPU and iGPU, the physically shared memory space is partitioned into different logical spaces and the CPU copies the data from its own partitions to the iGPU partitions (see Fig. 3.6(c)). The caches, which are all enabled, hide the data copy overhead, and cache coherence is guaranteed implicitly by flushing the caches before and after each GPU kernel invocation.

To ease the CPU-GPU programming and avoid explicit data transfer invocations, user-friendly solutions allow the programmer to implement CPU-iGPU communication through data pointers. In this communication model, which we call *Unified Memory (UM)*, the physically shared memory is still partitioned into CPU and GPU logic spaces although they are abstracted and used by the programmer as a virtually unified logic space. The runtime system maintains cache coherence through on-demand page migration (see Fig. 3.6(d)) [155].

Choosing the most efficient CPU-iGPU communication model amongst SC, UM, and ZC depends on both the SW application characteristics (i.e., compute vs. memory bound, usage of caches, etc.) as well as the characteristics of the target embedded device. Indeed, the overhead introduced by the cache coherence driver or the advantages provided by any I/O coherence implemented in hardware may affect the overall performance.

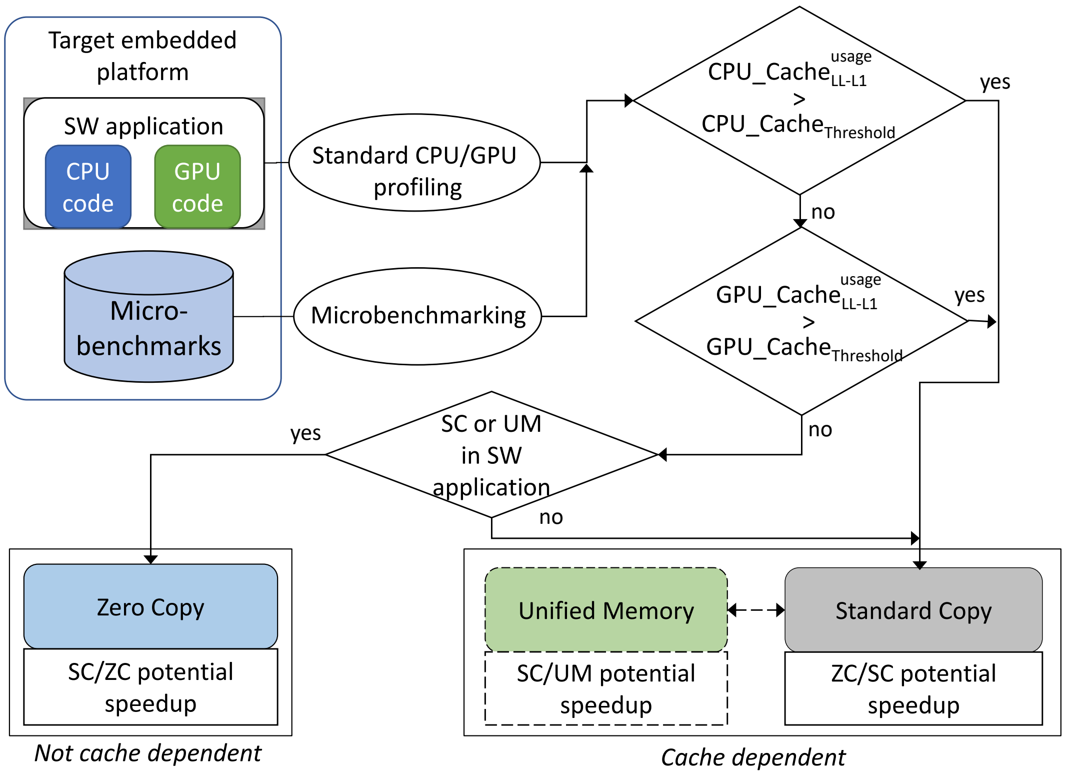


Fig. 3.7: Overview of the proposed framework.

In this section, we present a framework that, through the use of a set of micro-benchmarks and of a performance model, analyses the characteristics of the SW application and the target device to provide the potential performance the system can reach by changing the communication model. Switching from one communication model to another is often a time consuming and error prone task. Even more challenging is the switching from the models in which CPU routines and iGPU kernels are implicitly synchronized (i.e., SC and UM) to the model in which synchronization and task overlapping is the responsibility of the programmer. This framework (which is available for download at `github.com/FrancescoL96/Cache-Benchmark`) endeavours to support the programmer during the development and tuning of CPU-iGPU applications by proposing the most suited communication model and, in case of zero-copy, a communication pattern to best exploit this communication model's potential for improved performance.

This section introduces the proposed framework, as well as the following:

- A *set of micro-benchmarks* to characterize the CPU-iGPU communication performance of the device. The micro-benchmarks mix different amounts of computation and memory accesses on both the CPU and the iGPU to measure the performance impact of each communication model on the given embedded device.
- *A performance model* that combines the information provided by the micro-benchmarks and by any standard profiling tool to extrapolate the potential speedup the system performance can reach by switching from one model to another.
- A *zero-copy communication pattern* to enhance the system performance by taking advantage of a synchronized and overlapped execution of CPU and iGPU tasks.

The Framework

Fig. 3.7 shows the overview of the proposed framework. Given a SW application and a target embedded platform, a standard profiling tool is applied to extrapolate information on the usage levels of both CPU and GPU caches[2].

The idea is that if the application strongly benefits from both caches, then concurrent accesses of CPU and iGPU on the *pinned* shared memory space and thus the ZC communication model cannot provide the best overall performance. This is due to the fact that the cache coherence protocol, or the caches disabled for guaranteeing consistency with the zero-copy model, may elude the benefit of eliminating the data copy. In this case, SC or UM are the best solutions. In contrast, if the cache usage is low on the iGPU, the best model depends on the CPU cache usage. If the CPU cache usage is high, ZC could give the best performance if the device implements a sufficiently efficient cache coherence protocol (e.g., the hardware I/O coherence of the NVIDIA Jetson AGX Xavier). Otherwise, SC or UM are likely the best solutions.

In case the application makes low usage of both the CPU and iGPU caches (i.e., the caches do not affect the application performance), ZC could provide at least equivalent performance w.r.t. SC and UM. In this case, ZC is generally preferred, as shown in the experimental results section, because it can guarantee lower energy consumption due to the saved data transfers for the copies. The performance model proposed aims at characterizing such a *cache usage* of the application, and correlates this value with the potential performance of each communication model. As a result, considering the application, the implemented communication model, and the target device, the framework allows the user to accurately estimate the potential speedup the application may have by changing the communication model.

It is important to note that the characteristics of the target device strongly affect the choice of the best communication model and the potential speedup achieved from one model to another. The micro-benchmarks aim at extrapolating the device characteristics and accurately estimating such a potential speedup.

Finally, ZC may provide even better performance if combined with an overlapped execution of CPU and iGPU tasks. The performance model allows estimating the maximum performance improvement of ZC with task overlapping. We propose an access pattern to implement such an overlapping while guaranteeing data consistency.

Profiling and performance model. Given a SW application and an embedded device, we define the usage of the LLC of the CPU and iGPU as follows:

$$\begin{aligned} CPU\_Cache^{usage}_{LL\_L1}(\%) = {} & miss\_rate\_\mathrm{L1}_{CPU} \\ & \times \quad (1 - miss\_rate\_LL_{CPU}) \end{aligned} \tag{3.8}$$

$$\begin{aligned} GPU\_Cache^{usage}_{LL\_L1}(\%) = {} & \frac{t_n * t_{\text{size}} * (1 - hit\_rate\_L1_{GPU})}{kernel\_runtime} \\ & \times \quad \frac{1}{GPU\_Cache^{max\_throughput}_{LL\_L1}} \end{aligned} \tag{3.9}$$

where t_n and t_{size} represent the number of memory transactions and their size, respectively. The definitions assume an architecture with L2 as LLC. It can be generalized for different memory architectures. The two equations represent, in percentage, the amount of data that is obtained from the LLC of the CPU and iGPU, respectively out of all the requested data from the CPU/iGPU multiprocessors. All the miss/hit rate information are provided by any standard profiling tool. The maximum throughput in Eq. (3.9) is extrapolated by the micro-benchmarks.

[2] We consider the LLC for both CPU and iGPU as the caches involved in data coherency protocols for CPU-iGPU communication. We consider them as the only caches disabled with ZC.

We define the potential speedup a SW application may have by changing the communication model as follows:

$$\begin{aligned}\mathrm{SC/ZC}_{speedup} &= \frac{SC_{runtime}}{\dfrac{SC_{runtime} - copy\_time}{1 + (CPU_{time}/GPU_{time})}} \\ &\leq SC/ZC_{Max\_speedup}\end{aligned} \tag{3.10}$$

$$\begin{aligned}\mathrm{ZC/SC}_{speedup} &= \frac{ZC_{runtime}}{\dfrac{ZC_{runtime}}{1/[1 + (CPU_{time}/GPU_{time})]} + copy\_time} \\ &\leq ZC/SC_{Max\_speedup}\end{aligned} \tag{3.11}$$

where $SC_{runtime}$ ($ZC_{runtime}$) is the execution time of the SW application with the SC(ZC) communication model. $copy\_time$ is the time spent for CPU-iGPU data transfer, CPU_{time}(GPU_{time}) is the runtime of the only CPU task (GPU kernel).

`SC/ZC`$_{Max\_speedup}$ and `ZC/SC`$_{Max\_speedup}$ represent the maximum speedup that can be obtained by moving from one model to another on the given device. These values are independent from the application and are extrapolated by the micro-benchmarks. Eq. (3.10) defines the potential speedup an application that has been classified as not cache-dependent (first checks of the framework flow) may have by replacing the originally implemented SC model with ZC. It takes into account the SC runtime from which the data copy time is removed and the potential overlap between CPU and iGPU computation (CPU_{time}/GPU_{time}).

Eq. (3.11) defines the potential speedup an application classified as cache-dependent may have by switching from ZC to SC. It takes into account the overall time needed by SC to explicitly copy data. It also considers the serialization of the CPU and iGPU tasks since their overlapping is not allowed in SC. It is important to note that if an application is cache dependent and originally implemented with SC, the framework does not suggest any change to the communication model and any further potential speedup.

For the sake of space and without loss of generality, we consider the performance of UM similar to SC. In all our micro-benchmarks, the maximum difference between the two model performance ranges between ±8% in all the considered devices. The difference is strictly related to the driver implementation of the on-demand page migration. Compared to the difference between SC(UM) and ZC, we consider negligible the difference of performance between SC and UM. It is also important to note that the programming effort to switch between these two models (SC and UM) is minimal.

Micro-benchmarks. The micro-benchmark code is implemented with the aim of satisfying four main properties.

- *Stressing capability.* The micro-benchmarks apply extensive and heavy workloads to the memories and caches. This allows the framework to reach the steady state in which each functional component is fully work-loaded to measure the real (w.r.t. theoretical) peak performance while minimizing the side effects that can incur throughout the measurements.
- *Workload variability.* The communication model and the corresponding components are stressed with different workloads. This allows the framework to quantify the effect of moving from one model to another.
- *Selectivity.* The micro-benchmarks stress, as much as possible, only one target functional component at the time. This allows the framework to extrapolate the potential speedup obtained by moving from one communication model to another, considering that they can involve different functional components.
- *Portability.* The micro-benchmarks are implemented independently from any CPU/iGPU platform.

Since the compiler may optimize the code (e.g., code-block reordering, dead code elimination, etc.) and, by means of the consequent side effects, it may affect the target properties,

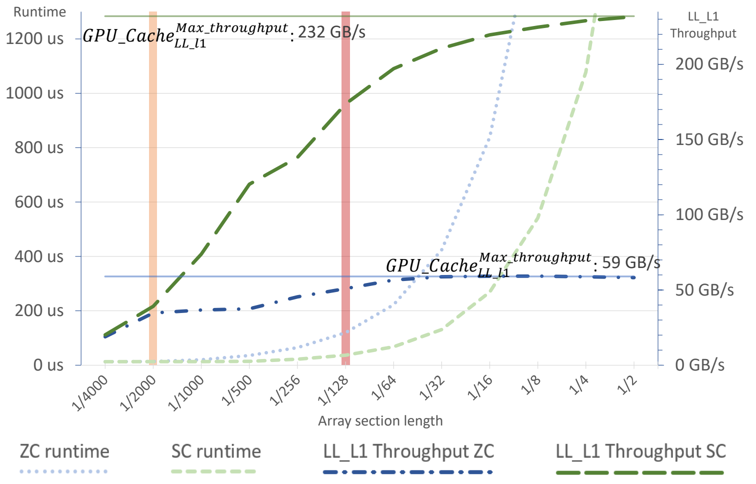


Fig. 3.8: Second micro-benchmark results on an NVIDIA Jetson Xavier. Relationship between LL_L1_throughput and kernel times of the iGPU.

the code has been checked and refined throughout the different steps of the compilation process.

The first micro-benchmark aims at finding the peak throughput of the GPU LL-L1 cache on the target device ($GPU\_Cache_{LL\_L1}^{max\_throughput}$). This value allows accurately classifying an application as GPU *cache dependent/not cache-dependent* (see Eq. (3.9)). Then, in case the application is cache dependent and originally implemented with the ZC model, this value is used to estimate the potential speedup obtained by switching from ZC to SC ($\texttt{ZC/SC}_{Max\_speedup}$ in Eq. (3.11)). It implements the elaboration of a matrix data structure computed by both CPU and GPU. In particular, the CPU performs a series of floating point operations with data read and written from and to a single memory address. These operations include square roots as well as divisions and multiplications. The GPU performs a 2D reduction multiple times through linear memory accesses. This is achieved through iterative loading of the operands (i.e., `ld.global` instructions), a sum (i.e., `add.s32`), and the result store (`st.global`). The two classes of operations (CPU and iGPU) allow the micro-benchmark to evaluate the peak usage of the CPU and iGPU caches. The two routines (CPU and GPU) are evaluated by considering the three communication models. ZC makes use of the concurrent execution of the routines and the concurrent access to the shared data structure. SC and UM explicitly exchange the data structure before the routine computation.

The second micro-benchmark implements extensive GPU computations, with varying levels of linear memory accesses. It aims at finding the GPU cache thresholds used by the framework to suggest the best communication model between ZC and SC/UM (see Fig. 3.7). The micro-benchmark routine accesses sections of different lengths of a fixed-size array (e.g., from 1/4000 to 1/2), through a single `ld.global` and `st.global`, combined with a `fma.rn` (i.e., fused multiply-add) that uses two locally calculated values. It is implemented with both ZC and SC communication models and extrapolates the $GPU\_Cache_{Threshold}$ value by comparing the LL-L1 caches. Fig. 3.8 shows, for example, the results for this micro-benchmark on an NVIDIA Jetson Xavier, in which the threshold has been found at 1/2000 accesses. A comparable throughput of the two models (from 1/4000 to 1/2000) translates into comparable system runtime with the two models (see light dotted lines in the figure). From 1/2000 onward, the difference between the throughput values and the runtime linearly increases. $GPU\_Cache_{Threshold}$, in percentage, is calculated by considering the last comparable value of the throughput over the peak cache throughput ($GPU\_Cache_{LL\_L1}^{max\_throughput}$) provided by the first micro-benchmark (i.e., 20 GB/s and 59 GB/s, respectively, for the SC model in the example). The micro-benchmark extrapolates the $CPU\_Cache_{Threshold}$ in a similar way.

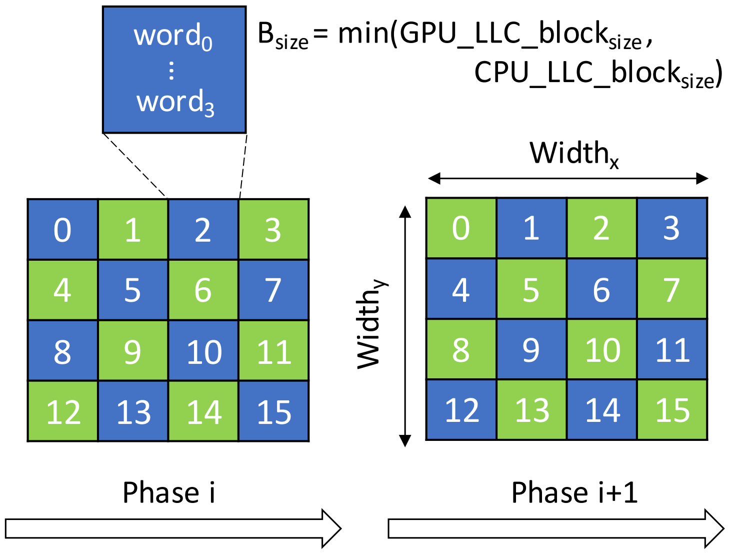


Fig. 3.9: Overview of the communication pattern for ZC.

The third micro-benchmark performs a balanced CPU+iGPU computation through a routine whose performance is fully independent from the GPU cache. The GPU kernel implements repetitive memory accesses with sufficiently sparse single read access (`ld.global`) and single write access (`st.global`) in order to guarantee the maximum miss rate. It implements a concurrent access pattern and a perfect overlapping of the CPU and iGPU computations to extrapolate the maximum communication performance (and thus $SC/ZC_{Max\_speedup}$ and $ZC/SC_{Max\_speedup}$) the given embedded platform can provide by considering both SC and ZC.

Zero-copy communication pattern. With ZC, data copy is removed while CPU and iGPU access concurrently to the same pinned logical and physical space. Concurrent accesses by heterogeneous processors requires both data consistency and race conditions to be solved by the programmer. Even though explicit synchronization points would allow these issues to be easily addressed, the overhead involved by synchronization strongly affects the overall system performance. To avoid explicit synchronizations at every memory access while guaranteeing deterministic results, we propose a concurrent pattern based on tiling [156], which is accessed by CPU and iGPU through alternate producer-consumer phases.

Fig. 3.9 shows an overview of the communication pattern. An n-dimensional data structure, where n depends on the problem (2D matrix for images in the example of Fig. 3.9), is created and its size ($Width_x \times Width_y$) is calculated depending on the available GPU LL cache. The data structure is partitioned into data blocks (tiles) which size (B_{size}) corresponds to the smaller size between GPU and CPU LLC cache block size. This allows each access to a tile to be performed by a coalesced memory transaction.

CPU-iGPU communication relies on a pipelined sequence of access phases in which at phase i the CPU first reads and then writes onto the even blocks while the iGPU reads and writes on the odd blocks. At phase $i + 1$, even and odd blocks are inverted for CPU and iGPU.

3.1.3 Improving performance for CPS on Edge computing embedded boards with UMA

Although some work has been proposed to study the best choice among the different communication models by considering the application characteristics [157], the effects of making these models compliant with the standard communication protocols adopted for the development of CPS have never been investigated. The design of a CPS, for example a robotic system, involves experts from different application domains [158–160], who must integrate

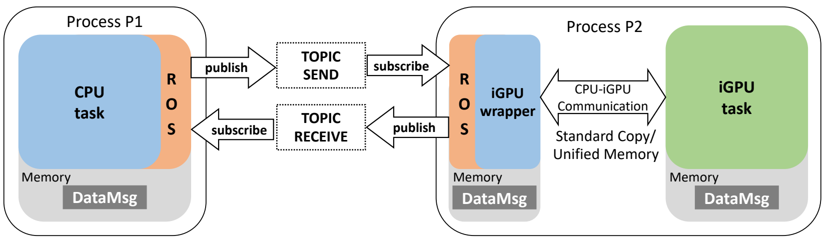


Fig. 3.10: CPU-iGPU standard copy with ROS topic paradigm.

the various software applications developed to address the specific issues of each distinct domain. However, such an integrated design is still an open problem [161–164]. In recent years, to facilitate the integration of such multi-domain components and (sub)systems, distributed frameworks like ROS have been proposed to provide the necessary support to develop distributed software for system components and devices. Compliance to ROS is nowadays a key aspect for application re-use and easy integration of software blocks in complex systems. However, it can cause performance degradation and can have a detrimental impact on the characteristics of the CPU-iGPU communication model.

This section presents different techniques to efficiently implement CPU-iGPU communication on UMA that complies with the ROS standard. We show that directly porting the most widespread communication protocols to the standard ROS architecture can lead to overheads of up-to 5000%, often eluding the benefits of GPU acceleration. The aforementioned issues are addressed through several dedicated solutions that implement ROS-compliant communication protocols that take advantage of the CPU-iGPU communication models provided by the embedded device, apply different mechanisms included in the ROS standard and are customized to different communication scenarios, from two to many ROS nodes.

Efficient ROS-compliant CPU-iGPU communication

We consider, as a starting point, communication between CPU and iGPU implemented through CUDA-SC or CUDA-UM (see Fig. 3.6(c) and Fig. 3.6(d)). For the sake of space and without loss of generality, we consider the performance of UM similar to SC, as in the previous section.

Fig. 3.10 shows the most intuitive solution to make such a communication model compliant to ROS. The main CPU task and the wrapped GPU task[3] are implemented by two different ROS nodes (i.e., processes P1 and P2). Communication between the two nodes relies on the publish-subscribe model. The CPU node (i.e., process P1) has the role of executing the CPU tasks and managing the synchronization points between the CPU and GPU nodes. The CPU node publishes input data into a send topic and subscribes on a receive topic to get the data elaborated by the GPU. The GPU node (i.e., process P2) is implemented by a GPU wrapper and the GPU kernel. The GPU wrapper is a CPU task that subscribes on the send topic to receive the input data from P1, it exchanges the data with the GPU through the standard CPU-iGPU communication model, and publishes the result on the receive topic.

This solution is simple and easy to implement. However, the system keeps three synchronized copies of the I/O data messages (i.e., for CPU task, GPU wrapper, and GPU task). For each new input data received by the wrapper through the send topic, the memory slot allocated for the message has to be updated both in the CPU logic space (i.e., for the GPU wrapper) and in the GPU logic space (i.e., for the GPU task) as for the CUDA-SC standard model.

[3] GPU-accelerated code (i.e., GPU kernels) cannot be directly invoked by a ROS-compliant node. To guarantee ROS-compliance for any task accelerated on the GPU, the GPU kernel has to be wrapped to become a ROS node.

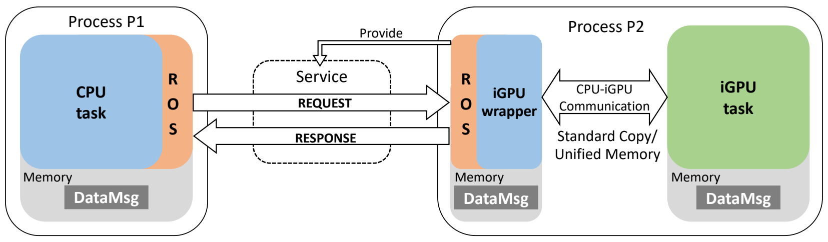


Fig. 3.11: CPU-iGPU standard copy with ROS wrapper service paradigm (RPC).

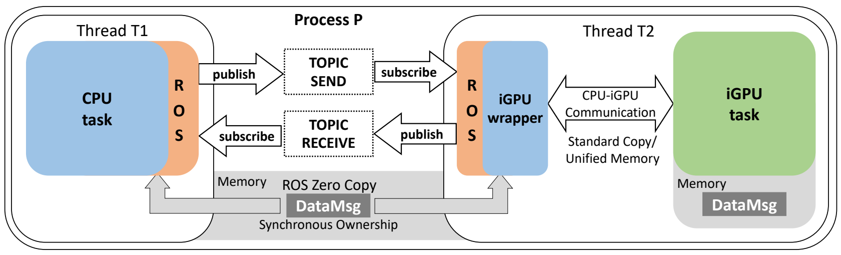


Fig. 3.12: CPU-iGPU standard copy with ROS native zero copy and topic paradigm.

CPU and GPU tasks can overlap, as the publish-subscribe protocol allows for asynchronous and non-blocking communication.

Fig. 3.11 shows a ROS-compliant implementation of the CUDA-SC model through the ROS service approach. Similarly to the publish-subscribe solution, the memory for the data message is replicated in each ROS node and in the GPU memory. The message has to be copied during every data communication, and since the service request can be performed asynchronously, CPU-iGPU operations can be performed asynchronously.

Fig. 3.12 shows a more optimized version of such a communication standardization, which is based on ROS-ZC and intraprocess communication. The ROS nodes are implemented as threads of the same process. As a consequence, each node shares the same virtual memory space, and communication can rely on the more efficient protocols based on shared memory. Nevertheless, the usage of ROS-ZC has several limitations:

- ROS-ZC can be implemented only for communication between threads of the same process. When a zero copy message is sent to a ROS node of a different process (i.e., inter-node communication), the communication mechanism falls back to the ROS standard copy.
- ROS-ZC does not allow for multiple nodes to subscribe on the same topic. If several nodes have to access a ROS-ZC message concurrently, ROS-ZC applies only to one of these nodes and returns to the ROS standard copy for the others. This condition holds for both intra-process and inter-process communication.
- ROS-ZC only allows for synchronous ownership of the memory address. A node that publishes a zero copy message over a topic will not be allowed to access the message memory address. For this reason, CPU and iGPU operations cannot be performed in parallel, as the CPU node cannot execute after sending the message (i.e., no overlapping computation is allowed over the shared memory address).

These limitations guarantee that race conditions in shared memory locations will not happen and provide better performance w.r.t. ROS standard copy in the case of intraprocess communication. Nevertheless, it does not apply to inter-process communication, which is fundamental for the portability of robotic applications.

Due to the several limitations involved by the previous solutions, we propose a new approach that maintains the standard ROS interface and its modularity advantages while taking advantage of the shared memory between processes when nodes are deployed in the same unified memory architecture (see Fig. 3.13). The idea is to implement a ROS interface

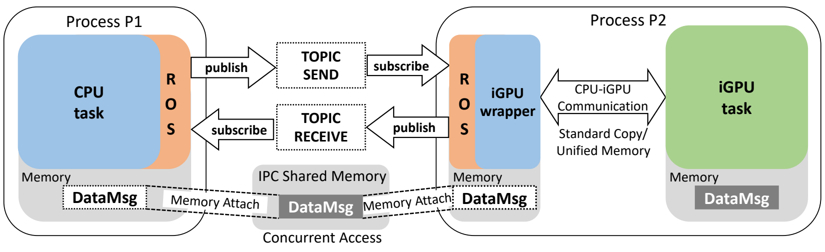


Fig. 3.13: CPU-iGPU standard copy with the proposed ROS-ZC solution and topic paradigm.

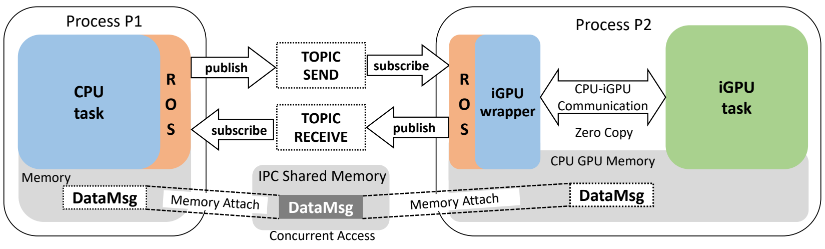


Fig. 3.14: CPU-GPU Zero Copy with our ROS Zero Copy solution and topic paradigm.

that shares the reference to the inter-process shared memory with the other ROS nodes, and exchanges only synchronization messages. The proposed solution, ROS-SHM-ZC, has the following characteristics:

- **Not only intra-process**: This solution also applies to inter-process communication by means of a Inter-Process Communication (IPC) shared memory managed by the operating system.
- **Unique data allocation**: the only memory allocated for data exchange is the shared memory.
- **Efficient CUDA-ZC (see Fig. 3.6(a) and (b))**: The reference to the shared memory does not change during the whole communication process between ROS nodes. As a consequence, it also applies to the CUDA-ZC communication between the wrapper and the iGPU task. (see Fig. 3.14).
- **Easy concurrency**: The shared memory can be accessed concurrently by different nodes, allowing concurrent access to the same memory space.

The two drawbacks of this implementation are the risk of race conditions, which have to be managed by the programmer, and the need to fall back to the standard ROS communication protocol when one of the nodes moves outside of the unified memory architecture.

Making multi-node CPU-iGPU communication compliant to ROS. The most intuitive ROS integration in a concurrent CPU-iGPU application is the partition of the CPU and iGPU tasks into two different nodes. In case of multiple nodes (CPUs and iGPU), we propose a different architecture in which a dedicated node exclusively implements the synchronization and scheduling (see Fig. 3.15). In particular, the CPU nodes wait for any new data message from the send topic, perform the defined tasks, and then return the results in the corresponding topic. The iGPU node(s) performs similarly for the CUDA kernel tasks. While the scheduler node acts as a synchronizer for the CPU and iGPU nodes, it provides the data to elaborate on the send topic, waits for CPUs and iGPU responses and, when both are received, it merges the responses and forwards the merged data.

Fig. 3.15 shows the multi-node architecture implemented as an extension of the two node architecture based on the *ROS topic paradigm*. It can be analogously implemented with the *ROS service paradigm*.

Comparing the two solutions with multi-node architecture, the topic paradigm is more efficient than the service paradigm in terms of communication, as the sending topic is shared between the other subscriber nodes. In the service paradigm, a new request has to be made

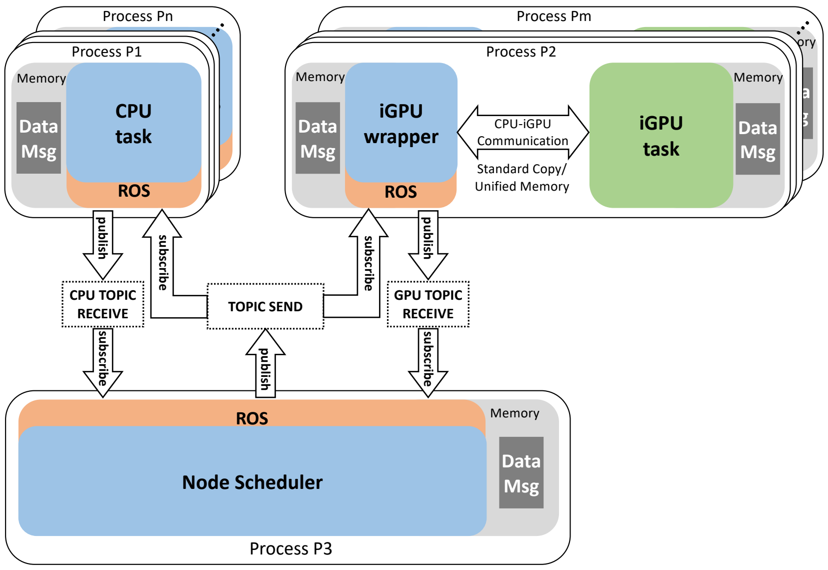


Fig. 3.15: CPU-GPU Standard Copy with ROS topic paradigm in multi-node architecture.

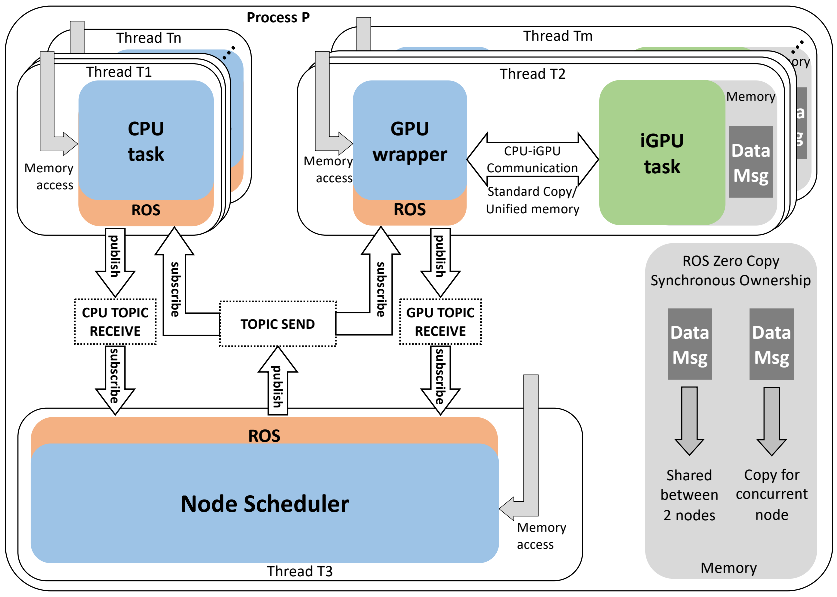


Fig. 3.16: CPU-GPU SC with ROS native Zero Copy, topic paradigm and multi-node architecture.

for each required service. This aspect is particularly important as it underlines the scalability advantages of the multi-node architecture compared to the two node architecture. Assume an application with N CPU tasks and M GPU tasks all sharing the same resource. The system can be ROS-compliant with one scheduler node, N CPU nodes, and M GPU nodes. All nodes are synchronized by the scheduler node with the ROS topic or ROS service paradigm.

With the topic paradigm, the system requires $N + M$ receive topics and one send topic, while in the service paradigm, each of the $N + M$ nodes has to create a service, which will be used by the scheduler node. Since the topic paradigm relies on a single send topic that can be shared by different subscribers, it is more efficient in the case of data-flow applications.

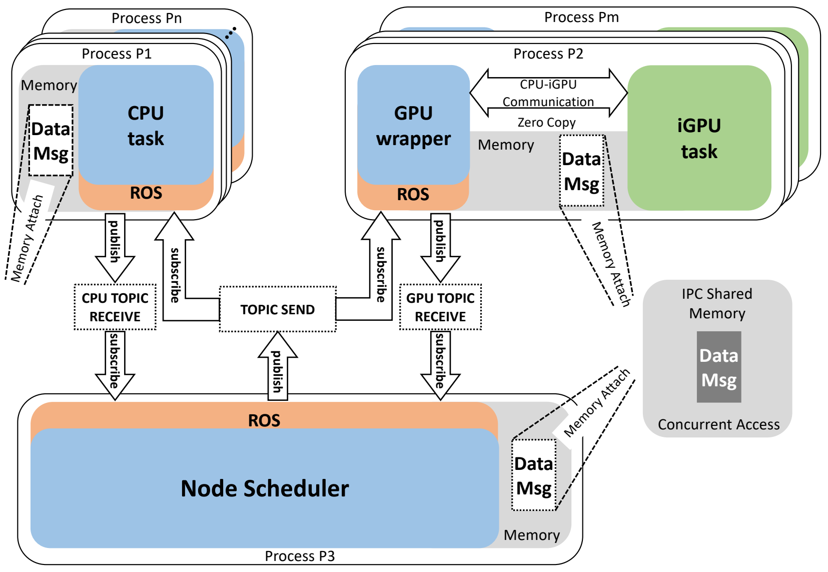


Fig. 3.17: CPU-GPU Zero Copy with our ROS Zero Copy solution, topic and 3 nodes architecture.

Fig. 3.16 shows the multi-node architecture implemented through the ROS-ZC for node communication. The CPU node, the iGPU node, and the scheduler node have the same roles as before, but they are executed as threads of the same process and the exchanged messages rely on the ROS-ZC paradigm. For this reason, two communicating nodes (i.e., scheduler-CPU or scheduler-iGPU) can exchange only the reference address of the data. The communication for the remaining, competing, node automatically switches to ROS-SC. The zero-copy exchange will be provided to the first node ready to receive the data and the choice of such a node is not predictable. In general, this architecture with N CPU tasks and M iGPU tasks requires $N + M - 1$ data instances in the virtual address memory space (e.g., for one CPU task and one iGPU the system requires two data instances, as shown in Fig. 3.16), as only one instance can be shared in zero copy between the scheduler node and another node. Therefore, due its limitations, the native ROS-ZC can reduce the memory used only by one data instance, compared to ROS-SC. The advantage of the multi-node architecture ROS-ZC is that it overcomes the two node limitations of the native ROS-ZC. Thanks to the scheduler node, which implements CPU-iGPU synchronization, all CPU and iGPU can be executed in overlap. However, since data copy is still needed, it is not possible to combine ROS-ZC to CUDA-ZC in the case of multiple nodes by fully taking advantage of the zero copy semantics. This is due to the fact that ROS does not guarantee that the same virtual address is maintained for the GPU node. As a consequence, and as confirmed by our experimental results, this solution cannot guarantee the best performance in multi-node communication compared to standard copy solutions in terms of both memory usage and GPU management.

To overcome such a limitation, we propose the solution represented in Fig. 3.17. Differently from the previous solution, the system manages the IPC shared memory and, unlike ROS-ZC, the nodes can be instantiated as processes. The scheduler node creates the IPC shared memory by using the standard Linux OS syscall, then it shares with the other nodes synchronization messages with the shared memory information. The CPU nodes and iGPU nodes wait for synchronization messages. They obtain the IPC shared memory with the OS syscalls and they perform the requested actions. This approach relies on the ROS-ZC mechanism implemented through shared memory. It applies to both intra-process and inter-process communication. However, since it aims at avoiding multiple copies of the data message, it involves more overhead to manage race conditions among the nodes sharing the same logical space.

3.2 Containerization and orchestration on heterogeneous Edge-Cloud computing architectures

In the context of containerization and orchestration for Edge-Cloud computing, existing task scheduling and resource management generally involve virtual machines or bare-metal models based on containers. Most of the studies ignore the high concurrency of computing nodes, and these studies focus mainly on how to reduce delays but do not consider a full set of extra-functional constraints, which are fundamental in the robotic context.

The proposed approach implements a container-based design flow that considers different extra-functional constraints along the robotic system programming and configuration.

Furthermore, bringing cloud-native environments to an open-hardware architecture, RISC-V, is a new necessity arising due to its increased adoption, both in research and industrial contexts. It allows unlocking the maximum potential of this new architecture in the realm of computing continuum. Furthermore, analyzing the performance of the RISC-V architecture with the porting of a complex SW infrastructure is critical.

Because of the aforementioned reasons, this section contributes the following:

- Section 3.2.1: Extending Docker and Kubernetes for ROS-compliant containerized robotic applications
- Section 3.2.2: Expanding the Edge-Cloud computing continuum to the RISC-V Open Hardware Architectures

3.2.1 Extending Docker and Kubernetes for ROS-compliant containerized robotic applications

The proposed methodology targets programming advanced robotic tasks, where every domain-specific SW component is developed and verified through a top-down design flow, while it is integrated with the other domains for SW co-design since early in the design flow for system-level (re)configuration (see bottom of Fig. 1.1). Rather than re-defining new model-based approaches for single/double domain SW development, for which there already exist solid and widespread solutions (e.g., Matlab Simulink [165], RobMoSys [166]) the proposed design flow applies *containerization*, *inter-domain* communication standards, and *orchestration* concepts to reuse existing approaches and integrate them in a multi-domain design flow.

The design methodology adopts and integrates industrial widespread applications like Kubernetes, Docker, and ROS since they are the de-facto reference standard in robotics for Industry 4.0/5.0. Other containerization and orchestration environments could still be considered, as the proposed methodology is independent from them. The main goal is the integration of multi-domain components since early in the design flow and their verification with HW-in-the-loop before deployment. We achieve this with a methodology organized in three main design and verification levels (see Fig. 3.18). We consider, as a starting point, the multi-domain SW components as ROS nodes and their communication based on the ROS standard as, at the state of the art, ROS is the de-facto reference standard for the design of robotic SW applications.

At L1, the application undergoes functional verification on server/desktop computing platforms. As verification is not the focus of this work, we will not explore it further, but any state-of-the-art technique found in the literature can be used, such as assertion based verification on ROS topics' values. Dynamics, robot model, and environment are simulated thorough a 3D simulator (e.g., Gazebo [167], Unity [168]). We also collect the data required to analyze communication between ROS nodes, which we use in the clustering (and also super-clustering phases, see Section 3.3.1).

Nevertheless, porting and testing these components to the target heterogeneous and distributed hardware and in the target software stack is a necessary step before the deployment

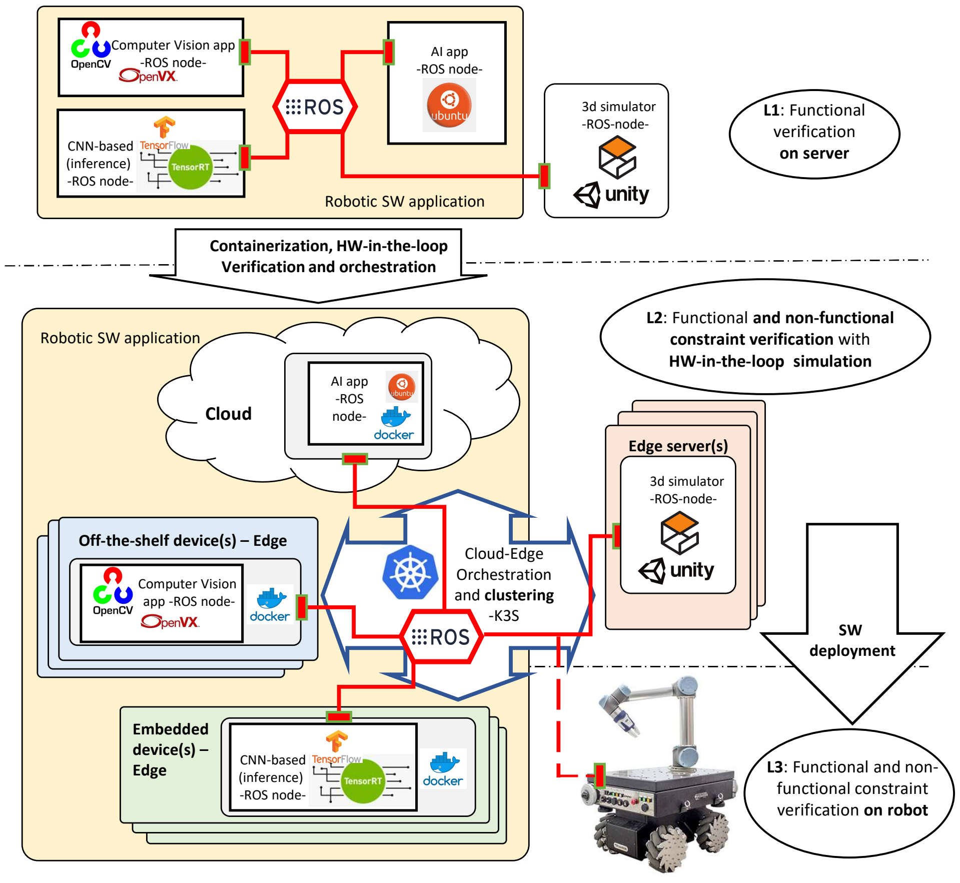

Computer Vision app -ROS node-
OpenCV
OpenVX
AI app -ROS node-
ubuntu
ROS
CNN-based (inference) -ROS node-
TensorFlow
TensorRT
Robotic SW application
3d simulator -ROS-node-
unity
L1: Functional verification on server
Containerization, HW-in-the-loop Verification and orchestration
Robotic SW application
Cloud
AI app -ROS node-
docker
L2: Functional and non-functional constraint verification with HW-in-the-loop simulation
Edge server(s)
3d simulator -ROS-node-
unity
Off-the-shelf device(s) – Edge
Computer Vision app -ROS node-
Cloud-Edge Orchestration and clustering -K3S
ROS
SW deployment
Embedded device(s) – Edge
CNN-based (inference) -ROS node-
L3: Functional and non-functional constraint verification on robot

Fig. 3.18: Overview of the design flow.

to verify extra-functional constraints like real-time vs. accuracy, and this prompts the creation of realistic yet accessible robotic platforms to close the gap between the simulation phase and the deployment phase. To avoid useless iterations of such a time-consuming step during the SW-to-HW mapping exploration, the proposed design flow first implements a containerization step of the SW components (i.e., ROS nodes) to abstract them from the target environment.

The containerization relies on the following main concepts:

- *ROS node hierarchy and abstraction levels.* We classify the nodes implementing the robotic application into abstraction levels, by which the multi-domain SW tasks are organized and integrated hierarchically, from the higher level nodes implementing the robot mission to the low-level nodes implementing the drivers. This allows for more efficient clustering of nodes and, as a consequence, for improved containerization efficiency in terms of memory overhead vs. quality of service.
- *Clustering of ROS nodes into containers.* Containerization of nodes involves both memory and CPU overhead. Such an overhead is negligible for a single node yet a problem in advanced SW applications with large numbers of nodes. In addition, containerization of nodes with intensive communication can lead to bandwidth bottlenecks. The methodology relies on a technique to cluster nodes into containers that considers bandwidth, system memory, and overhead as design constraints to optimize the system efficiency.
- *Inheritance-based container generation.* To guarantee modularity and portability of containers across Edge-Cloud devices, SW dependencies of containers have to be satisfied. Yet, the large number of containers may lead to prohibitive sizes of images that redundantly include SW packages on disks. To avoid such a useless redundancy, containers are generated by inheriting overlapping SW packages.

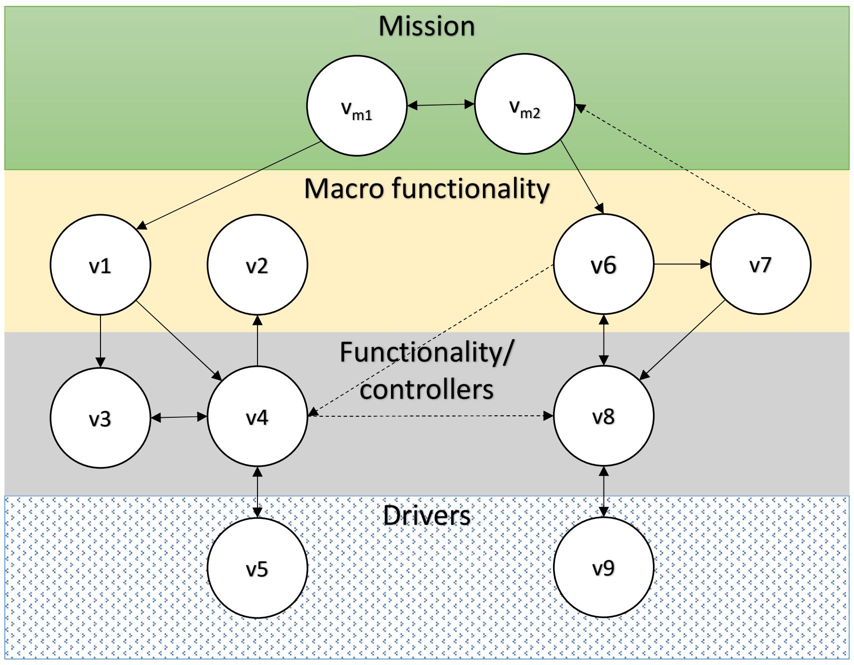


Fig. 3.19: Abstraction levels and hierarchy of ROS nodes.

- *ROS-based communication between containers.* The standard ROS efficiently implements communication only among nodes run on the same device. We implemented the *non-isolation of containers* to support node communication among nodes on distributed architectures, which is mandatory to target Edge-Cloud architectures.

The node clustering and containerization allow for the automatic porting of ROS nodes into distributed and heterogeneous devices (i.e., L2). The design methodology applies an orchestrator system that allows for the HW/SW/Network design space exploration by considering extra-functional constraints of the SW application (e.g., performance, network bandwidth, real-time computation) and the target Edge-Cloud infrastructure features. In this context, Kubernetes [169] is a standard platform to orchestrate containers and workloads, and it is largely and increasingly adopted in the context of Cloud-native SW applications. We combined Kubernetes and K3S [170] for the orchestration of the SW tasks across the HW devices (see Fig. 3.18). Functional and extra-functional constraints as well as architectural characteristics of the computing devices and network infrastructure (memory footprint, latency, bandwidth, etc.) are considered to extrapolate the best HW/SW/Network system configuration. It is important to note that, at L2, the robotic SW applications runs on the target HW (i.e., *HW-in-the-loop simulation*), while the 3D simulator still replaces the robot dynamics, model, and environment.

The 3D simulator is modularly removed from the SW system and replaced by the target robotic platform at the deployment phase (i.e., L3). At L3, functional and extra-functional correctness are verified on the robotic platform. Any simulation-based technique at the state of the art can be applied (e.g., [171]).

ROS node hierarchy and abstraction levels

We classify the ROS nodes implementing the robot application into four abstraction levels (see Fig. 3.19). Starting from the top, the *mission* level includes all nodes that implement high-level commands and behaviors, i.e. the main control of the robot. This is the most generic software and does not implement additional features, it is a collector of results from the lower levels, that combined in a specific order create recipes that achieve a specific goal (e.g., a sequence of commands to pickup an object and move it from position A to position

B). The *macro-functionality* level includes all the nodes that implement high level robotic tasks. These nodes synchronize and aggregate multiple results from lower level nodes and produce higher level abstractions (e.g., a global/local planner will use the results of the lasers, the mapping, the localization, etc. and produce a path). Functionality and controllers [172] handle the data coming from the drivers, but they are not always strictly dependent on the HW. They might modify the data from a specific format to another or prepare the instructions for a driver node. The driver level includes all nodes that interact with the robotic hardware and engines. While the nodes from the mission to the functionality/controllers level are common for the three levels of the design flow (L1, L2, L3), the driver nodes may differ from L1/L2 and L3. This is due to the fact that at L1 and L2, the drivers can be included in the 3D simulator, while at L3 the drivers are HW-dependent [173]. Through all these layers, there can be the standard ROS-based nodes, nodes custom-made by the manufacturer of the hardware or custom built by the user. Also, these last two categories of nodes are usually shared for the three levels of the design flow.

Clustering of ROS nodes into containers

To create the *images*, which are immutable objects that can be run through an isolated virtualization layer called *container*, we group ROS nodes into *clusters*. We define a cluster as a group of tightly communicating nodes. As confirmed by our experimental results, the execution of images in containers involves a system memory overhead, which is negligible for a single node yet not-negligible with the large number of nodes of real cases of study. In these cases, the trade-off between cluster granularity vs. involved overhead is a key factor for the overall system performance. The maximum granularity (i.e., one container per ROS node) guarantees the most flexibility when ported to Kubernetes but also the highest overhead. A reduced granularity, with a limited amount of containers, i.e., where each container includes a large number of nodes, involves less overhead but it also limits the orchestrator flexibility to distribute and balance the workload on the system. In addition, the clustering step also affects the bandwidth of communicating nodes. If two nodes with intensive communication are placed into two separate containers, and the containers are mapped into devices connected through a limited bandwidth network, the communication may become a system bottleneck. This can sensibly limit the usable mapping and scheduling techniques.

To create clusters, we use a greedy algorithm that groups nodes together through iterative steps by analyzing their inter-node communication. The algorithm is based on the Hierarchical Agglomerative Clustering (HAC) algorithm [174] that, when used in statistic analysis, agglomerates sets of input data points using a distance function. We adapted the algorithm to work on a graph G composed of ROS nodes. $V(G)$ represents all the vertexes and $E(G)$ represents all the edges connecting the vertexes. In the input graph, each vertex $v \in V(G)$ represents a *single* ROS node. Each $e \in E(G)$ is a non-directed weighted edge that connects two nodes v_i and v_j. The weight of the edge $(v_i, v_j) \in E(G)$ represents the communication bandwidth between two nodes. It represents the metric for clustering and is defined as follows:

$$\text{bandwidth}(v_i, v_j) = \frac{n\_topics \times m \times M_{\text{size}}}{T} \tag{3.12}$$

where n_topics is the number of ROS topics shared by (v_i, v_j), m is the number of messages exchanged between publishers and subscribers over the time T, and M_{size} is the average message size. These data are collected in the simulated environment at L1.

We define the distance function of the HAC algorithm to include multiple metrics. Unlike single-linkage or complete-linkage distance functions, which use a single edge as a metric, our distance function δ maximizes the custom metric by considering the total amount of bandwidth between clusters, i.e., the sum of the weights of the edges between connected nodes of two clusters, as follows:

Definition 3.2.1 (δ) *Given a graph G and two clusters of vertexes V_i and V_j with at least one edge between their vertexes, we define the weight δ of the edge as the sum of all the weights of the edges starting from vertexes of V_i, connected to vertexes of V_j.*

$$\forall(v_i, v_j) \in E(G) \ s.t. \ v_i \in V_i, v_j \in V_j$$
$$\delta(V_i, V_j) = \sum bandwidth(v_i, v_j)$$

The algorithm also considers the target number of clusters (n). We finally extrapolate the upper bound of n as follows:

$$n \leq \left\lfloor \frac{available\_system\_memory}{container\_overhead} \right\rfloor \tag{3.13}$$

where $available\_system\_memory$ is the system memory available for the containers and $container\_overhead$ is the overhead involved by each single container.

The output of the algorithm is a graph G' where each vertex $v \in V(G')$ is a cluster X of nodes of G such that v.rank ≥ 1 and $|X| \geq 1$. We define the rank of a vertex $v \in V(G')$ as v.rank $\in (1, .., |V(G')|)$ or as the number of nodes clustered inside each vertex.

Listing 3.20: Algorithm for cluster creation.

```
1  def cluster_nodes(G, n):
2    cull_edgeless_nodes(G)
3    while |V(G)| > n:
4      order_by_rank(G)
5      G_b = {}
6      cluster = []
7      for each node in V(G):
8        candidate = node.get_max_weight_node()
9        while (candidate in cluster):
10         candidate = node.get_max_weight_node()
11       if node not in cluster:
12         G_b.add(node + candidate)
13         cluster.add(node)
14         cluster.add(candidate)
15     refactor_graph(G_b)
16     G = G_b
```

Listing 3.20 shows the pseudo-code of the algorithm and Fig. 3.21a shows, for example, the control flow graph G representing a SW application, and $n = 2$ is the target number of clusters (i.e., containers). First, the algorithm cuts all the edge-less nodes (node $v0$ in the example of Fig 3.21a). Since these nodes do not communicate with any other node, they can be placed in any container without affecting the bandwidth. In the following steps, the algorithm iterates as long as the number of vertexes contained in the graph is higher than the target number of containers n (line 3). The number of vertexes in the new graph (i.e., G') decreases as the algorithm clusters the nodes that have higher communication rates together. The number of vertexes is checked after the creation of each cluster to reach the target number of running containers.

Then, the algorithm sorts the vertexes in the graph G by their rank. It starts from the nodes with the lowest rank to avoid grouping everything in the same cluster. As all nodes have the same rank when running the first iteration of the while cycle (i.e., $\forall v \in V(G), v$.rank $= 1$), the algorithm applies a lexicographic ordering, starting from $v1$, proceeding with $v2$, and so on. For each vertex in the graph (line 7), the algorithm selects the vertex connected by the edge with the highest value of δ and it checks if it has already been used in this iteration (line 9). In the example of Fig. 3.21a, node $v1$ is connected to nodes $v3$ and $v4$ with weights 5 and 4 respectively, so the candidate is $n3$. If the candidate has already been used in this iteration to create a cluster, the algorithm selects the next edge with the highest value of δ and the connected node. If no candidate is found, then *candidate* will be empty and the node is added to the new graph without being clustered in this iteration. The function add for the graph G_b will simply skip the empty candidate node. This happens for node $v5$ in the example of Fig 3.21c. Otherwise, the two nodes are clustered together and added as a single vertex (line 12), as for $v1$ and $v3$ in the example.

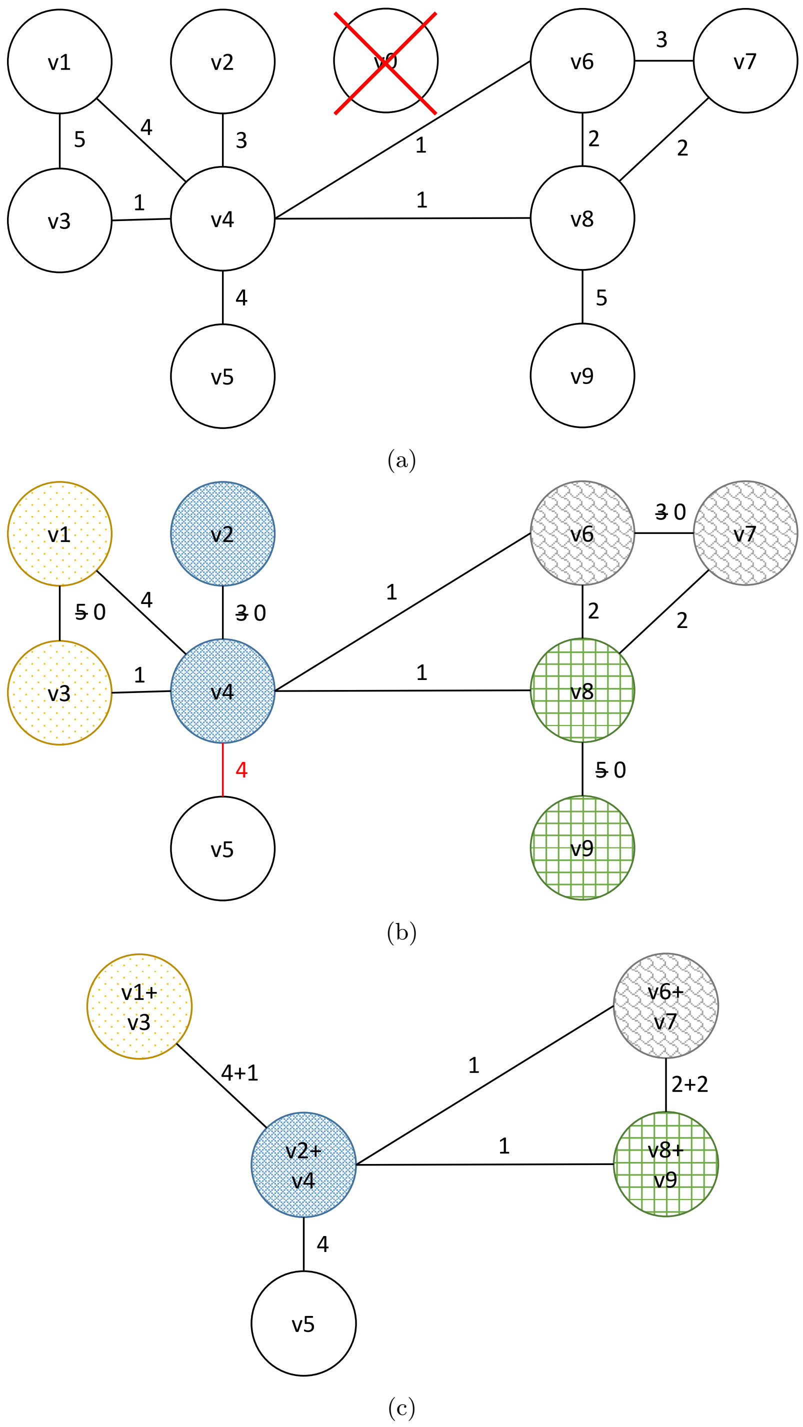


Fig. 3.21: An example graph before and after the first iteration of the clustering algorithm in listing 3.20.

The algorithm updates the clusters with the nodes considered in the iteration (lines 13 and 14). In the example, it repeats these steps for $v2, ..v9$, obtaining the clusters of Fig. 3.21b ($v2$ + $v4$, $v6$ + $v7$, $v8$ + $v9$). Finally, the algorithm refactors the graph by applying definition 3.2.1 (line 15), i.e., by merging edges towards clustered nodes together. Fig. 3.21c shows an example, in which the vertexes actually join to create G' and the weights on the edges going into nodes of the same cluster are summed together. The new graph is written back on the original graph variable to be used as the input graph for the next iteration (line

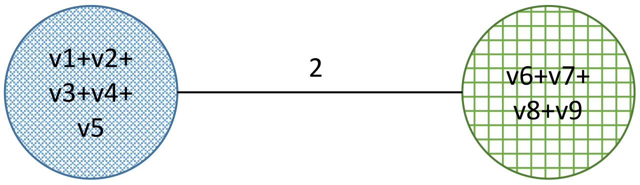


Fig. 3.22: Result of the algorithm in listing 3.20 applied to the graph of figure 3.21, with $n = 2$.

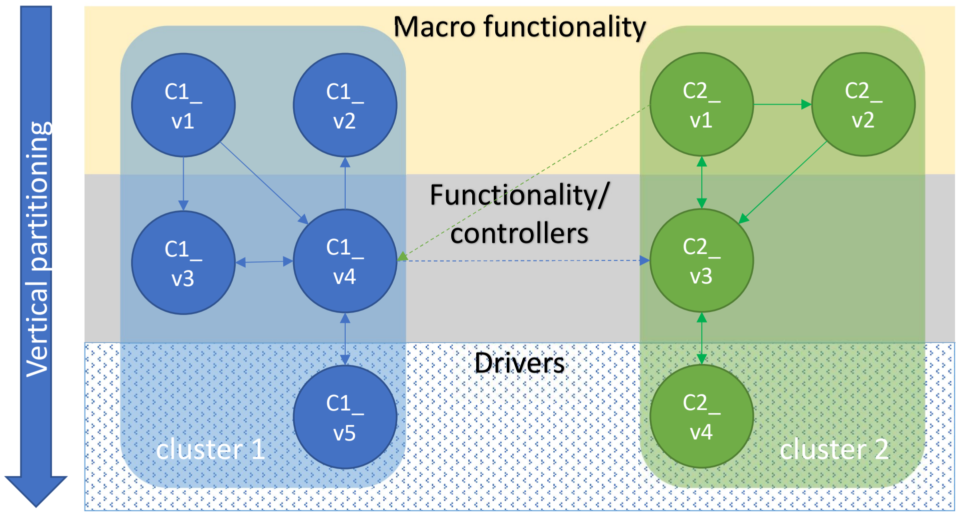


Fig. 3.23: Final result of the clustering on the example graph of figure 3.21, showing the vertical partitioning.

16). If the number of vertexes is higher than the target, the algorithm starts a new iteration (line 3).

The algorithm generates a context based clustering, as shown in the example of Fig. 3.22. We define a *context* as a group of nodes that share one or more functional properties (e.g., arm control and vision are two contexts in a robotic system).

Fig. 3.23 shows the results of the clustering algorithm for the running example. The generated clusters usually form a vertical partitioning (i.e., from macro-functionality to driver nodes) where nodes with the same context are grouped together. It may happen that large amounts of nodes cluster together, making the containerization counter-intuitive as all the nodes would be grouped together and thus defeating the purpose of modularity. In these cases, we use the resources needed by each node as tie-breakers. Through profiling, we identify resource-heavy nodes and split them into separate images as communication will always be impacted no matter which node is split.

Inheritance-based container generation

As containers are independent from each other and the rest of the system, they cannot run properly without installing the required software dependencies inside each image. In ROS-based applications, these can be installed automatically (by the rosdep tool [175]). The problem arises when considering overlapping dependencies between multiple images. These could inflate the total disk space used in a significant manner as several software packages and libraries might get replicated multiple times. To address this problem, we take advantage of the Layered File System (OverlayFS) built into Docker and the principle of inheritance.

Once all nodes are grouped in their respective clusters and all their dependencies are listed, we can perform a check for overlapping dependencies and create a parent image from

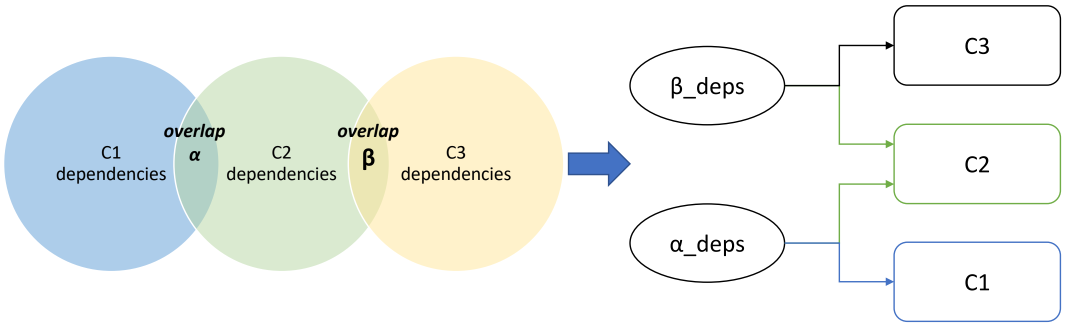


Fig. 3.24: Example of constructing containers with inheritance when there are overlapping dependencies, shown with a Venn diagram. $C2$ is also a multi-stage build.

which the others will inherit the common parts. This is done by gathering all common dependencies between every or some images and by creating parent images containing the overlaps. This procedure is expensive time-wise, but it only has to be done once at the image creation phase. The procedure also applies for images with multiple overlaps. Fig. 3.24 shows the main idea with, as an example, three containers. The procedure generates a Venn Diagram of the dependencies with the intersections as common parts. In the example, $C2$ represents an instance with multiple inheritance (in Docker, this is referred to as a *multi-stage build*).

ROS-based communication between containers

In standard ROS environments, nodes communicate through IP addresses and port numbers, where the IP corresponds to the device IP in the network and ports are assigned randomly. This allows communication and synchronization of ROS tasks to be easily implemented also when they are distributed on different devices of the computing cluster. In contrast, standard containers require the association to private (isolated) subnet IPs [176]. To tackle such a communication issue among *containerized ROS tasks* distributed across different cluster devices, the proposed solution implements the *non-isolation of containers*. All containers are launched with access to the networking interfaces of the host (through the option "hostNetwork: true" under Kubernetes). This eliminates any network overhead introduced by the containers [76]. It also removes the NAT, which is not required in our target applications.

3.2.2 Expanding the Edge-Cloud computing continuum to the RISC-V open hardware architectures

In recent years, there has been considerable interest in open hardware architectures, which allow innovation in processors to take place faster and faster, covering the whole computing continuum spectrum from cloud to edge and low-power systems. RISC-V architectures play a dominant role in this context for both academic and industrial product designs [8, 9].

Unlocking the potential of these architectures in the context of the computing continuum is essential to open the way to future systems entirely based on open and novel hardware. Indeed, the European Commission is pushing the development of HPC, cloud and edge computing systems based on RISC-V [177], which is already gaining momentum in edge and microcontroller architectures. Consequently, it is of utmost importance to concurrently bring the related software ecosystem at the same technology readiness level to support this evolution. A great effort is already happening in this direction, as indicated by the growing software libraries and tools for RISC-V [178]. This effort is still missing regarding software infrastructures and tools for the computing continuum. A first step in this direction requires a preliminary evaluation and profiling of the containerization software on RISC-V processors and platforms to identify possible bottlenecks to be faced in next-generation architectures. However, RISC-V processors are not yet present in commercial distributed computing platforms. Consequently, no container software version has yet been ported or profiled on these systems.

This section aims to fill this gap by porting and profiling an orchestration platform (Kubedge-V) to a RISC-V cluster prototype based on SiFive processors. The cluster, called Monte Cimone, was designed to open the way to the future edge computing systems based on RISC-V [179]. We identified the minimum components to support the basic features of container orchestration, the required network plugins (e.g., EdgeMesh), and the container runtime (e.g., runc with CRI-O) to be employed for the target RISC-V architecture.

In its current implementation, Monte Cimone performance per Watt lies in the class of high performance edge computing platforms. For this reason, we considered a reference architecture made of a Kubernetes node running on a high-performance server (with an Intel Xeon processor) and KubeEdge-V running on Monte Cimone. This setting allowed us to profile the execution of KubeEdge-V on RISC-V to characterize its overheads on different benchmark suites (Phoronix, OSBench, IPC-Benchmark and stress-ng). In addition, we show the overhead introduced by the KubeEdge-V system on both execution time and memory usage. Using a set of benchmarks and scaling the number of used containers, we show how the system performs under different load levels, finally comparing it with an equivalent system based on an ARM architecture featuring the same power envelope. The results show that the solution is feasible and that the performance degradation caused by containerisation on RISC-V systems is comparable to the performance degradation observed on the ARM system. We also identify which operations are mostly impacted by the container runtime and should be considered for future software or hardware optimizations.

The KubeEdge-V platform

We deployed the ported orchestration platform, KubeEdge-V, on a prototype RISC-V system. The prototype system is a computer cluster called "Monte Cimone". Monte Cimone nodes have been designed to explore RISC-V as the core for future edge nodes, as will be summarized in the following. Currently, the computational power delivered by these nodes makes them suitable for edge applications. For this reason, the reference system we consider is made of an x86 as the cloud node and RISC-V as the edge node, where we ported and profiled the KubeEdge platform.

Kubernetes for RISC-V. Fig. 3.25 shows a summary of the Kubernetes architecture, where each Kubernetes component is highlighted in blue, pods are dark red, and containers are green.

On the RISC-V edge nodes, we do not use Kubernetes but a lightweight, edge-oriented solution: KubeEdge. KubeEdge is an open-source system for extending native containerized application orchestration capabilities to hosts at the edge. It is built upon Kubernetes and provides fundamental infrastructure support for network, application deployment and metadata synchronization between cloud and edge [180]. Fig.3.25 shows the KubeEdge components in orange. KubeEdge is composed of two main components, the CloudCore, which is installed through a Kubernetes container on the master node, and the EdgeCore, which is a process that acts as a kubelet on the edge nodes. Unlike Kubernetes, KubeEdge can adapt to mesh networks and take into account nodes that momentarily become unreachable due to harsh conditions, e.g., remote sensors or drones. To achieve offline computing, KubeEdge maintains a copy of the cluster status locally on the edge device to then synchronize it with the master once connectivity is restored.

Fig. 3.26 shows a summary of the software stack required to run containers and KubeEdge on RISC-V. Container deployment requests are made through Kubernetes, which are then synchronized by the CloudCore to the appropriate EdgeCore, passing information through the mesh network. The EdgeCore on the target device, chosen by the Kubernetes scheduler, receives the deployment and requests a new container to CRI-O (Container Runtime Interface for the Open Container Initiative). If the container image is not already available on the device, CRI-O downloads it. CRI-O then starts *runc* (Container Runtime) with the requested parameters and network plugin (i.e., EdgeMesh). Runc starts the processes inside

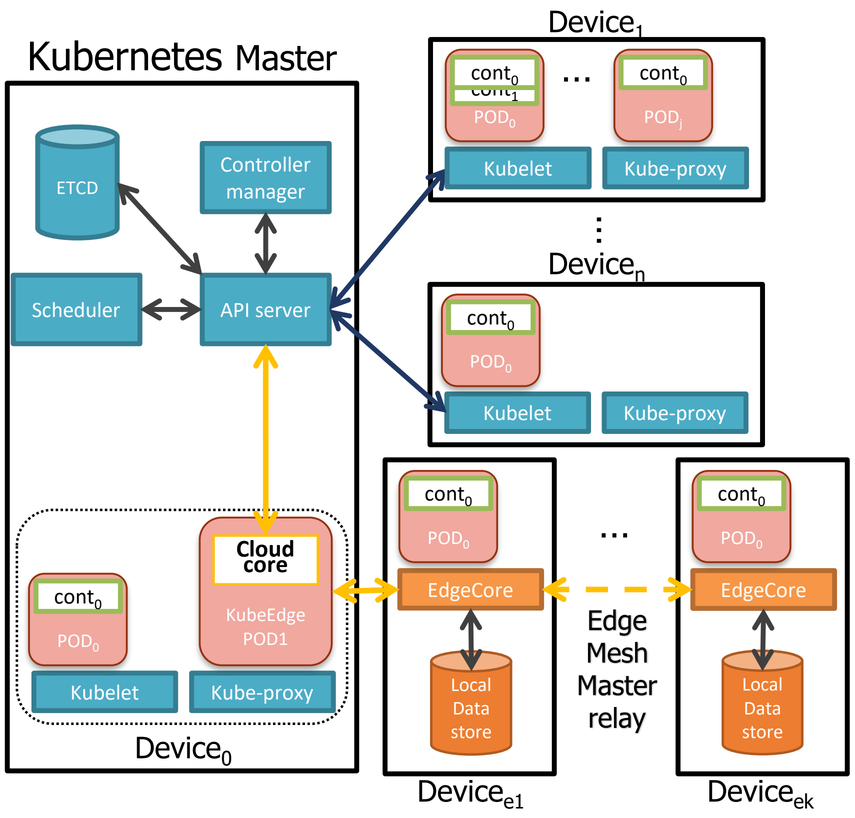


Fig. 3.25: Kubernetes and KubeEdge architecture.

the container and assigns them to the appropriate namespace and cgroup through Linux system calls.

We ported and re-compiled the whole environment for RISC-V, including the container images used to run Kubernetes. A detailed guide is also available on Gitlab[4]. The installation process is summarized in the following.

Installing KubeEdge on RISC-V. An up-to-date version of the Go language compiler is a main requirement to install the complete software stack. An older version of Golang can be obtained from the standard Ubuntu repository to bootstrap the latest. The installation process begins with runc and CRI-O. Configuration of CRI-O requires special attention, specifically adjusting the default container registry to a custom one, which is a requirement to use a custom pause image for RISC-V. The pause container is crucial for Linux to configure namespaces and cgroups before starting the actual container. However, since the pause image does not exist for RISC-V, it needs to be created and hosted on a custom registry. To achieve this, we extracted the Dockerfile from Kubernetes' pause and compiled it for RISC-V. Next, we require the EdgeMesh network plugin, which enables the communication between edge devices through a mesh network while relaying Kubernetes master data. The plugin requires a container running on each device for data handling and communication. Hence, we manually compiled the EdgeMesh executable for both x86 and RISC-V, to create a single multi-architecture container. To support RISC-V, modifications were made to one of the libraries used by EdgeMesh. Detailed information is available on the Gitlab page. Finally, we compiled the KubeEdge executables, created the container images, and installed them. These operations involved extracting Dockerfiles from the original KubeEdge, as the original containers rely on special "builder" containers not yet available for RISC-V. With these steps completed, KubeEdge can be successfully installed and used.

[4] https://gitlab.com/parco-lab/kubeedge-v

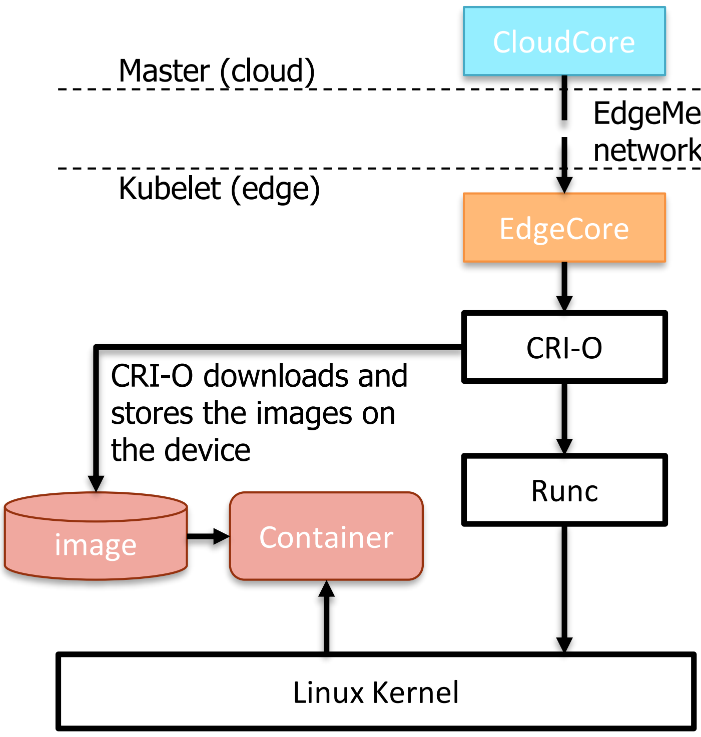


Fig. 3.26: Software stack for KubeEdge on RISC-V.

3.3 Re-configurability of software for Edge-Cloud computing continuum

Stringent accuracy standards for robotic software are imperative due to the frequent involvement of robots in safety-critical tasks. In addition to functional constraints, these standards encompass supplementary extra-functional constraints such as QoS, reliability, scalability, and energy efficiency [181]. Fulfilling these constraints necessitates the set up of robotic application software to function across *heterogeneous* computing architectures, which entails the allocation of software components across Edge-Cloud computing clusters [5].

Within this *multi-container* context, a significant challenge involves maintaining uninterrupted robot functionality despite disruptions. Consequently, there is a growing interest within robotics companies in embracing *orchestration* platforms for software deployment [15, 16].

This rises a new big challenge, as state-of-the-art orchestrators such as Kubernetes rely on "scheduling" policies designed to enhance system-level efficiency (e.g., load balancing, optimizing data transfer, and overall system energy efficiency) to allocate containers across the Edge-Cloud computing nodes (referred to as "nodes" hereafter). Due to their inherent focus on automating software deployment, these orchestrators do not support global synchronization of containers, which would require source code modifications. As a result, this limitation hinders the implementation of scheduling algorithms aimed at QoS constraints, such as minimizing the application network usage or makespan.

To optimize these two very important aspects of the Edge-Cloud computing architecture, this section makes the following contributions:

- Section 3.3.1: a super-clustering algorithm that extends the Kubernetes scheduler to perform efficient mapping of the containers into the cluster of computing devices. This enhances the clustering of ROS nodes into containers with the aim of reducing the network bandwidth utilization.

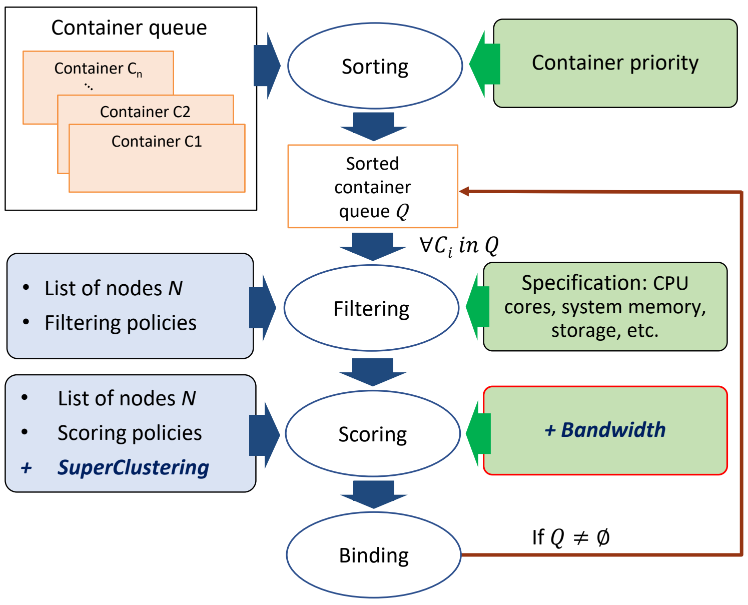


Fig. 3.27: K3S proposed extension for mapping of nodes based on ROS communication.

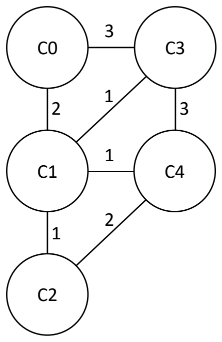
- Section 3.3.2: a new scheduling approach for Kubernetes called HEFT4K that, starting from the HEFT task ranking, takes advantage of the OS niceness concept to reduce priority inversions and preemptions of tasks to improve the application makespan.

3.3.1 Improving the Kubernetes schedule's efficiency for ROS-based applications: Network

Fig. 3.27 depicts an overview of the proposed orchestration platform with support for *super-clustering*. The orchestration begins with the queue of ROS nodes clustered into containers, and the list of computing nodes representing the computing platform (i.e., the *cluster* of computing devices). In this section, nodes are computing nodes of the cluster, i.e., devices. The instances where nodes refer to ROS nodes will be specified. A set of specifications is provided for each container (e.g., memory, CPU, storage requirements). The platform implements four main steps. First, *sorting* of containers using a hierarchical scheduling policy to generate a priority queue ordered by a priority class. The priority class allows the scheduler to determine which containers have to be scheduled first. This value is user-defined depending on the container requirements, and is set by the cluster administrator during the deployment phase. Then, for each container in the queue, the *filtering* phase selects the nodes of the cluster (i.e., the devices) that satisfy the container specifications, such as CPU cores required and available, system memory, and storage. The *scoring* phase creates a node ranking by considering user-defined policies, and schedules the container on the highest ranking node. An example of scoring policy is node affinity, where a preference for specific nodes is given, thus giving them higher ranking and favouring them in the scheduling. Finally, the *binding* phase maps the container to the node by allocating resources for the container.

The scoring also calculates the ranking for each node-container pair, depending on the bandwidth. We use an extended version of Eq. (3.12) where, instead of using ROS nodes, we use the clusters obtained from Algorithm 3.20 to calculate the bandwidth between containers. This allows us to compute the communication bandwidth between the new container and those that are already deployed on the node. We define the rank for the deployment of container λ on node n as follows:

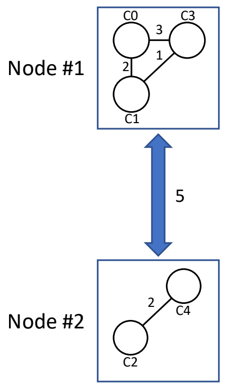
$$rank_{n,\lambda} = \sum_{c \in n} \text{bandwidth}(\lambda, c) \tag{3.14}$$



(a) Example graph representing communication between clusters of ROS nodes (i.e., containers)

Container	Priority
C_0	3
C_1	2
C_2	1
C_3	0
C_4	4

(b) Priorities that establish the deployment order for the containers.



(c) Example mapping created by equation (3.15) for the graph of Fig. 3.28a and two nodes.

Fig. 3.28: Example of Super Clustering.

where bandwidth is the extended definition of Eq. (3.12) and $c \in n$ are all the containers already deployed on node n. This formulation gives a higher ranking to those nodes where the bandwidth is higher. As a consequence, if container λ were to be deployed on node n, it would lead to that amount of saved bandwidth.

As other scoring policies might be active at the same time, we normalize the value such that $rank_n \in [0, 100]$. We achieve so with the normalized rank $rank\_norm$, obtained as follows:

$$rank\_norm_{n,\lambda} = \frac{rank_{n,\lambda}}{\max_{n\in N}(rank_{n,\lambda})} * 100 \tag{3.15}$$

where N is the list of all computing nodes in the Kubernetes cluster.

Fig. 3.28a shows an example graph. Each node in the graph represents a cluster of ROS nodes containerized together (i.e., containers). The edges represent communication between containers and their weights represent the involved bandwidth. There are five containers, C_0, C_1, C_2, C_3 and C_4, that need to be deployed onto two compute nodes with the super-clustering algorithm (Fig. 3.28c). The five containers have different priorities as listed in Table 3.28b. The containers are sorted by their priority, deploying the highest priority container first. Container C_4 has the highest priority and is deployed first on either node #1 or #2, since both are empty. For this example, we used node #1. Then, the containers are scheduled with the order C_0, C_1, C_2, C_3. To decide on which of the two nodes they should be deployed on, we compute the rank with Eq. (3.14). C_0 is calculated as follows:

$$C_0 = \left\{ \begin{array}{lcl} rank_{\#1,C_0} & = & \sum_{c\in\#1} bandwidth(C_0, c) = 0 \\ rank_{\#2,C_0} & = & \sum_{c\in\#2} bandwidth(C_0, c) = 0 \end{array} \right\} \rightarrow \begin{array}{c} \text{node } \#1 \\ \text{(free node policy)} \end{array}$$

As container C_0 has zero bandwidth towards both nodes, the one with more free resources is taken (node #1 in the example).

The rank for all subsequent containers is calculated as follows:

$$C_1 = \left\{ \begin{array}{lcl} rank_{\#1,C_1} & = & \sum_{c\in\#1} bandwidth(C_1, c) = 2 \\ rank_{\#2,C_1} & = & \sum_{c\in\#2} bandwidth(C_1, c) = 1 \end{array} \right\} \rightarrow \text{node } \#1$$

$$C_2 = \left\{ \begin{array}{lll} rank_{\#1,C_2} & = & \sum_{c\in\#1} bandwidth(C_2,c) = 1 \\ rank_{\#2,C_2} & = & \sum_{c\in\#2} bandwidth(C_2,c) = 2 \end{array} \right\} \quad \rightarrow \quad \text{node } \#2$$

$$C_3 = \left\{ \begin{array}{lll} rank_{\#1,C_3} & = & \sum_{c\in\#1} bandwidth(C_3,c) = 4 \\ rank_{\#2,C_3} & = & \sum_{c\in\#2} bandwidth(C_3,c) = 3 \end{array} \right\} \quad \rightarrow \quad \text{node } \#1$$

This example takes into consideration priority as well as bandwidth as the algorithm tends to cluster containers together as much as possible. If the bandwidth is the only metric considered, the best solution is to deploy all containers onto the same computing node, as that is the best possible solution. In our experimental results, other constraints such as CPU usage in time or cores, system memory minimum and maximum usage, storage available and network topology are considered for the deployment.

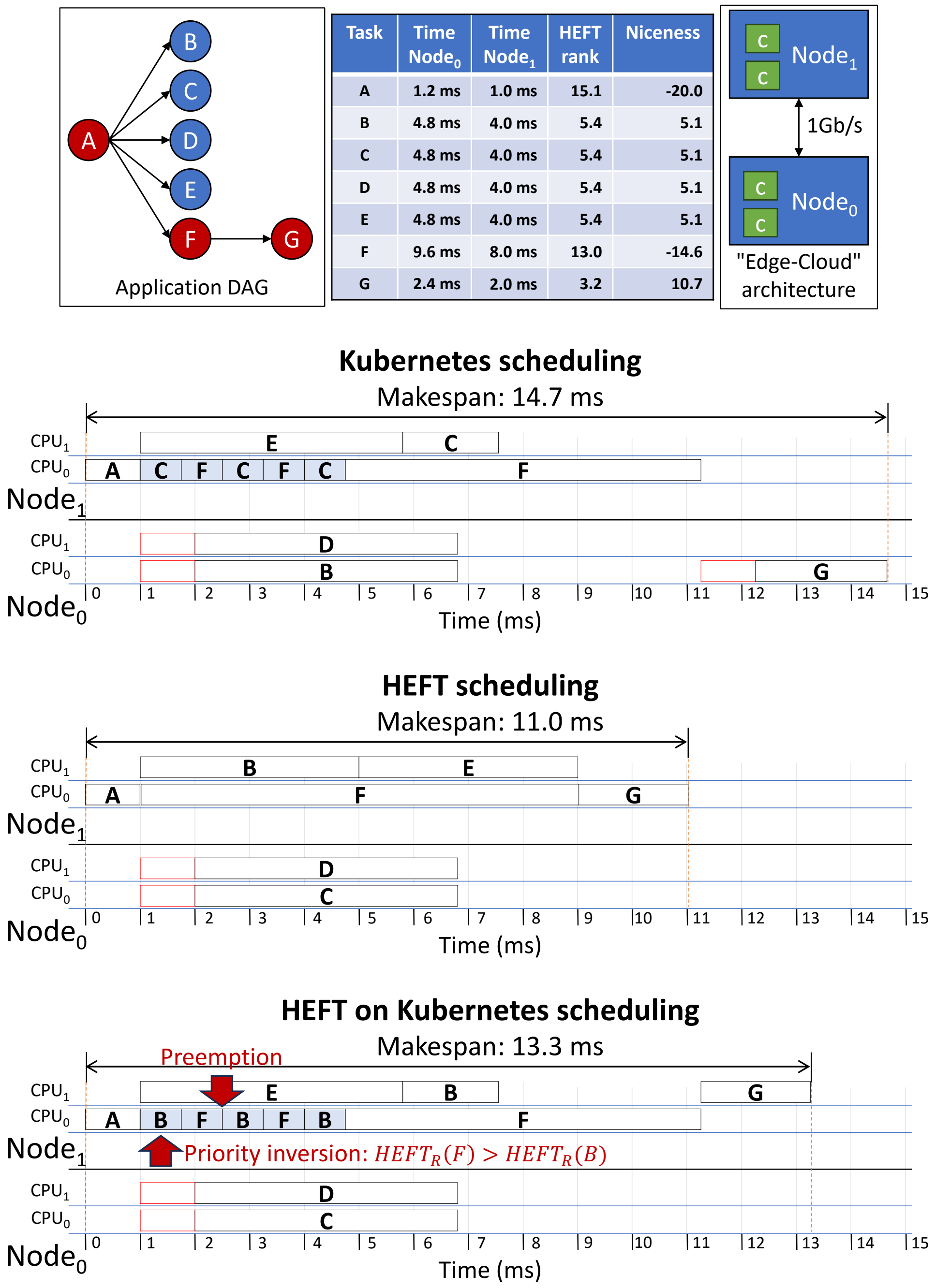
3.3.2 Improving the Kubernetes schedule's efficiency for ROS-based applications: Makespan

In the context of robotic applications, response times and application runtime are two important factors that contribute to the safety and reliability of a robotic system. Modern robotic software based on publish/subscribe paradigms (e.g., ROS), can often be represented as a DAG. This allows to apply more advanced scheduling techniques compared to those currently available in orchestration SW, such as Kubernetes.

As an example, Fig. 3.29 shows the outcome of the Kubernetes scheduling applied to an application represented as a DAG. This application runs on a heterogeneous (distributed) cluster consisting of two nodes, each equipped with two CPUs. Each graph node represents a containerized task, while the graph edges represent the temporal dependencies between tasks. In the optimal scenario, task communication relies on a standard API such as ROS or any other *publish-subscribe* paradigm. This approach ensures that temporal task-to-task dependencies are implicitly met. However, the lack of global synchronization leads to priority inversions between containers in the critical path (A, F, G) and other tasks. Furthermore, in such distributed architectures, factors like preemption mechanisms (shown in blue time slots in Fig. 3.29), which often cannot be disabled, and data transfers (red slots), impact the application's makespan.

To overcome this limitation, there is a need to reduce the makespan without centralized or distributed synchronization of containers. As a starting point, we used the HEFT algorithm [33], which is one of the most efficient mapping and synchronization algorithms in the current state-of-the-art for makespan optimization. HEFT typically demands centralized synchronization of tasks (containers), making it incompatible with Kubernetes. We take advantage only of the definition of the priority ranking of tasks and the mapping topology, while we discard synchronization information like the starting time of tasks. Fig. 3.29 (*HEFT on Kubernetes scheduling*) shows the outcome of implementing this approach in Kubernetes in the example and provides a comparison of the resulting makespan (13.3 ms) with the theoretically ideal scheduling according to HEFT (*HEFT scheduling*, 11 ms). While the ranking-based mapping method improves upon the initial Kubernetes scheduling (14.7 ms), it still struggles with the issues of priority inversion and preemption during container execution.

To improve the makespan, we propose a solution to adjust container priorities through their *niceness* during the deployment phase. Additionally, to align with the orchestrator's automated deployment, scaling, and recovery capabilities, our approach relies on event-driven re-scheduling. It starts with an initial offline mapping, and subsequent re-mappings are triggered as needed, such as in the event of a node failure. The re-mappings target the minimal subset of containers to ensure uninterrupted robot functionality to the greatest extent possible.



Task	Time Node$_0$	Time Node$_1$	HEFT rank	Niceness
A	1.2 ms	1.0 ms	15.1	-20.0
B	4.8 ms	4.0 ms	5.4	5.1
C	4.8 ms	4.0 ms	5.4	5.1
D	4.8 ms	4.0 ms	5.4	5.1
E	4.8 ms	4.0 ms	5.4	5.1
F	9.6 ms	8.0 ms	13.0	-14.6
G	2.4 ms	2.0 ms	3.2	10.7

Fig. 3.29: Application example represented by a DAG and the corresponding makespan achieved with native Kubernetes, (ideal) HEFT, and HEFT on Kubernetes schedulers. Red boxes represent data transfer, blue boxes represent preemption.

The HEFT4K scheduling

Let $G^c = (V^c, E^c)$ be a connected but not necessarily complete graph representing the distributed Edge-Cloud computing architecture. Each vertex $v_i^c \in V^c$ is a computing node and each edge $e_{j,k}^c \in E^c$ is the communication link between v_j^c and v_k^c. Each edge has a bandwidth value associated with $l(e_{j,k}^c)$.

We define $G^a = (V^a, E^a)$ as the DAG representing the software application a, where each vertex $v_i^a \in V^a$ is a containerized task to be scheduled on a distributed computing architecture. Each edge $e_{j,k}^a \in E^a$ is a temporal dependency between tasks (v_j^a, v_k^a). Each task has a corresponding *computational cost* $t(v^a_{i,j})$ for each node $v^c_j \in V^c$ (ms). Each edge has a corresponding *communication cost* $d(e^a_{j,k})$. The communication value is represented in Mbits. Combined with the network bandwidth, it allows to calculate the actual communication costs after the task-to-node mapping. The goal is to find a mapping of V^a in V^c that achieves the best makespan without the need for global synchronization among tasks.

Starting from the execution time of each task v^a_i in each node v^c_j and the communication latency of each link $e^c_{j,k}$ (which we obtain with the open shortest path first routing algorithm), we apply the HEFT algorithm (look ahead version [94]). The algorithm generates a ranking of the tasks. By following the rank list, it calculates, for each task, the task-to-node mapping, the start and the finish times of the task in the selected node. The mapping and the scheduling times guarantee that the execution of tasks follows the priority order of the generated rank (*condition 1*), and the temporal constraints between tasks (i.e., the execution order of the DAG nodes in each path) are satisfied (*condition 2*).

Nevertheless, due to the lack of global synchronization, the Kubernetes scheduling cannot apply the start and finish times of tasks on the distributed nodes. Although adopting the HEFT mapping in Kubernetes without task synchronization achieves makespan improvements over the standard Kubernetes scheduling (i.e., 17.7% on average), it suffers from priority inversion and preemption of tasks, which often lead to makespan increase. In addition, HEFT is a static scheduler, which requires re-execution at each deployment event of Kubernetes. It requires the shut down and re-start of the whole container set, which involves unsustainable interruptions of the robot functionality.

We propose *HEFT4K*, which relies on three points. First, we assume that the robotic software application implements task communication through a standard API, such as ROS or any other publish-subscribe paradigm. This guarantees that condition 2 is satisfied. At the deployment time, we associate a *niceness* value to each task for the preemption rules implemented by the target operating system. Although condition 1 cannot be guaranteed, the niceness values force the priority of the tasks by considering the HEFT ranking. Finally, it implements an event-driven remapping approach to deploy tasks after node failure without requiring the application deployment from scratch.

HEFT ranking-based priority through task niceness

In the Linux kernel, the standard scheduler relies on a dynamic priority-based scheduling algorithm. Initially, each process is assigned a base priority. This base priority increases each time the process remains idle or waits in the ready queue. The operating system allows adjustment of the base priority through the niceness value, which can elevate a process's base priority, resulting in more frequent scheduling. This scheduler ensures that every process receives a fair share of CPU time and prevents any process from becoming starved. Eventually, even a process with the lowest base priority will see its dynamic priority rise, making it the next to be executed. Once a process is scheduled to run, its dynamic priority is reset to its base value. It is important to note that, in this system, negative values indicate higher priorities compared to positive ones.

We leverage this dynamic priority system to assign a niceness value which depends on the HEFT rank of each task. The HEFT rank represents the task scheduling order computed by HEFT within each device's scheduler. This approach effectively mitigates priority inversions and drastically reduces preemptions, addressing the challenge of limited control over a process start and finish times.

HEFT4K calculates the task niceness as follows:

$$niceness(v^a_i) = \frac{(nice_H - nice_L) * \mathrm{HEFT}_R(v^a_i)}{max(\mathrm{HEFT}_R(G^a))} + nice_L \tag{3.16}$$

where $nice_H$ and $nice_L$ denote the upper and lower bounds for the niceness values to be assigned to each task. HEFT_R provides the HEFT rank for the given task.

Fig. 3.30 shows the resulting scheduling for the running example. Compared to HEFT on Kubernetes in Fig. 3.29, we observe the elimination of priority inversions and preemptions along the critical path, particularly with task *f*. This leads to a notable enhancement of

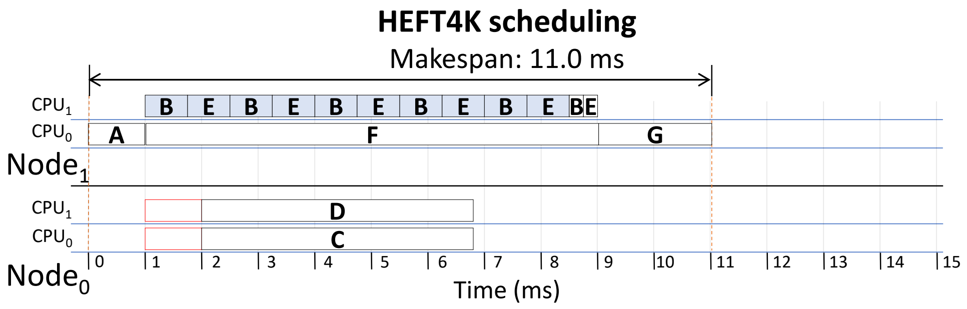


Fig. 3.30: Scheduling the example of Fig. 3.29 with HEFT4K.

approximately 20.9% in the makespan. In comparison to the scheduling implemented by Kubernetes, the speedup is 33.6%.

Event-based remapping

To complement the orchestrator functionality, we also account for scenarios where a computing node becomes unavailable and the tasks running on that node necessitate an online re-deployment (i.e., restart of the tasks on a different node without restarting the whole system). This operation can lead to suboptimal remapping and extended makespan as standard orchestrators like Kubernetes implement such a re-deployment of containers targeting workload balancing instead of makespan optimization. To address this issue, HEFT4K incorporates the *partial* remapping of tasks. This approach specifically remaps only the tasks that were lost onto operational nodes, with a focus on optimizing makespan to minimize the impact of node failures.

Conversely, there are scenarios where partial re-mapping might result in makespan increases, and a complete task re-mapping from the beginning, along with a corresponding temporary service interruption, would be more acceptable. HEFT4K employs a *threshold* mechanism to determine whether to choose partial or total remapping. If the cumulative HEFT ranks of the rescheduled tasks exceeds a certain percentage of the total sum of HEFT ranks for the entire DAG, the decision is made to reschedule the entire application. The idea is that, given that the HEFT rank encompasses task computation time, communication time, and priority, exceeding the HEFT rank threshold implies that either numerous less critical tasks or a few highly crucial ones were assigned to the failed node. In either scenario, this indicates a significant deterioration in the current mapping.

The threshold value is user-defined and system-dependent. We developed a benchmark that, starting from a given δ-value representing the maximum increase of makespan and the architecture characteristics (node compute capability, communication, etc.), it extrapolates the threshold for any DAG size. The benchmark iterates over the threshold values until the δ is reached, giving the best compromise between rescheduling frequency and performance.

Fig. 3.31, for example, depicts the results for $\delta = 10\%$ in our experimental setup, i.e., the performance difference between a re-mapping created from scratch and a partial re-mapping for different DAG sizes, when a node in the cluster is lost. We found the corresponding near-optimal value of the threshold at ≈20%. Green denotes the DAGs that lost less than 20% of the total HEFT rank of the application, while blue denotes applications that lost more than 20%.

Algorithm 2 outlines the complete re-mapping procedure. First, it identifies all tasks affected by an offline node (line 1). It checks if a critical task has been impacted or if the threshold has been surpassed (lines 3-6). If either condition is met, it performs a complete re-mapping of the DAG. Otherwise, a partial re-mapping is performed, and the list of tasks is sorted according to their HEFT ranks (lines 8-9). For each task in the graph, if it is not one of the affected tasks, and for each node in the Edge-Cloud cluster that differs from the node on which the current task was originally scheduled, it modifies the CPU time to the maximum. This adjustment compels the HEFT algorithm to remap tasks that were not

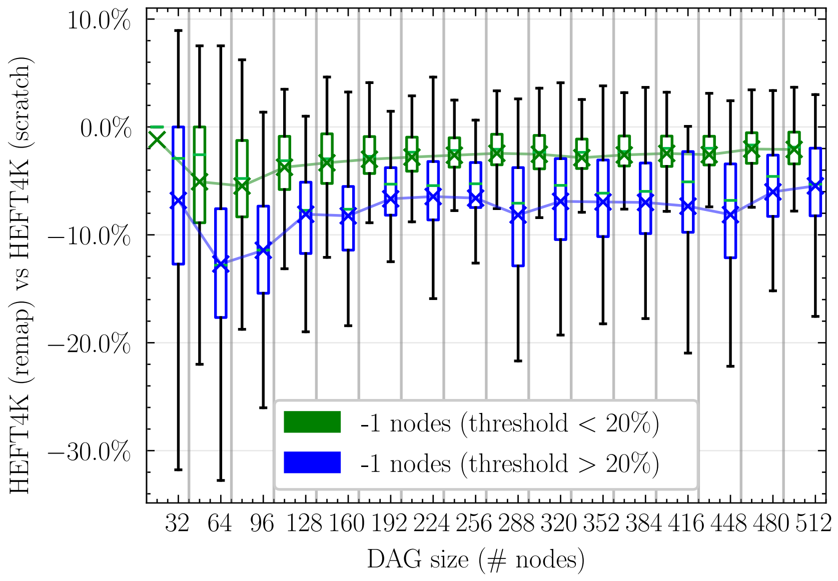


Fig. 3.31: Performance loss when the sum of the HEFT rank lost due to a node failure is higher than 20%.

affected on the same node as before, while dropped tasks are mapped around locked tasks in the most optimal manner possible (line 18). The algorithm computes the niceness value according to Eq. (3.16) (line 19). Finally, it restores all modified runtimes in G^a (line 20).

This algorithm allows for improved rescheduling efficiency and is also efficient itself. The overall complexity is the same as HEFT because the HEFT rank sorting takes $O(|V^a| \times \log |V^a|)$, the CPU time modification phase added to HEFT takes $O(|V^a| \times |V^c|)$ and, the niceness computation phase takes $O(|V^a|)$. Thus, the overall complexity remains $O(|V^a|^2 \times |V^c|)$ (from [33]).

Finally, HEFT4K implements the complete re-deployment also when a node failure involves a *critical task*, i.e., a task whose offline affects the entire application and causes the system downtime.

3.4 RT-Kube: real-time Kubernetes in the Edge-Cloud continuum

The standard Kubernetes architecture does not support any notion of RT containers, as it does not have the data structures or the modules required to handle the additional requirements of such containerized RT tasks[5]. This means that these RT tasks will be treated equally, even though their requirements are different.

This section introduces a platform for container orchestration onto edge-cloud architectures for mixed-criticality systems based on off-the-shelf technologies (i.e., Linux OS + Preempt-RT, Kubernetes-K3S). The platform is ROS- and Kubernetes-compliant and does not require any custom software patch to be used; thus, it is supported in provisioned installation of Kubernetes, as well as normal environments. It supports per-node WCET, different levels of criticality (e.g., A, B, and C for RT containers), real-time monitoring of all resources and overrun deadlines, and stateless migration of ROS tasks based on customizable policies. This framework allows for better meeting the functional and extra-functional constraints of advanced multi-domain software which are typical of autonomous mobile robots.

Fig. 3.32 presents an overview of the proposed extended architecture. RT-Kube evolves the standard platform with the following components (highlighted in green in Fig. 3.32).

[5] For the sake of clarity, this section will refer to ROS nodes as “software tasks” (or simply “tasks”), and real-time ROS nodes as “RT tasks”. We will use “computing nodes” or “nodes” to refer to the physical devices within the Edge-Cloud computing cluster.

Algorithm 2: Rescheduling algorithm in case of nodes that go offline.

```
input  : The graphs $G^a$ and $G^c$
output: The container-to-node-mapping.
Function reschedule ($G^a$, $G^c$):
    $OT \leftarrow get\_offline\_tasks()$
    if $\exists v_i^a \in OT \,|\, v_i^a$ is crit. or $\sum_{i \in OT} HEFT_R(v_i^a) > threshold$ then
        for $v_i^a \in G^a$ do
            sched $\leftarrow$ HEFT$_{sim}$ ($v_i^a$, $G^c$)
        end
    else
        $G^a \leftarrow$ sort($G^a$, HEFT$_R$)
        $G^a_{copy} \leftarrow G^a$
        for $v_i^a \in G^a$ do
            if $v_i^a \notin OT$ then
                for $v_j^c \in G^c$ do
                    if $v_j^c \neq n(v_i^a)$ then
                        $t(v_{i,j}^a) = 9999999$
                    end
                end
            end
            sched $\leftarrow$ HEFT$_{sim}$ ($v_i^a$, $G^c$)
            $optimize\_niceness(v_i^a)$
            $G^a \leftarrow G^a_{copy}$
        end
    end
return sched
```

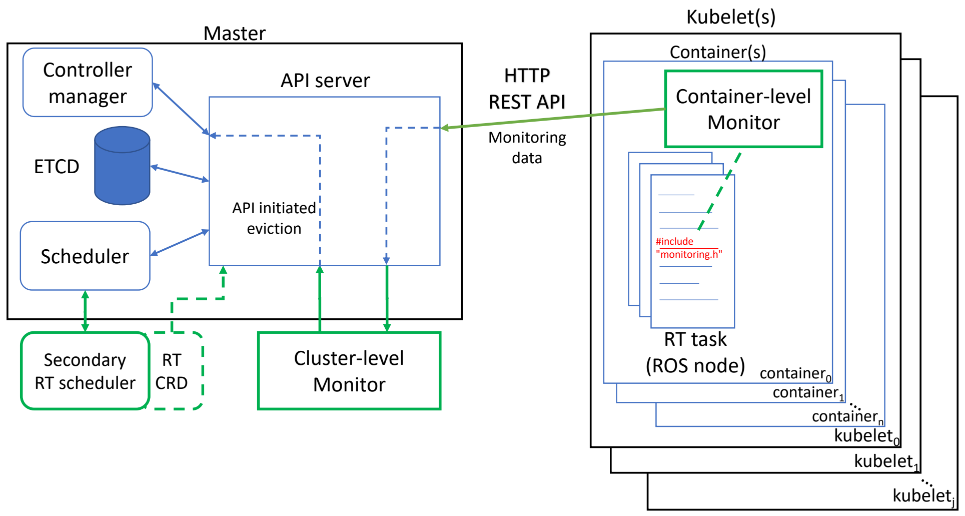


Fig. 3.32: The RT-Kube overview.

1. *A Custom Resource Definition (CRD) for RT containers (RT CRD).* It allows for the specification of RT parameters, such as deadline, period, WCET, and criticality level of containers.
2. *A secondary RT scheduler.* It implements a scheduler with RT plugins that uses the additional data (RT CRD) to perform the schedulability test of RT containers.
3. *Container-level and cluster-level monitors of RT tasks.* These implement the monitoring of RT containers at two levels. At the container-level, the monitors collect information at runtime about temporal violations (i.e., overrun WCET and missed deadlines). At the cluster-level, one main monitor combines such information to the cluster status to implement container migration and recover from temporal violations.

```
1   apiVersion: rt.scheduling/v1
2   kind: RealTime
3   metadata:
4     name: example-realtime-data
5   spec:
6     criticality: C
7     rtPeriod: 100
8     rtDeadline: 100
9     rtWcets:
10    - node: nodeA
11      rtWcet: 50
12    - node: nodeB
13      rtWcet: 60
14  ---
15  apiVersion: apps/v1
16  kind: Deployment
17  metadata:
18    name: nginx-deployment
19    labels:
20      app: nginx
21  spec:
22    replicas: 1
23    selector:
24      matchLabels:
25        app: nginx
26    template:
27      metadata:
28        labels:
29          app: nginx
30          rt-scheduling: example-realtime-data
31      spec:
32        schedulerName: RT-scheduler
33        containers:
34        - name: nginx
35          image: nginx:1.14.2
36          ports:
37          - containerPort: 80
38        tolerations:
39        - key: "RealTime"
40          operator: "Equal"
41          value: "RT"
42          effect: "NoSchedule"
```

Listing 3.33: Example of RT CRD (lines 1-13) and the corresponding Kubernetes configuration module for the container deployment configured with the real-time parameters.

We also implemented the non-isolation of containers to support communication among containerized ROS tasks. See Section 3.2.1.

The CRD module for RT containers

In standard orchestration platforms, the user provides a set of specifications for each container (e.g., memory, CPU, storage requirements). To schedule RT containers, we consider four additional specifications: criticality level, deadline, period, and the WCET of the corresponding containerized RT task. The platform takes advantage of this information to calculate the utilization of all containers in each node, and assesses the impact of a new RT container deployment on the system performance.

Figure 3.33 shows an example of CRD module (lines 1-13) with the extended specifications for the deployment of the Kubernetes *nginx* use case [182]. The first 13 lines create

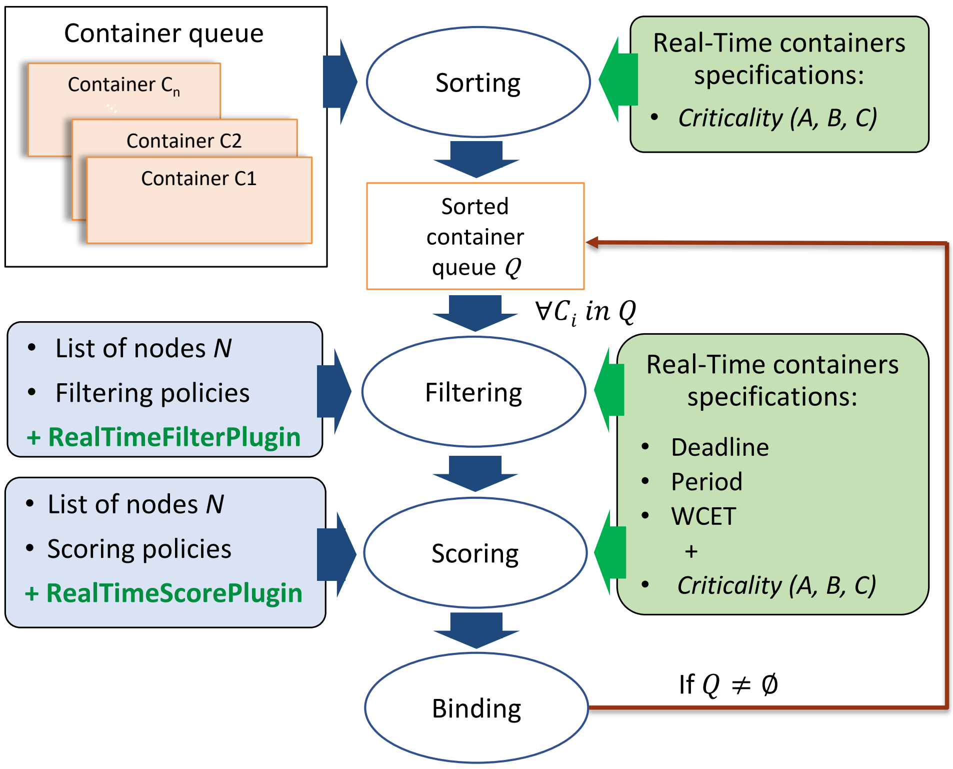


Fig. 3.34: The secondary RT-scheduler.

the CRD object “`example-realtime-data`”, where lines 6 to 13 contain the RT CRD with the four additional values. *Criticality (A, B, C, and D)* is used for the new sort and score phases, as well as in the monitoring (see Fig. 3.34). We borrowed such a criticality classification from the automotive safety integrity levels (ASIL) standard, with C being the highest the platform supports as of now. Note that it would be possible to support ASIL-D tasks, but it would require a compute platform that honours ASIL-D requirements as well. Further, criticality standards from other domains could also be adopted in a similar way.

The platform uses the deadline, period, and WCET information for the sorting, filtering, and scoring phases.

The CRD object is linked to the *deployment* object (*Kind* fields, lines 2 and 16) through a label that matches the RT specification at lines 4 and 30. These labels are compared at scheduling time for each deployment to find the matching RT CRD.

The secondary RT scheduler

Fig. 3.34 shows an overview of the secondary RT-scheduler. The orchestration begins with the queue of containers (i.e., standard and RT), and the list of nodes representing the compute platforms (i.e., the *cluster* of computing devices).

We assume that each real-time ROS task is mapped to one RT container. Our evaluation shows that this one-to-one configuration, when compared to other solutions, is the most flexible as it incurs negligible overhead and experiences minimal performance penalties for RT tasks. The scheduler implements the following four steps. First, *sorting* of containers using a hierarchical scheduling policy to generate a priority queue ordered by criticality. Then, for each container in the queue, the *filtering* phase selects the nodes of the cluster that satisfy the container specifications. The *scoring* phase creates a node ranking by considering user-defined policies, and schedules the container on the highest ranking node. Finally, the *binding* phase maps the container to the cluster node by allocating resources on the identified node for the container. To deploy RT containers, the platform relies on the *taint* and *tolerations* features of Kubernetes [182] to identify which nodes of the cluster run a

Algorithm 3: Real-time filtering extension for the Kubernetes scheduler.

```
input  : A list of nodes $N$, an RT container $x$
output: A list of schedulable nodes $M$
1  Function RealTimeFilterPlugin ($N$, $x$):
2      $M \leftarrow N$
3      for $\forall n \in N$ do
4          $\mathsf{currentUT}_n \leftarrow \sum_{c \in n} \frac{\mathrm{WCET}_c}{P_c}$
5          $\mathsf{newUT}_n \leftarrow \mathsf{currentUT}_n + \frac{\mathrm{WCET}_x}{P_x}$
6          if $\mathsf{newUT}_n > n.\mathsf{thresholdUT}$ then
7              $\mathsf{M} \leftarrow \mathsf{M} - \{n\}$
8          else if ($x.criticality$ = "C" $\oplus$ $n.criticality$ = "C") then
9              $\mathsf{M} \leftarrow \mathsf{M} - \{n\}$
10         end
11     end
12 return M
```

real-time operating system (Figure 3.33, line 38). Nodes with no RT operating system or RT capabilities are automatically excluded from the pool of schedulable nodes for RT tasks.
Filtering of nodes. The platform applies Eq. (2.4) extended to the containerized version of tasks to implement the *container admission test*. Algorithm 3 depicts the filtering phase, which relies on such a container admission test. The algorithm takes as input the list of nodes N of the cluster, and the container of task t_{new} that has to be scheduled, x. For each node of the cluster (line 3), the algorithm considers all the containers currently running (i.e., already deployed) in the node and sums up the utilization of each container (line 4). The result ($currentUT_n$) represents the left hand side of Eq. (2.4) extended to containers. The algorithm calculates the projected total utilization of the node ($newUT_n$) by considering the additional resources of container x under deployment (line 5). If the resulting projected utilization is greater than the threshold $n.\mathsf{thresholdUT}$ (which represents the right hand side of Eq. (2.4) extended for containers), the algorithm filters the current node from the pool of schedulable nodes. The algorithm also uses an `XOR` operator (line 8) to check whether both the node and task criticality are C. If only one of the two is C, the node n is marked as not schedulable[6].
Scoring of nodes. To implement the scoring phase, the platform relies on the following equation to obtain a normalized ranking ($nRank$) for each node:

$$\forall n \in N: \quad nRank = \begin{cases} 1 - \frac{n.\mathsf{thresholdUT} - \mathrm{newUT}_n}{n.\mathsf{thresholdUT}} & \text{if crit.} = A \\ \frac{n.\mathsf{thresholdUT} - \mathrm{newUT}_n}{n.\mathsf{thresholdUT}} & \text{if crit.} = C \end{cases} \tag{3.17}$$

where $n.\mathsf{thresholdUT}$ represents the threshold RT utilization ($M{*}sched\_rt\_runtime\_us/sched\_rt\_period\_us$), and $newUT_n$ represents the projected RT utilization after the deployment of x in n.

With a normalized utilization value $[0,1]$ for each node, independent of the total runtime and CPU cores available on the node, the platform applies a custom policy for scoring based on the value of the *criticality* field in the RT specification extension (see Algorithm 4). If a task has a criticality level of A, the algorithm assigns the task to the node with the highest RT load (i.e., the node with the highest normalized utilization), line 3. In contrast, for level C, the algorithm gives the highest rank to the node with the lowest normalized utilization (line 11). For the criticality level B (line 5), the algorithm maps the utilization to a function that gives the highest rank to the nodes with a utilization level closest to $\frac{1}{K}$, as follows:

[6] In the Kubernetes terminology, a *non-schedulable* node is a cluster node that does not satisfy the requirements of the container that has to be scheduled.

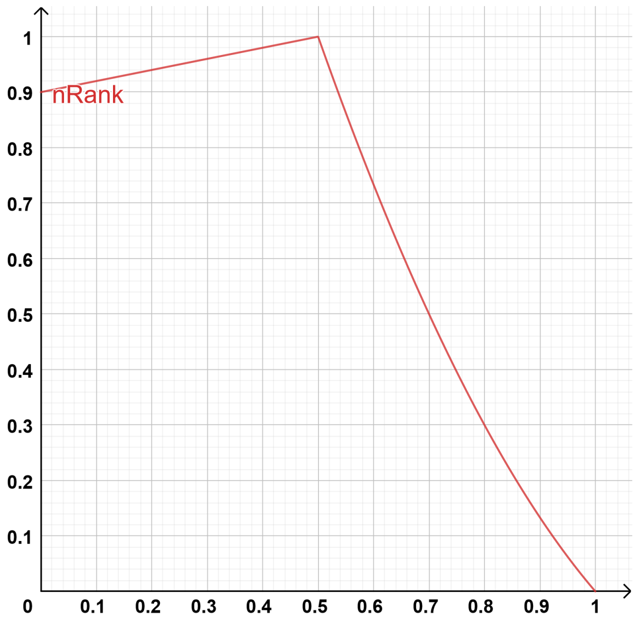


Fig. 3.35: The mapping results with Eq. (3.18) for the criticality level B with the coefficients adopted in the experimental results ($a = \frac{1}{5}$, $b = \frac{9}{10}$, $a' = \frac{5}{3}$, $b' = -\frac{9}{2}$, $c' = \frac{17}{6}$).

$$\forall n \in N:$$
$$nRank = \begin{cases} h\left(\frac{n.\mathsf{thresholdUT} - \mathrm{newUT}_n}{n.\mathsf{thresholdUT}}\right) & \frac{n.\mathsf{thresholdUT} - \mathrm{newUT}_n}{n.\mathsf{thresholdUT}} \leq \frac{1}{K} \\ g\left(\frac{n.\mathsf{thresholdUT} - \mathrm{newUT}_n}{n.\mathsf{thresholdUT}}\right) & otherwise \end{cases} \tag{3.18}$$

Scoring with this function allows us to modify the deployment behavior to best fit the needs and requirements of the tasks. For example, we can make a criticality B task resemble the behavior of a criticality A task with $K \to 1^-$ and a linearly increasing $h(\cdot)$, or closer to C with $K \to 0^+$ and a linearly decreasing $g(\cdot)$.

In our experiments, we applied Eq. (3.18) with $K = 2$, a linear function $h(\cdot) = a(\cdot) + b$, and a quadratic function for $g(\cdot) = a'(\cdot)^2 + b'(\cdot) + c'$. This allows us to linearly increase rank for nodes that have newUT_n lower than $\frac{1}{2}$, but greater than 0, and then a sharp decrease once that the threshold utilization value is reached. Fig. 3.35 shows the corresponding mapping, whereby the nodes with average RT load are classified as nodes with the highest ranking.

Container-level and cluster-level monitoring of RT containers

RT-Kube implements the monitoring of RT containers at two levels. At container-level, one monitor per container collects temporal information of the corresponding RT task and reports, at runtime, any temporal violation (i.e., overrun WCET or missed deadline) to the cluster-level monitor. This last implements the *reconcile phase*, which consists of checking the temporal constraints (i.e., threshold of missed deadlines) and eventually migrating containers to free resource and recover the system. To guarantee portability, each container-level monitor takes advantage of the `SIGXCPU` signal [183] that any Linux operating system can raise when a temporal violation occurs. Once the monitor receives such a signal, it communicates the updated counter of missed deadlines to the cluster-level monitor, which in turn implements the corresponding orchestration countermeasures. The injection of monitors in the SW application does not require any modification to the source code. Fig. 3.36 depicts the sequence diagram of the whole SW orchestration, starting from the container-level monitoring units. The figure reports the components of the standard Kubernetes release and the extension components in blue and green, respectively. To implement a continuous runtime monitoring while saving computational resources, we implemented each container-level monitor unit through two threads. The first receives the `SIGXCPU` signal and updates

Algorithm 4: Real-time scoring extension for the Kubernetes scheduler.

```
input  : A node $n$, the RT container $x$, the projected utilization $\mathsf{newUT}_n$
output: The score $s$ for RT container $x$ on node $n$
1  Function RealTimeScorePlugin ($n$, $x$, $\mathsf{newUT}_n$):
2      $\mathsf{nRank} \leftarrow \frac{n.\mathsf{thresholdUT} - \mathsf{newUT}_n}{n.\mathsf{thresholdUT}}$
3      if $x.criticality$ = "A" then
4          $s \leftarrow 1 - \mathsf{nRank}$
5      else if $x.criticality$ = "B" then
6          if $\mathsf{nRank} \leq \frac{1}{K}$ then
7              $s \leftarrow h(\mathsf{nRank})$
8          else
9              $s \leftarrow g(\mathsf{nRank})$
10         end
11     else if $x.criticality$ = "C" then
12         $s \leftarrow \mathsf{nRank}$
13     end
14 return $s$
```

```yaml
apiVersion: rt.monitoring/v1alpha1
kind: Monitoring
metadata:
  name: monitoring-test
spec:
  node: nodeA
  containerName: nginx
  missedDeadlinesPeriod: 0
  missedDeadlinesTotal: 0
```

Listing 3.37: Example of monitoring object created by the container-level monitor of RT-tasks.

the counters for the missed deadlines. The second communicates the RT container status (i.e., the updated number of missed deadlines) to the API server periodically through the HTTP-REST protocol. The container status is encoded into a Kubernetes compliant JSON object (see Figure 3.37) to guarantee modularity and scalability of the system. The API server validates the object syntactically (validation request in Fig. 3.36) and updates the RT container status in the ETCD database. It then requests for the system-level check to the cluster-level monitor (i.e., reconcile request).

The cluster-level monitor implements the reconcile phase. It collects the updated status of missed deadlines from each container-level monitor of the cluster. The updated status consists of the number of deadlines missed since the last update and the total number of missed deadlines in the RT container lifetime. If the number of missed deadlines, either in the period or total, is higher than the user-defined threshold (obtained from the centralized ETCD database), the cluster-level monitor selects a container to be immediately evicted and killed. The choice relies on the policy statically defined by the user. In our implementation, aside from RT containers with criticality C, any container with a lower criticality than the one with temporal violations and that has a stateless task can be selected. The cluster-level monitor notifies the eviction target to the controller manager (init. eviction in Fig. 3.36), which implements the *container eviction* and automatic re-deployment as for the standard protocol of Kubernetes' controllers. The monitor also implements *tainting* of the node that hosted the evicted container, which consists of marking the node as not available to host new containers for a period of time. This allows us to avoid immediate deployments of new containers in the node where the missed deadlines were observed. As a consequence, the evicted container is re-deployed on a different cluster node, thus implementing a de-facto stateless migration.

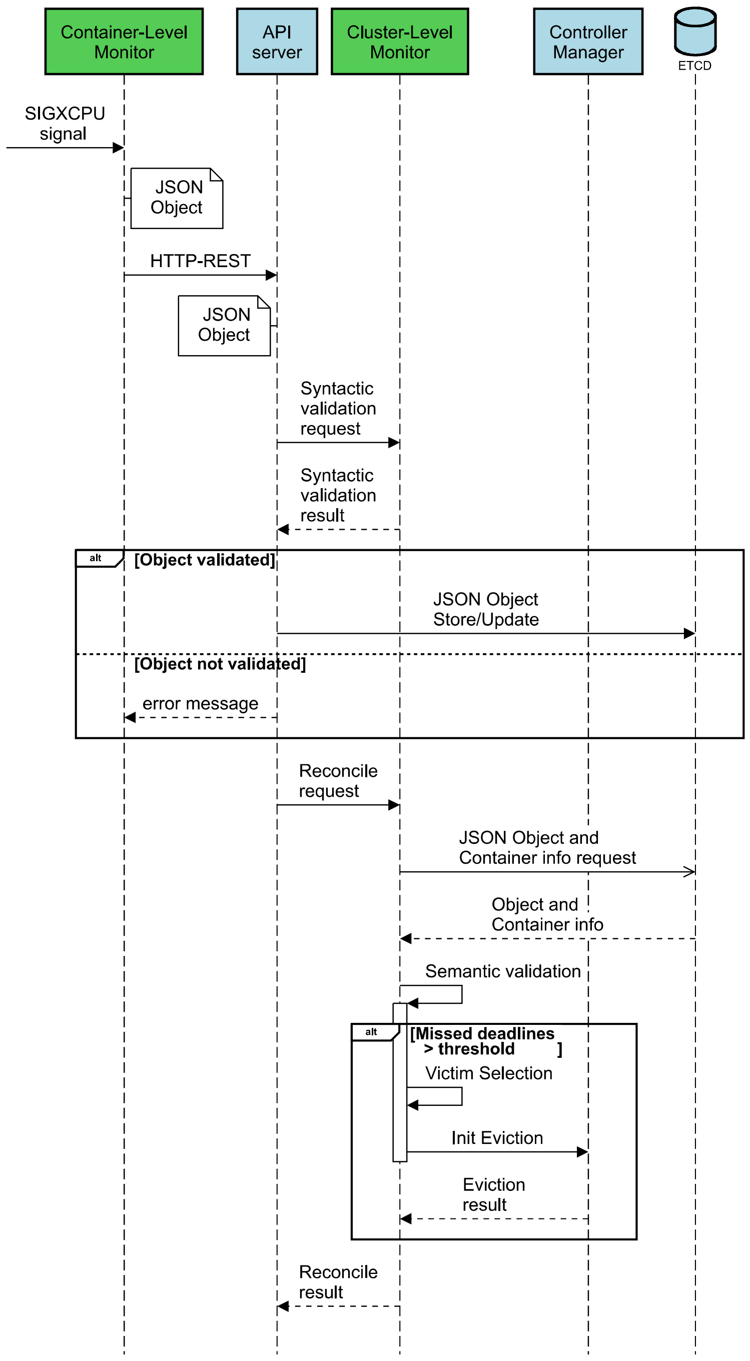


Fig. 3.36: Sequence diagram of the whole dynamic SW orchestration, starting from the container-level monitoring units.

Response time and predictability of the system recovery from temporal violations

We define *response time* (RT_i^c) of the system recovery from temporal violations of the RT container c on the cluster node i, as the time elapsed from the first deadline missed by c that leads c to exceed the threshold of missed deadlines to a container eviction on i. It is characterized by three components (see Fig. 3.38):

$$RT_i^c \leq \frac{1}{f_c} + HTTP\_trans(JSON_c) + Rec\_phase \tag{3.19}$$

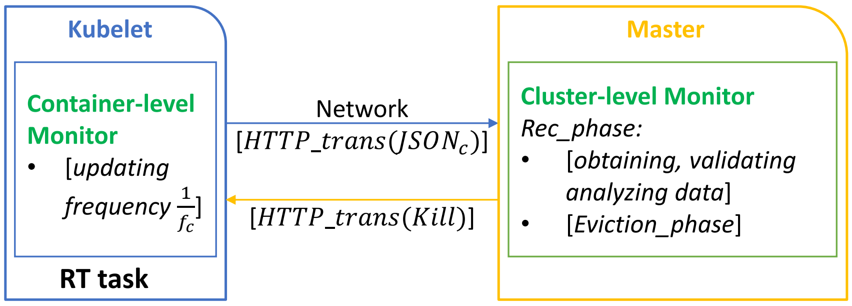


Fig. 3.38: The RT-Kube architecture considered for the analysis and formalization of the response time.

where f_c is the frequency the monitor in container c updates the master with the total number of missed deadlines; $HTTP\_trans(JSON_c)$ is the time spent for transferring the JSON object containing the counter information from node i to the master; Rec_phase is the time spent by the master for the reconcile phase.

Each cluster-level monitor updates the counter of missed deadlines at every SIGXCPU signal locally, while it updates the master periodically to save system resources. The period between two counter updating defines the worst case delay (i.e., first component of Eq. (3.19)).

We assume that the cluster-level monitor and the Kubernetes master are hosted on the same cluster node. As a consequence, we consider the communication latency between monitor and master being negligible. In contrast, we consider the time spent for transferring the updated data from the container-level monitor to the master over the network. This latency strongly depends on the static and dynamic characteristics of the communication network (i.e., bandwidth, traffic, etc.). Predictable networks for RT applications have been extensively studied in literature [184], and could be considered to increase the predictability of such a response time component. Nevertheless, we generalize the definition and consider $HTTP\_trans(JSON_c)$ as the worst case time spent for transferring the JSON object containing the counter information (e.g., $\approx 180 Bytes$ in our case study) to the master over HTTP.

The third component represents the time spent by the master for the reconcile phase. The master accesses the ETCD database for the semantic validation of the updating message (i.e., the JSON object), the request for specifications of each container (i.e., memory, CPU, storage requirements) and the additional specifications for RT containers (i.e., CRD module with deadline, period, WCET, criticality). Using this information, the master implements the victim selection and eviction through a message (i.e., standard killing Kubernetes message) over the network. We model the latency of the reconcile phase as follows:

$$\begin{aligned} Rec\_phase \ \leq \ & 2 \cdot n \cdot t_{SPEC} + n_{RT} \cdot t_{CRD} \\ & + Eviction\_phase \\ & + HTTP\_trans(Kill) \end{aligned} \tag{3.20}$$

where n is the total number of containers, n_{RT} is the number of RT containers ($n_{RT} \leq n$), t_{SPEC} and t_{CRD} are the latencies spent for retrieving the container and RT container specifications from the database. $HTTP\_trans(Kill)$ represents the latency spent for notifying a killing procedure by the master to the cluster node operating system. For this component, the considerations formulated before on the transfer time of a Kubernetes message over HTTP apply. The time for the eviction phase strictly depends on the complexity of the algorithm implementing the victim selection. We implemented three policies with different complexity. The first relies on an iterative linear search over the list of containers deployed on node i to find the container with the highest use of a single system resource (i.e., either CPU or memory). It starts from the lowest priority class of containers (i.e., non real-time) and, in case none of them is deployed in i, it iterates on the lists of the higher-priority containers.

The second policy implements a similar search, by considering the combination of two system resources. Being based on reordering and search phases, its complexity is linearithmic on the number of containers (i.e., $n \cdot log(n)$, with n the number of containers). The third considers inter-container communication dependencies and task semantics, and has quadratic complexity (n^2). We present a comparison among the three policies in terms of response time in Section 4.4.

3.5 Assertion-based verification and workload migration in Kubernetes for robotic systems

Thanks to the recent advances in robotics and artificial intelligence, autonomous mobile robots are now adopted in a wide spectrum of applications. Nevertheless, their safety and reliability requirements as well as their robustness guarantees remain major barriers to their large-scale adoption in real-world systems. Although various formal verification methods have been proposed to validate end-to-end software correctness [185–187], there is still a shortage of research and understanding in developing solutions to support *runtime system verification*. Runtime verification is a mandatory component for the validation process of robotic infrastructure. It is required to verify that the software implementing the robot's mission and behavior is correct and also satisfies extra-functional constraints (e.g., real-time, energy efficiency, reliability) when run on a real-world robotic platform.

Good design practice for runtime validation requires the use of Assertion-Based Verification (ABV) [188]. Assertions (i.e., user-defined properties that express desired behaviors of the target system) are first synthesized into *monitors* (i.e., chunks of code that implement the assertion semantics) and then checked at runtime to report any violation and possibly enforce fail-safe behaviors. In the state of the art, robotic applications monitors are generally applied to watch system resources and detect local faults [189]. Nevertheless, with the increased complexity in perception and control, modern robots and autonomous systems need more advanced and complex monitoring tasks during their daily missions. Such monitoring tasks range from enforcing security and safety properties while ensuring the correct execution of synthesized plans, to pattern matching over sensor readings to help perception [4].

Runtime verification based on such a monitor introduces computational overhead, whose amount depends on the number of active assertions, the complexity of the assertions, and the observed signals. When adopted in resource-constrained architectures, such workload variability can lead to system overloads and failures even with very few monitor instances. In similar contexts, migrating functional tasks from edge devices in the IoT to edge servers or to the cloud has been shown to be a valid solution to free resources and avoid system bottlenecks [190]. On the one hand, applying the same technique for migrating monitors may free resources for functional tasks at the edge. On the other hand, moving monitors far from the sensors or from the functional tasks that generate the observed events (i.e., signal values) requires continuous updating of these events through the communication network. As a consequence, the migration of monitors is a challenging task, as it could move the bottleneck problems from the edge to the network, with consequent inefficiency in the overall system performance.

We address this challenge by proposing an ABV platform for the automatic generation, orchestration, and deployment of monitors across Edge-Cloud computing architectures for robotic systems. Starting from STL assertions that express the behavior of the system as required by its specification, the platform synthesizes monitors that comply with the ROS [12] standard[7]. It then enables dynamic balancing of the SUV workload by considering a trade-off among verification accuracy, runtime constraints, and resource requirements. This is achieved by a novel approach that allows migrating the runtime evaluation of the monitors across the different layers of the Edge-Cloud computing platform. Finally, the verification environment is containerized to enable portability.

[7] We refer to the standard as ROS, even though the platform and the generated monitors are compliant to ROS2.

Problem statement

As a starting point, we consider the SUV being implemented as a set of distributed software tasks, which are modelled through ROS nodes. The ROS nodes execute on computing devices, which are distributed across an edge-cloud platform. This assumption is justified by the fact that ROS is the standard developing environment in the robotic community. It is based on a *publish-subscribe* communication paradigm, where a sender node publishes data on a *topic*, on which one or many nodes subscribe to receive the data (see Section 2.2). The adoption of ROS provides different advantages. First, it allows the modelling and simulation of complex systems through nodes running on different target devices. Second, it implements inter-node communication in a modular way to guarantee code portability. Finally, it adopts standards and widespread protocols requiring minimum intervention or modifications of the original code.

Based on this premise, we automate the generation and deployment of a monitoring platform for the runtime verification of ROS-compliant robotic systems. The challenge lies in formulating a strategy to seamlessly integrate monitors into the SUV implementation, ensuring accurate runtime verification without compromising its functionality (e.g., degrading the quality of the service or violating real-time constraints), which may happen if part of the computing resources have to be preempted for the verification tasks.

Each ROS node is located in a computational layer between the edge and the cloud. Its position is initially chosen with the ideal goal of keeping the source of data as close as possible to the computing unit that will elaborate it. At the edge, since data is elaborated immediately, monitoring and verification involve low response latency. However, edge devices are generally characterized by limited computational resources. As a consequence, execution of the SUV functionality may be hampered by the addition of runtime verification, as monitoring steals computational resources from other software tasks, possibly delaying their completion.

On the cloud, we assume no limitation in terms of computational capabilities. Nonetheless, end-to-end verification latency can become unpredictable due to the variability of bandwidth and traffic on the shared communication network. As a consequence, failure notifications could be delayed. We propose a verification architecture that generates ROS-compliant monitors that can automatically migrate between computing layers in accordance with the available resources and requirements of the functional tasks to best take advantage of the strengths of each platform.

Verification architecture

Fig. 3.39 shows an overview of the proposed verification platform. It is intended to support assertion-based verification of robotic systems at runtime by dynamically migrating the execution of monitors among the computational layers of the SUV implementation from edge to cloud. The platform automatically synthesizes monitors from STL assertions in the form $G(antecedent \rightarrow consequent)$. Monitors exploit the ROS publish-subscribe paradigm to collect at runtime the values of the SUV variables involved in the target assertions, thus allowing their evaluation. In the following, we call *event* the value assumed by a SUV variable at a time instant t. Whenever an event occurs, the system publishes such information on a topic. Then, each monitor subscribes to the topics associated with the variables included in the corresponding assertion, to receive all the events required for its evaluation. Thus, the platform enables the integration of monitors into the SUV without requiring any modifications to its source code.

The execution of monitors is finally managed by a set of *monitor handlers*. There is one monitor handler per computing device in the system. The handler is a ROS-compliant node containing an orchestrator and an instance of every monitor. At each execution instant, exactly one handler activates one instance per monitor, according to the decision taken by the orchestrator. Such a decision relies on the analysis of the trade-off between verification accuracy, runtime constraints, and resource requirements.

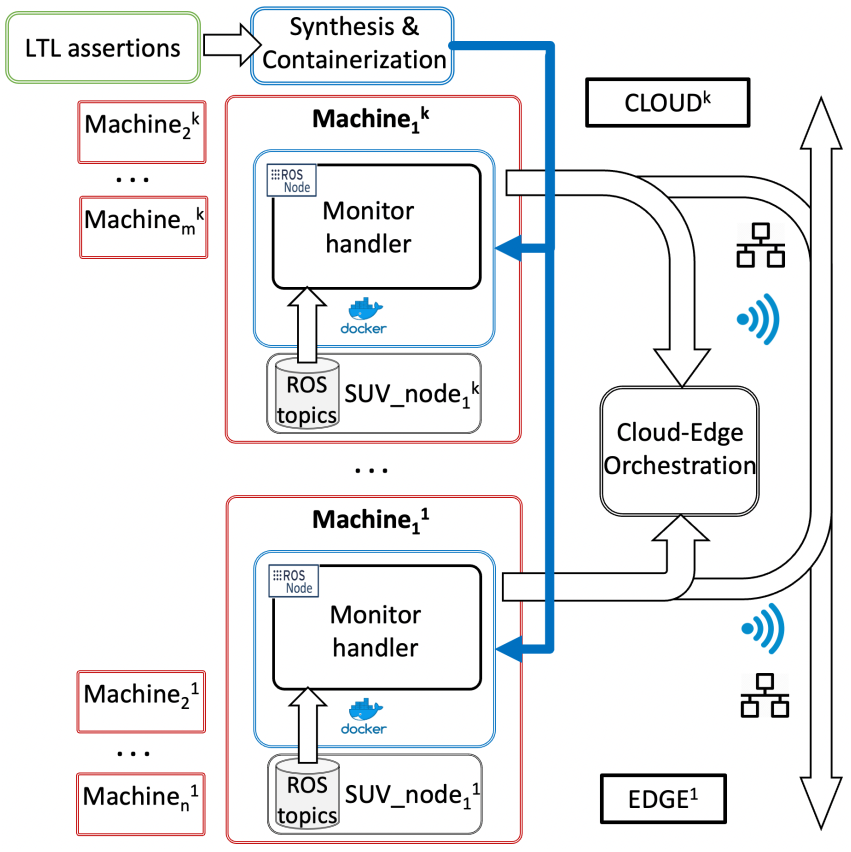


Fig. 3.39: Verification architecture.

In this way, we can set up an ABV environment that dynamically migrates the execution of monitors across computing devices, taking into consideration both resource constraints and communication latency.

Through *containerization* of the verification environment, the platform allows for the deployment of monitors across different hardware architectures and operating systems, as well as for handling the resources allocated for verification. The software application implementing the robotic tasks is also containerized through Docker and orchestrated through Kubernetes.

Monitor synthesis

The input of our verification architecture is a set of assertions expressed by using the STL logic. Assertions are then synthesized into monitors. They are composed of a C++ evaluation function to check the assertions dynamically and a ROS-compliant handler to capture the events necessary for performing their evaluation and enabling the edge-cloud migration. The evaluation function is created following state-of-the-art automatic procedures [131–133], and since this is a standard approach, we do not provide further details on it. On the contrary, we will focus more on the structure of the ROS-compliant handler in the following.
Assertion grammar. STL is a temporal logic formalism for specifying the properties of continuous signals introduced in [191]. STL is widely used to analyze programs in cyber-physical systems that interact with physical entities. STL extends the temporal operators of the Linear Temporal Logic (LTL) with intervals, forcing the formulae to resolve within a temporal window. The main temporal operators in LTL are:

- **F** (eventually): The formula $\mathbf{F}\phi$ is true if there exists a future time point at which ϕ is true.
- **G** (globally): The formula $\mathbf{G}\phi$ is true if ϕ is true at all future time points.

```
assertion :  G[N,N](tformula |-> tformula)

tformula: proposition
    | (tformula) | !tformula | tformula && tformula
    | tformula || tformula | tformula xor tformula
    | tformula U[N,N] tformula | tformula R[N,N] tformula
    | F[N,N](tformula)

N: unsigned integer
propostion: C/C++ expression of Boolean type
```

Fig. 3.40: Assertion grammar.

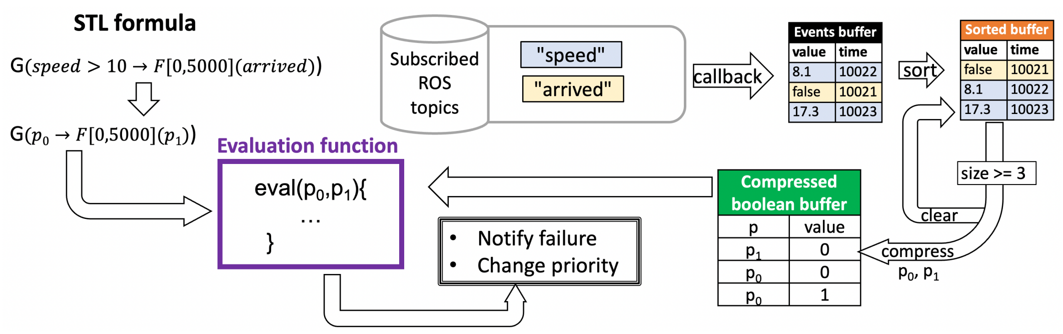


Fig. 3.41: Architecture of a monitor.

- **X** (next): The formula $\mathbf{X}\phi$ is true if ϕ is true at the next time point.
- **U** (until): The formula $\phi_1\mathbf{U}\phi_2$ is true if ϕ_2 is true at some future time point t and ϕ_1 is true at all time points up to t.

STL also includes temporal operators that specify a range of time intervals, such as:

- $\mathbf{F}[a,b]$ (eventually within interval): The formula $\mathbf{F}[a,b]\phi$ is true if there exists a future time point within the interval $[a,b]$ at which ϕ is true.
- $\mathbf{G}[a,b]$ (globally within the interval): The formula $\mathbf{G}[a,b]\phi$ is true if ϕ is true at all time points within the time interval $[a,b]$.
- $\phi_1\mathbf{U}[a,b]\phi_2$ (until within interval): is a temporal operator that specifies a range of time intervals. The formula is true if and only if there exists a time $t' \in [a,b]$ such that ϕ_2 is true at time t', and ϕ_1 is true for all time points in the interval $[t,t']$, where t is the current time. In other words, ϕ_1 must hold until ϕ_2 holds within the interval $[a,b]$.

Our verification architecture allows the formalization of assertions according to the grammar reported in Fig. 3.40, which implements a fragment of STL. In this work, the intervals in the STL operators specify the time window in milliseconds.
Monitor architecture. The process of translating an assertion to a C/C++ monitor consists of three main phases, as shown in Fig. 3.41:

1. Substitution of each proposition in the assertion with a placeholder.
2. Generation of the evaluation function.
3. Generation of the monitor handler (i.e., the infrastructure to capture the events).

Each proposition included in the target assertion is first substituted with a placeholder (i.e., a Boolean variable). This is motivated by the fact that Boolean information can be compressed through a single bit per value, which in turn allows a more efficient monitor migration. To illustrate, let us refer to the running example shown in Fig. 3.41. The assertion:

$$G(speed > 0 \rightarrow F[0,5000](arrived))$$

is first substituted by:

$$G(p_0 \rightarrow F[0,5000](p_1))$$

where p_0 and p_1 are the placeholders for *speed* > 0 and *arrived*, respectively. Once the substitution is completed, we generate a C++ evaluation function capable of checking the truth values of the assertion. The interface of the evaluation function is formalized as follows: $eval(p_0, p_1, ...p_n)$ where each p_i is a Boolean placeholder. The function is called each time a new event (new values for the assertion variables) must be evaluated.

Since all assertions are of the form $G(antecedent \rightarrow consequent)$, each new event at time t_i requires checking the truth value of the implication at time t_i; furthermore, multiple pending instances (evaluation instances that require future events to determine the truth value of the assertion) can accumulate in the monitor. The monitor is also capable of differentiating between antecedent and consequent instances. We will exploit the number of pending instances in the monitor to determine the priority of the monitor. Additionally, the pending instances correspond to the state (memory) of the monitor that will need to be migrated during the orchestration process.

During execution, the monitor evaluation function needs to receive as input the sequence of values assumed by the variables involved in the corresponding assertion during the SUV execution. We then describe below how to generate a ROS-compliant monitor handler to provide the evaluation function with such values.

The monitor handler subscribes to all topics used to exchange information (i.e., values of variables) included in the corresponding assertion. For each variable, a callback is used to retrieve the interesting events (i.e., variable changes) from the corresponding ROS topic. The system attaches a callback procedure to an independent thread that executes each time a message is processed from the subscriber queue. More formally, an *event* is a triple $\langle v, new\_value, timestamp \rangle$ to specify that the value new_value is assumed by the variable v at time $timestamp$, during the execution of the SUV. A captured event is added to an *event buffer* in the monitor handler each time a callback is executed, delaying its processing. the buffer enables the orchestrator to move the monitor evaluation across the edge-cloud layers, without considering the location where the events were observed. Furthermore, the buffer can be sorted by making use of timestamps, thus ensuring that the events are processed in chronological order. This minimizes the evaluation errors caused by synchronization problems and/or communication delays.

When the buffer's size reaches a certain threshold, it is sorted. A higher threshold improves verification accuracy since this increases the probability of evaluating the events in the correct order; however, this can severely reduce verification responsiveness. For this reason, every computing node in our architecture is synchronized with the precision time protocol, guaranteeing synchronization errors of the order of nanoseconds while requiring minimal bandwidth and little processing overhead. Consequently, the sorting threshold can be set to a low value, allowing for high verification responsiveness, while suffering from negligible accuracy errors.

After the buffer has been ordered, the Boolean placeholders of the target assertions are replaced by using the values corresponding to its events. These values are stored, in the same order as Boolean constants, in a new *compressed buffer*, where the timestamps associated with the events are removed (since the events are already ordered). Each time the evaluation function is called, an event is consumed from the compressed buffer and used to advance the verification.

In the example reported in the right part of Fig. 3.41, the ROS handler subscribes to two topics: *speed* and *arrived*. They correspond to the variables involved in the assertion shown in the left-hand side of Fig. 3.41. In this example, three events are captured and added to the *event buffer* by the callback functions: $\langle speed, 8.1, 10022 \rangle$, $\langle arrived, false, 10021 \rangle$, and $\langle speed, 17.3, 10023 \rangle$. If the sorting threshold, for example, was set to 3, after the arrival of the third event, the elements of the *event buffer* are ordered according to their timestamps and moved to the *sorted buffer*. Then, each event is compressed and added to the *compressed buffer*. Events $\langle speed, 17.3, 10023 \rangle$ and $\langle speed, 8.1, 10022 \rangle$ are compressed to 1 and 0, respectively, as the corresponding proposition $p_1 : speed > 10$ is true for *speed* equal to 17.3 and false for *speed* equal to 8.1; moreover, $\langle arrived, false, 10021 \rangle$ is directly translated to 0.

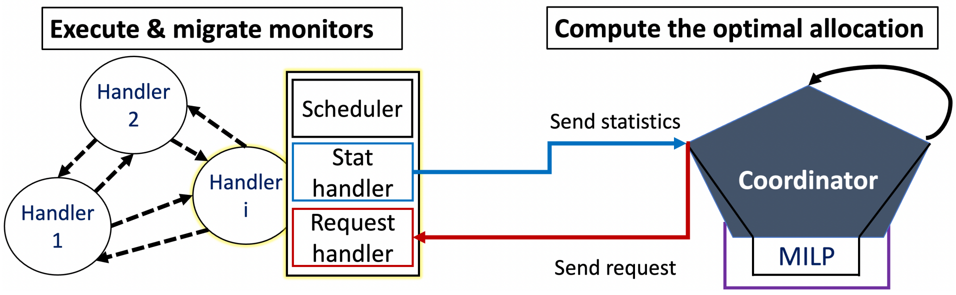


Fig. 3.42: Orchestrator's architecture.

Monitor containerization

The monitors are *containerized* after the synthesis process to make the verification environment portable across various HW/SW architectures. Different containerization techniques are at the state of the art for cloud-native applications. However, they provide each container with its own private (isolated) subnet IP addresses. As a result, they only allow ROS nodes to communicate if they are assigned to the same subnet IP. To solve this communication issues between distributed ROS nodes, we expanded the containerization process based on Docker for edge computing.

The proposed platform automatically maps each container IP address to the IP address of the host device (i.e., where the ROS node executes) while randomly allocating port numbers. This reduces communication latency by eliminating any network overhead caused by containers [76]. See Section 3.2.1 for further details.

By using *multi-architecture containers* (e.g., Docker `buildx`), the platform supports the simple generation and integration of containers for different HW/SW target architectures, from cloud to off-the-shelf edge devices (e.g., NVIDIA Jetson). These multi-architecture containers are concealed behind a single container that has multiple integrated versions, which allow for greater orchestration flexibility.

Monitor runtime management

The orchestration aim is to balance the responsiveness of verification and resource consumption during the SUV execution. Low verification responsiveness may cause delays in the identification of assertion failures. Conversely, intensive use of computational resources for verification may cause the violation of run-time constraints related to functional tasks running on the same device, thus causing assertion failures.

In the optimal situation, a monitor is executed on the same edge node that generates the values for the variables observed by the monitor itself. When computing resources at this node become not enough to support both the verification effort and the execution of the functional tasks, our orchestration system migrates the execution of the monitors towards another computing unit, possibly belonging to a higher computational layer, in the edge-cloud architecture. This guarantees that additional resources are available at the destination device to handle the monitor, but at the cost of lower responsiveness. In fact, in this new configuration, the monitor is evaluated farther from the source of the observed events.

Architecture and workflow of the orchestrator. The orchestrator consists of two main elements: a set of handlers, one per each computing unit in the SUV, and a global coordinator (see Fig. 3.42), which make up a fully connected verification network.

The coordinator orchestrates the allocation of monitors to the computing units in the SUV by continuously searching for the optimal allocation of resources. It then issues requests to the handlers to force such allocation. The coordinator requests are of the following types:

- EXEC request: the coordinator commands a handler to execute a certain set of monitors;
- MIGRATE request: the coordinator commands a handler to migrate a certain set of monitors to another handler;

- SHUTDOWN request: the coordinator commands a handler to stop executing.

Each handler is responsible for collecting statistics about the state of the execution in the corresponding computing unit (through the *stat handler* module) and satisfying the requests from the coordinator (through the *request handler* module). It also controls the execution of monitors assigned by the coordinator through its *scheduler* module. Statistics are periodically sent by the handlers to the coordinator. They mainly include the current CPU usage dedicated to the verification process, and the time elapsed from the publishing of a ROS topic (corresponding to the change of a variable) to the evaluation of the monitors that have subscribed for the topic.

The scheduler of each handler manages the dynamic allocation of computational resources to evaluate the elements stored in the event buffers of allocated monitors. Each monitor evaluation is seen as a request to be fulfilled by the scheduler. The scheduler satisfies these requests by spawning as many worker threads as the available cores of the device. A thread that receives a request keeps executing the evaluation function of the corresponding monitor until a fixed time slice runs out or until the event buffer is empty. After that, the request is pushed back to the scheduler's queue. The scheduler handles the requests by using a priority delay queue. To avoid starvation, requests with low priority are promoted to higher priorities as their waiting time increases.

The orchestrator life cycle is then divided into three main parts: init, allocation, and termination.

At the *init* time, the verification network is initialized. Each handler discovers the existence of all the other computing units by sending a broadcast message. Then the handlers run an election algorithm to determine which unit will execute the coordinator. The algorithm selects the handler executed on the unit with the highest available computational resources; the handler spawns an additional thread that executes the coordinator. After that, each handler starts sending statistics to the coordinator and listening for requests. Note that the verification network is designed to automatically include new nodes (handlers arrived late) and to elect a new coordinator if it becomes unresponsive for an extended period of time.

After that, the *allocation* phase takes place. The coordinator periodically analyses the statistics from the handlers of the SUV computing units and calculates the optimal allocation of monitors that maximizes the responsiveness of the verification without saturating the CPU of any machine. Then, the coordinator sends EXEC or MIGRATE requests to handlers to enforce optimal allocation in the network. This is repeated throughout the SUV execution.

At *termination*, when the SUV execution stops, the coordinator issues a SHUTDOWN request to all handlers, and all nodes of the verification network terminate.

Computation of the optimal allocation. During the allocation phase, the coordinator periodically solves a Mixed Integer Linear Programming (MILP) problem to determine where to allocate the monitors among the various computing units comprising the SUV. The solution to the MILP problem corresponds to the allocation of monitors that maximizes the responsiveness of the verification without saturating any CPUs. The coordinator constructs the MILP problem by using the statistics received from the handlers. The statistics are formalized as follows:

- $monitorCPU_{c_j}^{m_i}$ percentage of CPU usage spent to execute monitor m_i on computing unit c_j;
- $topicCPU_{c_j}^{t_k}$ percentage of CPU usage spent to receive data from ROS topic t_k on computing unit c_j;
- $availableCPU_{c_j}$ percentage of unused CPU plus the CPU consumption of the verification process on computing unit c_j;
- $delay_{c_j}^{t_k}$ time (in milliseconds) to receive data from ROS topic t_k on computing unit c_j.

Formally, for M monitors, C computing units and T topics, we search for the allocation of any monitor m_i to a computing unit c_j that minimizes the objective function:

$$\sum_{k=1}^{T}(\sum_{j=1}^{C} getDelay\_t_k\_c_j(\bigvee_{i=1}^{M} m_i == c_j)). \tag{3.21}$$

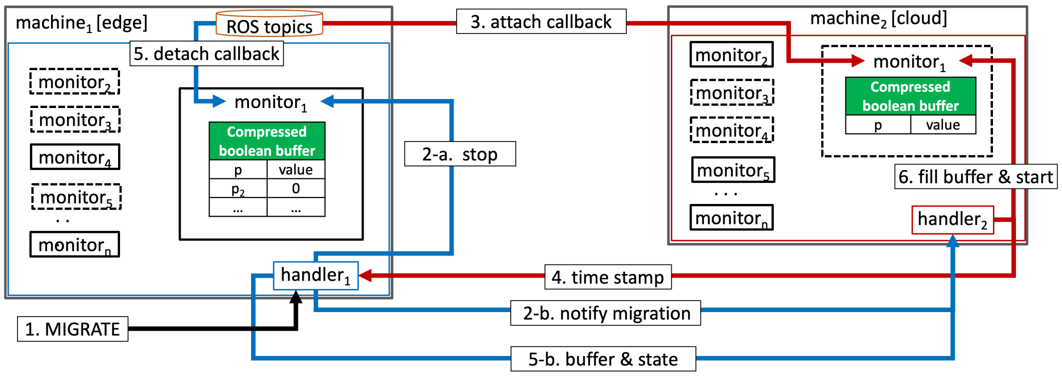


Fig. 3.43: Example of buffer migration.

The objective function is subject to the following constraint:

$$\bigwedge_{j=1}^{C}(availableCPU_{c_j} >=$$
$$(\sum_{i=1}^{M} getMonitorCPU\_m_i\_c_j(m_i == c_j)+ \tag{3.22}$$
$$\sum_{k=1}^{T} getTopicCPU\_t_k\_c_j(\bigvee_{i=1}^{M} m_i == c_j)))$$

where, m_1, $m_2, ..., m_M$ are enumerated integer variables, associated with the monitors, which can assume one value among $c_1, c_2, ...c_C$, associated with the computing units. Intuitively, the above constraint ensures that the CPU usage of the resulting allocation of monitors does not exceed the available CPU on any computing unit.

The result of the MILP is an assignment of variables that minimizes the objective function. The functions $getDelay\_t_k\_c_j$, $getMonitorCPU\_m_i\_c_j$ and $getTopicCPU\_t_k\_c_j$ take as input a Boolean value, and they return zero if such a value is *false*; otherwise, they return, respectively, $delay_{c_j}^{t_k}$, $monitorCPU_{c_j}^{m_i}$, and $topicCPU_{c_j}^{t_k}$.

Since the MILP contains only linear constraints and integer arithmetic, the dynamism of the system is not affected by the execution time of the MILP, as it is capable of computing the optimal allocation of monitors in just a few milliseconds, even when there are hundreds of monitors.

Monitor migration. The migration of a monitor takes place when the coordinator sends a MIGRATE request. Since each machine has a copy of all monitors, to move a monitor m_j from the computing unit c_1 to the computing unit c_2 the migration procedure operates as follows: it first turns off m_j on c_1, then it sends the state and event buffer of m_j to $c2$, and, finally, it activates m_j on c_2. This procedure requires data to be moved through the network. However, the proposed migration strategy is extremely effective because it is lightweight by design, even in slower networks. This is also thanks to data compression. Additionally, since no event is lost during migration, the suggested approach does not experience any false negatives (the monitor does not fail when it should) or false positives (the monitor fails when it should not).

To explain the migration protocol between two machines, let us consider the example shown in Fig. 3.43. It shows $monitor_1$ that, executing at level l_i of the edge-cloud hierarchy, moves from $handler_1$ to $handler_2$ at level l_{i+1}. Before migration, $handler_1$ executes monitors 1 and 4, while $handler_2$ executes monitors 2 and 5. Let us assume that, at a given point, $handler_1$ receives a MIGRATE request from the coordinator (step 1). Then, $handler_1$ removes $monitor_1$ from the scheduler (step 2_a), but the process of adding events to its buffer is not interrupted. Sequentially, $handler_1$ notifies that the migration has started to $handler_2$ (step 2_b). As a consequence, $handler_2$ starts adding events to the buffer of $monitor_2$ by attaching the callbacks (step 3). At this time, $monitor_2$ is not yet on the scheduler. Once

$monitor_2$ receives enough events to make the first ordering, $handler_2$ returns to $handler_1$ the timestamp of the oldest event in $monitor_1$ (step 4). In this way, as soon as $monitor_1$ receives the timestamp, it recognizes which buffer events must be sent (i.e., the events evaluated with a timestamp lower than the received timestamp). When this happens, $handler_1$ detaches the callbacks from $monitor_1$ to stop adding events to its buffer (step 5_a) and sends the correct events to $handler_2$ together with the state of $monitor_1$ (step 5_b). At that point, $monitor_1$ is inactive on $handler_1$ and $handler_2$ finishes the migration by filling the buffer of $monitor_1$ with the received events and by putting the monitor on the scheduler to start its evaluation (step 6).

Handling overload scenarios. The buffer migration procedure is intended to free computational resources on heavily loaded machines to balance the verification effort among the cloud-edge computation layers. This is of utmost importance in systems where the lack of computational resources could severely affect the reactivity of the SUV, making its functional tasks miss deadlines or, in the worst scenario, causing the whole system to crash. However, it might happen that no allocation of monitors exists that does not saturate any CPU in the SUV. To handle this overload scenario, we implemented two additional strategies to reduce the computational load of verification. First, by assigning the scheduling priorities according to the run-time criticality of the monitors, and second, by reducing the observation frequency of the events evaluated in the monitors. The first strategy affects the responsiveness of the verification, while the second may affect the accuracy of the verification.

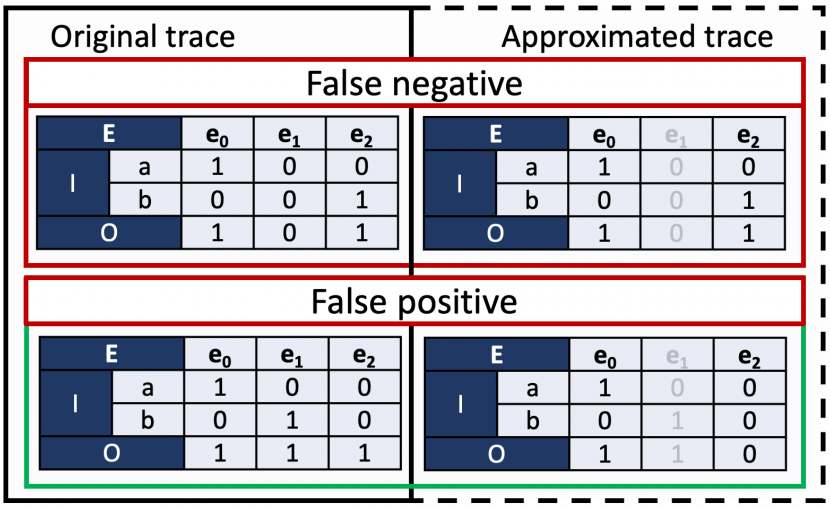
The first strategy consists of assigning a higher scheduling priority to monitors containing active instances (i.e., whose antecedent has been fired and the consequent is still pending), purposely increasing their responsiveness. The evaluation function is capable of counting the number of active instances in a monitor: after each evaluation, the priority of a monitor is increased proportionally to the number of active instances. The priority is the lowest when no instance is active.

The second strategy consists in discarding not-yet evaluated events from the event buffer and resetting the monitor to its initial state. Thus, performing an approximation of the event trace. When a monitor is left behind in the evaluation, its event buffer keeps filling with past events. This can happen mainly for two reasons: either those events cannot be evaluated because the executing machine does not provide enough resources to the verification environment, or because of the low priority of the monitor, which is considered non-critical by the scheduler, and thus its evaluation function is executed with a lower frequency.

Approximating the event trace by removing "old" events from the buffer decreases the required resources, as the discarded events no longer have to be evaluated, and increases the responsiveness of the monitor, as more up-to-date events are used in the evaluation. However, discarding events may affect verification accuracy. In fact, this approach can either cause a monitor failure, when it should instead succeed (false positive), or can make the monitor miss a failure (false negative).

Fig. 3.44 shows an example to clarify this concept. Let us consider the simple assertion $always(a \rightarrow next(b))$. The variables a and b represent the input of the monitor (reported in row I) in three different instants. The values in row O are the outputs of the monitor. Evaluated events are shown in row E. In the upper part of Fig. 3.44, we show an example of a false negative. In this scenario, the original event trace causes the monitor failure after e_1. In the approximated trace, e_1 is instead discarded, making the monitor miss the failure due to e_1. The approximation then produced a false negative. In the lower part of Fig. 3.44, we show an example of a false positive. In this case, the values originally assumed by a and b, i.e., $(1,0),(0,1),(0,0)$, at event e_0, e_1, e_2 (table on the left) do not make the monitor fail; therefore, the monitor output is always 1. When the event approximation is applied (table on the right), event e_1 is discarded. This causes the monitor failure after receiving event e_2. Thus, the approximation produced a false positive.

It should be noted that, in our verification architecture, monitors synthesized from assertions following the template $G(antecedent \rightarrow consequent)$ are guaranteed to not produce false positives because the monitor is reset after discarding the events. Thus, only false negatives are possible. Consequently, in scenarios where the presence of a few false negatives is not the main concern, this approach is useful to reduce resource consumption while main-



E		e_0	e_1	e_2
I	a	1	0	0
	b	0	0	1
O		1	0	1

E		e_0	e_1	e_2
I	a	1	0	0
	b	0	0	1
O		1	0	1

E		e_0	e_1	e_2
I	a	1	0	0
	b	0	1	0
O		1	1	1

E		e_0	e_1	e_2
I	a	1	0	0
	b	0	1	0
O		1	1	0

Fig. 3.44: False negative/positive example.

taining good responsiveness. The cost is paid in terms of degradation in the verification accuracy or in terms of a delay in detecting a failure. This makes sense, for example, when assertions are intended to monitor soft tasks that affect the system performances or the quality of service, or when a *stable property* (e.g., a deadlock occurs, a token gets lost, the program terminates) is monitored, where a delayed detection does not cause catastrophic effects. Nevertheless, this second approach should not be applied to monitors that check hard tasks and safety-critical requirements, as a missed failure might be catastrophic.

4

Experimental Results

This chapter reports the results obtained for each of the five steps required to reach the objectives of Fig. 1.2. Section 4.1 analyzes the benefits of edge-related optimizations. Section 4.2 shows how the porting of containerization and orchestration to robotic environments allows for improved performance thanks to the Edge-Cloud computing continuum. Section 4.3 demonstrates how adapting containerization and orchestration allows for the reconfigurability of software and more advanced deployment configurations. Section 4.4 proves how containerization is compatible with real-time, as well as orchestration software. Finally, Section 4.5 shows the results of the proposed runtime migration of tasks and containers.

4.1 Optimizing performance on heterogeneous devices at the edge

This Section focuses on the results obtained for the optimizations related to the edge-computing components, specifically:

- Section 4.1.1 analyzes the improvements obtained with XEFT and the pipelined DAG-executions (see Section 3.1.1).
- Section 4.1.2 analyzes the benefits of ZC techniques on UMA (see Section 3.1.2).
- Section 4.1.3 analyzes the improvements obtained by applying ROS-ZC (see Section 3.1.3).

4.1.1 Improving the scheduling on DAG-based embedded vision applications

To compare the different algorithms and heuristics, we considered two categories of benchmarks. The first is a real-world SLAM application [192] combined with an inference application based on CNN for object detection [193]. Such a computer vision application implements the simultaneous localization and mapping algorithm with one or more RGB camera sensors. It computes, in real-time, the camera(s) trajectory, a 3D scene reconstruction, and car/pedestrian detection. We tested the different scheduling solutions by considering three versions of the applications: *Monocular* with a DAG composed of 41 tasks, *stereo* (81 tasks), and *4-stereo* (161 tasks). We adopted the standard KITTI input dataset [194] for the evaluation, which is the de-facto standard evaluation benchmark in computer vision and robotics. It consists of video streams taken by a car driven around city blocks.

The second category is a large set of synthetic DAGs. We implemented a parametric DAG generator which we used for our evaluation by creating around 40,000 DAGs with different characteristics: size (from 20 to 250 nodes), degree of nodes, execution times of the node implementations for CPU and GPU, multiple implementations vs. exclusive implementation of nodes, and CPU/GPU speedup for nodes with multiple implementations. We collected the generated DAGs in two classes: *Tree*, for DAGs with high average degree, medium small diameters, and low standard deviation; *linear* for the rest. The idea was to classify the DAGs depending on the average level of topological constraints among nodes.

For all benchmarks, we used an NVIDIA Jetson TX2 device as edge computing architecture, which is a prevalent low-power heterogeneous device used in industrial robots, machine

ORB-SLAM version	**CPU cores (#)**	**DL time (ms)**	**Max XO (ms)**	**Overlap (ms)**		**XO (ms)**		**Idle time (%)**		**XO / Max_XO (%)**		**Makespan (ms)**			**Speedup**			**SLR**	
				HEFT	**XEFT**	**HEFT**	**XEFT**	**HEFT**	**XEFT**	**HEFT**	**XEFT**	**VW**	**HEFT**	**XEFT**	**HEFT on VW**	**XEFT on VW**	**XEFT on HEFT**	**HEFT**	**XEFT**
Monocular	2	15	30.0	43.1	46.0	10.7	24.5	45.9%	27.6%	35.8%	81.8%	74.7	39.9	31.7	46.6%	57.5%	20.4%	2.45	1.95
Monocular	3	15	44.1	62.1	70.2	18.7	32.5	31.5%	20.7%	42.3%	73.7%	74.7	30.2	25.3	59.6%	66.2%	16.3%	1.86	1.56
Monocular	4	15	44.1	76.0	80.2	26.5	29.4	28.2%	40.5%	60.0%	66.6%	74.7	26.5	25.3	64.6%	66.2%	4.4%	1.63	1.56
Monocular	5	15	44.1	90.1	82.0	37.8	31.3	21.8%	40.9%	85.6%	70.9%	74.7	23.0	24.4	69.2%	67.4%	-5.7%	1.42	1.50
Monocular	2	20	40.0	51.6	55.2	7.0	32.5	43.9%	15.5%	17.6%	81.2%	79.7	46.0	32.7	42.3%	59.0%	29.0%	2.16	1.54
Monocular	3	20	44.1	72.3	80.2	19.1	32.5	31.9%	30.8%	43.4%	73.7%	79.7	35.4	30.3	55.6%	62.0%	14.3%	1.66	1.43
Monocular	4	20	44.1	92.9	98.2	31.3	34.4	21.2%	30.1%	71.0%	77.9%	79.7	29.5	28.7	63.0%	64.0%	2.6%	1.39	1.35
Monocular	5	20	44.1	103.2	100.4	41.0	37.4	23.1%	35.1%	93.0%	84.7%	79.7	26.9	27.9	66.3%	65.0%	-3.8%	1.26	1.31
Monocular	2	25	44.1	60.3	70.4	0.3	36.8	42.9%	14.4%	0.7%	83.5%	84.7	52.8	35.3	37.7%	58.3%	33.2%	2.01	1.34
Monocular	3	25	44.1	84.9	92.0	16.6	34.2	29.3%	25.1%	37.6%	77.4%	84.7	40.0	34.4	52.8%	59.5%	14.1%	1.52	1.31
Monocular	4	25	44.1	107.3	109.4	32.9	37.4	20.2%	31.2%	74.7%	84.7%	84.7	33.6	32.9	60.3%	61.2%	2.2%	1.28	1.25
Monocular	5	25	44.1	119.2	119.2	42.3	42.3	22.0%	22.0%	95.9%	95.9%	84.7	30.6	30.6	63.9%	63.9%	0.0%	1.16	1.16
Stereo	2	15	30.0	66.4	67.8	5.7	21.1	50.7%	43.8%	18.9%	70.3%	118.8	67.4	60.3	43.3%	49.2%	10.5%	4.15	3.71
Stereo	3	15	45.0	92.5	96.6	8.8	25.5	40.7%	31.3%	19.6%	56.7%	118.8	52.0	46.8	56.3%	60.6%	9.9%	3.20	2.88
Stereo	4	15	60.0	115.4	121.4	17.2	45.1	32.8%	15.3%	28.7%	75.2%	118.8	42.9	35.9	63.9%	69.8%	16.5%	2.64	2.21
Stereo	5	15	75.0	133.0	137.0	15.6	41.5	32.4%	25.8%	20.8%	55.3%	118.8	39.4	34.2	66.9%	71.2%	13.1%	2.42	2.11
Stereo	2	20	40.0	74.2	76.6	9.8	30.4	48.8%	38.7%	24.4%	75.9%	123.8	72.4	62.5	41.5%	49.5%	13.7%	3.41	2.94
Stereo	3	20	60.0	101.9	110.0	6.4	46.1	42.2%	17.6%	10.7%	76.9%	123.8	58.8	**44.5**	52.6%	64.1%	24.3%	2.76	2.09
Stereo	4	20	80.0	122.1	127.1	12.4	50.6	39.3%	19.0%	15.5%	63.2%	123.8	50.3	**44.2**	59.3%	68.3%	22.1%	2.37	1.85
Stereo	5	20	88.2	146.3	147.0	16.0	41.5	34.2%	35.2%	18.1%	47.0%	123.8	**44.4**	**39.2**	64.1%	68.3%	11.7%	2.09	1.85
Stereo	2	25	50.0	81.0	87.0	0.0	36.7	51.2%	31.7%	0.0%	73.4%	128.8	83.0	63.7	35.6%	50.5%	23.2%	3.16	2.43
Stereo	3	25	75.0	110.8	123.1	0.0	57.3	44.2%	10.0%	0.0%	76.4%	128.8	66.2	45.6	48.6%	64.6%	31.1%	2.52	1.74
Stereo	4	25	88.2	134.1	147.1	6.2	50.6	40.7%	28.1%	7.0%	57.3%	128.8	56.6	44.2	56.1%	65.7%	21.9%	2.16	1.68
Stereo	5	25	88.2	163.4	167.8	13.0	44.9	33.4%	36.4%	14.8%	50.9%	128.8	49.1	43.3	61.9%	66.4%	11.8%	1.87	1.65
4-stereo	2	15	30.0	112.7	114.0	1.8	1.0	57.3%	54.3%	6.1%	3.2%	207.0	131.9	124.7	36.3%	39.8%	5.5%	5.63	5.32
4-stereo	3	15	45.0	155.5	159.6	8.3	0.9	47.8%	43.2%	18.5%	2.1%	207.0	99.3	93.7	52.1%	54.8%	5.6%	4.23	3.99
4-stereo	4	15	60.0	189.2	195.5	10.5	0.0	44.1%	39.0%	17.5%	0.0%	207.0	84.6	80.2	59.1%	61.3%	5.2%	3.61	3.42
4-stereo	5	15	75.0	217.8	228.7	12.4	13.3	42.0%	35.0%	16.5%	4.3%	207.0	75.1	70.4	63.7%	66.0%	6.3%	3.20	3.00
4-stereo	2	20	40.0	120.3	126.7	0.0	16.6	56.3%	48.6%	0.0%	41.4%	212.0	137.7	123.2	35.0%	41.9%	10.6%	5.87	5.25
4-stereo	3	20	60.0	164.2	179.5	7.8	21.8	48.1%	34.3%	13.0%	36.4%	212.0	105.5	91.0	50.2%	57.1%	13.7%	4.50	3.88
4-stereo	4	20	80.0	202.2	215.6	9.7	16.1	43.8%	32.9%	12.2%	20.1%	212.0	89.9	80.3	57.6%	62.1%	10.6%	3.83	3.43
4-stereo	5	20	100.0	237.0	248.6	13.7	61.4	40.1%	29.1%	13.7%	61.4%	212.0	79.1	70.1	62.7%	67.0%	11.4%	3.37	2.99
4-stereo	2	25	50.0	130.7	137.3	0.0	33.4	54.6%	43.9%	0.0%	66.8%	217.0	144.0	122.4	33.7%	43.6%	15.0%	5.48	4.66
4-stereo	3	25	75.0	181.4	188.8	0.0	27.1	45.1%	32.9%	0.0%	36.1%	217.0	110.1	93.8	49.3%	56.8%	14.8%	4.20	3.57
4-stereo	4	25	100.0	222.4	236.0	3.8	89.0	40.3%	22.9%	3.8%	89.0%	217.0	93.2	76.6	57.1%	64.7%	17.8%	3.55	2.92
4-stereo	5	25	125.0	258.3	276.5	8.1	114.4	38.1%	17.4%	6.4%	91.5%	217.0	83.5	66.9	61.5%	69.2%	19.8%	3.18	2.55

Table 4.1: Experimental results with ORB-SLAM+DL on Jetson TX2.

vision cameras, and portable medical equipment. It consists of a quad-core ARM Cortex-A57 MPCore, a dual-core NVIDIA Denver 2 64-Bit CPU, and a 256-core NVIDIA Pascal GPU with 256 NVIDIA CUDA cores. We tested the mapping and scheduling activity over 2, 3, 4, and 5 CPUs core + 1 GPU. We reserved one Denver CPU core for the OpenVX runtime system and CPU/GPU synchronization.

Evaluation of rank-based heuristics: HEFT vs. XEFT

Table 4.1 summarizes the results we obtained with different configurations of the ORB-SLAM run with the inference application (ORB-SLAM+DL) on the Jetson TX2 device with the different CPU/GPU scenarios (i.e., #CPU cores enabled combined to the GPU). The table reports the comparison of the rank-based mapping and scheduling solutions considered, i.e., the proposed HEFT implementation and XEFT. NVIDIA VisionWorks (VW) [30] has been taken as reference. The schedule length ratio (SLR) represents the normalization of the makespan over the maximum critical path.

We found that, as expected, HEFT sensibly improves (from 33.7% to 69.2%) the system performance w.r.t. the scheduling system currently implemented and released with the NVIDIA VisionWorks library. The table also shows that XEFT generates an exclusive overlapping among nodes that is higher than that provided by HEFT in almost all cases (see double column XO). We observed a XO reduction only with the simpler application versions (monocular) run on a large number of CPU cores (i.e., 5). This is due to the fact that the scheduler easily finds computing resources for the few executing nodes, thus any clusterization in the ranking cannot give better results. For this reason, in these two cases, HEFT also gives slightly better system performance w.r.t. XEFT (-5.7% and -3.8%).

The idle time values underline that this category of benchmarks scheduled with HEFT suffers from workload imbalance. The HEFT efficiency to generate exclusive overlapping is measured in column XO/Max_XO. Such a value is higher with the simpler applications that have also low levels of maximum potential XO. The clustering effect of XEFT is an increase of the XO efficiency and a reduction of the idle times in the CEs. The two values improve by increasing the application complexity.

We found that, in general, XEFT provides better performance w.r.t. HEFT up to 33.2% and, more importantly, it can provide the same or better performance w.r.t. HEFT with less architectural resources (e.g., with one ore more CPU cores less). As an example, in the configuration Stereo with 20ms DL, XEFT provides the same performance of HEFT with 2 CPU cores less.

Figures 4.1 and 4.2 show the performance comparison between XEFT and HEFT on the synthetic benchmarks. The two figures identify each benchmark on the plot by considering two metrics evaluated with HEFT: the XO and the idle time. As an example, the rightmost side and top side of the plot group the benchmarks for which HEFT provides the highest XO and the highest idle time, respectively. For each benchmark, the circle and the cross represent the improvement and loss of performance, respectively. The size of circles(crosses) represents the improvement(loss) measure.

The figures aim at understanding the correlation between the XEFT efficiency w.r.t. HEFT and the DAG characteristics. For both the synthetic classes, the results underline that XEFT generally outperforms HEFT with benchmarks for which (i) HEFT suffers from idle time, and (ii) the XO efficiency of HEFT is low. This class of benchmarks are grouped, in both figures, in the leftmost top side. XEFT cannot improve the HEFT performance if the benchmark, with HEFT, is already well balanced (low idle time) or already presents high XO efficiency. In these cases, XEFT can lead to a loss of performance up to 19%.

Figure 4.1 shows that HEFT already performs well with the main parts of DAGs of class *Tree*. This is due to the fact that they provide high potential overlapping, less node constraints and, as a consequence, HEFT generates low idle time. Since the potential XO is also related to the number of exclusive nodes, which has been generated with a Gaussian distribution for the analysis, XEFT improves the performance mostly in benchmarks with less than 80% XO of HEFT. Figure 4.2 shows that the rest of the benchmarks are more distributed over the space. Here it is evident the main contribution of XEFT on the leftmost top side of the plot. The ORB-SLAM+DL benchmarks belong to this category of DAGs.

Evaluation of scheduling with pipelined DAG executions

In this section we first present the evaluation of the benefits provided by the pipelined DAG execution in terms of performance improvement for all the considered scheduling alternatives. Then we present a global comparison among the alternatives. They include G-FL [31], the proposed HEFT implementation, and XEFT. Since we found that, in all the adopted benchmarks, the batched pipelined G-FL approach often causes GPU bottlenecks, as shown in Figure 3.5, generally leading to low performance w.r.t. the rank-based approaches, we also implemented an improved version of the G-FL algorithm. This modified version, which we call G-FL$_c$, uses the mapping produced by the HEFT heuristic and then schedules all the nodes using the normal G-FL scheduling rules.

For brevity, we focus the analysis and comparison on the batched version (without the constraint of Eq. (3.7)) for all the approaches. In general, the experimental results confirmed that such a constraint almost always leads all the approaches to not provide any performance improvement w.r.t. the non-pipelined version.

Table 4.2 shows what kind of benefits we can expect when combining a batched pipeline of DAGs with each scheduling compared to the non-pipelined makespan of the same sched-

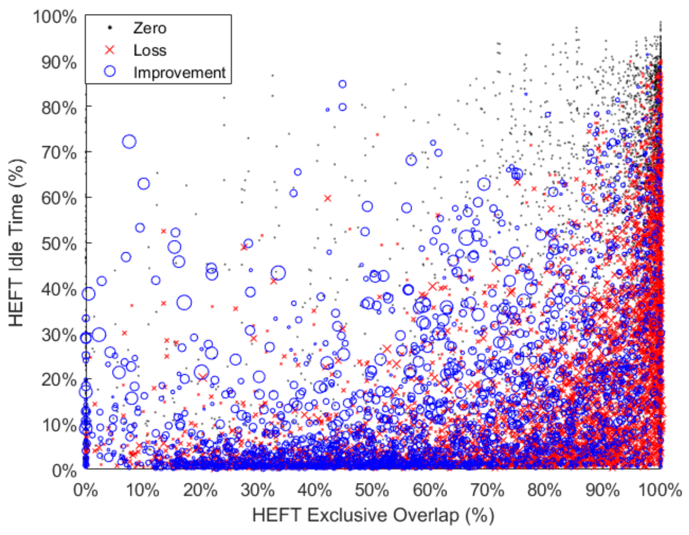


Fig. 4.1: Experimental results with the *Tree* class of synthetic DAGs on the Jetson TX2.

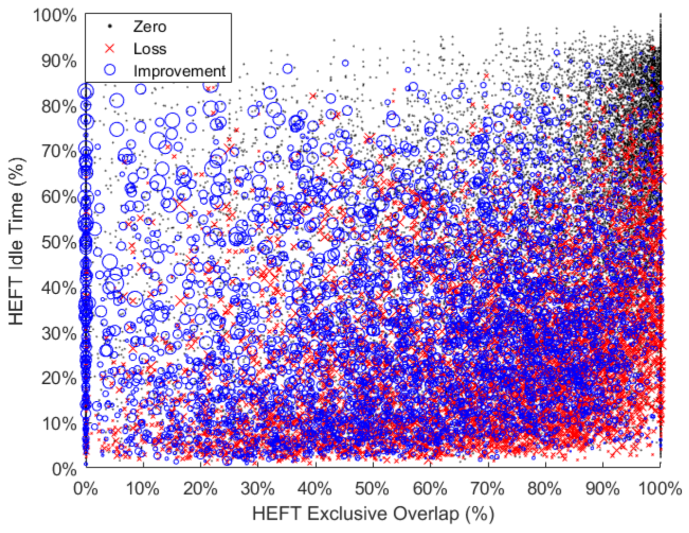


Fig. 4.2: Experimental results with the *Linear* class of synthetic DAGs on the Jetson TX2.

uler. In linear graphs, it is evident that the batched pipeline, on average, leads to better performance. We also found that, as expected, the gain increases with the number of CPU cores available to the scheduler.

With G-FL, moving from two CPU cores to three yields a 40% higher improvement on average for any number of pipeline levels (Frames in the table). Moving past three cores does not give any further benefit.

When looking at the length of the pipeline (number of frames), moving from three to five frames allows the biggest improvement out of any configuration with an average increase of 51% when compared to three frame. We also tested a ten frames pipeline, which still showed a slight improvement while sensibly less than from three to five.

G-FL$_c$ behaves almost identically, even though it showed the best average improvement, ranging from 7.6% when using two cores to 11.9% when using five. This suggests that high idle times generated by the schedule allow to take advantage of nodes from successive DAG instances.

Class	CPUs - Frames	G-FL	HEFT	G-FL$_c$	XEFT
Linear	2 - Any	2.66%	3.57%	7.62%	3.23%
Linear	3 - Any	3.72%	5.77%	9.63%	5.56%
Linear	4 - Any	3.87%	7.54%	10.92%	7.51%
Linear	5 - Any	3.73%	8.88%	11.94%	8.89%
Linear	Any - 3	2.79%	4.71%	7.93%	4.62%
Linear	Any - 5	4.20%	8.17%	12.12%	7.98%
Linear	Any - 10	5.15%	11.28%	15.16%	10.71%
Tree	2 - Any	0.55%	0.62%	0.71%	0.72%
Tree	3 - Any	0.71%	0.01%	0.03%	0.14%
Tree	4 - Any	0.81%	0.53%	0.51%	0.80%
Tree	5 - Any	0.88%	0.97%	0.83%	2.14%
Tree	Any - 3	0.67%	0.05%	0.09%	0.23%
Tree	Any - 5	0.81%	0.51%	0.44%	0.68%
Tree	Any - 10	0.94%	1.90%	1.93%	2.11%

Table 4.2: Evaluation of benefits provided by the batch pipeline on the scheduling algorithms with the different classes of benchmarks (*Linear* and *Tree*) considering multiple configurations of CPU cores and batch sizes.

	vs G-FL	vs G-FL$_c$	vs XEFT	vs G-FL (p)	vs G-FL$_c$ (p)	vs XEFT (p)
G-FL		-22.66%	-25.05%	-2.07%	-27.34%	-28.14%
G-FL$_c$	29.29%		-3.10%	26.62%	-6.06%	-7.09%
XEFT	33.43%	3.20%		30.67%	-3.05%	-4.12%
G-FL (p)	2.11%	-21.02%	-23.47%		-25.81%	-26.62%
G-FL$_c$ (p)	37.63%	6.45%	3.15%	34.78%		-1.10%
XEFT (p)	39.16%	7.63%	4.29%	36.28%	1.11%	

Table 4.3: Overall comparison of the scheduling approaches without and with the batched pipeline (p).

XEFT and HEFT run with the batched pipeline show moderate improvements, from around 3% to 9%. The improvement increases from two to three cores, allowing an average 72% better improvement, a further 35% when moving from three to four, and a further 18% when moving from four to five. Looking at the size of the pipeline, the change from three to five frames creates an improvement 73% higher, which, just like G-FL, is better than what we find when moving from five to ten, showing only a 34% better improvement. HEFT behaves basically the same as XEFT, only showing slightly lower margins when moving from two to three cores (62%).

In general, with a lower number of cores, XEFT shows fewer improvements than G-FL$_c$, due to the fact that XEFT generates less idle times. G-FL shows very limited improvements (2-3%) in almost all situations due to the bottleneck problems analyzed in Section 3.1.1.

The second part of the table shows how, in almost every instance of the *tree* graph class, the improvements are less than 1% and thus negligible for all the scheduling approaches. This is due to the fact that, as explained in Section 4.1.1, the characteristics of this DAG instances lead the schedulers to generate very small idle periods, for which the pipeline cannot provide any benefits.

The correlation between the idle periods of the non-pipelined HEFT and XEFT and the potential improvements given by the pipeline is better represented in Figure 4.3. The figure shows the speedup obtained by applying the pipeline for both HEFT and XEFT considering 4 CPU cores+GPU and 5 overlapped DAG instances for the linear benchmark class. For a large number of these benchmarks, the non-pipelined XEFT already provides very low idle times (leftmost side of the plot). For these cases, the speedup is very limited as the scheduler, even on the multi DAG instances, has no actual room for improvements. In case of benchmarks for which XEFT generates idle periods (center, rightmost side of the plot),

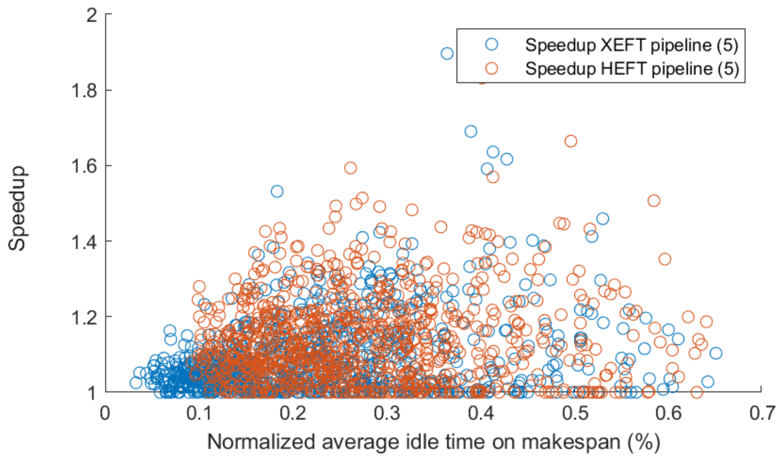


Fig. 4.3: Performance improvement with DAG pipelining for HEFT and XEFT considering 4 CPU cores+GPU and 5 frames with the *linear* class of benchmarks.

the pipeline can further improve the results. Similar trends can be observed with the HEFT scheduler, even though the higher idle time degree of the benchmarks with the non-pipeline HEFT is evident.

Table 4.3 shows the global comparison of the different schedulers. The first row shows how G-FL is up to 25.09% slower than all other solutions, and up to 28.14% slower when compared to the corresponding pipelined version. XEFT provides the best performance globally, being significantly faster than G-FL, and slightly faster than G-FL$_c$. As shown before, the pipelined G-FL does not improve enough particularly over G-FL, because the pipeline cannot improve on the bottlenecks that come from poor mappings. Even when compared to non-pipelined XEFT, it is significantly worse, begin on average 23.46% slower. The situation improves significantly when looking at G-FL$_c$, while this solution is still slower than XEFT, it is 29.29% faster than G-FL. It is important to remark that, the mapping implemented by G-FL$_c$ is extrapolated from the execution of the HEFT scheduling.

The pipelined version of G-FL$_c$ is comparable to XEFT and slightly behind the pipelined XEFT. This underlines that, both with and without the pipeline, XEFT is the fastest algorithm, but also that the mapping is of utmost importance to obtain the best possible result.

Pipeline memory footprint and effectiveness

When adopting the scheduling with batched pipeline, it is important to also consider the additional costs that comes with pipelining multiple DAG instances. As the addressed strategies relies on static scheduling to limit any run time overhead, the situation changes when looking at memory. Pipelining multiple DAGs means keeping them in the system memory while performing the computation and until all the operations on the corresponding frames are completed (i.e., all the nodes from the same DAG are scheduled and executed). This translates in a much higher memory usage. Because of this, when scheduling an embedded vision application with the pipeline, the number of DAGs we can overlap heavily depends on how much memory is available.

In addition, increasing the number of potential pipeline *levels* does not always translate into a corresponding number of DAG instances effectively overlapped at run time (see mapping and temporal constraints issues analysed in Section 3.1.1). To understand the effectiveness of the scheduling approach to exploit the pipeline levels, we report the average number of memory allocations reserved for the different frames that are actually exploited concurrently by the schedulers during the execution (see Table 4.4). For this analysis, we allowed each scheduler to reuse, for the computation of a DAG instance, the allocated resources used for the computation of any previous DAG as long as there is no more temporal overlap

	Schedulers with batched pipeline			
# DAG instances	G-FL	G-FL$_c$	HEFT	XEFT
3	2.0-2.1	2.6-2.9	2.5-2.9	2.5-2.9
5	2.8-3.1	3.7-4.6	3.4-4.6	3.4-4.6
10	4.6-5.3	5.7-8.1	6.3-8.2	5.4-8.0

Table 4.4: Average number of actual frame buffers exploited during the scheduling of pipelined DAG instances for the synthetic graphs (2 cores - 5 cores).

between the two instances. Reuse of resources for single nodes has not been considered (i.e., either all the allocated space can be reused or new space gets allocated).

The results in Table 4.4 underline that, on average, G-FL does not exploit all the resources instantiated for the pipeline. By allocating memory space for 3 overlapped DAG instances, G-FL uses in average the resources for 2 (2.0 represents the average number of levels exploited by considering all benchmarks with 2 cores, while 2.1 represents the same value with 5 cores). The waste of memory resources gets worse as soon as resources for more levels of the pipeline are allocated (e.g., for 5 overlapping DAG instances, G-FL actually uses in average 2.8 and 3.1 levels with 2 and 5 cores, respectively). With 10 available levels of pipeline, it can take advantage of half of the allocated resources.

The table also shows that all the other approaches take advantage of all the allocated resources until 5 levels of pipeline and by considering 5 CPU cores. Further than this level, these schedulers also cannot fully take advantage of the allocated resources, and the trend of performance improvement decreases (see Table 4.3).

4.1.2 Improving performance on Edge computing embedded boards with Unified Memory Architecture

To verify the benefits obtained with the UMA ZC framework (see Section 3.1.2), we start with the analysis of the micro-benchmarks on three edge computing devices, i.e., NVIDIA Jetson Nano, TX2, and AGX Xavier. We then utilize the results obtained for the tuning of two different real cases of study: an application for the extraction of centroids in Shack–Hartmann wave front sensors [195] and an ORB-SLAM application for the simultaneous localization and mapping [192].

Device micro-benchmarking

Fig. 4.4 shows the results of the first micro-benchmark on the Jetson TX2 and Xavier. ZC has side-by-side bars to show the overlapping execution. For the sake of space, the results on the Nano, which are equivalent to those of the TX2, have been omitted. The figure shows that the execution time of both the CPU routine and GPU kernel of the micro-benchmark with ZC are higher than those of SC or UM. This is due to the fact that the system disables the GPU cache when adopting the concurrent accesses of ZC.

With TX2, the performance difference is sensibly higher (up to 70%) since, differently from Xavier, TX2 disables also the CPU cache with ZC. The results shown in Table 4.5 ($GPU\_Cache_{LL\_L1}^{max\_throughput}$) confirm that the GPU memory accesses with ZC form an important bottleneck with a GPU throughput that is up to 77 times lower than the GPU throughput provided by SC and UM.

With Xavier, which implements I/O coherency and the CPU cache is always enabled, the difference between the GPU kernel performance with ZC and SC is “limited” to 3.7 times. Table 4.5 shows that the $GPU\_Cache_{LL\_L1}^{max\_throughput}$ of ZC in Xavier is still significantly worse than that in SC (or UM), even though the difference is sensibly reduced when compared to TX2 (i.e., 7 times lower in Xavier vs. 77 times lower in TX2).

In conclusion, when considering cache-dependent applications originally implemented with ZC, the ZC to SC switching can lead to a $\texttt{Max}_{ZC/SC\_speedup}$ equal to 70 in the TX2,

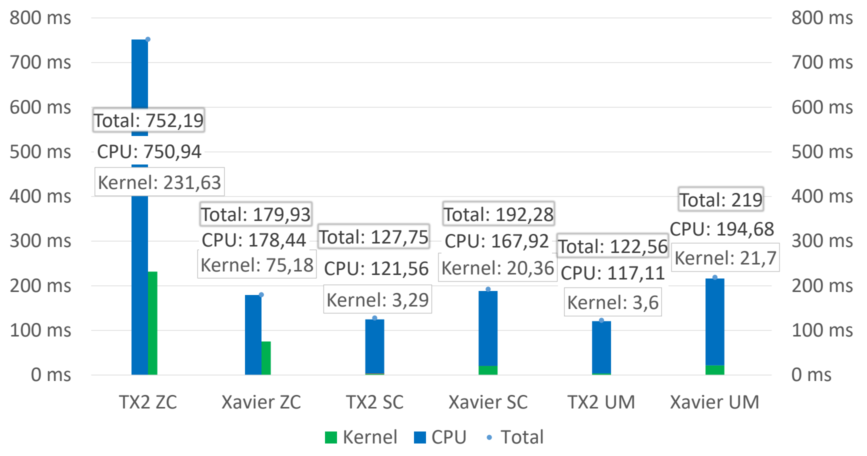


Fig. 4.4: First benchmark results: Execution times on the Jetson TX2 and Xavier with ZC, SC, and UM.

Board	$GPU\_Cache_{LL\_L1}^{max\_throughput}$		
	Zero Copy	Unified Memory	Standard Copy
TX2	1.28 GB/s	97.34 GB/s	104.15 GB/s
Xavier	32.29 GB/s	214.64 GB/s	231.14 GB/s

Table 4.5: First benchmark results: Maximum throughput of the GPU cache on the Jetson TX2 and Xavier.

while equal to 3.7 in Xavier. Since these values represent upper bounds, the micro-benchmark results underline that Xavier likely gives positive performance by adopting ZC *also* in many cache-dependent applications.

Figures 4.5 and 3.8 show the results of the second benchmark on the Jetson TX2 and Xavier, respectively. With TX2 (Fig. 4.5), from 1/16,000 to 1/8000 accesses, the GPU cache throughput between ZC and SC is comparable. This allows us to identify the GPU cache threshold (2.7%). Over 1/8000, the difference on throughput as well as on performance sensibly increases.

With Xavier (Fig. 3.8), we identified three zones (delimited by the vertical lines in the figure). In the first zone (left-most), ZC and SC provide the same performance. This allows us to identify the GPU cache threshold (16.2%) to switch to ZC. In the second zone (between the two vertical lines) the performance difference is below 200% (cache usage between 16.2% and 57.1%). In this case, the device may still provide equal or better performance by switching to ZC. In the third zone the performance difference sensibly increases over 200%, which suggests to adopt SC. This underlines that, when compared to SC, ZC can offer identical performance when there is limited cache usage, with linear performance degradation up to a hard limit for bandwidth (i.e., 59 GB/s on Xavier). The closer we move towards the third zone, the higher must be the time the application gains with concurrent execution and task overlapping. After 57.1% of GPU cache usage, the GPU is severely bottlenecked and the recommendation is to not use ZC.

Fig. 4.6 shows the results of the third benchmark, which are used to extrapolate the $\texttt{SC/ZC}_{Max\_speedup}$. The runtime of the CPU and GPU tasks are comparable and the tasks can be fully overlapped. Due to the large data set used, 2^{27} floats (i.e., 512 MB), transfer times contribute significantly to the system performance. ZC is up to 164% faster than UM and up to 152% faster than SC.

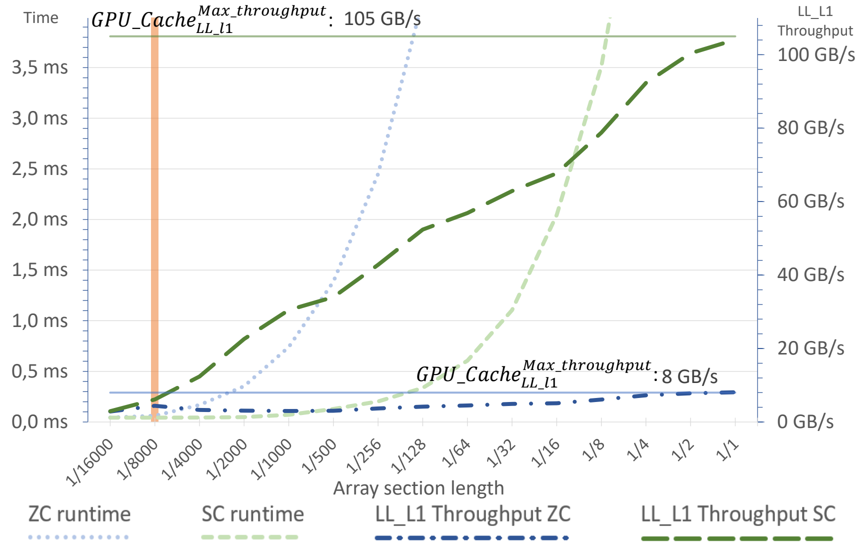


Fig. 4.5: Second benchmark results on the NVIDIA Jetson TX2.

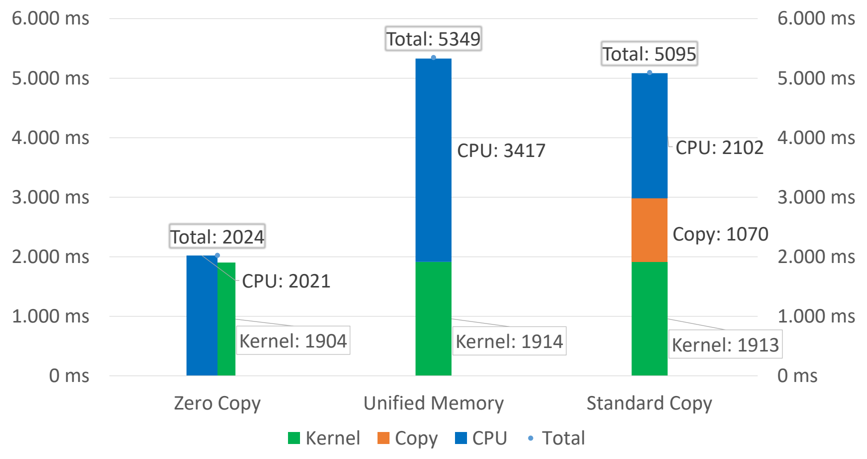


Fig. 4.6: Third benchmark results.

Board	CPU $Cache_{LL\_L1}^{usage}$ (%)	CPU cache thresh. (%)	GPU $Cache_{LL\_L1}^{usage}$ (%)	GPU cache thresh. (%)	Kernel times (μs)	Copy time/k-ernel (μs)	SC/ZC *speedup* (up to, %)
Nano	19.8	15.6	1.7	2.5	453.5	44.8	–
TX2	19.8	15.6	3.7	2.7	175.2	22.4	–
Xavier	6.1	100	7.0	16.2-57.1	41.2	16.88	69.3

Table 4.6: Profiling results of the SH-WFS application.

Tuning the Shack–Hartmann adaptive optics application

Adaptive optic algorithms measure and compensate optical aberrations when capturing images. We applied the proposed framework to tune an implementation of the adaptive optics with Shack-Hartmann sensors algorithm for edge computing [195]. Table 4.6 shows the profiling results, which quantify the dependency of the application performance on the CPU and GPU caches. In particular, the CPU cache usage of the application on Nano and TX2

Board	SC time (CPU only)	SC kernel time	UM time (CPU only)	UM kernel time	UM speedup (vs SC)	UM kernel speedup (vs SC)	ZC time (CPU only)	ZC kernel time	SC/ZC *speedup* (actual vs SC)	ZC kernel speedup (vs SC)
Nano	1070.1μs (238.6μs)	453.54μs	1021.5μs (259.7μs)	454.92μs	5%	0%	1796.1μs (1120.7μs)	467.21μs	−67%	−3%
TX2	765.04μs (79.6μs)	175.18μs	783.67μs (217.2μs)	177.16μs	−2%	−1%	801.24μs (307.4μs)	244.17μs	−5%	−39%
Xavier	304.57μs (41.9μs)	41.24μs	305.80μs (88.8μs)	47.08μs	0%	−14%	220.15μs (45.4μs)	47.14μs	38%	−14%

Table 4.7: SH-WFS centroid extraction algorithm performance results.

Board	$\overset{CPU}{Cache}{}^{usage}_{LL\_L1}$ (%)	CPU cache thresh. (%)	$\overset{GPU}{Cache}{}^{usage}_{LL\_L1}$ (%)	GPU cache thresh. (%)	Kernel times (μs)	Copy time per kernel (μs)	SC/ZC *speedup* (up to, %)
TX2	0	15.6	25.3	2.7	93.56	1.57	–
Xavier	0	100	20.1	16.2-57.1	24.22	1.35	5.9

Table 4.8: Profiling results of the ORB-SLAM application.

exceeds the threshold. This suggests that the application on these devices likely takes more advantage from the SC or UM communication models. With Xavier, the framework suggests to switch to ZC, with an estimated potential speedup of up-to 69%.

To evaluate the performance model, we implemented the application with the three communication models. Table 4.7 shows the results. As expected, the difference between SC and UM is negligible (below ±5%). Switching from SC to ZC on Nano and TX2 lead to a significant performance degradation. A sensible loss of performance has been measured with Nano (-67%), which was expected as, in that board, the micro-benchmarks classified the application CPU cache dependent while not GPU cache dependent. A loss of performance has been measured and was expected on TX2, since both the CPU and GPU cache usage were behind the corresponding thresholds. With Xavier, we observed a performance improvement of the system equal to 38%. Thanks to ZC and the corresponding data transfer elimination, we measured, in average, 0.12J and 0.09J per second energy saving on Xavier and TX2, respectively, w.r.t. SC. We did not consider the energy saving on Nano as the performance loss is not negligible.

ORB-SLAM application

For the sake of space, we report the comparison between the ORB-SLAM application considering only SC and ZC. We do not report the result with the Nano device as it does not allow satisfying the real time constraints of the (heavy) application. Table 4.8 shows the profiling results, which classify the application as GPU cache-dependent with both TX2 and Xavier. However, with Xavier, the profiling maps the application in the second zone of the GPU cache usage (see Fig. 3.8).

Table 4.9 shows the application performance we obtained with the application implemented with both SC and ZC on TX2 and Xavier. As expected, ZC on TX2 strongly limits the application performance. Instead, Xavier provides the same performance by considering the application implemented with SC and ZC. In this case, the performance of the GPU kernel that slightly decreases (−10%) is fully compensated by the absence of data transfers and task overlapping. With a 30Hz camera as input sensor, we measured an energy saving of 0.17J per second on the Xavier, on average.

Board	SC time	SC kernel time	ZC time	ZC kernel time	SC/ZC *speedup* (actual)	ZC kernel speedup
TX2	70ms	93.56μs	521ms	824.20μs	-744%	-880%
Xavier	30ms	24.22μs	30ms	26.99μs	0%	-10%

Table 4.9: ORB-SLAM performance results.

Fig. 4.7: The embedded platforms used for testing. The Nvidia Jetson Xavier (left) and Nvidia Jetson TX2 (right).

NVIDIA device	**Benchmark**	**GPU Copy**	**Time** (ms)		**SD** (ms)
Jetson TX2	Cache	CUDA-SC	833.0	±	3.8
Jetson TX2	Cache	CUDA-ZC	8 509.1	±	131.7
Jetson TX2	Concurrent	CUDA-SC	1 053.0	±	8.2
Jetson TX2	Concurrent	CUDA-ZC	1 316.1	±	33.9
Jetson Xavier	Cache	CUDA-SC	207.7	±	9.8
Jetson Xavier	Cache	CUDA-ZC	244.7	±	11.0
Jetson Xavier	Concurrent	CUDA-SC	381.2	±	8.3
Jetson Xavier	Concurrent	CUDA-ZC	256.9	±	6.1

Table 4.10: Time reference of cache benchmark and concurrent benchmark in NVIDIA Jetson TX2 and Xavier with CUDA-SC and CUDA-ZC.

4.1.3 Improving performance for CPS on Edge computing embedded boards with UMA

To verify the performance of the proposed ROS-compliant communication models, we carefully tailored two different benchmarks: cache-dependent and concurrent benchmark. For details on the benchmarks, see Section 4.1.2.

For all tests, we used an NVIDIA Jetson TX2 and a Jetson Xavier as embedded computing architectures (see Fig. 4.7). The routines of both benchmarks are optimized on I/O coherent hardware through the use of the `cudaHostRegister` API.

These two synthetic tests aim at maximizing the communication bottleneck and are representative of a worst-case scenario communication-wise. Real world applications, especially in the field of machine learning, will see a lesser bottleneck because of a more coarse-grained communication. In contrast, a high number of nodes with limited communication will still benefit from the application of the proposed methodology to reduce the overall communication overhead.

Table 4.10 shows the performance results obtained by running the two benchmarks on the two different devices with different communication models (i.e., CUDA-SC, CUDA-ZC) without ROS. The reported times are the averaged results of 30 runs. Standard deviation is also considered in the fifth column (i.e., column "SD"). As expected, the TX2 device provides

Arch.	**GPU Copy**	**ROS Type**	**ROS Copy**	**Time** (ms)		**SD** (ms)	**Overhead**
2 nodes (fig. 3.10)	CUDA-SC	Topic	ROS-SC	22 825.2	±	5 763.5	2 640%
2 nodes (fig. 3.11)	CUDA-SC	Service	ROS-SC	22 715.0	±	3 929.8	2 627%
2 nodes (fig. 3.12)	CUDA-SC	Topic	ROS-ZC	1 056.0	±	32.2	27%
2 nodes (fig. 3.12)	CUDA-ZC	Topic	ROS-ZC	10 254.2	±	189.0	21%
2 nodes (fig. 3.13)	CUDA-SC	Topic	ROS-SHM-ZC	852.6	±	7.7	2%
2 nodes (fig. 3.14)	CUDA-ZC	Topic	ROS-SHM-ZC	9 474.0	±	226.3	11%
3 nodes (fig. 3.15)	CUDA-SC	Topic	ROS-SC	6 334.2	±	3 562.1	660%
3 nodes (fig. 3.15)	CUDA-SC	Service	ROS-SC	6 755.5	±	4 589.0	711%
3 nodes (fig. 3.16)	CUDA-SC	Topic	ROS-ZC	1 087.6	±	39.4	31%
3 nodes (fig. 3.17)	CUDA-SC	Topic	ROS-SHM-ZC	862.7	±	9.7	4%
3 nodes (fig. 3.17)	CUDA-ZC	Topic	ROS-SHM-ZC	9 408.1	±	213.0	11%
5 nodes (fig. 3.15)	CUDA-SC	Topic	ROS-SC	12 234.2	±	3 487.9	1 369%
5 nodes (fig. 3.17)	CUDA-SC	Topic	ROS-SHM-ZC	1 737.1	±	9.1	109%
5 nodes (fig. 3.17)	CUDA-ZC	Topic	ROS-SHM-ZC	11 381.6	±	65.6	34%

Table 4.11: Results for the Cache Benchmark on the NVIDIA Jetson TX2. The reference execution time for CUDA-SC is 833.0 ms and 8 509.1 ms for CUDA-ZC.

less performance than the Xavier. In the Jetson TX2, the cache benchmark with the CUDA-ZC model, which disables the LLC of CPU and GPU, provides the worst performance. The concurrent benchmark, even with a light cache workload and concurrent execution, still leads to performance loss. The I/O coherency implemented in hardware in the Xavier reduces this performance loss. In contrast, such a performance loss is extremely evident in the TX2. The concurrent benchmark shows how lighter cache usage combined with I/O coherency and concurrent executions, thanks to CUDA-ZC, allows for significant performance improvements compared to CUDA-SC.

The results of Table 4.10 will be used as reference times to calculate the *overhead* of all the tested ROS-based configurations.

We present the results obtained with the proposed ROS compliant models into four tables. Tables 4.11 and 4.12 present the results on the Jetson TX2 with the cache and concurrent benchmarks, respectively. Tables 4.13 and 4.14 present the results on the Jetson Xavier with the cache and concurrent benchmarks, respectively. In the tables, the first column indicates the proposed ROS-compliant communication model, with a reference to the corresponding figure in the methodology section. The second column indicates the CPU-iGPU CUDA communication type. The third and fourth columns indicate the ROS communication protocol. The fifth and sixth show the average run time of 30 executions and the standard deviation respectively. The last column shows the overhead.

In Table 4.11, the first two rows show how a *standard* implementation of the ROS protocol (i.e., ROS-SC) decreases the overall performance to unacceptable levels, for both services and topics. The ROS-ZC standard shows noticeable improvements compared to ROS-SC, by reducing the overhead by a factor of ≈ 100. The proposed ROS-SHM-ZC improves even further by reducing the overhead down to 2% and 11% when compared to CUDA-SC and CUDA-ZC respectively.

Moving from two to three nodes, we found a performance improvement by combining CUDA-SC with ROS-SC in both topics and services. While there is a slight overhead caused by the addition of the third node, these models lead to better overall performance due to the easier synchronization between nodes. The standard deviation shows high variance between results, suggesting a communication bottleneck that can be exacerbated by the system network conditions, outside of the programmer's control. This communication bottleneck is greatly reduced in the ROS-ZC and ROS-SHM-ZC configurations, thanks to the reduced size of the messages. The third node overhead still reduces the performance when compared to two nodes. Overall, the difference between two nodes and three nodes in the zero-copy configurations is negligible.

The same considerations hold for the five node architecture, which also proves to be very costly due to the additional nodes. Nevertheless, it is still better than the two nodes ROS-

Arch.	**GPU Copy**	**ROS Type**	**ROS Copy**	**Time** (ms)	**SD** (ms)	**Overhead**
2 nodes (fig. 3.10)	CUDA-SC	Topic	ROS-SC	4 666.0	± 68.9	343%
2 nodes (fig. 3.11)	CUDA-SC	Service	ROS-SC	4 930.0	± 73.3	368%
2 nodes (fig. 3.12)	CUDA-SC	Topic	ROS-ZC	1 584.6	± 46.3	50%
2 nodes (fig. 3.12)	CUDA-ZC	Topic	ROS-ZC	1 910.6	± 73.1	45%
2 nodes (fig. 3.13)	CUDA-SC	Topic	ROS-SHM-ZC	1 046.1	± 82.3	-1%
2 nodes (fig. 3.14)	CUDA-ZC	Topic	ROS-SHM-ZC	1 203.2	± 47.2	-9%
3 nodes (fig. 3.15)	CUDA-SC	Topic	ROS-SC	8 244.6	± 2 938.4	683%
3 nodes (fig. 3.15)	CUDA-SC	Service	ROS-SC	8 505.7	± 767.6	708%
3 nodes (fig. 3.16)	CUDA-SC	Topic	ROS-ZC	2 261.7	± 47.1	115%
3 nodes (fig. 3.17)	CUDA-SC	Topic	ROS-SHM-ZC	997.9	± 75.6	-5%
3 nodes (fig. 3.17)	CUDA-ZC	Topic	ROS-SHM-ZC	1 172.5	± 41.5	-11%
5 nodes (fig. 3.15)	CUDA-SC	Topic	ROS-SC	41 773.4	± 12 118.6	3 867%
5 nodes (fig. 3.17)	CUDA-SC	Topic	ROS-SHM-ZC	1 923.	± 89.7	131%
5 nodes (fig. 3.17)	CUDA-ZC	Topic	ROS-SHM-ZC	1 978.3	± 131.9	50%

Table 4.12: Results for the Concurrent Benchmark on the NVIDIA Jetson TX2. The reference execution time for CUDA-SC is 1 053.0 ms and 1 316.1 ms for CUDA-ZC.

Arch.	**GPU Copy**	**ROS Type**	**ROS Copy**	**Time** (ms)	**SD** (ms)	**Overhead**
2 nodes (fig. 3.10)	CUDA-SC	Topic	ROS-SC	8 371.3	± 3 210.8	3 930%
2 nodes (fig. 3.11)	CUDA-SC	Service	ROS-SC	10 385.2	± 3 423.4	4 900%
2 nodes (fig. 3.12)	CUDA-SC	Topic	ROS-ZC	338.1	± 48.8	63%
2 nodes (fig. 3.12)	CUDA-ZC	Topic	ROS-ZC	653.6	± 35.4	167%
2 nodes (fig. 3.13)	CUDA-SC	Topic	ROS-SHM-ZC	230.0	± 15.5	11%
2 nodes (fig. 3.14)	CUDA-ZC	Topic	ROS-SHM-ZC	251.9	± 25.3	3%
3 nodes (fig. 3.15)	CUDA-SC	Topic	ROS-SC	5 139.1	± 3 484.6	2 374%
3 nodes (fig. 3.15)	CUDA-SC	Service	ROS-SC	6 744.4	± 4 367.4	3 147%
3 nodes (fig. 3.16)	CUDA-SC	Topic	ROS-ZC	333.9	± 30.5	61%
3 nodes (fig. 3.17)	CUDA-SC	Topic	ROS-SHM-ZC	236.9	± 13.1	14%
3 nodes (fig. 3.17)	CUDA-ZC	Topic	ROS-SHM-ZC	400.9	± 20.8	64%
5 nodes (fig. 3.15)	CUDA-SC	Topic	ROS-SC	7 466.4	± 5 225.2	3 495%
5 nodes (fig. 3.17)	CUDA-SC	Topic	ROS-SHM-ZC	492.5	± 15.6	137%
5 nodes (fig. 3.17)	CUDA-ZC	Topic	ROS-SHM-ZC	847.6	± 36.4	246%

Table 4.13: Results for the Cache Benchmark on NVIDIA Jetson Xavier. The reference execution time for CUDA-SC is 207.7 ms and 244.7 ms for CUDA-ZC.

SC configuration, overhead-wise. In this instance, for the sake of clarity, the only reported solutions are the proposed ROS-SHM-ZC. The overhead obtained, while not negligible, is still limited, especially in the CUDA-ZC configuration.

Considering the concurrent benchmark on the Jetson TX2 (Table 4.12), we found performance results similar to the cache benchmark but with lower overall overheads in the two nodes ROS-SC configurations. In this benchmark, the service mechanism of ROS is consistently slower than the topics mechanism. The speedups obtained with ROS-SHM-ZC are notable. With CUDA-SC they are within a margin of error at −1% with two nodes and −5% with three nodes. With CUDA-ZC they are more significant at −9% and −11% with two and three nodes, respectively. This is thanks to the caches not being disabled on the CPU. As the hardware is not I/O coherent, we had to manually handle the consistency of the data, and because the cache utilization is low but still present, the CPU computations have better performance. This allows for an improvement when compared to the original CUDA-ZC solution of Table 4.10. Nevertheless, this also means that, while there should be no copies in these two configurations, the combination of the ROS mechanisms with the CUDA communication model (CUDA-ZC) actually forces explicit data copies. One from CPU to iGPU and one in the opposite direction. These copies are responsible for the loss in performance when comparing ROS-SHM-ZC + CUDA-ZC to ROS-SHM-ZC + CUDA-SC.

Arch.	**GPU Copy**	**ROS Type**	**ROS Copy**	**Time** (ms)	**SD** (ms)	**Overhead**
2 nodes (fig. 3.10)	CUDA-SC	Topic	ROS-SC	3 501.0	± 3 052.9	818%
2 nodes (fig. 3.11)	CUDA-SC	Service	ROS-SC	5 307.0	± 5 841.7	1 292%
2 nodes (fig. 3.12)	CUDA-SC	Topic	ROS-ZC	658.5	± 41.7	73%
2 nodes (fig. 3.12)	CUDA-ZC	Topic	ROS-ZC	672.4	± 34.9	162%
2 nodes (fig. 3.13)	CUDA-SC	Topic	ROS-SHM-ZC	403.3	± 43.0	6%
2 nodes (fig. 3.14)	CUDA-ZC	Topic	ROS-SHM-ZC	266.7	± 23.4	4%
3 nodes (fig. 3.15)	CUDA-SC	Topic	ROS-SC	17 054.5	± 5 763.9	4 374%
3 nodes (fig. 3.15)	CUDA-SC	Service	ROS-SC	25 033.2	± 14 269.0	6 467%
3 nodes (fig. 3.16)	CUDA-SC	Topic	ROS-ZC	655.7	± 42.1	72%
3 nodes (fig. 3.17)	CUDA-SC	Topic	ROS-SHM-ZC	408.0	± 56.9	7%
3 nodes (fig. 3.17)	CUDA-ZC	Topic	ROS-SHM-ZC	451.3	± 62.6	76%
5 nodes (fig. 3.15)	CUDA-SC	Topic	ROS-SC	43 808.4	± 8 735.0	11 392%
5 nodes (fig. 3.17)	CUDA-SC	Topic	ROS-SHM-ZC	761.9	± 86.5	100%
5 nodes (fig. 3.17)	CUDA-ZC	Topic	ROS-SHM-ZC	859.0	± 66.5	234%

Table 4.14: Results for the Concurrent Benchmark on the NVIDIA Jetson Xavier. The reference execution time for CUDA-SC is 381.2 ms and 256.9 ms for CUDA-ZC.

When analyzing the results obtained with the Jetson Xavier, (Tables 4.13 and 4.14), the ROS-SC model is much slower compared to the reference performance and services are consistently slower than topics. ROS-ZC is faster than ROS-SC by a wide margin. It is also important to note that, for ROS-SHM-ZC, there are no negative overheads on the Xavier. This is due to the hardware I/O coherency, which already extrapolates the maximum performance for CUDA-ZC and the manual handling of the coherency does not lead to performance improvements. In both benchmarks, the three node configurations for ROS-SHM-ZC show higher overhead compared to the two node variants, highlighting the higher cost of the third node and, thus, leading to a significant performance loss when compared to the optimal performance of the native configuration.

4.2 Containerization and orchestration on heterogeneous Edge-Cloud computing architectures

This Section focuses on the results obtained for the optimizations related to the porting and application of containerization and orchestration components to edge-cloud and edge environments, specifically:

- Section 4.2.1 analyzes the improvements obtained with the containerization and orchestration methodology to a robotic real-case of study (see Section 3.2.1).
- Section 4.2.2 analyzes the performance of Kubernetes on RISC-V and compare it to an equivalent ARM-based platform (see Section 3.2.2).

4.2.1 Extending Docker and Kubernetes for ROS-compliant containerized robotic applications

We evaluated the proposed design methodology to program the mission of a Robotnik RB-Kairos mobile robot in an industrial agile production chain. Such a mobile robot consists of a skid-steering platform equipped with an Universal Robots UR5 manipulator and a Schunk WSG50 end-effector for grasping. The robot is also equipped with different sensors including two Sick S300 laser scanners for localization, and an RGB-D Intel RealSense D415 camera. The computing HW architecture consists of four edge devices installed on-board. One Main Control Board (MCB) is equipped with an Intel i7 9700 3.0 GHz limited to 4 cores (20W power constraint), 4GB of RAM, and Ubuntu 16.04 OS with ROS Kinetic. Two additional devices are installed for the real-time control SW of the UR5 manipulator and WSG50 gripper, respectively. They run a real-time Ubuntu OS and real-time kernels that communicate with the driver nodes for low-level tasks. They are not available for the

Node type	L1	L2	L3
Mission	1	1	1
Macro-functionality	4	4	4
Functionality/controllers	25	25	52
Drivers	2	2	12
Total:	32	32	69

Table 4.15: ROS-node classification and abstraction levels.

orchestrator. The fourth edge device consists of an Nvidia Jetson Nano with JetPack 4.2. The on-board devices are connected through a local (Ethernet) network, which is controlled by an on-board router. The computing HW architecture also includes an external server equipped with an Intel i5-7400 3.5 GHz, 8GB of RAM, Ubuntu 18.04. The server is connected to the on-board devices through a 300 Mbps wireless network.

We configured a K3s cluster, version 1.20.4+k3s1, on the MCB, on the Jetson Nano and on the external server. The external server also runs the K3s master agent.

We started from the ROS-compliant SW application implementing the robot mission. It consists of four main tasks that implement the interaction of the mobile robot with an industrial agile production chain. The init task initializes the robot on a starting position, aligns with the production chain, and waits in idle for the start of the following tasks. The first task consists of a series of arm and gripper operations to grasp production pieces from the conveyor belt through the gripper and to move them to the cargo bay. The second task implements movements of the robot base, arm, and gripper. It moves the base towards the storage area, then it unloads the pieces from the cargo bay. The third and last task implements the movement of the base and the arm. The robot returns to the production line and re-aligns itself with the conveyor belt. It then moves the arm in the ready position.

ROS node classification and abstraction levels

The final SW application deployed on the robot consists of 69 ROS nodes. 32 nodes implement the robot mission (i.e., the four tasks described above), macro-functionality (i.e., local/global planner, localization and mapping, arm motion planner), functionality/controllers, and drivers that are used for the simulation and automatically offloaded on the target HW architecture for the system deployment. We extended the starting SW with a further macro-functionality to implement a visual SLAM, i.e. an ORB-SLAM [192]. Further 27 nodes implement low-level controllers and drivers which are used only on the real HW at the deployment phase (L3). Table 4.15 summarizes the node classification over the abstraction levels.

Clustering of ROS nodes and inheritance-based container generation

Considering the available system memory and the container overhead, we run the algorithm presented in Section 3.2.1 to cluster the nodes into 11 containers. Table 4.16 reports the generated clusters (i.e., image name) and the corresponding image size without and with the proposed inheritance-based optimization. The last five images represent the overlapping packages, which are included in every image without the optimization, while they are shared by the 11 images in the optimized version. Overall, we found that the proposed inheritance-based optimization led to a reduction of the memory footprint of 83.2% when all images are downloaded on the same device.

At L3, we analysed the impact of the containerization process as well as the impact of the node orchestration system in terms of memory and CPU overhead on the target HW architecture.

As the cluster includes devices with different architectures (i.e., AMD64 and ARM64), we created container images that are compatible with both types of architectures using Docker and the `buildx` command for `linux/arm64/v8` and `linux/amd64`.

Image name	Size (GB)	Size optimized (GB)
mission-rb	4.087	0.367
hmi-webserver-rb	0.699	0.567
arm-rb	4.343	0.623
rostful-rb	3.777	0.129
local-control-rb	3.967	0.320
rms-rb	3.769	0.134
perception-rb	3.770	0.135
map-nav-rb	3.886	0.251
navigation-rb	3.771	0.135
main-rb	4.136	0.416
hmi-rb	3.872	0.237
wsg50-ros-pkgs	–	0.379
summit-xl-bringup	–	0.052
summit-xl-robot-local-control	–	0.000
summit-xl-navigation	–	0.000
base	–	3.634
Total:	39.897	6.705

Table 4.16: Docker image sizes on disk before and after the proposed optimization.

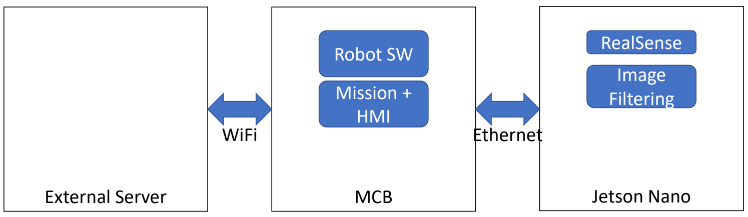


Fig. 4.8: native configuration.

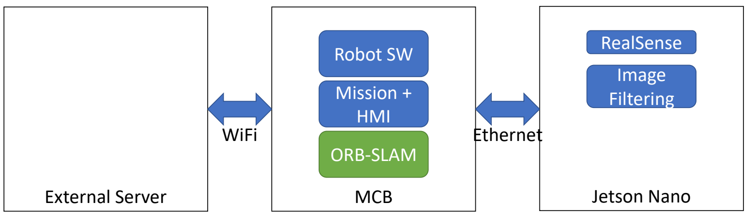


Fig. 4.9: native + orb configuration.

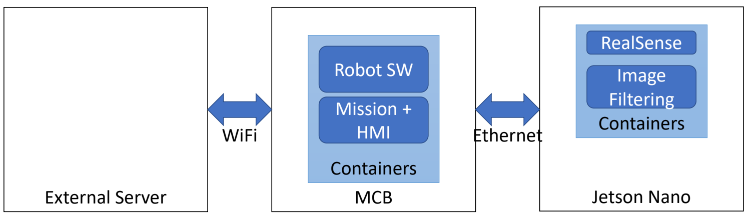


Fig. 4.10: cont(native) configuration.

We developed six different versions of the SW. Then, thanks to the containerization process, we rapidly analysed the containerized versions with different orchestration alternatives across the distributed computing HW devices.

Figures 4.8 through 4.13, summarize the configurations. The robot's SW is categorized into `Robot SW` and `Mission + HMI`, where the latter represents the containers `demo-rb`, `hmi-rb` and `hmi-webserver-rb` from Table 4.16, while the former represents the remaining containers listed in the same table. The ORB-SLAM has its own block.

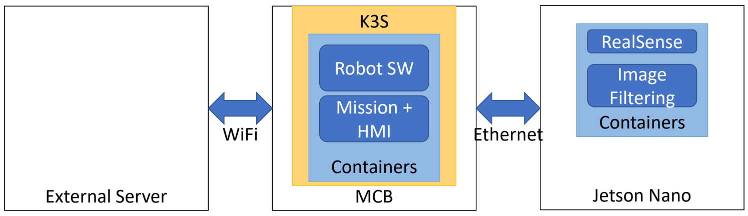


Fig. 4.11: k3s-a(native) configuration.

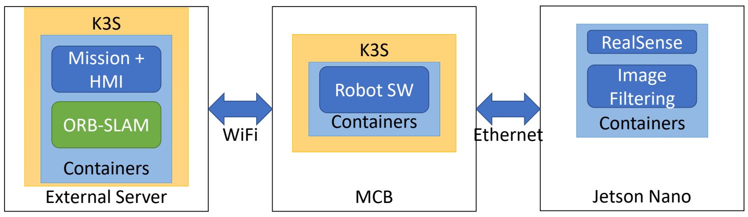


Fig. 4.12: k3s-b(native + orb) configuration.

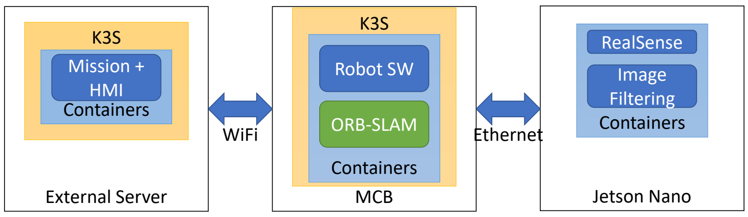


Fig. 4.13: k3s-c(native + orb) configuration.

Fig. 4.8 shows the starting point, which consists of the software stack without containerization (i.e., native) run on the MCB. Fig. 4.9 represents the native stack extended with the ORB-SLAM task (i.e., nat.+orb) on the MCB. Fig. 4.10 depicts the native SW application containerized through Docker and run on the MCB through the Docker runtime (i.e., cont.). We then extended the containerized version by including the k3s orchestrator (k3s-a), which maps the whole SW on the MCB (Fig. 4.11). In a further configuration (k3s-b), the orchestration system maps the mission nodes, the ORB-SLAM, and the human-machine-interface (HMI), which includes web and mysql servers, into the external server, while the rest of the nodes are mapped on the MCB (Fig. 4.12). In the last configuration (k3s-c), the orchestrator maps the HMI, the web and mysql servers, and the mission nodes into the external server, while the rest of the nodes including the ORB-SLAM are mapped on the MCB (Fig 4.13). In all configurations, the nodes implementing the video frame control and image filtering are mapped on the edge device (i.e., Jetson Nano) connected to the RGB-D camera.

Table 4.17 summarizes, for each SW version and HW/SW configuration, and for each mission task, the memory footprint and the overhead involved by containers and orchestrator. The table reports the total overhead, the overhead per container, and the overhead in terms of average CPU usage. The overhead per container has been calculated by considering 11 containers running on the containerized and k3s-a configurations (due to the omitted ORB-SLAM), while 12 in all the other versions.

We found that the system incurs, on average, in an overhead of 29 MB of system memory per container when using the Docker runtime (cont. rows). The orchestrator involves an additional 34.5% overhead per container (around 39 MB per container). In addition, the k3s master agent, which runs on the external server, requires 508.1 MB on average.

In general, we found that the CPU overhead involved by containerization and orchestration is negligible. The measured variance on the CPU usage is in line with the fluctuations due to the operating system, considering that the robot SW application runs on a standard installation of Ubuntu 16.04. We also found that K3s is consistently more expensive

Task	configuration	Memory Usage (MB)	Overhead (MB, total)	Overhead (MB, per cont.)	Avg CPU usage (%)
init	native	3453.5	–	–	64.4
	nat. + orb	3905.6	–	–	91.0
	cont(native)	3771.1	317.6	28.9	55.6
	k3s-a(native)	3880.0	426.5	38.8	60.8
	k3s-b(native+orb)	2684.8	–	–	60.0
	k3s-c(native+orb)	3136.9	–	–	93.4
1	native	3454.4	–	–	63.2
	native + orb	3906.5	–	–	89.8
	cont(native)	3773.0	318.6	29.0	55.6
	k3s-a(native)	3881.9	427.5	38.9	59.4
	k3s-b(native+orb)	2685.4	–	–	58.3
	k3s-c(native+orb)	3137.5	–	–	97.1
2	native	3454.7	–	–	62.6
	native + orb	3906.8	–	–	89.2
	cont(native)	3773.8	319.1	29.0	72.8
	k3s-a(native)	3885.2	430.5	39.1	63.6
	k3s-b(native+orb)	2685.8	–	–	58.6
	k3s-c(native+orb)	3137.9	–	–	91.6
3	native	3454.1	–	–	63.8
	native + orb	3906.2	–	–	90.4
	cont(native)	3775.2	321.1	29.2	58.4
	k3s-a(native)	3886.2	432.1	39.3	61.4
	k3s-b(native+orb)	2684.2	–	–	59.6
	k3s-c(native+orb)	3136.3	–	–	94.5

Table 4.17: Resource usage on the robot MCB with all proposed configurations.

Configuration	Computation time (ms)	Network latency (ms)	Supported FPS	Network usage (Mbps)
ORB-SLAM	34.6	–	29.6	–
native + orb	58.3	–	17.2	–
cont(native + orb)	–	–	–	–
k3s-a(native + orb)	–	–	–	–
k3s-b(native + orb)	33.5	3.5	27.0	4.1
k3s-c(native + orb)	60.0	–	16.7	0.0

Table 4.18: ORB-SLAM supported frame rates.

in terms of CPU consumption, with 4% overhead on average, when the CPU is overcharged (e.g., k3s-c in our experimental results). The overhead caused by k3s is negligible in all the other cases.

We finally evaluated the real-time constraint on the ORB-SLAM task by considering 20 FPS as minimum supported rate for the 20Hz RGB-D camera input stream. We tested six different configurations at L3 (see Table 4.18). The first implements the only ORB-SLAM application run on the MCB, in order to measure the maximum FPS supported by the original code. Then, we measured the ORB-SLAM FPS when it is run concurrently with the rest of the nodes implementing the native application on the same edge device (MCB). The results show that, even though the overall SW correctly executes, the CPU cannot maintain a real-time behavior (i.e., 17.2 FPS). The containerization process leads to a memory footprint larger than total memory available on the MCB and, thus, to out-of-memory issues. As a consequence, the orchestration system cannot run when the native SW and the ORB-SLAM are mapped on the MCB (k3s-a in Table 4.18).

We then set the orchestrator to map the containerized native SW on the MCB and the ORB-SLAM on the external server (k3s-b in Table 4.18). In this configuration, we also considered the overhead involved in the network to send the image data flow from the RGB-D camera to the ORB-SLAM. To achieve the lowest possible network latency, the ROS node implementing the image reading and control sends compressed images from the edge device to the external server, in which a republish node of the *image_transport* ROS package decompresses the image as input for the ORB-SLAM. Such a data transfer requires, on average, 4.1 Mbps of bandwidth compared with 47.2 Mbps when uncompressed. This network usage also adds latency to the node communication as the idle round trip time from the server to the robot increased from 3ms, with a standard deviation of 0.7ms, to a round trip time of 3.5ms, with a standard deviation of 1.2ms. This translates into an elaboration time of the ORB-SLAM of 37ms per frame (27FPS) for the k3s-b configuration. We finally evaluated the configuration in which the native and ORB-SLAM applications run on the MCB, while the mission+HMI nodes are mapped on the external server (k3s-c). The results show that, even though the resource hungry mission+HMI nodes are moved to the external server, the rest of the nodes and the ORB-SLAM suffer from resource (CPU) contention and communication overhead when running concurrently. This leads the ORB-SLAM application to not satisfy the real-time constraint. In conclusion, the k3s-b configuration guarantees the real-time performance of the ORB-SLAM at the cost of a negligible overhead on the network.

4.2.2 Expanding the Edge-Cloud computing continuum to the RISC-V open hardware architecture

We tested a set of benchmarks to quantify the containerization impact on the performance of the RISC-V architecture. We used *sysbench* for CPU integer performance, the *Stream* benchmark for system memory throughput, the *Phoronix test suite* for CPU-related benchmarks and finally, a sequence of system-related tests to verify the OS performance. We run each benchmark both natively and through KubeEdge. After benchmarking each test on the RISC-V board, we also verified the performance overhead on an ARM-based board, which allowed us to compare the results and verify if there was any odd behavior on the new architecture.

To run the containerized benchmarks, we configured a Kubernetes cluster with the master running on an x86 device. Then, we connected two boards with KubeEdge. The first board is a SiFive HiFive Unmatched, part of the Monte Cimone cluster, and has a RISC-V architecture. It runs Ubuntu 21.10 with Linux Kernel 5.11 on 4 CPU cores running at 1GHz and 16GB of system memory. The second board is an ARM-based Jetson Xavier AGX running Ubuntu 20.04 with Linux kernel 5.10 on 8 CPU cores running at 2.3GHz and 16GB of system memory. We configured the Jetson to run with only 4 cores at 1.2GHz, achieving a comparable CPU power target to the RISC-V board (i.e., $\approx 5W$).

In our analysis, we focused on a single node because our testing methodology is specifically designed to assess the architectural impact of containerization, regardless of orchestration policies. Because of this, the network impact of orchestration has not been measured as it would not influence the outcome of this analysis. Still, it is important to note that the results obtained with the profiling we conducted are not restricted by the use of a single node and can be applied in broader scenarios.

Benchmark results

Sysbench. Table 4.19 shows the results obtained with sysbench, averaging 15 runs. The benchmark was run with 4 threads, calculating up to 1 million primes with a 1-hour time limit. The Unmatched board exhibits significantly lower performance compared to the Jeston, but it experiences less overhead from containerization.
Stream. Table 4.19 also shows the results for the STREAM benchmark run on the system memory, averaging 15 runs. The test was configured to run on 1.8GB of data with 4 threads. The memory subsystem of the Unmatched board is composed of 16GB of DDR4 memory running in dual channel with a 64bit bus at 1866MT/s, resulting in a theoretical maximum

Table 4.19: STREAM benchmark results with native and KubeEdge configurations on both RISC-V and ARM64. ▲ higher is better, ▼ lower is better.

Suite	Test	RISC-V			ARM64			Unit
		Native	KubeEdge	Overhead	Native	KubeEdge	Overhead	(▲▼)
Sysbench	CPU	2.47 ± 0.39%	2.46 ± 0.19%	-0.4%	6.40 ± 0.06%	6.17 ± 1.72%	-3.7%	event/s ▲
Stream	Copy	1 294 ± 0.28%	1 274 ± 2.19%	-1.5%	22 765 ± 0.51%	20 500 ± 0.15%	-10.0%	MiB/s ▲
	Scale	1 079 ± 0.50%	1 096 ± 1.01%	1.6%	23 929 ± 0.42%	22 229 ± 0.09%	-7.1%	MiB/s ▲
	Add	1 181 ± 0.20%	1 180 ± 0.89%	-0.1%	24 866 ± 0.10%	24 719 ± 1.46%	-0.6%	MiB/s ▲
	Triad	1 165 ± 0.18%	1 191 ± 1.94%	2.2%	24 499 ± 0.25%	25 044 ± 0.16%	2.2%	MiB/s ▲
			Average:	0.4%		*Average*:	-3.8%	

Table 4.20: Phoronix test suite results with native and KubeEdge configurations on both RISC-V and ARM64. ▲ higher is better, ▼ lower is better. The results include the relative standard deviation.

Suite	Test	RISC-V			ARM64			Unit
		Native	KubeEdge	Overhead	Native	KubeEdge	Overhead	(▲▼)
Rodinia	LavaMD	12 298 ± 1.05%	13 462 ± 0.41%	-8.7%	1 731 ± 2.41%	1 742 ± 2.05%	-0.6%	s ▼
x265 3.4 [1e-3]	Bos. 1080p	150 ± 0.00%	140 ± 0.00%	-6.7%	1 240 ± 0.00%	1 230 ± 0.24%	-0.8%	fps ▲
	Bos. 4K	30 ± 0.00%	30 ± 0.00%	0.0%	340 ± 0.00%	330 ± 0.00%	-2.9%	fps ▲
7-Zip	Comp.	1 782 ± 0.80%	1 805 ± 0.77%	1.3%	7 972 ± 2.63%	6 983 ± 3.60%	-12.4%	MIPS ▲
	Decomp.	3 433 ± 0.54%	3 430 ± 0.13%	-0.1%	5 921 ± 1.36%	5 309 ± 1.10%	-10.3%	MIPS ▲
POV-Ray	Trace	2 948 ± 1.33%	2 948 ± 1.79%	-0.0%	541 ± 2.38%	541 ± 2.38%	0.0%	s ▼
OpenSSL	SHA256	66.1 ± 0.38%	63.1 ± 5.89%	-4.7%	1 589.5 ± 2.45%	1 593.1 ± 0.69%	0.2%	MB/s ▲
	SHA512	92.7 ± 1.08%	90.4 ± 0.63%	-2.5%	398.5 ± 0.38%	387.1 ± 0.40%	-2.9%	MB/s ▲
	RSA4096_s	41 ± 0.00%	42 ± 0.73%	0.5%	155 ± 0.47%	147 ± 0.48%	-5.2%	sign/s ▲
	RSA4096_v	3 139 ± 0.28%	3 124 ± 0.69%	-0.5%	11 01 ± 0.01%	10 532 ± 0.05%	-4.4%	verify/s ▲
	AES-128	65.0 ± 0.37%	64.1 ± 0.12%	-1.5%	4 964.8 ± 0.07%	4 866.4 ± 0.09%	-2.0%	MB/s ▲
	AES-256	53.9 ± 0.55%	53.2 ± 0.62%	-1.3%	3 677.2 ± 0.01%	4 027.0 ± 0.05%	9.5%	MB/s ▲
	ChaCha20	231.1 ± 0.27%	227.3 ± 0.09%	-1.7%	2 213.2 ± 0.01%	2 080.9 ± 0.08%	-6.0%	MB/s ▲
	Poly1305	168.0 ± 0.26%	165.7 ± 0.33%	-1.4%	1 497.4 ± 0.04%	1 415.9 ± 0.03%	-5.4%	MB/s ▲
			Average:	-1.9%		*Average*:	-3.1%	

bandwidth of ≈ 30GB/s. The Jetson board has a much more sophisticated memory subsystem, composed of 16GB of LPDDR4x memory running in dual channel with a 256-bit bus at 1333MT/s, resulting in a theoretical maximum bandwidth of ≈ 86GB/s. As expected, due to these significant hardware differences, we observed that the RISC-V board is much slower in this test compared to the Jetson. However, the KubeEdge overhead is lower on the RISC-V architecture across all memory tests.

Phoronix. Table 4.20 shows the results obtained with the Phoronix test suite. We used the following benchmarks, and each run 15 times:

- Rodinia: this suite is focused on accelerator-based computing. We picked the LavaMD test based on OpenMP to benchmark multicore performance;
- x265: a CPU-based encoding test;
- 7-Zip: uses the integrated compression and decompression benchmarks;
- POV-Ray: Persistence of Vision Raytracer, it creates 3D graphics using ray tracing;
- OpenSSL: tests SSL (Secure Sockets Layer) and TLS (Transport Layer Security) protocols, including encryption and hashing functions.

This suite presents similar results to the other benchmarks. The Jetson is much faster, but the container overhead is, on average, 37.5% smaller on the RISC-V Unmatched board.

OSBench, IPC-Benchmark and stress-ng. Table 4.21 shows the results for OSBench, IPC-Benchmark and stress-ng, with each test run 15 times. OSBench performs a series of activities that are connected to running applications. The IPC benchmark tests the interprocess communication speed and bandwidth. Finally, stress-ng contains a multitude of tests, and those targeting the operating system were chosen.

Table 4.21: OSBench, IPC-Benchmark and stress-ng results with native and KubeEdge configurations, on both RISC-V and ARM64. ▲ higher is better, ▼ lower is better. The results include the relative standard deviation.

Suite	Test	RISC-V			ARM64			Unit
		Native	KubeEdge	Overhead	Native	KubeEdge	Overhead	(▲▼)
OSBench	Create Files	426 ± 5.48%	596 ± 4.32%	-28.4%	183 ± 2.28%	279 ± 4.71%	-34.3%	μs ▼
	↳ mount	-	425 ± 5.42%	0.2%	-	190 ± 3.64%	-3.7%	μs ▼
	C. Thread	197 ± 2.15%	212 ± 0.28%	-7.1%	136 ± 2.27%	171 ± 2.06%	-20.6%	μs ▼
	C. Processes	402 ± 2.37%	452 ± 2.47%	-10.9%	262 ± 1.59%	272 ± 1.05%	-3.7%	μs ▼
	Launch Prog.	618 ± 1.07%	673 ± 0.23%	-8.2%	924 ± 0.71%	818 ± 1.32%	13.0%	μs ▼
	Malloc	1 399 ± 0.43%	1 424 ± 1.19%	-1.8%	565 ± 1.49%	542 ± 0.23%	4.2%	μs ▼
IPC bench.	TCP Socket	117 705 ± 5.01%	112 507 ± 14.08%	-4.4%	137 263 ± 2.35%	136 135 ± 0.99%	-0.8%	msg/s ▲
	PIPE Un.	270 793 ± 0.36%	274 776 ± 1.96%	1.5%	154 689 ± 0.25%	152 937 ± 0.38%	-1.1%	msg/s ▲
	PIPE FIFO	269 931 ± 1.24%	266 265 ± 1.35%	-1.4%	155 666 ± 0.33%	157 966 ± 0.47%	1.5%	msg/s ▲
	Unix Socket	95 177 ± 2.17%	95 469 ± 0.94%	0.3%	90 560 ± 1.13%	92 266 ± 1.37%	1.9%	msg/s ▲
Stress-ng	Mutex	158 964 ± 2.16%	135 805 ± 0.52%	-14.6%	57 472 ± 4.30%	56 235 ± 2.49%	-2.2%	ops/s ▲
	Malloc	99 067 ± 0.17%	97 948 ± 0.98%	-1.1%	54 320 ± 1.16%	52 001 ± 0.22%	-4.3%	ops/s ▲
	Forking	1 851 ± 2.34%	2 020 ± 1.98%	9.1%	1 601 ± 3.07%	1 684 ± 12.59%	5.2%	ops/s ▲
	Pthread	2 690 ± 1.07%	2 340 ± 0.53%	-13.0%	3 633 ± 2.06%	3 473 ± 0.91%	-4.4%	ops/s ▲
	CPU cache	23 254 ± 4.07%	23 549 ± 8.91%	1.3%	182 042 ± 1.59%	177 345 ± 1.07%	-2.6%	ops/s ▲
	Semaphores	845 130 ± 2.05%	793 821 ± 2.37%	-6.1%	201 253 ± 9.87%	194 155 ± 5.54%	-3.5%	ops/s ▲
	Matrix Math	518 ± 0.22%	507 ± 0.17%	-2.1%	3 698 ± 0.15%	3 770 ± 0.05%	1.9%	ops/s ▲
	Vector Math	348 ± 0.33%	354 ± 0.02%	1.8%	6 484 ± 0.53%	7 084 ± 0.69%	9.3%	ops/s ▲
	Functions	2 294 ± 0.36%	2 249 ± 0.36%	-2.0%	18 766 ± 2.58%	16 004 ± 2.43%	-14.7%	ops/s ▲
	Cntx switch	84 110 ± 4.90%	66 082 ± 5.53%	-21.4%	47 990 ± 2.48%	47 865 ± 2.39%	-0.3%	ops/s ▲
			Average:	−5.4%		*Average*:	−2.9%	

The most interesting results are related to the file creation (i.e., the first two rows of Table 4.21) and process-related activities, i.e., launch programs, create processes, pthread, and context switching. The former because it shows how the file system architecture of containers creates significant overhead in I/O-based operations and when excluded by mounting a native folder of the underlying file system inside the container, this overhead is not present anymore. This applies to RISC-V and to some extent to ARM64, where there is still some overhead even with the native mount. The latter shows how RISC-V incurs significant overhead when operations related to processes or threads are made. This is especially true when considering context switching, where RISC-V loses 21.4% of its performance.

To find the root cause of this delay on RISC-V, we run the `stress-ng` context switch benchmark again, along with the `perf` profiler. We found that system calls were taking significantly longer, up to 40% more time. This additional delay is probably caused by the container runtime system call interception procedure.

When a containerized process makes a system call, it is intercepted by the container runtime, which verifies permissions and resources before initiating the system call. To verify this theory, we formulated the following hypothesis: if the overhead is due to the container runtime rather than additional data structures from namespaces and cgroups, and if we manually associate a native process with the same namespaces and cgroups as a KubeEdge process, then:

1. system call speed should be comparable to a native execution;
2. cgroups/namespaces should work exactly like in KubeEdge.

To verify these hypotheses, we conducted the following experiment. We used two test applications: the first performs around 10 million system calls and calculates their average time, which should verify the initial hypothesis. The second test allocates 1GB of system memory using `malloc` with a cgroup limitation of 128MB, checking if the cgroup works properly by killing the application when it exceeds the limit. We used three configurations: native as a reference, containerized using KubeEdge, and manually associating the processes within namespaces/cgroups using `nsenter` and the `cgroup` parameter.

Table 4.22 shows the results. Native system calls and manual namespaces/cgroups have the same execution time, while there is a slowdown when executing them from KubeEdge. However, namespaces and cgroups were working correctly, as the memory allocation application was killed once it exceeded 128MB. Therefore, the overhead is definitely not caused by

Table 4.22: Overhead analysis for RISC-V system calls under KubeEdge.

	Native	Manual ns/cgroup	KubeEdge
Syscall time	192.18 ns	191.72 ns	206.48 ns
OOM kill	No	Yes	Yes

Table 4.23: Average memory usage for KubeEdge software stack. The results include the relative standard deviation.

Process	Avg.Memory [MiB]
KubeEdge	87.4 ± 2.10%
EdgeMesh	82.6 ± 3.10%
cri-o	76.6 ± 3.00%
runc	1.5 ± 5.30%

Table 4.24: Scaling overhead of containers.

Containers [#]	Available Memory [MiB]	Overhead [MiB]
0	15 827	-
4	15 821	1.4
16	15 804	1.4
32	15 775	1.6
64	15 704	1.9

namespaces/cgroups, but it is reasonable to think that it is caused by the container runtime. This hypothesis does not justify the difference in overhead between RISC-V and ARM, but, as we manually ported the container runtime in this work, there may be differences due to implementation or architectural optimizations missing for RISC-V.

Memory footprint and scaling analysis

We also analyzed the memory usage of this lightweight virtualization technique to assess potential overhead beyond CPU performance. This involved testing applications running inside and outside containers, as well as the additional software overhead incurred by the system for containerization and orchestration.

Firstly, we measured the average memory usage of the runc, CRI-O, KubeEdge and EdgeMesh processes. All these applications are required to run containers. Table 4.23 shows the results. The highest impacts are caused by KubeEdge, EdgeMesh and CRI-O, which combined use almost 250MB of system memory. This overhead is substantial and could limit the applicability of such an orchestration system for edge applications.

The second test uses a varying number of identical containers to analyze what is the memory overhead for each container started. The containers only start the sleep process and thus use no memory or CPU. We obtain the available memory readings by accessing the `/proc/meminfo` variable. Table 4.24 shows the results. The overhead is measured at 1.51MB per container when using 4 containers, but it grows to 1.92MB per container when there are 64 containers deployed. These overheads can be attributed to runc, which was measured at 1.5MB per container in Table 4.23. Overall, the memory impact of 64 containers running is 122.52MB, which is quite significant.

The last test compares memory utilization and performance in the *sysbench* memory benchmark. It compares five configurations: native execution with one thread in total and one thread per CPU core, containerized execution with one container and either one thread

Table 4.25: Results for running sysbench memory benchmark in different process, thread and containerization configurations.

Conf.	Result [MiB/s]	CPU [%]	Mem. [MiB]
Native - 1T	306.7	24.9%	1000.0
1 cont. - 1T	305.9	25.0%	1000.4
Native - 4T	1058.5	92.7%	1000.0
1 cont. - 4T	1065.6	88.6%	1000.1
4 cont. - 1T	1078.9	95.7%	1068.4

in total or one thread per CPU core, and containerized execution with one container per CPU core.

Table 4.25 shows the results. The first two rows compare native and containerized execution with only one thread. The difference is negligible despite the containerized benchmark using slightly more memory. This may be caused by the nature of libraries in containers, requiring static linking and resulting in higher system memory usage. When comparing versions with four workers, performance is similar across configurations, but memory usage increases significantly with multiple containers due to the less efficient nature of running multiple identical copies compared to letting the benchmark handle four threads independently.

4.3 Re-configurability of software for Edge-Cloud computing continuum

This section analyzes the improvements for the re-configuratibility of SW for Edge-Cloud computing, specifically:

- Section 4.3.1: ROS bandwidth-aware orchestration with superclustering allows for improve mapping and a reduction of network usage for the Kubernetes scheduler.
- Section 4.3.2: HEFT4K further optimizes the Kubernetes scheduler to improve the makespan of deployed applications that can be represented by a DAG.

4.3.1 Improving the Kubernetes schedule's efficiency for ROS-based applications: Network

ROS bandwidth-aware orchestration with superclustering

We consider a similar computing cluster for the RB-Kairos to Section 4.2, but the Jetson Nano has been replaced with a Jetson Xavier NX. It consists of 6 ARMv8 CPU cores, 8GB of system memory and Jetpack 4.5. This edge board allows us more freedom in the orchestration thanks to the improved performance and similar power consumption. The remaining computing nodes of the cluster are unchanged. We verified the super clustering algorithm presented in Section 3.3.1 with the containerized software stack of Tables 4.15 and 4.16, and compared the resource usage to the results of Table 4.17.

Table 4.26 shows the containers considered for super-clustering, along the specifications used for the orchestration in K3S. The `main-rb` and `arm-rb` containers have a required affinity for the MCB node, as these containers have drivers that need a direct communication with their respective devices. As these containers are the only ones with a physical constraint due to the device drivers presence, we set their priority higher than all other containers. The `map-nav-manager-rb` container is quite resource-hungry as it executes most of the macro-functionality nodes (i.e., localization, local and global planning). We set the affinity to the Jetson to avoid the mapping of such a container to the same node of the arm motion planner or too far away from the actual robot's HW. Nodes that do not have a specific amount of

Container name	Priority	Allocated CPU (cores)	Allocated System Memory (GB)	Affinity
main-rb	1	2	1.0	MCB
arm-rb	1	1	1.0	MCB
mission-rb	0	1	1.0	–
perception-rb	0	–	1.0	–
local-control-rb	0	–	1.0	–
map-nav-man.-rb	0	3	2.0	Jetson
navigation-rb	0	1	2.0	–
hmi-rb	0	1	1.0	Server

Table 4.26: Parameters used for the automatic scheduling of the containers onto the k3s computing cluster.

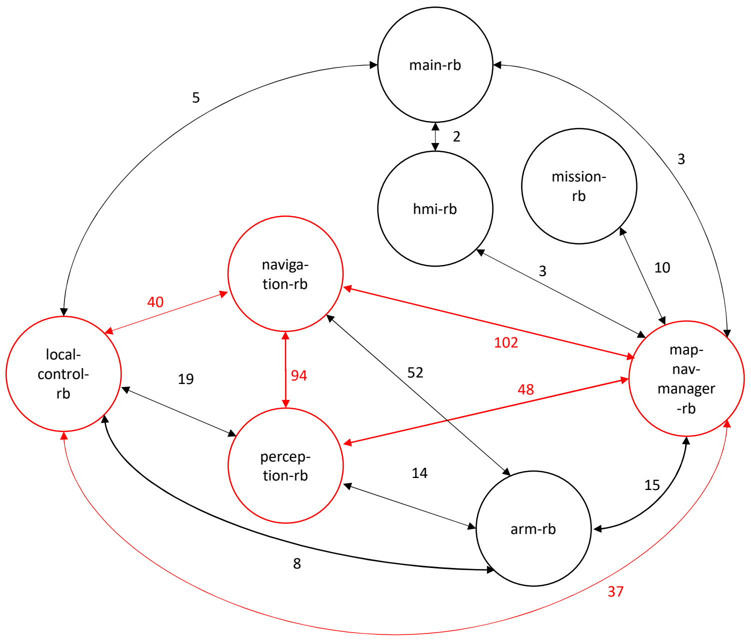


Fig. 4.14: ROS communication graph for the Kairos mobile robot with the highest communicating nodes highlited.

CPU cores allocated are not particularly resource intensive and can be executed with the "left-over" resources.

The remaining containers `hmi-webserver-rb`, `rms-rb` and `rostful-rb` (see Table 4.16) create a convenient interface to communicate with the robot from a web accessible format. These containers have been deployed manually (shown in green in Fig. 4.15).

Fig. 4.14 shows the graph with the bandwidth between each container pair. For the sake of readability, some low-weight edges have been omitted. The nodes and edges highlighted in red represent the highest communicating containers: `local-control`, `navigation`, `perception` and `map-nav-manager`. These containers should be kept together or as close as possible. Fig. 4.15 shows the resulting mapping obtained with the optimized (super-clustering) K3S scheduler. The results underline that the affinities have been respected and that the super clustering actually kept together the tightly communicating nodes.

With this optimized mapping, we can analyze the resource usage on the nodes of the computing cluster. Table 4.27 compares the obtained results to those of Table 4.17 for the two similar configurations: *native* from Fig. 4.8 and *k3s-a(native)* from Fig. 4.11. The task *init* corresponds to the *init* task of Table 4.17, while *mission* refers to the mission's tasks 1, 2 and 3 (see the introduction of Section 4.3.1). The memory usage on the MCB is significantly reduced, thus allowing for new robotic hardware to be added with its respective software

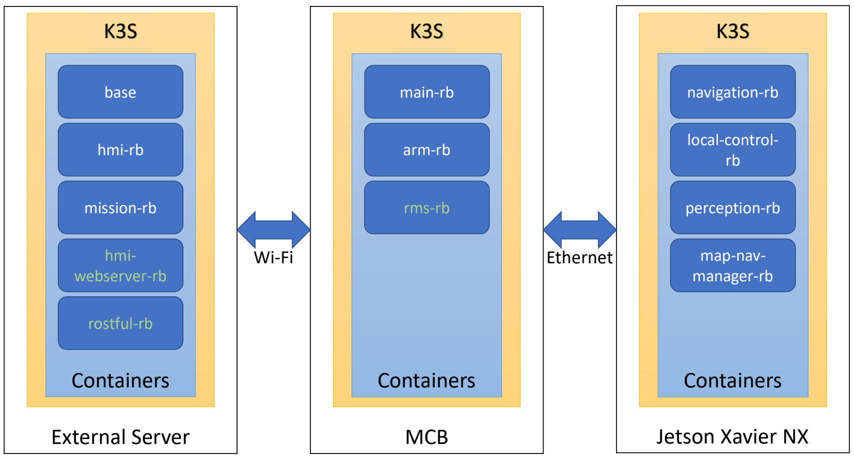


Fig. 4.15: Super clustering and mapping for the Kairos containers on the K3S computing platform.

Configuration	Task	Node Name	Memory Usage (MB)	Avg CPU usage (%)	Avg outgoing Network traffic (Mbps)
Native	init	MCB	3453.5	64.4	–
	mission	MCB	3454.4	63.1	–
k3s-a(native) [196]	init	MCB	3880.0	60.8	–
	mission	MCB	3884.4	61.5	–
SuperClustering	init	MCB	2011.0	42.2	10.29
		Jetson	2568.1	16.6	0.91
		Server	3103.5	45.8	1.59
	mission	MCB	2014.6	48.4	11.69
		Jetson	2585.2	18.3	0.93
		Server	3221.5	72.6	1.69

Table 4.27: Resource usage with SuperClustering.

stack. Also, the CPU utilization is lower due to the ROS nodes being moved to a different computing node, which allows a native installation of the ORB-SLAM (see first row of Table 4.18). The Jetson supports most of the navigation nodes with low CPU and system memory load. The server undergoes a significant increase in CPU load, especially when the robot is moving. This is due to the `base` container, which deploys the `roscore` node that handles communication setup between nodes and the parameter server. The network communication increases due to the distributed nature of the software. The highest traffic is found on the MCB while running the mission, which has to communicate with both the server and the Jetson on its two network interfaces (i.e., 6.46 Mbps and 5.23 Mbps, towards the server with Wi-Fi and the Jetson with Ethernet, respectively). The network traffic on the Wi-Fi interface is comparable to deploying the ORB-SLAM on the external server, but this configuration yields improved frame times.

4.3.2 Improving the Kubernetes schedule's efficiency for ROS-based applications: Makespan

We first tested HEFT4K with 100k synthetic DAGs on a computing platform simulating an Edge-Cloud architecture composed of 6 nodes, with 2 CPUs per node. The DAGs topology and characteristics as well as the cluster configuration were randomly generated (Table 4.28 summarizes the range of the considered values).

Graph characteristics	min.	max.	avg.	median
Nodes (#)	4	511	181.8	130
Edges (#)	3	1576	305.6	158
Outdegree	0.9	3.1	1.4	1.2
Critical path (ms)	4	446	64.3	41
Task comp. cost (ms)	1	20	10	10
Task comm. cost (Mb)	1	5	2.5	2.5
Node comm. (Mb/s)	500	1000	750	750

Table 4.28: Range of values considered for the random generation of DAGs and cluster configuration.

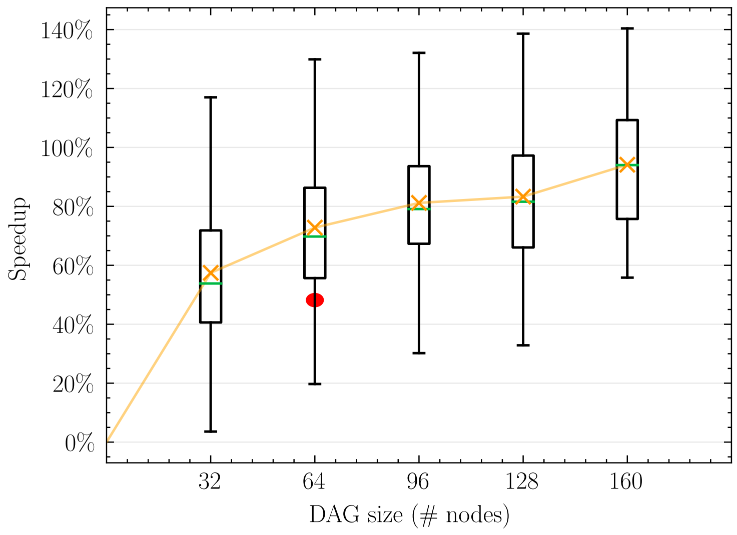


Fig. 4.16: Speedup of HEFT4K vs. the standard Kubernetes on different DAG sizes.

We then tested HEFT4K on the software that implements the mission of a Robotnik RB-Kairos, which is a skid-steering mobile platform equipped with a Universal Robots UR5 and a Schunk WSG50 gripper. The computing cluster consists of three nodes: two on-board programmable devices, i.e., an NVIDIA Jetson Xavier and a Jetson Nano. The onboard nodes communicate through a Gigabit Ethernet switch (802.3ab). The third node consists of an off-board desktop with an octa-core CPU and 16GB of RAM. It communicates with the other cluster nodes through WiFi (802.11ac).

The software is a ROS2-compliant application that executes a robot mission through a 42-task DAG. It allows for interaction between the mobile robot and an agile industrial production chain. The robot starts at a designated position, aligned with the production chain, and remains idle until objects arrive on the conveyor belt. Using a sequence of arm and gripper operations, the robot grasps production pieces from the conveyor belt and stores them in its cargo bay. Subsequently, it moves towards the storage area, unloads the pieces from the cargo bay, and then returns to the production line, aligning itself with the conveyor belt for the next cycle.

For both the synthetic benchmarks and the real case study, we compared the results obtained with the standard Kubernetes scheduler, HEFT on Kubernetes, the most efficient scheduler for Edge-Cloud architectures targeting QoS in literature (DPE) [96], and the proposed HEFT4K. We also tested the event-driven rescheduling of HEFT4K, analyzing its ability to maintain a positive makespan when sequentially removing nodes from the cluster.

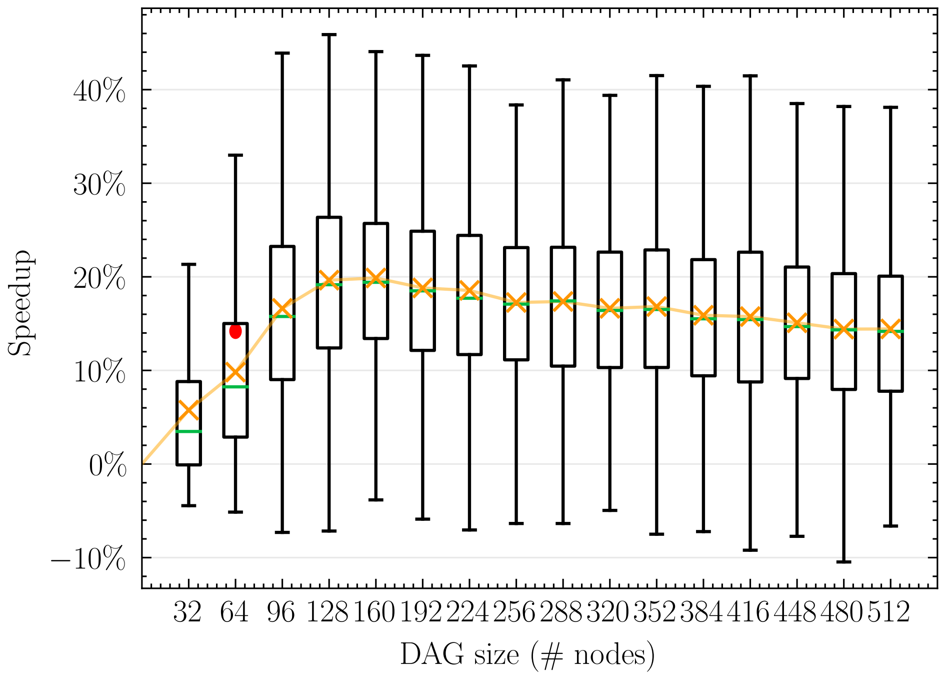


Fig. 4.17: Speedup of HEFT4K vs DPE scheduler on different DAG sizes.

HEFT4K vs. state of the art Kubernetes scheduling

Fig. 4.16 summarizes the comparison results between HEFT4K and the standard Kubernetes scheduler. Even with graphs of size 32 or less (first box plot from the left), HEFT4K achieves a significant speedup of 40%+ and, in the upper 25% of cases, the speedup grows to over 60%. With larger graphs, the speedup exceeds 100%. The red dot in Fig. 4.16 represents the speedup (48.2%) measured with the real case of study. With DAG sizes larger than 160 tasks, the Kubernetes scheduler fails. This is due to the fact that, differently from HEFT4K, Kubernetes stops the deployment of tasks when there are no resources available (i.e., free computing nodes).

Fig. 4.17 shows the comparison results between HEFT4K and DPE [96][1]. On average, HEFT4K achieves a speedup of 16% and reaches up to 40% speedup with DAG sizes larger that 64 tasks. In a limited number of cases ($<$ 25%), the DPE is slightly faster, on average, with 4% speedup w.r.t. HEFT4K. This is due to the fact that DPE does not use preemption and takes advantage of global synchronization. In the smaller graphs, small scheduling inefficiencies caused by the Linux scheduler, can translate in large percentage slowdowns.

We also compared HEFT4K and DPE on the Robotnik RB-Kairos case study. We obtained a 14.2% speedup on our 42-task DAG workload, which is in line with the average speedup obtained for the same DAG size in the synthetic dataset.

HEFT4K vs. HEFT on Kubernetes

The speedup achieved by HEFT4K on HEFT on Kubernetes is on average ≈ 13% across all DAGs and cluster configurations. Figure 4.18 shows a three dimensional heat map to represent the speedup of HEFT4K over HEFT on Kubernetes, by considering the DAG size and the critical path length. The speedup is limited with small graphs with short critical paths (lower left-most part of the plot), but it increases linearly with the DAG size and with the critical path length, albeit not as quickly with the critical path length (i.e., the

[1] In [96], DPE achieves a performance improvement of ≈40% over a standard HEFT [33] with one CPU per node. We measured a loss of performance ≈15% of DPE when compared to HEFT with the look-ahead variant [94] with multiple CPUs per node.

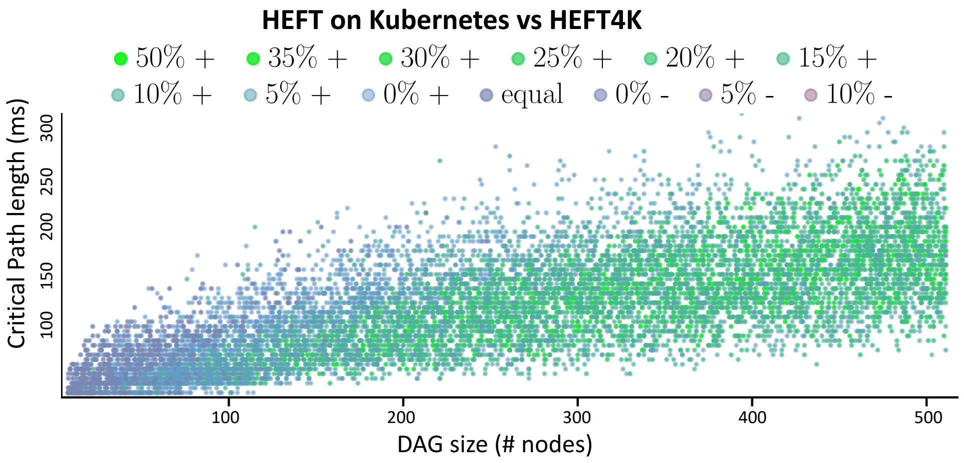


Fig. 4.18: Heatmap comparing the performance of HEFT4K and HEFT on Kubernetes, plotting the number of tasks in the DAG and the length of the critical path.

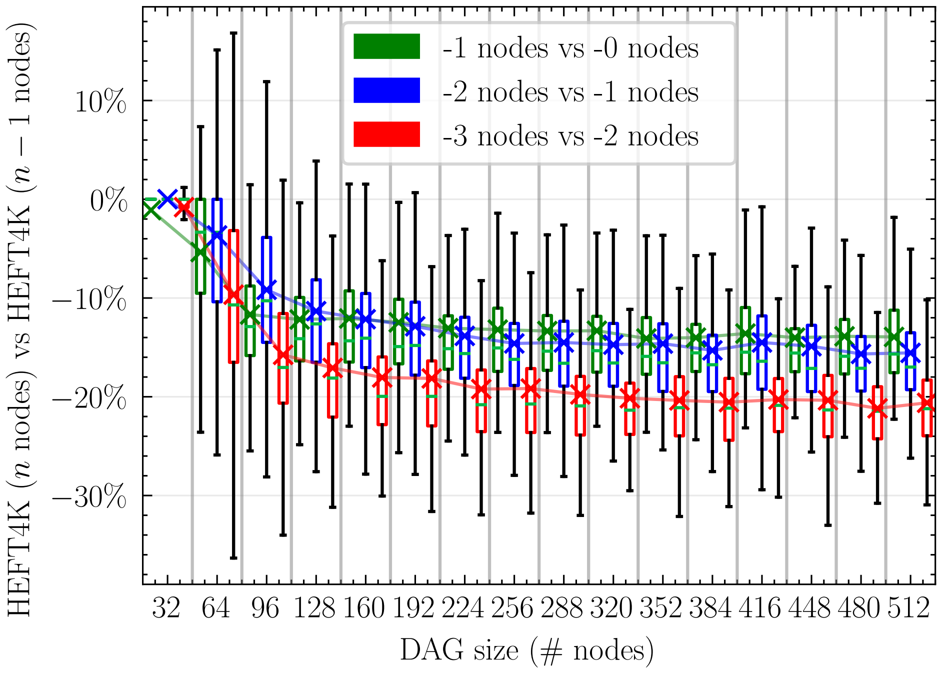


Fig. 4.19: Speedup of HEFT4K event-based remapping when rescheduling DAGs in case of node shutdown, compared to the previous makespan obtained with HEFT4K.

bottom half of the Figure has lower speedups compared to the top half). This is due to the fact that larger graphs with a longer critical path are generally more sequential and thus have less scheduling freedom. Therefore, the niceness optimization cannot really improve the scheduling, resulting in lower speedups.

Performance of event-based remapping

We reused the same six node architecture and 100k DAGs as the previous tests, and we repeated the scheduling five times. In each iteration, we removed one random node from the cluster until only one node was left. Fig. 4.19 shows the result in terms of makespan increase (i.e., negative speedup in %).

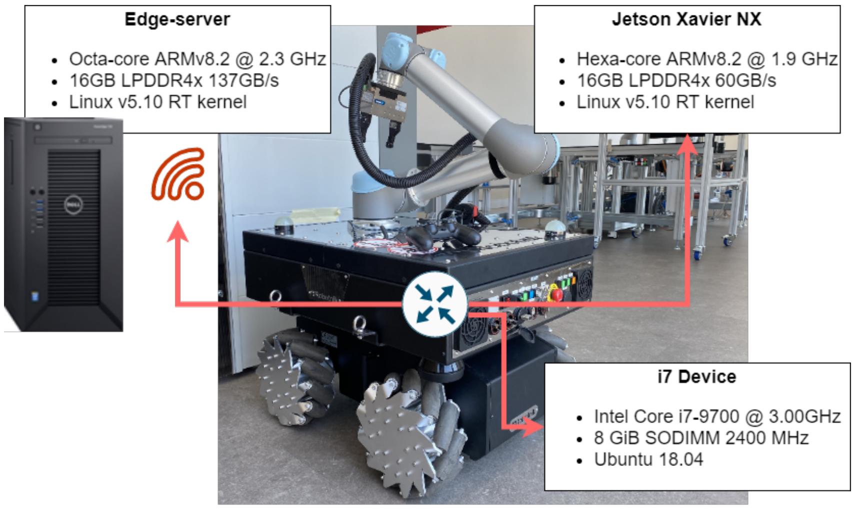


Fig. 4.20: RB-Kairos compute architecture.

The loss of one to three nodes of the cluster translates into a performance loss of ≈ 6%, for each lost node, when looking at DAG sizes smaller than 96 nodes. When considering larger DAG sizes, the performance degradation increases, although remaining below 15% on average when one or two nodes are lost. When three nodes are lost, the average trends towards −20%. However, at this point, half the cluster is offline, and the mapping strategy becomes quite limited.

The results for losing 4 and 5 nodes are similar, trending toward an average performance loss of ≈ 25% (not reported for readability).

In general, the algorithm provides resilience against node faults. It achieves positive performance especially when losing one to two nodes of the computing cluster, which represents the most common scenario in real application cases.

The overall time required to return the application to a working state after a node failure, and thus its downtime, depends strongly on its size and architecture. To recover from a node failure, the system needs to first execute the rescheduling algorithm and assess whether a partial rescheduling would significantly degrade performance. The time required by the rescheduling algorithm is negligible (few ms in our setup) since it is as efficient as HEFT. In addition, it needs to restart some (or all) containers on a new cluster node. However, restarting a container is a rather fast process that takes on average less than a second, and is parallelized, restarting multiple containers at once.

4.4 RT-Kube: real-time Kubernetes in the Edge-Cloud continuum

We evaluated RT-Kube efficiency on the software that implements the mission of a Robotnik RB-Kairos, which is a skid-steering mobile platform equipped with a Universal Robots UR5 and a Schunk WSG50 gripper. The software is distributed on computing cluster that consists of three nodes (see Fig. 4.20): two on-board programmable devices, i.e., an NVIDIA Jetson Xavier with the RT operating system (Linux kernel $v5.10$, and the *preempt_rt* patch) and a desktop with an i7 9700 locked at 3.0 GHz with 8GB RAM and a standard *non real-time* Linux-based operating system. The onboard nodes communicate through a Gigabit Ethernet switch (802.3ab). The third node consists of an off-board desktop with an octa-core CPU and 16GB of RAM, with Linux kernel $v5.10$, and the *preempt_rt* patch. It communicates with the other (on-board) cluster nodes through WiFi (802.11ac).

We first evaluated empirically the impact of containerization on RT tasks in terms of WCET overruns and missed deadlines. We then compared the efficiency of the proposed Kubernetes extension to support RT containers in terms of task rejections and missed deadlines with a synthetic software benchmark and with the real software implementing the

# tasks	config.	average runtime (ms)	average runtime with stress (ms)	Observed WCET (ms)	Observed WCET with stress (ms)	WCET over-runs (#)	WCET overruns with stress (#)	missed dead-lines (#)	missed dead-lines with stress (#)	total dead-lines per config (#)
1	native	1.219	1.256	3.143	3.655	0	0	0	0	30k
	1 container	1.164	1.172	3.055	3.119	0	0	0	0	
2	native	1.234	1.251	3.118	3.755	0	0	0	0	60k
	1 container	1.154	1.169	3.092	3.052	0	0	0	0	
	2 containers	1.144	1.162	3.091	3.090	0	0	0	0	
4	native	1.225	1.262	3.100	3.845	0	0	0	0	120k
	1 container	1.155	1.160	3.522	4.014	0	1	0	1	
	4 containers	1.134	1.154	3.783	3.704	0	0	0	0	
8	native	1.187	1.204	3.565	4.120	0	9	0	5	8M
	1 container	1.155	1.146	4.337	4.511	28	23	24	21	
	8 containers	1.140	1.141	4.405	4.322	26	18	10	14	

Table 4.29: Comparison of RT-tasks running natively, containerized in a single and multiple Docker containers.

robot mission. Then, we analysed the response time of the system recovery from temporal violations achieved by RT-Kube (see Section 3.4).

Impact of containerization on RT-tasks

Workload configuration. We evaluated the impact of containerization on RT tasks by using a large set of standard software bencharmarks for RT tasks. For brevity, we only report the results obtained with the *cyclicdeadline* benchmark from *rt-tests* [197] (the results with the other benchmarks show similar behavior). The benchmark is configured to run with either 1, 2, 4 or 8 identical tasks, maximum utilization of 95% per task, WCET of 3.8ms, period of 4.0ms, and deadline equal to the period. Note that we set these parameters to experience a full workload capacity on the system, which allows us to assess the performance impact of containerization. We evaluated three scenarios: (1) All RT tasks running natively on the edge platform to establish a baseline performance metric; (2) All RT tasks running in one container to analyze whether an application composed of multiple RT tasks is affected by the overhead; (3) All RT tasks running, where each RT task is mapped into its own container to evaluate how the overhead of containerization scales with multiple isolated tasks and if the overhead caused by a high number of containers can interfere with RT deadlines. We also run these three scenarios with additional tasks that overload the system with memory accesses to assess the behavior of the RT tasks under stress.
Key results. Table 4.29 shows the average actual runtime among n tasks, observed WCET among all the n tasks, total WCET overruns, and total of missed deadlines for all n tasks. Columns with "stress" identify those scenarios where the system was overloaded with memory accesses.

Our results indicate that containerization does introduce overhead on observed WCET, and that such an overhead is evident only in the higher CPU-load configurations, i.e., 4 or more tasks on the 8-core CPU. However, containerization does not impact the average-case runtime of tasks with or without stress. When compared to tasks running natively on the device, containerization impacts the WCET overruns and missed deadlines only when the RT utilization is close to 100% (last two rows of the table). This is due to the fact that when the task overruns the deadline and all CPU cores are allocated to RT tasks, the system cannot remap the task to a different available core. Interestingly, the number of containers created to group the RT tasks does not influence the observed WCET as, both when using one container and n containers, the observed times are similar. This is highlighted by the increased value of observed WCET (both with or without stress) by around 9.5% when the tasks are grouped into one container w.r.t. the native configuration, while the value is

	1st deployment (#)	2nd deployment (#)
Criticality A	32	16
Criticality B	24	24
Criticality C	16	32
Stress	1	1

Table 4.30: RT containers distribution for the RT orchestration.

comparable between 1 container and n containers with any number of tasks. We observed the same behavior for the WCET overruns and missed deadlines. The task WCET was overrun 28 times with 8 tasks in one Docker container, and this led to 24 missed deadlines over 8 millions (i.e., 0.00035%). The values do not increase by increasing the number of containers.
Summary. We noticed a negligible overhead when containerizing applications in general. Only at maximum utilization (e.g., 8 tasks over 8-CPU cores with 95% utilization) does the overhead lead to missed deadlines. With appropriate provisioning, we contend that containerization with the proposed off-the-shelf technology can support real-time applications. In our experimental analysis, we obtained no missed deadlines for the containerized configurations also with eight tasks by slightly relaxing the deadline w.r.t. to the period (e.g., increasing the period from 4.0 to 4.5ms, and the WCET from 3.8ms to 4.3ms, still with 95% maximum utilization per task).

Benchmarking the orchestration with RT support

Orchestration platform configurations. We used the following three configurations for the orchestration platform:

1. *Native K3S-standard*: The standard Kubernetes configuration for orchestration.
2. *Native K3S-best configuration*: The *best* configuration for orchestration with the native Kubernetes scheduler. We statically and manually assign an optimal task-to-node orchestration solution.
3. *RT-Kube*: Our proposed orchestration platform with the extended RT specifications and secondary RT scheduler.

Workload and deployment setups. The workload consists of 73 tasks. 72 of them have been containerized into 72 RT containers. The remaining task has been containerized into a non-RT container and implements stress activity through memory accesses on the platform.

All containers have one instance of "cyclicdeadline", the WCET has been set to 2.9ms for all tasks, and the deadlines at 24.2ms, 18.1ms and 14.5ms for levels A, B, and C, respectively. The resulting RT utilizations are: 12% for A, 16% for B, and 20% for C. We tested two deployment setups, shown in table 4.30. With these configurations and deployments, an improper orchestration would result in higher than 100% RT utilization on a single cluster node and, thus, rejected tasks.
Key results. Table 4.31 summarizes the workload and deployment setup in the first three columns. The fourth column shows the number of RT tasks that have been mapped into the corresponding RT nodes by the orchestrator, but that have been rejected by the Linux kernel admission test (see Eq. (2.4)). The last column shows, for all criticality levels, the number of missed deadlines. This column reports the deadlines missed by the running tasks (not rejected) and the total missed deadlines (those missed by the running + those missed because of the task rejection).

We found that the orchestration of the standard Kubernetes produces a task-to-node mapping that is inadequate for RT tasks. This is underlined by the amount of RT tasks that, after mapping, have been rejected by the Linux Kernel RT admission test and by the number of missed deadlines w.r.t. the best (manually configured) orchestration configuration.

Table 4.32 reports the efficiency comparison between the proposed solution when compared with the native Kubernetes, with a dynamic workload. The benchmark implements a sequential number of RT container deployments. We set up a first scenario (first two rows

Deployment	Config.	Critical tasks/ Nodes	Tasks rejected	Deadlines missed (running tasks - running+rejected)
1st	Native K3S-standard	A: J(32) B: J(24) C: J(16) oth.: J(1)	A: 8 B: 7 C: 6	A: 3.3% - 27.5% B: 9.1% - 35.6% C: 6.5% - 41.5%
	Native K3S-best config.	A: I(32) B: I(24) C: J(16) oth.: R(1)	A: 0 B: 0 C: 0	A: 0.0% - 0.0% B: 0.0% - 0.0% C: 0.0% - 0.0%
	RT-Kube	A: I(32) B: I(24) C: J(16) oth.: R(1)	A: 0 B: 0 C: 0	A: 0.0% - 0.0% B: 0.0% - 0.0% C: 0.0% - 0.0%
2nd	Native K3S-standard	A: J(16) B: J(24) C: J(32) oth.: J(1)	A: 8 B: 10 C: 10	A: 0.0% - 50.0% B: 2.9% - 44.6% C: 4.3% - 34.2%
	Native K3S-best config.	A: I(16) B: I(24) C: J(32) oth.: R(1)	A: 0 B: 0 C: 0	A: 0.0% - 0.0% B: 0.0% - 0.0% C: 0.0% - 0.0%
	RT-Kube	A: I(16) B: I(24) C: J(32) oth.: R(1)	A: 0 B: 0 C: 0	A: 0.0% - 0.0% B: 0.0% - 0.0% C: 0.0% - 0.0%

Table 4.31: Comparison among Kubernetes standard orchestration, best orchestration, and the proposed RT-Kube.

	T_0		$T_0 + t_\alpha$	
Configuration	Deployed	Pending - Rejected	Deployed	Pending - Rejected
Native-K3S	N_1: 8 N_2: 8	1 - 0	N_1: 9 N_2: 8	0 - 1
RT-Kube	N_1: 8 N_2: 8	1 - 0	N_1: 8 N_2: 8	1 - 0
Native-K3S	N_1: 8 N_2: 0	1 - 0	N_1: 9 N_2: 0	0 - 1
RT-Kube	N_1: 8 N_2: 0	1 - 0	N_1: 8 N_2: 1	0 - 0

Table 4.32: Experimental results for dynamic orchestration.

of Table 4.32) in which 16 RT containers are deployed and run correctly onto the cluster for 100% RT utilization. In this context, there is CPU available for the (standard) orchestrator, while the CPU is fully loaded for the RT admission test (see Eq. (2.4)). When a new RT container has to be deployed (instant $T_0 + t_\alpha$), the orchestrator of the native Kubernetes immediately deploys the container on the cluster. This leads to an overload of the cluster RT utilization (i.e., more than 100%) and, as a consequence, to a failed admission test by the Linux kernel. This *failing deployment* keeps getting restarted by Kubernetes. The admission test succeeds and the container is deployed only when RT resources of the same node become

					Missed deadlines	
Configuration	Avg runtime (ms)	Std. dev.	Observed WCET (ms)	Max. lateness (ms)	arm driver	SLAM
Native	7.15	3.58	32.32	-	24.50%	-
non-RT + RT [108, 112]	0.20	0.11	5.72	0.95	1.28%	≈ 0%
non-RT + A + B + C	0.15	0.07	2.06	0.49	0.17%	≈ 0%

Table 4.33: Experimental results with the RB-Kairos.

available and the new RT utilization is within 100%. Nevertheless, if any other different RT node becomes available, the issue persists as Kubernetes has no way of knowing the cause of the failure. In contrast, with the proposed solution, the new container remains pending *at orchestration level* until any RT resources in the cluster become available.

We set up a second scenario (last two rows of Table 4.32), in which the first RT node has utilization 100%, with 8 containers, while the second RT node has utilization 0%. When a new RT container has to be deployed, with the native Kubernetes, there is only a 50% chance for the correct node to be picked, as the orchestrator has no knowledge of the RT utilization. In contrast, the proposed orchestration platform guarantees that the correct node (one with 0% RT utilization) is always selected.

Orchestration with RT support of the robot's mission software

Software configuration. The robot's mission is implemented through a SW application composed of 80 ROS tasks. It performs pick and place operations, delivering goods from a conveyor belt to a storage area and vice-versa. The most critical part of the software is the driver controlling the arm operations. This task needs to maintain a control loop that communicates directly with the arm hardware at 120Hz (8ms deadline, 7.6ms WCET, 90% utilization). Exceeding the deadline and delaying such a communication beyond a certain threshold causes the arm to go into safe mode and halting operations. This translates into a measured maximum lateness of 0.7ms, after which the connection is dropped.
Key results. As expected, we found that, with the native K3S orchestration, all tasks were mapped onto the three devices randomly. This resulted in missed deadlines to exceed the threshold and to the safety stop of the robot. We then evaluated two alternative scenarios with the proposed RT orchestrator. The first supports multiple criticality levels (i.e., non real-time tasks with A, B, and C real-time tasks), while the second only supports non real-time with real-time tasks (e.g., [108, 112]). In the first solution, we classified 3 tasks implementing the robot SLAM as RT criticality B. Then, the arm driver as an RT task with criticality C. We defined the edge server as an RT node for criticalities A and B, while the on-board Jetson for criticality C. In the second scenario, we classified the SLAM, as well as the arm driver, as RT. Both are deployed on the RT-capable Jetson onboard. In both scenarios the rest of the software was configured as non real-time. These two test cases allow us to evaluate the improvement for the automatic separation of different classes of criticality as well as the benefits of RT orchestration.

Table 4.33 shows the results. Our solution based on multi-level criticality (third row in the table) manages to take advantage of criticality-aware load distribution, which allows the arm driver to perform substantially better than native and other alternatives, with lower observed WCET and reduced deadline misses. The much less restrictive control loop of the SLAM nodes allows for performance improvement in both solutions.
Summary. Unlike other alternatives, the proposed solution did not suffer from any safety-related stop as the maximum lateness bound was never exceeded. This is due to the isolation of the highest criticality tasks from the lower criticality ones implemented by the orchestrator. We also observed very different average runtime and WCET when comparing the native Kubernetes approach and RT-Kube. With the native Kubernetes, our case study caused an

Containers (#)	Cont. Distribution			Reconcile phase (ms)		
	non-RT	–	RT	Linear	Linearithmic	Quadratic
1	1	–	0	41.07	40.81	40.83
1	0	–	1	41.75	41.49	41.51
8	8	–	0	40.26	39.27	57.27
8	4	–	4	42.98	41.99	59.99
8	0	–	8	45.70	44.71	62.71
16	16	–	0	51.32	51.62	157.94
16	8	–	8	56.76	57.06	163.38
16	0	–	16	62.20	62.50	168.82
32	32	–	0	65.43	106.07	173.57
32	16	–	16	70.87	111.51	179.01
32	0	–	32	76.31	116.95	184.45
64	64	–	0	115.23	152.39	687.02
64	32	–	32	126.11	163.27	697.90
64	0	–	64	136.99	174.15	708.08

Table 4.34: Cluster-level monitor reconcile response times for various combinations of RT scheduling objects and containers.

order of magnitude more missed deadlines w.r.t. RT-Kube, which make the native solution practically unusable.

Analysis of response time for container migration

Software configuration. We first measured the response time of the system recovery from temporal violations implemented by RT-Kube (Eq. (3.19) in Section 3.4) through a large set of benchmarks, which consist of different numbers and distributions of non-RT and RT containers on the cluster. We deployed the container-level monitors on each device of the cluster and, the cluster-level monitor on the K3S master node (i.e., the external server). In all tests, we set the updating frequency of the container-level monitors to 10Hz. We measured, on average, 53ms as the time elapsed from the missed deadline to the update instant (the first component of Eq. (3.19)).
Network impact. We analyzed the time taken to transfer the updating messages from the container-level monitors to the master across the Ethernet+WiFi network, with and without network congestion. Without network congestion, we measured an average latency of ≈ 3ms for both transmissions ($HTTP\_trans(JSON_c)$ and $HTTP\_trans(Kill)$). With a congested network, we measured, on average, a communication latency of 9.9ms for the two transmissions. We measured the worst-case transfer time to send each update message to the server as 115.2ms (i.e., $HTTP\_trans(JSON_c)$) and 84.99ms to send the $HTTP\_trans(Kill)$ message to the node. The congestion was obtained through 1Gb/s of traffic from the master to the container-level monitor, and vice versa (obtained with the *netcat* and *pv* emulation software).
Key results. Table 4.34 reports the average time spent for the reconcile phase in the tested configurations. The first column indicates the total number of containers deployed in the cluster node. The second column reports the distribution of standard and RT containers. The last three columns report the time spent in the reconcile phase with the linear, linearithmic, and quadratic policies.

We found that the time required by the three policies is comparable with a low number of containers deployed in the cluster node (i.e., 8 in our experimental setup). Linear and linearithmic still achieve comparable performance for up to 16 containers. With more containers, the different impacts of the three policies on the response time become evident. When analyzing how each policy is affected by the number of standard and RT container, we observed that, when moving from 1 to 32 *standard* containers, the latency of the reconcile

Cont. (#)	Container Distribution non-RT − RT	Eq. (3.19) (# element)	AVG Time (ms)	WCET (ms)
18	14 − 4	(1) updating freq.	50.00	100.00
		(2) $trans(JSON_c)$	9.89	115.21
		(3) Rec_phase	23.47	108.09
		(3.1) $trans(Kill)$	8.67	84.99
		Total	92.03	408.29

Table 4.35: Cluster-level monitor response time on the autonomous mobile robot case study.

phase increases by 59.3%, 159.9%, and 325.1% with the linear, linearithmic, and quadratic policies, respectively. When moving from 1 to 32 *RT* containers, the time increases by 82.3%, 181.9% and 344.5%.

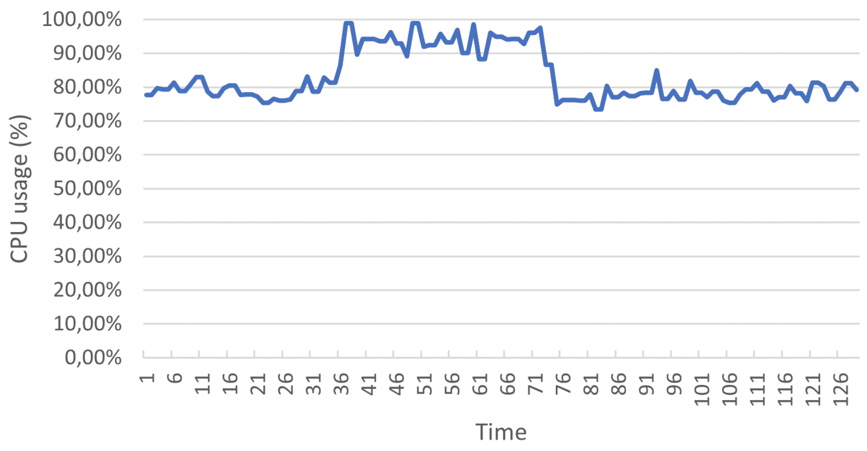
Summary. In general, the response time with both linear and linearithimc policies is less than 200ms, in which, as expected, there is a negligible contribution of the transfer latency. The contribution of the monitor updating to the overall latency could become negligible at higher updating frequencies. The trade-off comes at the cost of more computational resources spent on operations, particularly on the master node. The *smarter* quadratic policy leads to a response time of around 750ms. In general, RT containers impact the reconcile time slightly more than standard containers. Such an additional overhead is negligible and becomes even less relevant as the complexity of the algorithm grows. This is due to the fact that the overhead for each container is constant.

Analysis of response time for system recovery with the robot's mission software

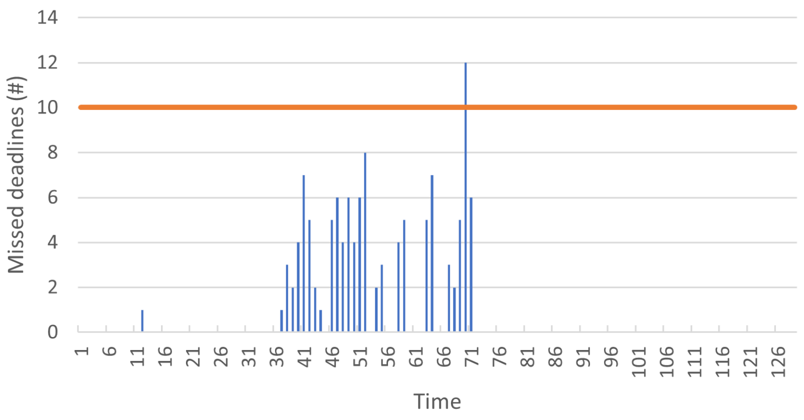
Software configuration. We measured the response time of the system recovery with the robot's mission software. The 80 ROS tasks implementing the robot's mission were organized into 13 containers, for a total of 18 containers by including the RT containers of the SLAM software and arm drivers. The first two columns of table 4.35 show the configuration and distribution. We deployed all the non-RT containers on the robot's on-board i7 and all the RT containers on the on-board Jetson.

Key results. Table 4.35 shows the results. The fourth column reports the average time and the last column reports the worst case time for each component, with the total time in the last row. On average, the delay caused by the updating frequency is 50ms, the communication both ways takes ≈ 20ms, and the computation is ≈ 20ms, with a total time of ≈ 90ms. In the worst-case scenario, communication consumes half of the total response time, with one fourth being computation for the reconcile phase. The last one fourth is the configurable updating frequency, where the same considerations of the previous section apply. Nonetheless, the observed worst-case response time, from detecting a missed deadline to the Kubelet on the target node receiving a command is below half a second (≈ 408ms).

We also analyzed the CPU usage and the number of missed deadlines through Prometheus and Grafana (i.e., two popular monitoring and visualizing tools for Kubernetes). Figure 4.21 shows the collected data. To showcase cluster-level monitor eviction, we deployed an RT container with a CPU-intensive task to stress the system, bypassing the schedulability check on the Jetson and forcing the system into a particularly unsustainable scenario. At instant 36, Fig. 4.21a underlines the start of a sequence of CPU overload, while Fig. 4.21b reports the corresponding increase of missed deadlines. At instant 71, the missed deadlines cross the threshold and the cluster-level monitor starts the eviction process. The RT container of the CPU-intensive task was selected for eviction as it had the highest CPU utilization. In the following time instants, Figures 4.21a and 4.21b, depict the decrease of both CPU usage and missed deadlines. Meanwhile, the RT container of the CPU-intensive task was sent back to the scheduling queue and restarted on the server, as it is the only other node of the cluster with RT support.



(a) CPU usage with the RT tasks for the Jetson Xavier on the Robotnik RB-Kairos.



(b) Missed deadlines for the RT tasks for the Jetson Xavier on the Robotnik RB-Kairos.

Fig. 4.21: Real-Time monitoring on the Robotnik RB-Kairos.

During the eviction phase, the victim container is immediately killed and restarted on a different node. This results in the evicted task experiencing downtime due to the required cold-starting on the new node. RT-Kube migrates non-critical or less-critical tasks that steal resources to higher criticality tasks in devices where temporal violations are observed. In our system, all container images are already downloaded onto permanent storage for all nodes to avoid unpredictable network latency. This is possible thanks to the proposed multi, smaller, and modular container organization of the software. With this configuration, the downtime is in average less than one second (less than three seconds in the worst case). Since it involves only "less-critical tasks", it is acceptable and does not involve any disruption.

4.5 Assertion-based verification and workload migration in Kubernetes for robotic systems

We evaluated the effectiveness of the proposed orchestration- and assertion-based verification approach, discussed in Section 3.5, in two case studies. In the first, we set up a cluster of three nodes and implemented a synthetic benchmark to quantitatively evaluate the behavior of the monitor orchestration. In the second, we applied the platform in a real industrial case study, which implements the mission of a Robotnik RB-Kairos mobile robot in a smart manufacturing line.

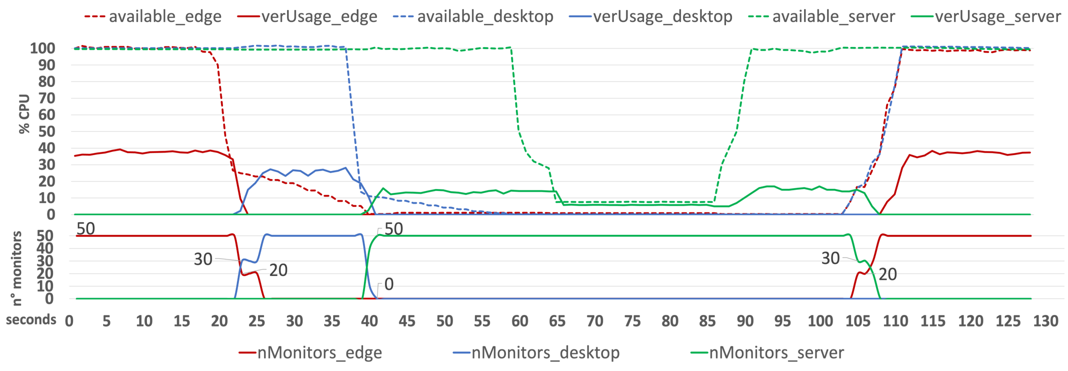


Fig. 4.22: Verification statistics of the first case study.

4.5.1 Case Study 1: synthetic benchmark on a three-level cluster

We first tested the orchestration functionality on a distributed edge-cloud system composed of three computing nodes. At the edge level, the cluster includes a 4-core Intel i7 (2.8 GHz) node with 4GB of RAM (i.e., *edge node*). At the cloud level, it includes a computing node equipped with a 16-core AMD Ryzen (3.9 GHz) CPU and 16 GB of RAM (i.e., *server node*). It then includes a desktop machine equipped with an 8-core Intel CPU (3.2 GHz, 8 GB of RAM), which is connected between the edge and the cloud nodes (i.e., *desktop node*). The server node lies at a 100 ms (average) ping distance from the other nodes while the edge and desktop nodes lie at a 10 ms (average) ping distance from each other. The verification environment consists of 50 monitors requesting input values from 5 different topics. All topics are published on the edge machine, which represents the most common configuration in most practical use cases. The orchestrator is set to recompute the optimal allocation of monitors every second. The SUV does not contain any tasks, while the CPU consumption of the computing nodes is artificially influenced by a custom test bench capable of generating a variable amount of computational workloads. The purpose of the testbench is to gradually saturate the CPU of the nodes, allowing us to study the behavior of the orchestrator under such conditions.

Fig. 4.22 reports the quantitative history of the monitor orchestration and the CPU utilization of each node while executing both the testbench and the verification environment. It includes the available CPU of the different nodes (*available_edge*, *available_desktop*, *available_server*), which represents, in percentage, the total CPU minus the CPU used by the SUV tasks. The verification environment usage (*verUsage_edge*, *verUsage_desktop*, *verUsage_server*) corresponds to the percentage of CPU used to execute the verification environment on the corresponding machine. The values *nMonitors_edge*, *nMonitor_desktop* and *nMonitor_server* represent the number of monitors currently executing on the corresponding node. On the x-axis, the figure reports the time (in seconds) that elapsed from the beginning of the simulation. At time 0, all nodes had 100 percent of the available CPU and the edge node was executing all 50 monitors using, on average, 38 percent of the available CPU. After 20 seconds, the testbench started saturating the CPU of the edge node. As a consequence, the node was no longer able to handle the computational load of the verification environment. Therefore, the orchestrator enabled the migration of monitors to the desktop node, which had available computing resources and guaranteed the least reduction in verification responsiveness. The monitor migration, as expected, was performed incrementally at the varying of the testbench workload. Since the edge CPU was not immediately saturated, the orchestrator started moving 20 monitors after 22 seconds, and then, at the next allocation period (second 23), as the edge CPU had been further loaded by the testbench, it migrated the remaining 30 monitors. From 23 to 40 seconds, the desktop node executed all the monitors. After that, the testbench saturated the CPU of the desktop node to test the migration from the desktop to the server. Similarly to the previous iteration, the orchestrator migrated the monitors to the server node, which was the only device remaining with available CPU resources. Saturating such a node caused the system to fall into the overload

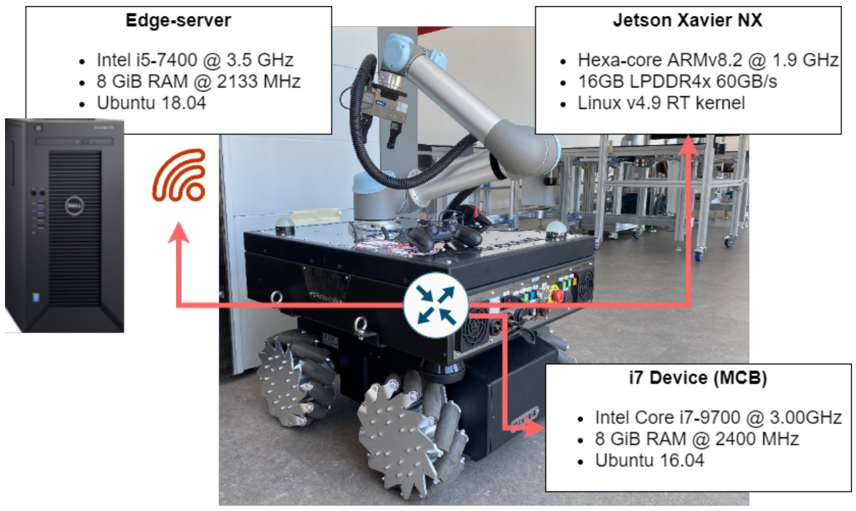


Fig. 4.23: Overview of the programmable cluster nodes in the second case study.

scenario. Note that this time, the monitors were transferred all at once towards the server, since the testbench reduced the available CPU on the desktop machine more quickly than in the previous transfer (from the edge machine), forcing the orchestrator to quickly free computational resources on the desktop machine. To observe the behavior of the verification environment in the overload scenario, the testbench started partially saturating the CPU of the server node (65 seconds), leaving only 8% of the available CPU. Since the verification environment required 18% of the CPU, it started dropping events from the buffers of the monitors, effectively reducing the computational cost of verification to 7.5% (on average). Then, the testbench removed the computational load on the server machine (87 seconds), restoring the execution of the monitors to their original accuracy and CPU consumption. Finally, the testbench removed the computational load on all nodes. As a consequence, the orchestrator moved all the monitors to the edge node where verification could achieve the best responsiveness (105 seconds). During the entire simulation, the orchestrator spent on average 27ms (every second) to solve the MILP problem, proving the scalability of this approach.

4.5.2 Case Study 2: Autonomous mobile robot for a smart manufacturing line

We evaluated the proposed methodology to assess the mission of a Robotnik RB-Kairos mobile robot[2] in an industrial agile production chain. Such a mobile robot consists of a skid-steering platform equipped with a Universal Robots UR5 manipulator and a Schunk WSG50 end-effector for grasping (see Fig. 4.23). The robot is also equipped with different sensors including two Sick S300 laser scanners and an RGB-D Intel RealSense D415 camera for localization. The computing HW architecture consists of two edge devices installed onboard. One MCB is equipped with an Intel i7 9700 3.0 GHz, 8GB of RAM, and Ubuntu 22.04 OS with ROS2 Humble. The second edge device consists of an Nvidia Jetson Xavier NX with JetPack 4.5. The onboard devices are connected through a local gigabit Ethernet router (802.3ab). The cluster of nodes also includes an external server equipped with an Intel i5-7400 3.5 GHz, 8GB of RAM, and Ubuntu 18.04. The server is connected to the onboard devices through an 866Mbps wireless network (802.11ac).

We configured a K3s cluster, version 1.20.4+k3s1, on the MCB, on the Jetson Xavier and on the external server. The server also runs the K3s master agent.

We measured the system performance while executing the ROS-compliant SW application that implements the robot mission. It consists of several tasks that implement the

[2] Note that, while it is the same robot as Fig. 4.20, the computing architecture is different. This is caused by the different computing requirements, i.e., in Fig. 4.20 the Edge-server needed to have an RT kernel.

Component	Devices	No monitors	Monitors
CPU	Jetson	38.64%	45.49%
	MCB	23.44%	28.50%
	Server	30.96%	50.98%
Network	Wi-Fi	0.31 MiB/s	0.39 MiB/s
	Ethernet	1.31 MiB/s	1.41 MiB/s

Table 4.36: Resource usage overhead with and without monitors.

interaction of the mobile robot with an industrial agile production chain. See Section 4.2 for details on the mission.

The software for the Kairos is split between all three computing nodes inside the K3s cluster. The MCB handles most of the robotic functionality such as mission, arm and gripper planning, etc. The Jetson node is tasked with the navigation stack, while the server node handles HMI and monitoring. To collect the performance metrics we used Prometheus connected to Grafana. As the server node is tasked with scraping the data, its CPU usage fluctuates significantly when there is a data update.

We configured the MILP solver with threshold CPU usage for the MCB of 40%, 60% for the Jetson, and 90% for the external server. These thresholds allow the Kairos HW to continue working properly and not be overwhelmed by the computational load when the monitors are running together with the robotic software. The system includes 201 monitors running on the cluster, on all three computing nodes. We also used a moving window to average the last 10 values obtained from the ROS topics to create a low-pass filter that removes signal spikes that often happen when analyzing real sensor data.

Functional analysis: With the use of RGB cameras and an inference-based image recognition system [193], the robotic software implements an ORB-SLAM [192] application for localization and mapping. It is then connected to a *move-base* system that relies on a global and local planner for obstacle avoidance.

Considering the temporal constraints of the software system, the aim was to check the functional correctness of the applications. In particular, we designated the global planner as a non-critical task on the server. We considered the ORB-SLAM and local planner running in real time on the Jetson (i.e., 125 ms application makespan). We enforced a corresponding minimum supported rate of 8 FPS for the RGB camera input stream.

To evaluate the effect of orchestration, we consider a scenario in which the robot must reach a user-defined location $\langle x_2, y_2 \rangle$ from the starting point $\langle x_1, y_1 \rangle$. First, the global planner finds the path to reach the arrival point. Subsequently, while the robot moves, the local planner reschedules the path trajectory (waypoints) to take care of changes in the environment that may cause unwanted collisions with moving obstacles.

After the initial allocation of 201 monitors on the Jetson, we observed that the monitor corresponding to the assertion $G((robot\_x_1 = x_1 \;\&\&\; robot\_y_1 = y_1 \;\&\&\; newGoal) \rightarrow (F[0, timeout](robot\_x_2 = x_2 \;\&\&\; robot\_y_2 = y_2)))$ failed. This monitor was set up to check if the robot can reach $\langle x_2, y_2 \rangle$ before a given timeout. The monitor failed due to the overhead introduced by the verification environment, which causes an increment in the execution time. Consequently, the robot moved in the wrong direction because the robot's controller was unable to support the minimum updating frequency for the motor velocities. For proper operations, the controller for the motor velocities needs an update at least every 125 ms (8 Hz). The system guarantees a makespan of 115 ms without the verification environment (8.7Hz). When the verification overhead is introduced, this value increases to 140 ms (7.1 Hz). When, thanks to the buffer migration approach, 150 monitors are automatically transferred from the Jetson to the server during execution; therefore, some of the verification load on the edge device is relieved. The robot's controller resumed functioning normally with a makespan of 125 ms (8 Hz), which prevented the aforementioned assertion from failing.

Orchestration and overhead: Table 4.36 summarizes the CPU and network usage with and without monitors. We can observe higher CPU usage overall, distributed across the computing cluster, and a slight increase in network usage of $\approx$ 0.1 MiB/s. The high

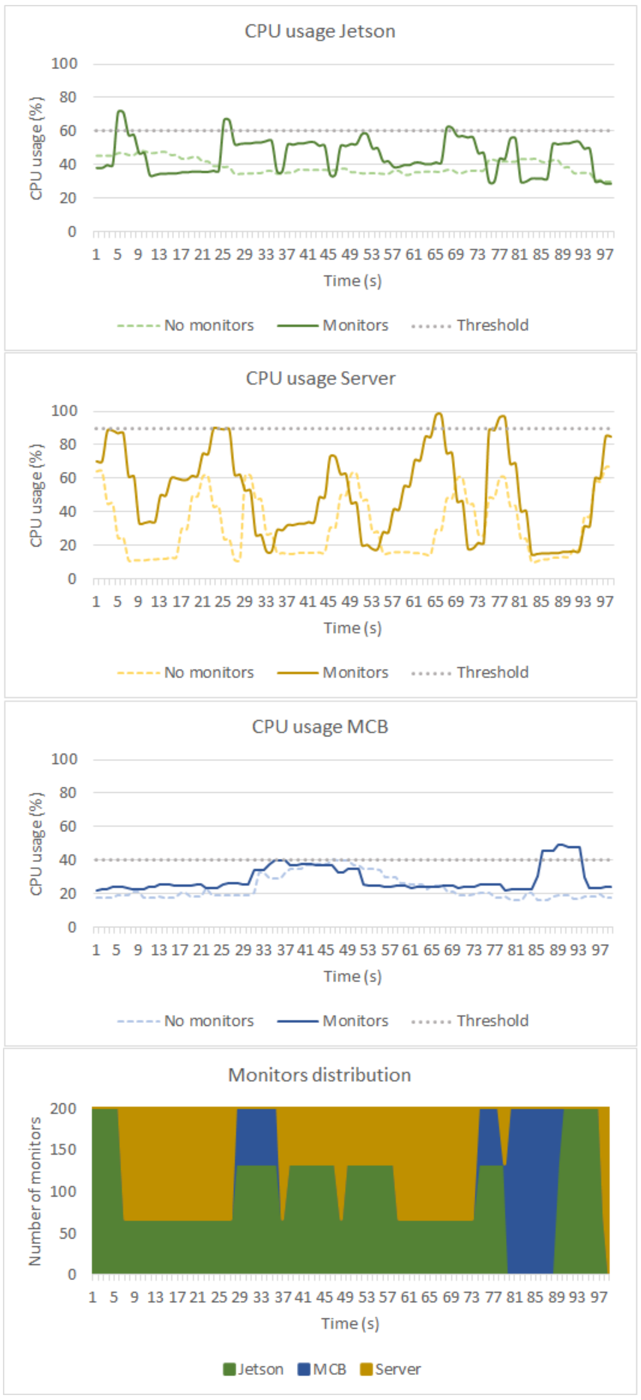


Fig. 4.24: CPU overhead for all the nodes correlated to the number of monitors during the execution of the robot's mission.

bandwidth on the Ethernet interface is caused by the constant communication to handle the navigation SW components deployed on the Jetson. We also observed a negligible and static increase of ≈ 40MB of system memory usage on all nodes of the cluster due to the allocation of the resources needed by the monitors (not reported in Table 4.36).

Figure 4.24 shows the overhead in more detail, as well as the number of running monitors on each node during the mission of the RB-Kairos. As reported in the summary table, CPU usage is higher overall for all devices due to the additional SW for the monitors running, but the highest average increase is found on the server and Jetson. On these two devices, the monitors spend most of their time as reported in the last graph (yellow and green for server and Jetson respectively). We also observed notable CPU usage spikes when the monitors are moved onto a particular device. An example is at the time instant 29, where some monitors are moved to the MCB and we can see the CPU usage reach the threshold, causing a migration towards the server. It is also evident at the time instant 85, where all monitors are migrated towards the MCB and its CPU usage exceeds the threshold, also causing a reduction in CPU usage on the other two devices. When the threshold value is exceeded, the MILP begins a migration towards the Jetson and also towards the server, causing spikes in usage on all the involved CPUs, but freeing the crucial resources needed by the MCB to correctly run the robot SW.

5 Conclusions and Future Work

This thesis focused on optimizing performance in Edge-Cloud architectures, leading to strong results for containerized environments in robotics, industrial automation, and emerging architectures like RISC-V.

It introduced a design methodology based on Docker and Kubernetes to program advanced robotic tasks. It aimed to facilitate the containerization and orchestration of ROS-based robotic software applications across diverse hardware architectures. Through comprehensive case studies, this thesis demonstrated the methodology's ability to integrate and verify multi-domain components early in the design process, enabling functional and extra-functional verification prior to system deployment. It also introduced and evaluated an innovative node clustering algorithm, which improves the efficiency of the organization of ROS nodes within containers. This approach significantly optimized memory usage, CPU overhead, and network bandwidth.

Additionally, since the adoption of containerization and Kubernetes orchestration in resource-constrained edge computing environments, including robotics and CPSs, has gained significant traction due to the complexity of AI-based software on modern autonomous platforms, this thesis introduced RT-Kube, an orchestration platform tailored for mixed-criticality systems, seamlessly integrating real-time containers into standard Kubernetes. This research enabled containerized mixed-criticality environments, previously unattainable, and offered a transparent monitoring solution for temporal violations. The experiments demonstrated the negligible overhead when deploying real-time tasks on embedded systems and showcased RT-Kube's ability to reduce missed deadlines by up to 50% compared to standard Kubernetes, ensuring improved reliability and robust real-time performance.

Another significant result was effectively integrating the essential components of containerization and orchestration, such as network plugins and container runtime, into a RISC-V platform. The thesis demonstrated that these technologies can be efficiently implemented on a distributed computing system based on the RISC-V architecture with a small performance degradation, comparable to the performance degradation observed on the ARM architecture. The experimental findings have significant implications for the development of open digital infrastructures. They broaden the possibilities for creating large computing ecosystems with an open and scalable infrastructure, leveraging the open hardware design of RISC-V as well as the adaptability and effectiveness of containerization and orchestration technologies.

Furthermore, this research addressed performance improvements at both the application level and system level. At the application level, it tackled task mapping and scheduling challenges for OpenVX DAG-based embedded vision applications on heterogeneous CPU-GPU devices. It presented innovative scheduling algorithms and frameworks that enhance the efficiency of embedded vision applications while considering device heterogeneity and multiple node implementations. These approaches were extensively evaluated through a considerable set of benchmarks and real-world applications, providing valuable insights into their effectiveness.

At the system level, it introduced HEFT4K, a scheduling approach based on the HEFT task ranking, optimized for orchestrator platforms such as Kubernetes. Using the OS niceness concept and event-driven remapping, HEFT4K significantly reduced the makepan of DAG-based applications by an average of 43% on synthetic datasets and real case studies, outperforming existing Kubernetes schedulers.

In summary, this research has advanced the field of Edge-Cloud architectures and introduced innovative methodologies and frameworks that significantly enhance the efficiency, reliability, and real-time performance of robotic systems and embedded vision applications. The improved resource management reduces the need for over-provisioning and hardware upgrades, meaning that systems can handle more tasks more efficiently, leading to less stress on hardware components while reducing costs. Moreover, better performance allows for more reliable runtime calculation and prediction mechanisms that minimize operational risks, leading to safer industrial and robotic environments. The goals established in Fig. 1.2 were achieved by following five research and implementation steps, which methodically and consistently improved the architecture for the Edge-Cloud computing continuum.

5.1 Future work

In the context of the Edge-Cloud computing continuum, research across various fields has been making strong leaps toward more sustainable computing, thanks to improved algorithms and methodologies that allow for improved efficiency. Containerization especially has allowed the flexibility required by these techniques to reconfigure and adapt the workloads to more dynamic infrastructures. However, many challenges remain open.

First and foremost, there is a need for improved communication. The problem of reliable network has been extensively studied for wired connections, but, especially in the case of mobile robots, which rely on wireless communications standards like 5G or Wi-Fi, there is a need to transform an unreliable medium into a reliable one. This could be achieved with better and stronger connections, but also with SW redundancy and failsafes. Simple solutions could be applied, such as stopping the robot in case the connection is lost or applying a redundant controller for navigation. However, these solutions come with tangible drawbacks. The first one interrupts the robot's mission, which may result in catastrophic failures or costly delays. And using onboard resources for a redundant controller, resources that were freed by migrating the navigation controller to the cloud completely negates the advantages of such a methodology. This translates into a concrete need to improve communication or techniques to improve redundancies and failsafe.

Another challenge that is still open in this context is to integrate more diverse accelerators into the workflow of robotic and industrial workloads. With the increased adoption of RISC-V, the proliferation of customized and dedicated accelerators will probably require adapting the SW architecture to accept these new computing nodes in the edge-cloud paradigm. Containerization and orchestration allow for the partitioning of memory or CPU resources, but there is a need for more in-depth segmentation that allows control over dedicated accelerators that do not enter into the simple categorization of "GPUs". This translates into the need for more modular approaches for the development of SW, which is a strong attraction for research opportunities.

The hardware itself could also be assessed for optimization since we are considering a heterogeneous and flexible computing architecture. Some computing hardware can be dynamically altered at "runtime" to fit the computational needs of a specific application, for example, by adapting a computing platform to the software requirements of a robotic system. By incorporating reconfigurable hardware architectures such as Field-Programmable Gate Array (FPGA) or Coarse-Grained Reconfigurable Array (CGRA), robots could modify their computational behavior in real-time to optimize for different tasks or environmental conditions. These customized devices, including task-specific accelerators, such as Application-Specific Integrated Circuit (ASIC), could be seamlessly integrated into the computing architecture thanks to its dynamic and heterogeneous composition, allowing for even more flexibility and performance.

References

1. X. Xu, Y. Lu, B. Vogel-Heuser, and L. Wang, “Industry 4.0 and industry 5.0 - inception, conception and perception,” *Journal of Manufacturing Systems*, vol. 61, pp. 530–535, 2021.
2. M. Di Nardo and H. Yu, “Special issue “industry 5.0: The prelude to the sixth industrial revolution”,” *Applied System Innovation*, vol. 4, no. 3, 2021.
3. A. Al-Yacoub, Y. Zhao, W. Eaton, Y. Goh, and N. Lohse, “Improving human robot collaboration through force/torque based learning for object manipulation,” *Robotics and Computer-Integrated Manufacturing*, vol. 69, 2021.
4. H. Abbas, I. Saha, Y. Shoukry, R. Ehlers, G. Fainekos, R. Gupta, R. Majumdar, and D. Ulus, “Special session: Embedded software for robotics: Challenges and future directions,” in *Proc. of ACM/IEEE International Conference on Embedded Software, EMSOFT*, 2018.
5. P. Thakur and V. Kumar Sehgal, “Emerging architecture for heterogeneous smart cyber-physical systems for industry 5.0,” *Computers and Industrial Engineering*, vol. 162, 2021.
6. J. Chen and X. Ran, “Deep learning with edge computing: A review,” *Proceedings of the IEEE*, 2019.
7. F. Lumpp, F. Fummi, H. Patel, and N. Bombieri, “Containerization and orchestration of software for autonomous mobile robots: a case study of mixed-criticality tasks across edge-cloud computing platforms,” in *2022 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2022, pp. 1–6.
8. Sifive website. [Online]. Available: https://www.sifive.com/
9. E. Flamand, D. Rossi, F. Conti, I. Loi, A. Pullini, F. Rotenberg, and L. Benini, “GAP-8: A RISC-V SoC for AI at the Edge of the IoT,” in *2018 IEEE 29th International Conference on Application-specific Systems, Architectures and Processors (ASAP)*. IEEE, 2018, pp. 1–4.
10. NVIDIA. (2021) Isaac SDK-Powered Robots Collaborating in Simulated Factory Environment. [Online]. Available: https://developer.nvidia.com/isaac-sdk
11. AMAZON. (2021) AWS RoboMaker. [Online]. Available: https://aws.amazon.com/robomaker/?nc1=h_ls
12. Open Source Robotics Foundation. (2023) Robot Operating System. [Online]. Available: www.ros.org
13. D. Merkel *et al.*, “Docker: lightweight linux containers for consistent development and deployment,” *Linux journal*, vol. 2014, no. 239-art.2, 2014.
14. P. Melo, R. Arrais, and G. Veiga, “Development and deployment of complex robotic applications using containerized infrastructures,” in *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*, 2021, pp. 1–8.
15. M. Shaik and et al., “Enabling fog-based industrial robotics systems,” in *IEEE Symposium on Emerging Technologies and Factory Automation, ETFA*, vol. 2020-September, 2020, pp. 61–68.
16. N. Nikolakis, R. Senington, K. Sipsas, A. Syberfeldt, and S. Makris, “On a containerized approach for the dynamic planning and control of a cyber - physical production system,” *Robotics and Computer-Integrated Manufacturing*, vol. 64, p. 101919, 2020.
17. M. Sollfrank, F. Loch, S. Denteneer, and B. Vogel-Heuser, “Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation,” *IEEE Transactions on Industrial Informatics*, vol. 17, no. 5, pp. 3566–3576, 2021.
18. AUTOSAR: AUTomotive Open System ARchitecture, “The standardized software framework for intelligent mobility,” 2021, www.autosar.org.
19. Collins Aerospace, “Connectivity and network services,” 2021, www.arinc.com.
20. A. Burns and R. Davis, “Mixed criticality systems-a review,” *Department of Computer Science, University of York, Tech. Rep*, pp. 1–69, 2013.

21. ——, “A survey of research into mixed criticality systems,” *ACM Computing Surveys*, vol. 50, no. 6, 2017.
22. T. Tasci, J. Melcher, and A. Verl, “A container-based architecture for real-time control applications,” in *2018 IEEE International Conference on Engineering, Technology and Innovation, ICE/ITMC*, 2018.
23. M. Cinque, R. Corte, A. Eliso, and A. Pecchia, “Rt-cases: Container-based virtualization for temporally separated mixed-criticality task sets,” in *Leibniz International Proceedings in Informatics, LIPIcs*, vol. 133, 2019.
24. L. Abeni, A. Balsini, and T. Cucinotta, “Container-based real-time scheduling in the linux kernel,” *SIGBED Rev.*, vol. 16, no. 3, p. 33–38, Nov. 2019.
25. M. Thiyyakat, S. Kalambur, and D. Sitaram, “Improving resource isolation of critical tasks in a workload,” *Lecture Notes in Computer Science*, vol. 12326 LNCS, pp. 45–67, 2020.
26. F. Hofer, M. Sehr, A. Sangiovanni-Vincentelli, and B. Russo, “Industrial control via application containers: Maintaining determinism in iaas,” *Systems Engineering*, vol. 24, no. 5, pp. 352–368, 2021.
27. D. Bernstein, “Containers and cloud: From lxc to docker to kubernetes,” *IEEE Cloud Computing*, vol. 1, no. 3, pp. 81–84, 2014.
28. A. Singh, M. Shafique, A. Kumar, and J. Henkel, “Mapping on multi/many-core systems: Survey of current and emerging trends,” in *Proc. od ACM/IEEE Design Automation Conference*, 2013.
29. A. Goens, R. Khasanov, J. Castrillon, M. Hähnel, T. Smejkal, and H. Härtig, “TETRiS: A Multi-Application Run-Time System for Predictable Execution of Static Mappings,” in *Proc. of ACM International Workshop on Software and Compilers for Embedded Systems*, 2017, pp. 11–20.
30. NVIDIA Inc., “VisionWorks,” https://developer.nvidia.com/embedded/ visionworks.
31. M. Yang, T. Amert, K. Yang, N. Otterness, J. Anderson, F. Smith, and S. Wang, “Making OpenVX Really ’Real Time’,” in *Proc of IEEE Real-Time Systems Symposium (RTSS)*, 2019, pp. 80–93.
32. G. Elliott, K. Yang, and J. Anderson, “Supporting Real-Time Computer Vision Workloads Using OpenVX on Multicore+GPU Platforms,” in *Proceedings - Real-Time Systems Symposium*, 2016, pp. 273–284.
33. H. Topcuoglu, S. Hariri, and M.-Y. Wu, “Performance-effective and low-complexity task scheduling for heterogeneous computing,” *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.
34. A. Maurya and A. Tripathi, “Performance Comparison of HEFT, Lookahead, CEFT and PEFT Scheduling Algorithms for Heterogeneous Computing Systems,” in *Proc. of ACM International Conference on Computer and Communication Technology*, 2017, pp. 128–132.
35. S. AlEbrahim and I. Ahmad, “Task scheduling for heterogeneous computing systems,” *Journal of Supercomputing*, vol. 73, no. 6, pp. 2313–2338, 2017.
36. L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira, “DAG Scheduling Using a Lookahead Variant of the Heterogeneous Earliest Finish Time Algorithm,” in *Proc. of Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010, pp. 27–34.
37. H. Zhao and R. Sakellariou, “An experimental investigation into the rank function of the heterogeneous earliest finish time scheduling algorithm,” in *Euro-Par 2003 Parallel Processing*, H. Kosch, L. Böszörményi, and H. Hellwagner, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2003, pp. 189–194.
38. K. R. Shetti, S. A. Fahmy, and T. Bretschneider, “Optimization of the HEFT Algorithm for a CPU-GPU Environment,” in *Proc. of International Conference on Parallel and Distributed Computing, Applications and Technologies*, 2013, pp. 212–218.
39. A. Yudha, R. Pulungan, H. Hoffmann, and Y. Solihin, “A simple cache coherence scheme for integrated CPU-GPU systems,” in *Proc. of ACM/IEEE Design Automation Conference (DAC)*, 2020.
40. X. Ren, D. Lustig, E. Bolotin, A. Jaleel, O. Villa, and D. Nellans, “Hmg: Extending cache coherence protocols across modern hierarchical multi-gpu systems,” in *Proc. of IEEE International Symposium on High Performance Computer Architecture, HPCA 2020*, 2020, pp. 582–595.
41. S. A. Mojumder, Y. Sun, L. Delshadtehrani, Y. Ma, T. Baruah, J. L. Abellán, J. Kim, D. R. Kaeli, and A. Joshi, “HALCONE : A hardware-level timestamp-based cache coherence scheme for multi-gpu systems,” *CoRR*, vol. abs/2007.04292, 2020.
42. N. Agarwal, D. Nellans, E. Ebrahimi, T. Wenisch, J. Danskin, and S. Keckler, “Selective gpu caches to eliminate cpu-gpu hw cache coherence,” in *Proc. of International Symposium on High-Performance Computer Architecture*, 2016, pp. 494–506.

43. S. Mittal, "A survey of techniques for managing and leveraging caches in gpus," *Journal of Circuits, Systems and Computers*, vol. 23, no. 8, 2014.
44. N. Bombieri, F. Busato, and F. Fummi, "Power-aware performance tuning of gpu applications through microbenchmarking," in *Proc. of ACM/IEEE Design Automation Conference*, 2017.
45. R. Zheng, Q. Hu, and H. Jin, "Gpuperfml: A performance analytical model based on decision tree for GPU architectures," in *Proc. of International Conference on High Performance Computing and Communications*, 2019, pp. 602–609.
46. S. Madougou, A. Varbanescu, C. De Laat, and R. Van Nieuwpoort, "The landscape of GPGPU performance modeling tools," *Parallel Computing*, vol. 56, pp. 18–33, 2016.
47. T. Exley and A. Jafari, "Maximizing energy efficiency of variable stiffness actuators through an interval-based optimization framework," *Sensors and Actuators A: Physical*, vol. 332, 2021.
48. V. Dutta and T. Zielińska, "Cybersecurity of robotic systems: Leading challenges and robotic system design methodology," *Electronics (Switzerland)*, vol. 10, no. 22, 2021.
49. G. Giri, Y. Maddahi, and K. Zareinia, "A brief review on challenges in design and development of nanorobots for medical applications," *Applied Sciences (Switzerland)*, vol. 11, no. 21, 2021.
50. I. Calvo, E. Villar, C. Napole, A. Fernández, O. Barambones, and J. Gil-García, "Reliable control applications with wireless communication technologies: Application to robotic systems," *Sensors*, vol. 21, no. 21, 2021.
51. C. Mouradian, D. Naboulsi, S. Yangui, R. Glitho, M. Morrow, and P. Polakos, "A comprehensive survey on fog computing: State-of-the-art and research challenges," *IEEE Communications Surveys and Tutorials*, vol. 20, no. 1, pp. 416–464, 2018.
52. C. Pahl, "Containerization and the paas cloud," *IEEE Cloud Computing*, vol. 2, no. 3, pp. 24–31, 2015.
53. R. Morabito, "Virtualization on internet of things edge devices with container technologies: A performance evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017.
54. P. González-Nalda, I. Etxeberria-Agiriano, I. Calvo, and M. Otero, "A modular cps architecture design based on ros and docker," *International Journal on Interactive Design and Manufacturing*, vol. 11, no. 4, pp. 949–955, 2017.
55. L. Yin, J. Luo, and H. Luo, "Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, pp. 4712–4721, 2018.
56. Open Source Robotics Foundation, "Ros 2 documentation," https://docs.ros.org/en/dashing/index.html.
57. ——, "Ros wiki," http://wiki.ros.org/.
58. ——, "Efficient intra-process communication," https://docs.ros.org/en/foxy/Tutorials/Intra-Process-Communication.html.
59. Cloud Native Computing Foundation. (2023) Kubernetes. [Online]. Available: kubernetes.io
60. Architecture. [Online]. Available: https://kubernetes.io/docs/concepts/architecture/cloud-controller/
61. F. Carpio, M. Delgado, and A. Jukan, "Engineering and experimentally benchmarking a container-based edge computing system," *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, Jun 2020.
62. V. Ibarra-Junquera, A. González, C. M. Paredes, D. Martínez-Castro, and R. A. Nuñez-Vizcaino, "Component-based microservices for flexible and scalable automation of industrial bioprocesses," *IEEE Access*, vol. 9, pp. 58 192–58 207, 2021.
63. G. Kurtzer, V. Sochat, and M. Bauer, "Singularity: Scientific containers for mobility of compute," *PLoS ONE*, vol. 12, 05 2017.
64. A. Moga, T. Sivanthi, and C. Franke, "Os-level virtualization for industrial automation systems: Are we there yet?" in *Proc. of the ACM Symposium on Applied Computing*, vol. 04-08-April-2016, 2016, pp. 1838–1843.
65. K. Kaur, S. Garg, G. Kaddoum, S. H. Ahmed, and M. Atiquzzaman, "Keids: Kubernetes-based energy and interference driven scheduler for industrial iot in edge-cloud ecosystem," *IEEE Internet of Things Journal*, vol. 7, no. 5, pp. 4228–4237, 2020.
66. H. Sami and A. Mourad, "Dynamic on-demand fog formation offering on-the-fly iot service deployment," *IEEE Transactions on Network and Service Management*, vol. 17, no. 2, pp. 1026–1039, 2020.
67. H. A. Ozmen, S. Işık, and C. Ersoy, "A hardware and environment-agnostic smart home architecture with containerized on-the-fly service offloading," *Computers & Electrical Engineering*, vol. 92, p. 107090, 06 2021.
68. T. Goldschmidt, S. Hauck-Stattelmann, S. Malakuti, and S. Grüner, "Container-based architecture for flexible industrial control applications," *Journal of Systems Architecture*, vol. 84, pp. 28–36, 2018.

69. A. Valente, M. Mazzolini, and E. Carpanzano, “An approach to design and develop reconfigurable control software for highly automated production systems,” *International Journal of Computer Integrated Manufacturing*, vol. 28, no. 3, pp. 321–336, 2015.
70. S. Aldegheri, N. Bombieri, F. Fummi, S. Girardi, R. Muradore, and N. Piccinelli, “Late breaking results: Enabling containerized computing and orchestration of ros-based robotic sw applications on cloud-server-edge architectures,” in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–2.
71. R. White and H. Christensen, “Ros and docker,” *Studies in Computational Intelligence*, vol. 707, p. 285 – 307, 2017. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85019705507&doi=10.1007%2f978-3-319-54927-9_9&partnerID=40&md5=146008c1aecf7228fec0c73479272c66
72. S. Wen, B. Ding, H. Wang, B. Hu, H. Liu, and P. Shi, “Towards migrating resource-consuming robotic software packages to cloud,” in *2016 IEEE International Conference on Real-time Computing and Robotics (RCAR)*, 2016, pp. 283–288.
73. P. Melo, R. Arrais, and G. Veiga, “Development and deployment of complex robotic applications using containerized infrastructures,” in *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*, 2021, pp. 1–8.
74. A. Waterman, Y. Lee, D. A. Patterson, and K. Asanovic, “The risc-v instruction set manual, volume i: Base user-level isa,” *EECS Department, UC Berkeley, Tech. Rep. UCB/EECS-2011-62*, vol. 116, 2011.
75. Eu roadmap open hardware. [Online]. Available: digital-strategy.ec.europa.eu/en/library/recommendations-and-roadmap-european-sovereignty-open-source-hardware-software-and-risc-v
76. 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, pp. 171–172.
77. A. Torrez, T. Randles, and R. Priedhorsky, “HPC Container Runtimes have Minimal or No Performance Impact,” in *2019 IEEE/ACM International Workshop on Containers and New Orchestration Paradigms for Isolated Environments in HPC (CANOPIE-HPC)*, 2019, pp. 37–42.
78. Y. Pan, I. Chen, F. Brasileiro, G. Jayaputera, and R. Sinnott, “A Performance Comparison of Cloud-Based Container Orchestration Tools,” in *2019 IEEE International Conference on Big Knowledge (ICBK)*, 2019, pp. 191–198.
79. Y. Jing, Z. Qiao, and R. O. Sinnott, “Benchmarking Container Technologies For IoT Environments,” in *2022 Seventh International Conference on Fog and Mobile Edge Computing (FMEC)*, 2022, pp. 1–8.
80. A. Das, S. Patterson, and M. Wittie, “EdgeBench: Benchmarking Edge Computing Platforms,” in *2018 IEEE/ACM International Conference on Utility and Cloud Computing Companion (UCC Companion)*, 2018, pp. 175–180.
81. V. Noronha, E. Lang, M. Riegel, and T. Bauschert, “Performance Evaluation of Container Based Virtualization on Embedded Microprocessors,” in *2018 30th International Teletraffic Congress (ITC 30)*, vol. 01, 2018, pp. 79–84.
82. T. Goethals, M. Sebrechts, M. Al-Naday, B. Volckaert, and F. De Turck, “A Functional and Performance Benchmark of Lightweight Virtualization Platforms for Edge Computing,” in *2022 IEEE International Conference on Edge Computing and Communications (EDGE)*, 2022, pp. 60–68.
83. T. Wei, M. Malhotra, B. Gao, T. Bednar, D. Jacoby, and Y. Coady, “No such thing as a “free launch”? Systematic benchmarking of containers,” in *2017 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, 2017.
84. F. Carpio, M. Delgado, and A. Jukan, “Engineering and Experimentally Benchmarking a Container-based Edge Computing System,” in *ICC 2020 - 2020 IEEE International Conference on Communications (ICC)*, 2020, pp. 1–6.
85. C.-C. Chang, S.-R. Yang, E.-H. Yeh, P. Lin, and J.-Y. Jeng, “A kubernetes-based monitoring platform for dynamic cloud resource provisioning,” in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, 2017, pp. 1–6.
86. F. Katenbrink, A. Seitz, L. Mittermeier, H. Müller, and B. Bruegge, “Dynamic scheduling for seamless computing,” in *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, 2018, pp. 41–48.
87. G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, “Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes,” in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 33–40.

88. M. Chima Ogbuachi, C. Gore, A. Reale, P. Suskovics, and B. Kovács, "Context-aware k8s scheduler for real time distributed 5g edge computing applications," in *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2019, pp. 1–6.
89. M. C. Ogbuachi, A. Reale, P. Suskovics, and B. Kovács, "Context-aware kubernetes scheduler for edge-native applications on 5g," *Journal of communications software and systems*, 2020.
90. I. Rocha, C. Göttel, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, "Heats: Heterogeneity- and energy-aware task-based scheduling," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2019, pp. 400–405.
91. G. El Haj Ahmed, F. Gil-Castiñeira, and E. Costa-Montenegro, "Kubcg: A dynamic kubernetes scheduler for heterogeneous clusters," *Software: Practice and Experience*, vol. 51, no. 2, pp. 213–234, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2898
92. J. Li, B. Liu, W. Lin, P. Li, and Q. Gao, "An improved container scheduling algorithm based on pso for big data applications," in *Cyberspace Safety and Security*, J. Vaidya, X. Zhang, and J. Li, Eds. Cham: Springer International Publishing, 2019, pp. 516–530.
93. Y. Fu, S. Zhang, J. Terrero, Y. Mao, G. Liu, S. Li, and D. Tao, "Progress-based container scheduling for short-lived applications in a kubernetes cluster," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 278–287.
94. L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira, "Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010, pp. 27–34.
95. T. Hai, J. Zhou, D. Jawawi, D. Wang, U. Oduah, C. Biamba, and S. K. Jain, "Task scheduling in cloud environment: optimization, security prioritization and processor selection schemes," *Journal of Cloud Computing*, vol. 12, no. 1, 2023.
96. S. Deng, H. Zhao, Z. Xiang, C. Zhang, R. Jiang, Y. Li, J. Yin, S. Dustdar, and A. Y. Zomaya, "Dependent function embedding for distributed serverless edge computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 10, pp. 2346–2357, 2022.
97. Intel. (2023) Telemetry Aware Scheduling. [Online]. Available: /www.intel.com/content/www/us/en/developer/articles/technical/telemetry-aware-scheduling.html
98. IBM. (2023) Trimaran: Real Load Aware Scheduling. [Online]. Available: github.com/kubernetes-sigs/scheduler-plugins/tree/master/kep/61-Trimaran-real-load-aware-scheduling
99. J. Abella, D. Hardy, I. Puaut, E. Quiñones, and F. J. Cazorla, "On the comparison of deterministic and probabilistic wcet estimation techniques," in *2014 26th Euromicro Conference on Real-Time Systems*, 2014, pp. 266–275.
100. Linux scheduler. [Online]. Available: docs.kernel.org/scheduler/index.html
101. S. Aldegheri, N. Bombieri, S. Germiniani, F. Moschin, and G. Pravadelli, "A containerized ros-compliant verification environment for robotic systems," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 222–225.
102. F. Lumpp, M. Panato, F. Fummi, and N. Bombieri, "A container-based design methodology for robotic applications on kubernetes edge-cloud architectures," in *2021 Forum on specification Design Languages (FDL)*, 2021, pp. 01–08.
103. S. Aldegheri, N. Bombieri, F. Fummi, S. Girardi, R. Muradore, and N. Piccinelli, "Late breaking results: Enabling containerized computing and orchestration of ros-based robotic sw applications on cloud-server-edge architectures," in *2020 57th ACM/IEEE Design Automation Conference (DAC)*, 2020, pp. 1–2.
104. F. Mastrogiovanni, A. Paikan, and A. Sgorbissa, "Semantic-aware real-time scheduling in robotics," *IEEE Transactions on Robotics*, vol. 29, no. 1, pp. 118–135, 2013.
105. L. Han, L. Xu, D. Bobkov, E. Steinbach, and L. Fang, "Real-time global registration for globally consistent rgb-d slam," *IEEE Transactions on Robotics*, vol. 35, no. 2, p. 498 – 508, 2019.
106. V. Struhár, M. Behnam, M. Ashjaei, and A. V. Papadopoulos, "Real-Time Containers: A Survey," in *Workshop on Fog Computing and the IoT*, 2020, pp. 7:1–7:9.
107. F. Hofer, M. A. Sehr, A. Iannopollo, I. Ugalde, A. Sangiovanni-Vincentelli, and B. Russo, "Industrial control via application containers: Migrating from bare-metal to iaas," in *In Proc. of IEEE CloudCom*, 2019, pp. 62–69.
108. S. Fiori, L. Abeni, and T. Cucinotta, "Rt-kubernetes: Containerized real-time cloud computing," in *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*, ser. SAC '22. New York, NY, USA: Association for Computing Machinery, 2022, p. 36–39. [Online]. Available: https://doi.org/10.1145/3477314.3507216
109. F. Hofer, M. A. Sehr, A. Sangiovanni-Vincentelli, and B. Russo, "Odre workshop: Probabilistic dynamic hard real-time scheduling in hpc," 2020.

110. M. Cinque, R. Della Corte, and R. Ruggiero, “Preventing timing failures in mixed-criticality clouds with dynamic real-time containers,” in *2021 17th European Dependable Computing Conference (EDCC)*, 2021, pp. 17–24.
111. L. De Simone and G. Mazzeo, “Isolating real-time safety-critical embedded systems via sgx-based lightweight virtualization,” in *Proc. of International Symposium on Software Reliability Engineering Workshops, ISSREW 2019*, 2019, pp. 308–313.
112. V. Struhár, S. S. Craciunas, M. Ashjaei, M. Behnam, and A. V. Papadopoulos, “React: Enabling real-time container orchestration,” in *2021 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, 2021, pp. 1–8.
113. S. Choudhury, S. Maheshwari, I. Seskar, and D. Raychaudhuri, “Shareon: Shared resource dynamic container migration framework for real-time support in mobile edge clouds,” *IEEE Access*, vol. 10, pp. 66 045–66 060, 2022.
114. A. E. González and E. Arzuaga, “Herdmonitor: Monitoring live migrating containers in cloud environments,” in *2020 IEEE International Conference on Big Data (Big Data)*, 2020, pp. 2180–2189.
115. S. Zheng, F. Huang, C. Li, and H. Wang, “A cloud resource prediction and migration method for container scheduling,” in *2021 IEEE Conference on Telecommunications, Optics and Computer Science (TOCS)*, 2021, pp. 76–80.
116. P. Meredith, D. Jin, D. Griffith, F. Chen, and G. Roşu, “An overview of the mop runtime verification framework,” *International Journal on Software Tools for Technology Transfer*, vol. 14, no. 3, pp. 249–289, 2012.
117. E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan, “Specification-based monitoring of cyber-physical systems: A survey on theory, tools and applications,” *Lecture Notes in Computer Science*, vol. 10457, pp. 135–175, 2018.
118. C. Hu, W. Dong, Y. Yang, H. Shi, and G. Zhou, “Runtime verification on hierarchical properties of ros-based robot swarms,” *IEEE Transactions on Reliability*, vol. 69, no. 2, pp. 674–689, 2020.
119. K. Havelund, G. Reger, and G. Roşu, “Runtime verification past experiences and future projections,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 10000, pp. 532–562, 2019.
120. G. Rosu and K. Havelund, “Rewriting-based techniques for runtime verification,” *Automated Software Engineering*, vol. 12, no. 2, pp. 151–197, 2005.
121. D. Ulus, T. Ferrère, E. Asarin, and O. Maler, “Online timed pattern matching using derivatives,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 9636, pp. 736–751, 2016.
122. H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard, “Quantified event automata: Towards expressive and efficient runtime monitors,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 7436 LNCS, pp. 68–84, 2012.
123. A. Pnueli and A. Zaks, “On the merits of temporal testers,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 5000 LNCS, pp. 172–195, 2008.
124. D. Basin, F. Klaedtke, and E. Zălinescu, “Algorithms for monitoring real-time properties,” *Acta Informatica*, vol. 55, no. 4, pp. 309–338, 2018.
125. A. Dokhanchi, B. Hoxha, and G. Fainekos, “On-line monitoring for temporal logic robustness,” *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 8734, pp. 231–246, 2014.
126. K. Havelund, D. Peled, and D. Ulus, “First-order temporal logic monitoring with bdds,” *Formal Methods in System Design*, 2019.
127. T. Zabinski, T. Maoczka, J. Kluska, M. Madera, and J. Sep, “Condition monitoring in industry 4.0 production systems - the idea of computational intelligence methods application,” in *12th CIRP Conference on Intelligent Computation in Manufacturing Engineering, 18-20 July 2018, Gulf of Naples, Italy*, vol. 79, 2019, pp. 63–67.
128. W. Zhang, M.-P. Jia, L. Zhu, and X.-A. Yan, “Comprehensive overview on computational intelligence techniques for machinery condition monitoring and fault diagnosis,” *Chinese Journal of Mechanical Engineering (English Edition)*, vol. 30, no. 4, pp. 782–795, 2017.
129. O. Ljungkrantz, K. Åkesson, M. Fabian, and C. Yuan, “Formal specification and verification of industrial control logic components,” *IEEE Transactions on Automation Science and Engineering*, vol. 7, no. 3, p. 538 – 548, 2010.
130. G. Chen, P. Wei, and M. Liu, “Temporal logic inference for fault detection of switched systems with gaussian process dynamics,” *IEEE Transactions on Automation Science and Engineering*, vol. 19, no. 3, p. 2187 – 2202, 2022.

131. J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia, "Robust online monitoring of signal temporal logic," *Formal Methods in System Design*, vol. 51, pp. 5–30, 2017.
132. S. Jakšić, E. Bartocci, R. Grosu, R. Kloibhofer, T. Nguyen, and D. Ničković, "From signal temporal logic to fpga monitors," in *2015 ACM/IEEE International Conference on Formal Methods and Models for Codesign (MEMOCODE).* IEEE, 2015, pp. 218–227.
133. B. Finkbeiner, M. Fränzle, F. Kohn, and P. Kröger, "A truly robust signal temporal logic: Monitoring safety properties of interacting cyber-physical systems under uncertain observation," *Algorithms*, vol. 15, no. 4, p. 126, 2022.
134. J. Huang, C. Erdogan, Y. Zhang, B. M. Moore, Q. Luo, A. Sundaresan, and G. Roşu, "Rosrv: Runtime verification for robots," in *Runtime Verification*, 2014.
135. A. Ferrando, R. Cardoso, M. Fisher, D. Ancona, L. Franceschini, and V. Mascardi, "Rosmonitoring: A runtime verification framework for ros," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, Dec. 2020, pp. 387–399.
136. A. Francalanza, J. A. Pérez, and C. Sánchez, "Runtime verification for decentralised and distributed systems," in *Lectures on Runtime Verification*, 2018.
137. Embedded Vision Alliance, "Applications for Embedded Vision," https://www.embedded-vision.com/applications-embedded-vision.
138. F. Lin, X. Dong, B. Chen, K.-Y. Lum, and T. Lee, "A robust real-time embedded vision system on an unmanned rotorcraft for ground target following," *IEEE Transactions on Industrial Electronics*, vol. 59, no. 2, pp. 1038–1049, 2012.
139. E. Gudis, P. Lu, D. Berends, K. Kaighn, G. Van Der Wal, G. Buchanan, S. Chai, and M. Piacentino, "An embedded vision services framework for heterogeneous accelerators," in *Proc. of IEEE CS Conference on Computer Vision and Pattern Recognition*, 2013, pp. 598–603.
140. K. Group. (2023) OpenVX: Portable, Power-efficient Vision Processing. [Online]. Available: www.khronos.org/openvx
141. AMD, "AMD OpenVX - AMDOVX," http://gpuopen.com/compute-product/amd-openvx/.
142. INTEL, "Intel Computer Vision SDK," https://software.intel.com/en-us/computer-vision-sdk.
143. M. Popp, S. v. Son, and O. Moreira, "Automatic control flow generation for openvx graphs," in *2017 Euromicro Conference on Digital System Design (DSD)*, 2017, pp. 198–204.
144. D. Dekkiche, B. Vincke, and A. Merigot, "Investigation and performance analysis of OpenVX optimizations on computer vision applications," in *Proc. of IEEE International Conference on Control, Automation, Robotics and Vision*, 2016, pp. 1–6.
145. G. Tagliavini, G. Haugou, A. Marongiu, and L. Benini, "ADRENALINE: An OpenVX Environment to Optimize Embedded Vision Applications on Many-core Accelerators," in *Proc. of IEEE International Symposium on Embedded Multicore/Many-core SoCs*, 2015, pp. 289–296.
146. S. Aldegheri and N. Bombieri, "Extending OpenVX for model-based design of embedded vision applications," in *Proc. of IEEE International Conference on VLSI and System-on-Chip, VLSI-SoC*, 2017, pp. 1–6.
147. E. Rainey, J. Villarreal, G. Dedeoglu, K. Pulli, T. Lepley, and F. Brill, "Addressing system-level optimization with OpenVX graphs," in *IEEE Computer Society Conference on Computer Vision and Pattern Recognition Workshops*, 2014, pp. 658–663.
148. A. Syschikov, B. Sedov, K. Nedovodeev, and V. Ivanova, "Openvx integration into the visual development environment," *International Journal of Embedded and Real-Time Communication Systems*, vol. 9, no. 1, pp. 20–49, 2018. [Online]. Available: www.scopus.com
149. S. Aldegheri and N. Bombieri, "Enhancing performance of computer vision applications on low-power embedded systems through heterogeneous parallel programming," in *Proc. of IEEE International Conference on VLSI and System-on-Chip, VLSI-SoC*, 2018, pp. 1–6.
150. Khronos, "OpenVX lib," https://www.khronos.org/openvx.
151. T. C. Koopmans and M. Beckmann, "Assignment problems and the location of economic activities," *Econometrica*, vol. 25, no. 1, pp. 53–76, 1957.
152. J. Fickenscher, S. Reinhart, F. Hannig, J. Teich, and M. Bouzouraa, "Convoy tracking for adas on embedded gpus," in *Proc. of IEEE Intelligent Vehicles Symposium*, 2017, pp. 959–965.
153. X. Wang, K. Huang, and A. Knoll, "Performance optimisation of parallelized adas applications in fpga-gpu heterogeneous systems: A case study with lane detection," *IEEE Transactions on Intelligent Vehicles*, vol. 4, no. 4, pp. 519–531, 2019.
154. N. Otterness, M. Yang, S. Rust, E. Park, J. Anderson, F. Smith, A. Berg, and S. Wang, "An evaluation of the nvidia tx1 for supporting real-time computer-vision workloads," in *Proc. of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS*, 2017, pp. 353–363.

155. Nvidia Inc., "Nvidia tootlkit documentation, Unified Memory," https://docs.nvidia.com/cuda/cuda-c-programming-guide/index.html#um-unified-memory-programming-hd.
156. J. Cheng, M. Grossman, and T. McKercher, *Professional Cuda C Programming.* John Wiley & Sons, 2014.
157. F. Lumpp, H. Patel, and N. Bombieri, "A Framework for Optimizing CPU-iGPU Communication on Embedded Platforms," in *proceedings of IEEE Design Automation Conference (DAC).* IEEE, 2021, pp. 1–6.
158. E. Lee and S. Seshia, "Introduction to embedded systems—a cyber-physical systems approach," *MIT Press*, 2015.
159. K. Adam, A. Butting, R. Heim, O. Kautz, B. Rumpe, and A. Wortmann, "Model-driven separation of concerns for service robotics," in *DSM 2016 - Proceedings of the International Workshop on Domain-Specific Modeling, co-located with SPLASH 2016*, 2016, pp. 22–27.
160. N. Hammoudeh Garcia, L. Deval, M. Lüdtke, A. Santos, B. Kahl, and M. Bordignon, "Bootstrapping mde development from ros manual code - part 2: Model generation," in *Proceedings - 2019 ACM/IEEE 22nd International Conference on Model Driven Engineering Languages and Systems, MODELS 2019*, 2019, pp. 95–105.
161. G. Bardaro, A. Semprebon, and M. Matteucci, "A use case in model-based robot development using aadl and ros," in *Proceedings - International Conference on Software Engineering*, 2018, pp. 9–16.
162. M. Wenger, W. Eisenmenger, G. Neugschwandtner, B. Schneider, and A. Zoitl, "A model based engineering tool for ros component compositioning, configuration and generation of deployment information," in *IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, vol. 2016-November, 2016.
163. P. Neis, M. Wehrmeister, and M. Mendes, "Model driven software engineering of power systems applications: Literature review and trends," *IEEE Access*, vol. 7, pp. 177 761–177 773, 2019, cited By 6. [Online]. Available: https://www.scopus.com/inward/record.uri?eid=2-s2.0-85077213525&doi=10.1109%2fACCESS.2019.2958275&partnerID=40&md5=2462fd65ed3be7846a22631f3536726a
164. E. Estévez, A. Sánchez-García, J. Gámez-García, J. Gómez-Ortega, and S. Satorres-Martínez, "A novel model-driven approach to support development cycle of robotic systems," *International Journal of Advanced Manufacturing Technology*, vol. 82, no. 1-4, pp. 737–751, 2016.
165. Matlab. (2021) MATLAB and Simulink for Robotics and Autonomous Systems. [Online]. Available: https://www.mathworks.com/solutions/robotics.html
166. Robmosys. (2021) Composable Models and Software for Robotics Systems. [Online]. Available: https://robmosys.eu/
167. J. Cacace, N. Mimmo, and L. Marconi, "A ros gazebo plugin to simulate arva sensors," in *Proceedings - IEEE International Conference on Robotics and Automation*, 2020, pp. 7233–7239.
168. E. Sita, C. Horváth, T. Thomessen, P. Korondi, and A. Pipe, "Ros-unity3d based system for monitoring of an industrial robotic process," in *SII 2017 - 2017 IEEE/SICE International Symposium on System Integration*, vol. 2018-January, 2018, pp. 1047–1052.
169. N. Zhou, Y. Georgiou, M. Pospieszny, L. Zhong, H. Zhou, C. Niethammer, B. Pejak, O. Marko, and D. Hoppe, "Container orchestration on hpc systems through kubernetes," *Journal of Cloud Computing*, vol. 10, no. 1, 2021.
170. H. Fathoni, C.-T. Yang, C.-H. Chang, and C.-Y. Huang, "Performance comparison of lightweight kubernetes in edge devices," *Communications in Computer and Information Science*, vol. 1080 CCIS, pp. 304–309, 2019.
171. S. Aldegheri, N. Bombieri, S. Germiniani, F. Moschin, and G. Pravadelli, "A containerized ros-compliant verification environment for robotic systems," in *2021 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, 2021, pp. 222–225.
172. ROS. Control. [Online]. Available: http://wiki.ros.org/ros_control
173. Gazebo. Control. [Online]. Available: http://gazebosim.org/tutorials/?tut=ros_control
174. A. K. Jain and R. C. Dubes, *Algorithms for Clustering Data.* USA: Prentice-Hall, Inc., 1988.
175. rosdep tool. [Online]. Available: https://wiki.ros.org/rosdep
176. Docker, "Configure networking," 2021, docs.docker.com/network.
177. Eurohpc risc-v. [Online]. Available: https://eurohpc-ju.europa.eu/new-call-developing-hpc-ecosystem-based-risc-v-2023-02-01_en
178. Risc-v wiki. [Online]. Available: https://wiki.riscv.org/display/HOME/RISC-V+Software+Ecosystem
179. A. Bartolini, F. Ficarelli, E. Parisi, F. Beneventi, F. Barchi, D. Gregori, F. Magugliani, M. Cicala, C. Gianfreda, D. Cesarini *et al.*, "Monte Cimone: Paving the Road for the First Generation of RISC-V High-Performance Computers," in *2022 IEEE 35th International System-on-Chip Conference (SOCC).* IEEE, 2022, pp. 1–6.

180. KubeEdge. (2023) KubeEdge. [Online]. Available: https://kubeedge.io
181. T. Iqbal, S. Rack, and L. D. Riek, "Movement coordination in human-robot teams: A dynamical systems approach," *IEEE Transactions on Robotics*, vol. 32, no. 4, pp. 909–919, 2016, cited By :45. [Online]. Available: www.scopus.com
182. Deployments. [Online]. Available: kubernetes.io/docs/concepts/workloads/controllers/deployment/
183. sched_deadline: Implement "runtime overrun signal" support. [Online]. Available: git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=34be39305a77b8b1ec9f279163c7cdb6cc719b91
184. M. Felser, "Real-time ethernet - industry prospective," *Proceedings of the IEEE*, vol. 93, no. 6, pp. 1118–1129, 2005.
185. A. T. Praveen, A. Gupta, S. Bhattacharyya, and R. Muthalagu, "Assuring behavior of multi-robot autonomous systems with translation from formal verification to ros simulation," *IEEE Systems Journal*, 2022.
186. R. Halder, J. Proença, N. Macedo, and A. Santos, "Formal verification of ros-based robotic applications using timed-automata," in *IEEE/ACM International FME Workshop on Formal Methods in Software Engineering (FormaliSE)*, 2017, pp. 44–50.
187. T. Klotz, J. Schonherr, N. Sesler, B. Straube, and K. Turek, "Automated formal verification of routing in material handling systems," *IEEE Transactions on Automation Science and Engineering*, vol. 10, no. 4, p. 900 – 915, 2013.
188. H. Foster, *Applied Assertion-Based Verification: An Industry Perspective.* Now Foundations and Trends, 2009.
189. X. Zheng, C. Julien, R. Podorozhny, and F. Cassez, "Braceassertion: Runtime verification of cyber-physical systems," in *Proceedings - 2015 IEEE 12th International Conference on Mobile Ad Hoc and Sensor Systems, MASS 2015*, 2015, pp. 298–306.
190. H. Ko, M. Jo, and V. Leung, "Application-aware migration algorithm with prefetching in heterogeneous cloud environments," *IEEE Transactions on Cloud Computing*, 2021.
191. O. Maler and D. Nickovic, "Monitoring temporal properties of continuous signals," in *FORMATS/FTRTFT*, 2004.
192. R. Mur-Artal and J. D. Tardós, "ORB-SLAM2: an open-source SLAM system for monocular, stereo and RGB-D cameras," *IEEE Transactions on Robotics*, vol. 33, no. 5, pp. 1255–1262, 2017.
193. NVIDIA. (2021) Deep-learning inference networks and deep vision primitives with TensorRT and NVIDIA Jetson. [Online]. Available: https://github.com/dusty-nv/jetson-inference
194. A. Geiger, P. Lenz, C. Stiller, and R. Urtasun, "Vision meets robotics: The kitti dataset," *International Journal of Robotics Research (IJRR)*, 2013.
195. F. Kong, M. Polo, and A. Lambert, "Centroid estimation for a shack-hartmann wavefront sensor based on stream processing," *Applied optics*, vol. 56, no. 23, 2017.
196. F. Lumpp, M. Panato, F. Fummi, and N. Bombieri, "A container-based design methodology for robotic applications on kubernetes edge-cloud architectures," in *Forum on Specification and Design Languages*, vol. 2021-September, 2021.
197. Rt-tests. [Online]. Available: git.kernel.org/pub/scm/utils/rt-tests/rt-tests.git