Evaluation of OpenStack as Middleware for Automotive Embedded Systems

AUTHOR: MAX MALINOWSKI

A thesis submitted in fulfillment of the
requirements for the degree
Bachelor of Science
in the course of studies
Aircraft and Vehicle Informatics
of the faculty
Computer Science

Declaration of Authorship

I, Max Malinowski, hereby declare that this thesis is my own work, that I have not presented it elsewhere for examination purposes, and that I have not used any sources or aids other than those stated. I have marked verbatim and indirect quotations as such.

....

          Date   Max Malinowski

Abstract

Cloud Computing is already in extensive use by many car manufacturers in their backend, providing scalability, performance, and availability. However, the in-vehicle software is still definitely bound and statically executed on particular hardware. This thesis explores the basic feasibility and performance of operating an OpenStack cloud on automotive hardware for in-vehicle use. For evaluation, the introduced overhead and the cases of VM distribution and VM migration are considered and examined. Two automotive ARM SoCs and two high-performance x86 platforms are used for measurements and evaluation. The results indicate that on both hardware, OpenStack is executable and operable. On performable hardware, the results reveal OpenStack to introduce a low overhead and function reliably. However, low resources can influence OpenStack's performance. Furthermore, VM migration and distribution were found to be practicable, although not for real-time applications.

Table of Contents

Abbreviations

ECU   Electrical Control Unit

VM   Virtual Machine

SoC   System on a Chip

NIST   National Institute for Standards and Technology

IaaS   Infrastructure-as-a-Service

PaaS   Platform-as-a-Service

SaaS   Software-as-a-Service

OS   Operating System

CPU   Central Processing Unit

KVM   Kernel-based Virtual Machine

API   Application Programming Interface

RPC   Remote Procedure Call

SDK   Software Development Kit

vCPU   Virtual CPU

SSH   Secure Shell

EPS   Events per Second

VNC   Virtual Network Computing

ATF   ARM Trusted Firmware

List of Figures

List of Tables

List of Listings

1. Introduction

At the latest, since automated driving is gaining increased attention, it becomes clear that the automotive industry is not only about mechanics anymore. Information technology and computers play a big part in today's vehicles and are responsible for crucial tasks and controls. Today's vehicles consist of over 100 Electrical Control Units (ECUs), as shown by Figure 1.1, creating a big network of distributed hardware. A similar topology can be found in other distributed systems like data centers, where a variety of different, locally distributed hardware provides various services. However, data centers mostly do not provide the hardware separately or bind applications to specific servers. Instead, using a cloud middleware, the hardware and its resources are provisioned on-demand and independent from the actual hardware. With topics like image recognition, artificial intelligence, and data processing of, for example, sensor data, new requirements arise for automotive embedded hardware. Cloud Computing is already being used for similar tasks and is gaining popularity in general [11, 14]. The current usage of Cloud Computing in the backend, the similarities between data centers and vehicle topologies, and the benefits of extending the backend cloud into the vehicle [36], suggest a great potential for an in-vehicle cloud.
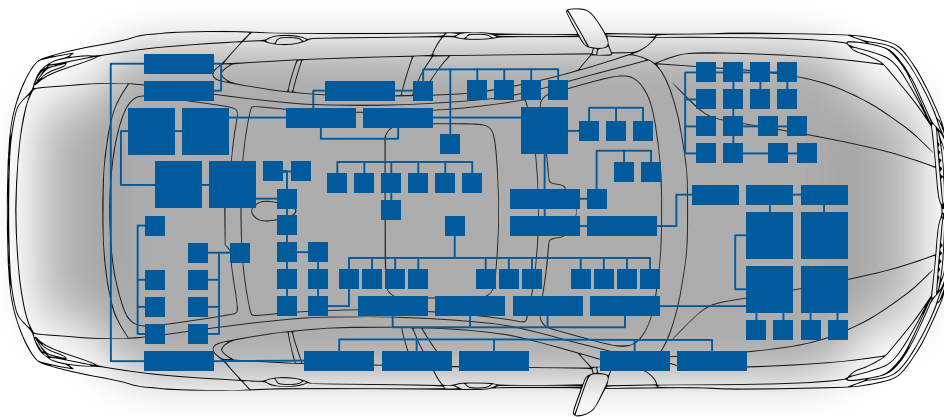
Figure 1.1: Schematic ECU distribution in modern vehicles according to [6, 30]

1.1 Problem Statement

"Cloud is about how you do computing, not where you do computing."
$\sim$*Paul Maritz, CEO of VMware [21]*

In various disciplines and areas, the future is already introducing unknown challenges. Challenges like autonomous driving or electric vehicles call for new approaches and require rethinking today's situation. The increasing demand for computational power and performance contradicts the embedded system characteristics, being very resource-aware and efficient. The rising number of ECUs installed in vehicles already exceeds 100 and makes their management nearly impossible, not even considering the software. Through various undertakings towards domain or vehicle centric E/E-Architectures [38], the number of ECUs will decrease, however, multiple ECUs will still be present in a vehicle. Developing software for embedded systems requires engineers not only to solve tomorrow's challenges but also to face more restrictions and environmental factors than in most software development areas. Restrictions like cost, weight, or efficiency lead to highly optimized software at the expense of a challenging and complex development process and possibly abandoned functionality.

To manage requirements and standardize automotive software development, Autosar plays an important role. Nevertheless, with the Autosar Classic documentation consisting of over 20.000 pages in the latest release, the standard's complexity is exploding. While Autosar enables the development of applications and functionality independently from the hardware, from a certain point, the software has to be statically assigned to specific hardware and, therefore, ECUs. As Paul Maritz said, today's challenges force us to concentrate on the problem itself and on the *how* we compute rather than concern ourselves with the *where* we compute. With the high number of ECUs, the rising software quantity and complexity, along with casual embedded system constraints and safety requirements, concentration on the *where* often shadows the *how*.

Figure 1.2a depicts schematically how applications are currently assigned to particular ECUs. Cloud Computing already addresses these static assignations through a middleware representing separate, distributed systems as one pool of shared resources. Although Cloud Computing is used in many areas, including the automotive backend, it is yet to be used inside the vehicle. Using a middleware that virtualizes and provisions resources to Virtual Machines (VMs) could facilitate the distribution of applications and software among multiple ECUs, as depicted by Figure 1.2b, and overcome the static assignations.

(a) Today: Applications → Hardware

(b) Vision: Applications → Shared Pool
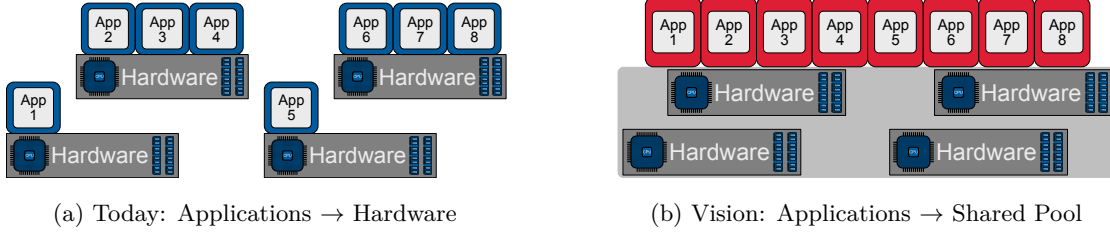
Figure 1.2: Symbolic depiction of distributed and pooled resources

1.2 Goals and Research Question

Providers of Cloud Computing services like Amazon with Amazon Web Services or Microsoft with Microsoft Azure only provide the actual services. In contrast, the Open-Source project OpenStack [7] aims at providing all the necessary tools and software to create an own cloud infrastructure. With OpenStack, one can deploy a cloud infrastructure on any hardware with sufficient resources. This thesis focuses on using OpenStack and its software components for providing computational resources on Systems-on-a-Chip (SoCs) in embedded systems. Therefore, the goal of the thesis presented shall be the successful execution of an OpenStack cluster on automotive embedded hardware. However, OpenStack's primary target hardware are servers in data centers. By consolidating different hardware into one OpenStack cluster, all resources can be pooled, represented as one shared pool, and provisioned to applications inside VMs. While embedded systems represent an area with specific requirements, OpenStack is developed for typical server environments. These contrasts and contradictions have an impact on the use of OpenStack on embedded systems. The extent of this impact shall be the research object of this thesis. Therefore, the main research question is:

Is it possible to leverage the advantages of Cloud Computing on embedded hardware?

This question can be further divided into the following goals:

1. **Install and execute OpenStack on embedded automotive hardware.** As neither OpenStack is intended for embedded hardware, nor embedded hardware is intended for software like OpenStack, the installation has to be successful prior to actual evaluation. Due to custom Operating Systems (OSs) and kernels, as well as limited resources, the installation might pose difficulties.

2. **Determine the impacts of Cloud Computing on embedded systems.** Due to neither working in the typical data center environment nor working with typical embedded software, special requirements and environmental constraints

for OpenStack are present. It has to be determined how the contradictory requirements affect the embedded system and also OpenStack. On the other hand, it has to be determined whether and which Cloud Computing advantages can be leveraged on embedded systems.

3. **Measure and evaluate identified impacts.**
   Having determined the positive and negative impacts of Cloud Computing, these have to be objectively quantified. To evaluate whether the impacts are substantial or negligible, measurements have to be performed in a defined way.

4. **Evaluate OpenStack's performance on embedded hardware.**
   With impacts and their metrics defined, a cloud infrastructure on embedded hardware can be evaluated. Executing tests for the relevant metrics enables an assertion on OpenStack's performance on embedded hardware.

1.3   Structure of this Thesis

The answer to the main research question is elaborated along the following structure: First, Chapter 2 establishes the basics in, for this thesis, relevant areas. Second, in Chapter 3, the preparation of the hardware used for the testing and the installation process of OpenStack itself is described. In Chapter 4, the test methodology for accessing OpenStack's overhead is presented, followed by the discussion of the results in Chapter 5. Chapter 6 further examines two exemplary use cases as advantages of OpenStack, leading to a conclusion and outlook in Chapter 7.

2. Basics

In order to reach the defined goals of this thesis, more profound knowledge in different areas is necessary. Besides Cloud Computing and virtualization, common characteristics and features of embedded systems should be known as well. Also, familiarization and understanding of OpenStack are crucial. This chapter aims at establishing the fundamentals needed to understand this thesis.

2.1 Cloud Computing

Considering the importance and extensive usage of Cloud Computing in today's world, it is hard to define the exact meaning. In 2011, the National Institute for Standards and Technology (NIST) defined *Cloud Computing* in a special publication. According to NIST, Cloud Computing enables *"ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction"* [20]. Due to the broad utilization of Cloud Computing through third parties like Amazon or Microsoft, terms like pay-as-you-go or payment, in general, are often perceived as part of the definition [2, 29]. Although this often applies, strictly reviewing, Cloud Computing only refers to the provisioning of resources. The business models of companies do not relate to the concept itself.

Additionally, in [20] NIST established five essential characteristics of Cloud Computing. Through *on-demand self-service* and *broad network access*, consumers can provision computing capabilities without human interaction over the network. The providers' resource pooling allows serving computing resources to multiple consumers. *Rapid elasticity* and a *measured service* automatically control and optimize resource use by elastically provisioning and releasing them.

2.1.1   Service Models

As mentioned previously, the main component of Cloud Computing is the provisioning of resources. This provisioning can be done on different levels, enabling different possibilities for the user. NIST defined the following three service models for Cloud Computing, depicted by Figure 2.1.

Software-as-a-Service (SaaS) As the name suggests, only the software is provisioned. In this model, the software is executed on cloud infrastructure and consumers can access it through thin-client interfaces like web browsers. The consumers do not manage or even have access to the underlying cloud infrastructure or the environment like the OS.

Platform-as-a-Service (PaaS) The service provisioned here is a platform for deployment of applications. The provider offers control over deployed applications and possibly over environmental settings for hosting applications. The consumers do not manage or even have access to the underlying cloud infrastructure.

Infrastructure-as-a-Service (IaaS) The consumer has access to fundamental computing resources and is able to provision them. The user has no access or control over the underlying hardware but can run arbitrary software and control the OS, storage, and deployed applications. This thesis aims at establishing an IaaS cloud on the automotive hardware presented in Chapter 3 using OpenStack.
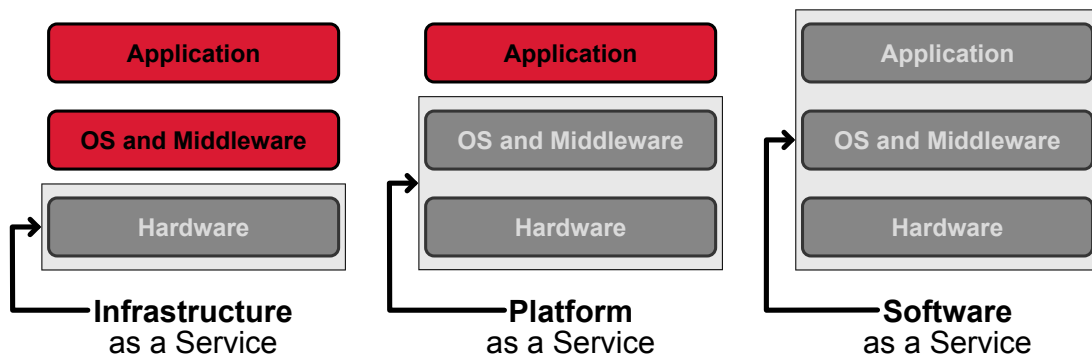
Figure 2.1: Cloud computing service models

2.1.2   Architectures

Except for different service models, Cloud Computing can be characterized by its architecture. Depending on the accessibility to the infrastructure, the following architectures are defined.

Private Cloud The infrastructure is available for only one organization. Related consumers, e.g., business units, can access and use the infrastructure. The infrastructure itself can be managed either by the organization itself, a third party, or a combination of both. Throughout this thesis, a private architecture is used, as no access for other parties is considered.

Community Cloud In addition to private clouds, in community clouds, the usage is not exclusive to one organization. Organizations with shared interests and concerns have access to the infrastructure and can access the provided services. The infrastructure itself can be managed by either one of the organizations itself, a third party, or a combination of both.

Public Cloud As the name suggests the cloud infrastructure is accessible by the general public for use. Various institutions, like businesses, academics, or governments, can be responsible for managing the cloud infrastructure.

Hybrid Cloud This infrastructure is a composition of two or more different cloud infrastructures. Both clouds remain unique entities but are connected to each other for, for example, data exchange.

2.1.3 Advantages and Disadvantages

Cloud Computing is not without reason a prevalent and beneficial model. Various reasons exist why Cloud Computing should be adopted, from a technical and business point of view. However, Cloud Computing also comes with negative aspects, especially for embedded systems. Based on [1, 10], some advantages and disadvantages of Cloud Computing shall be pointed out.

2.1.3.1 Advantages

Cloud Computing introduces a new degree of **flexibility**. Virtualizing available, distributed resources as one coherent pool makes the allocation of applications to specific systems like a specific ECU redundant.

Moreover, **scalability** and, therefore, **performance** can be tremendously improved. Regarding the topic of upgradability, in current embedded systems, this is not even considered or possible. Through Cloud Computing, upgrading or simply adding resources, for example, computing power or memory, would instantaneously become possible. Especially in today's environmentally and resource-conscious society, this could be an interesting topic.

Additionally, Cloud Computing is ideal if **redundancy** is needed. Considering embedded systems with high safety requirements, this is essential. While today additional hardware is used, Cloud Computing could enable applications to be easily shifted or duplicated in case of hardware failure. Chapter 6 examines the advantages of flexibility and redundancy in greater detail.

2.1.3.2 Disadvantages

Deploying and managing cloud infrastructures introduces a significant level of **complexity**. Understanding the exact behavior of the whole software or software components becomes impossible.

This introduces new problems in the areas of **security**. Especially for embedded systems, where updating software is not necessarily intended, security flaws and vulnerabilities are critical. With big and complex software, vulnerabilities can be overlooked and later discovered during production. While in data centers, this can quickly be resolved through updates, embedded systems are scarcely updated. Using cloud infrastructure could therefore create a conflict with this tradition

Further, this contradicts embedded systems with high safety requirements that must have **deterministic behavior**. Besides the inability to understand the whole software, possible exploitation of security flaws adds to the fact that it is impossible to know the software's behavior with confidence.

2.2 Virtualization

Cloud Computing relies on the provisioning of resources. To do that, resources like the Central Processing Unit (CPU), CPU cycles, or memory have to be virtualized. Virtualization describes a real-world object's mapping onto a virtual object with the same or similar properties [8]. This allows applications and services to be deployed onto the virtualized hardware instead of directly onto the physical resources. From now on, the hardware providing the resources is called the *host*, while the software, for example, a VM, using those resources is called the *guest*. The management of the virtualized resources is handled by a software called *hypervisor*. The following subsections will describe two common concepts of virtualization, as well as two common hypervisors using these concepts.

2.2.1 Hypervisors

Hypervisors are systems or pieces of software initially developed for monitoring purposes in virtual computing [5]. In addition, hypervisors are responsible for the allocation of physical resources to individual guests. Guest VMs, for example, can execute their own OS and run as an individual computing platform. By providing a hardware abstraction layer, hypervisors provide interfaces for guests to interact with the physical hardware. Two widely used hypervisors are Kernel-based Virtual Machine (KVM) and XEN. In this thesis, the OpenStack cluster will use KVM as the hypervisor.

2.2.2 Virtualization Concepts

For the virtualization of resources, especially the CPU and CPU cycles, two main approaches established themselves. Their main difference resides in the hypervisor and the hypervisors reaction to non-virtualizable machine instructions. During execution, atomic machine instructions define the actions performed by the processor. While most machine instructions are independent of specific architectures, some instructions behave differently or require specific privileges [17]. As stated by Popeck and Goldberg already in 1974, to provide efficient behavior to guests, a certain percentage of instructions must be executed without the hypervisors intervention [28]. In cases where hypervisors' interventions are necessary, the instructions require a specific behavior not known by the hardware. To resolve such non-virtualizable instructions, two different strategies can be used, resulting in two different virtualization approaches visualized by Figure 2.2.

Full-Virtualization Guests run on a hardware abstraction layer, being an exact copy of the host hardware [17]. In the case of non-virtualizable instructions, the hypervisor identifies these instructions in memory during execution and replaces them with suitable instructions [4]. The guest is not aware of this replacement and continues its execution unhindered. Therefore, no modifications need to be performed to the guest system in advance, but the host hardware must support hardware virtualization to provide an efficient solution. KVM is a widely adopted hypervisor using full-virtualization.

Para-Virtualization Guests run on a hardware abstraction layer, being a slightly modified copy of the host hardware [17]. In the case of non-virtualizable instructions, the hypervisor provides a specific interface to the virtualized hardware to translate these instructions [4]. The guest must be extended with specific drivers to make use of this interface. Therefore modifications to the guest are needed, but contrary

to full-virtualization, para-virtualization does not require hardware support for virtualization. XEN is a popular hypervisor using para-virtualization.
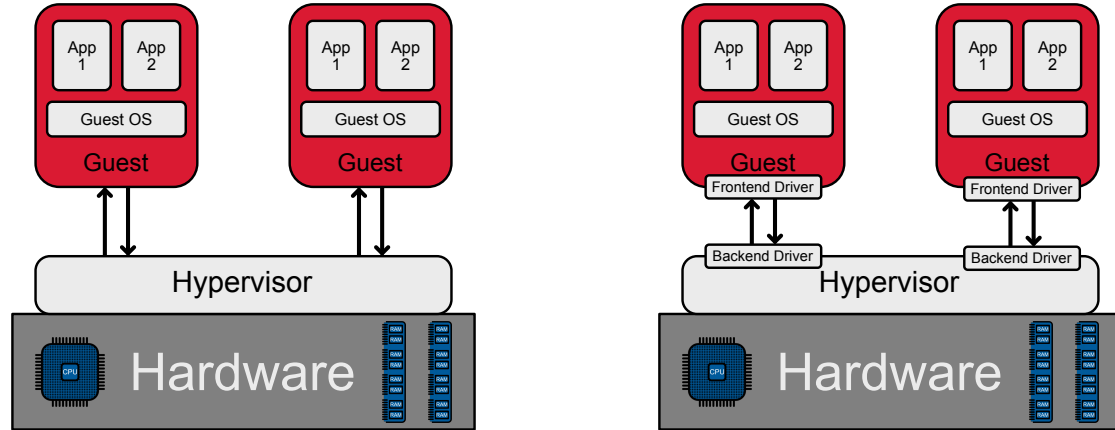
Figure 2.2: Full-Virtualization (left) and Para-Virtualization (right)

2.3 Embedded Systems

In today's world, embedded systems are widely adopted and in use. From areas like robotics and the Internet of Things, over automotive and avionic systems to nuclear power plants, embedded systems perform various essential tasks. Although these areas all come with different requirements, the function of embedded systems mostly stays the same. P. Marwedel defined embedded systems as *"information processing systems embedded into enclosing products"* [18]. Often, embedded systems are linked to the physical world and interact with physical processes. In such cases, the systems can be named *Cyber-Physical Systems*. In [15], E. Lee defined Cyber-Physical Systems as *"integrations of computation and physical processes"*. The following sections are based on P. Marwedel's book on Embedded Systems Foundations of Cyber-Physical Systems [18].

2.3.1 Characteristics

Despite different environments, embedded systems share several characteristics and challenges these systems have to meet. According to P. Marwedel, the following attributes are commonly found:

Dependability Due to the direct interaction with the environment, embedded systems have an immediate effect and impact on the physical world. Additionally, as described later, the lack of a minimal user interface makes a human reaction in case of failure or misbehavior very hard or even impossible. Therefore, embedded

systems must be dependable by providing safety, security, confidentiality, reliability, maintainability, and availability.

Time Constraints The interaction with physical processes binds the execution or computation of specific things to determined deadlines. Depending on the computation and the physical process involved, these limits can be either short or long. Moreover, depending on the physical process, a deadline miss can be either catastrophic or less noticeable.

Resource Awareness Representing not the primary system but being embedded into an encapsulating product, embedded systems often are restricted in various resources. Resources like code size and performance or run-time efficiency are optimized as far as possible. Through this, further resources like energy, weight, or space can be optimized. Due to very high production volumes of embedded systems, for example, automotive ECUs, the cost is a crucial factor. By minimizing any resource usage, general costs can be reduced, which is very much desirable.

Dedicated User Interface The integration into encapsulating products implies that the interaction will most probably happen with the outer product. Therefore, user interfaces of embedded systems are specific to the application and held as minimal as possible. If no interaction is anticipated, no user interface is provided. Only interfaces for debugging or flashing may then be implemented, consisting of simple switches or interfaces for console access.

2.3.2 Software

To successfully overcome challenges like time constrains and dependability, OSs for embedded systems are specially tailored. Limited resources and defined applications make it possible and necessary to specifically configure OSs according to individual needs. The lack of untested and unnecessary software allows the assumption about the presence of only reliable and safe software, which further allows to leave out standard protection mechanisms. For the embedded environment, access to OS service calls or interrupts is essential to react fast and on time to specific events, like sensor data. As many embedded systems do not underlie only time constraints but must be real-time capable, further functionality is required by the OS.

Real-Time OS Real-Time OSs are necessary for the construction of real-time systems. Real-Time systems describe systems, which not only depend on the functional correctness but equally depend on the in-time computation of results. In other words, a particular deadline for a computation must not be missed, or else catastrophic events could occur. To ensure such deadlines are not missed, real-time

OSs must provide very predictable timing. The OS must manage the scheduling of threads, processes, and tasks through appropriate scheduling algorithms to enable an in-time computation of every task. Naturally, the OS must additionally be fast and deliver a good performance. These requirements are achieved through the use of fast, customized, and proprietary kernels, as well as with real-time extensions to standard OSs.

Embedded Linux With more complex applications and more requirements, embedded OSs must provide increased functionality. When developing new software components and integrating them, using an existing and extensively tested OS can be preferable. Using Linux, which relies on an Open-Source kernel, eliminates commercial costs and allows the kernel's desired configurability. The primary target environment for Linux being servers and desktops, Linux is designed for general-purpose computing. Aiming for high average performance and fair scheduling, additional scheduling policies must be added to the kernel to satisfy real-time requirements. Similarly, the usage of memory differs in desktop and server environments. While in desktop and server environments, the memory is often used for caching, this is a waste of resources in the context of embedded systems. Although there are techniques to eliminate this duplication of memory, they come at the cost of high complexity and often more expensive storage solutions. This again leads to the reconsideration of allowing, for example, swap space and using larger memory.

2.4 OpenStack

OpenStack is a free and Open-Source Cloud Computing software. It aims at providing *"a ubiquitous Open Source Cloud Computing platform that is easy to use, simple to implement, interoperable between deployments, works well at all scales, and meets the needs of users and operators of both public and private clouds"* [24]. In other words, OpenStack enables the deployment of own clouds on the hardware of choice. The project itself originated from two separate projects at Rackspace and Anso Labs, developing both the corresponding counterparts for each other. In 2010, the project OpenStack was officially announced and the first release published. At the time of writing, 22 releases happened, with the newest one having the series name *Victoria*. This release was used throughout this thesis for all OpenStack software and components.

OpenStack is built upon multiple different software components, each providing specific functionality. Currently, OpenStack consists of 45 different components. Figure 2.3 shows all available OpenStack software components along with additional

software and tools. The relevant components for this thesis will be presented in the next section.
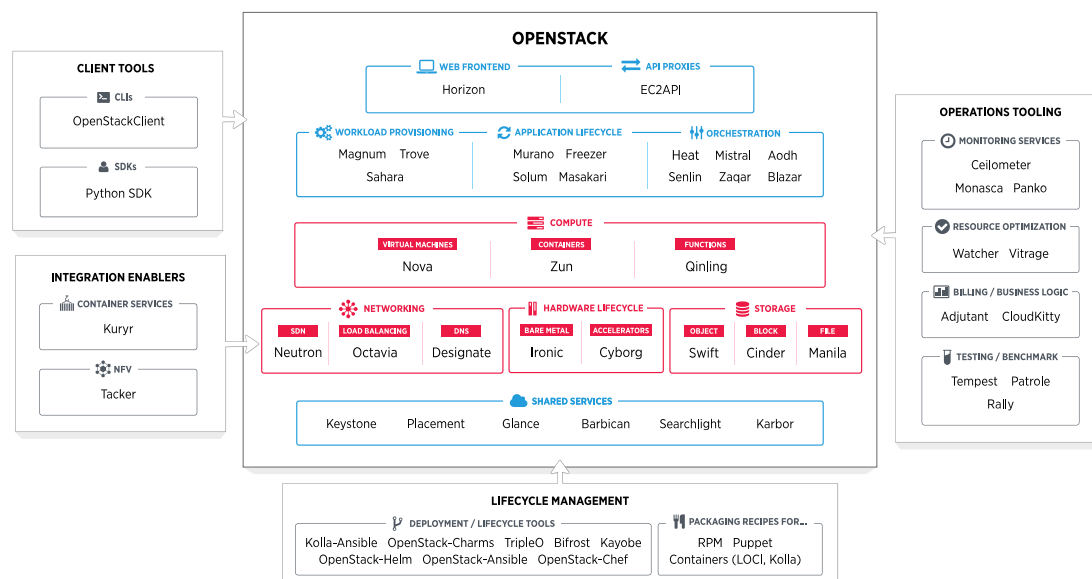
Figure 2.3: Map of the OpenStack's software landscape [27]

With OpenStack's numerous components and their primarily intended environment being data centers, an OpenStack cluster is designed to consist of multiple hosts. However, it is possible to install and execute all services on one host. Commonly, an OpenStack cluster consists of one controller node and multiple compute nodes. Depending on the goals and resources, dedicated storage and networking nodes are possible. Throughout this thesis, only dedicated compute nodes will exist, while all other services and components will be executed on one controller node.

2.4.1 Software Components

As depicted in Figure 2.3, many different software components exist. For the scope of this thesis, only the crucial components of an OpenStack cluster will be used. Figure 2.4 illustrates these components along with their interactions. The following sections briefly describe these components.

2.4.1.1 Shared Services

Regarding shared services, the identity service *Keystone* and *Glance* are critical to a functioning OpenStack cluster.

Keystone provides authentication for clients in the OpenStack environment. Clients hereby refer to other OpenStack components, which need to authenticate their
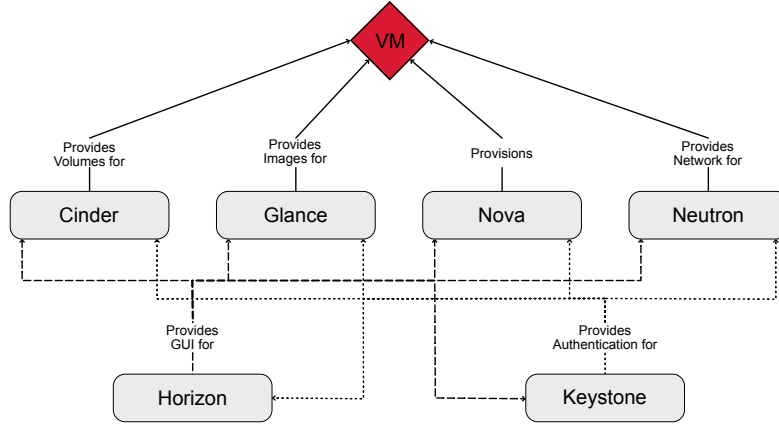
Figure 2.4: Interaction of OpenStack components [34]

operations, as well as to actual clients, authenticating themselves to access the OpenStack cluster.

Glance provides the functionality of registering and retrieving images for VMs. The actual images can be stored in various locations, and using Glance's Application Programming Interface (API), image metadata and the actual retrieval of the image is possible.

Although not essential, the placement services *Placement* provides the functionality of tracking cloud resources and their usage. Using an API, Placement helps other services to manage and allocate their resources accordingly, which is an important factor in this thesis's context.

2.4.1.2 Web Frontend

Although not strictly necessary for a functioning OpenStack Cluster, it was decided to use the canonical web interface *Horizon*. While also in an automotive or embedded context, a web interface would not be used, for the research and development performed in this thesis, it considerably increases the usability of OpenStack.

2.4.1.3 Computing Services

The computing service *Nova* provides the provisioning of compute instances. Nova provisions instances via VMs. Throughout this thesis, only VMs will be used to provision and consume compute services. Nova consists of three core services being the *Nova-Conductor*, *Nova-Scheduler*, and *Nova-Compute*.
Nova-Compute represents the resource provisioning. Its main objective is the communication with the hypervisor on the host and the VM.
The Nova-Scheduler is responsible for choosing appropriate computing hosts. Based on a broad space of parameters, the scheduler can be configured to distribute VMs

based on specific rules to compute hosts. It first filters all eligible hosts for, for example, satisfying the required resources, and second determines the best host based on the mentioned parameters. Chapter 6 will examine the Nova-Scheduler in greater detail.

The Nova-Conductor handles all requests requiring cluster-wide coordination. Figure 2.5 shows the Nova service and the external components it is dependent upon.
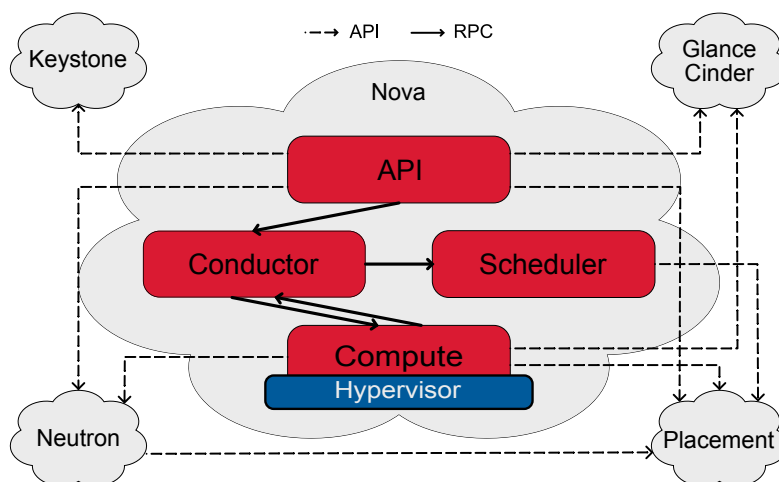
Figure 2.5: OpenStack Nova architecture [26]

2.4.1.4 Networking Services

For the networking services, the OpenStack component *Neutron* is responsible. It focuses upon the delivery of Networking-as-a-Service to other OpenStack services, for example, to Nova-Compute instances. Neutron also supports creating multiple, on-demand networks along with all necessities like routers, switches, dynamic host configuration protocol, or domain name system services.

2.4.1.5 Storage Services

OpenStack offers three different storage services. In this thesis, only the block-storage service *Cinder* will be considered. Cinder virtualizes and provides block-storage volumes to other components like Glance as well as for VMs themselves as storage.

2.4.2 Communication Services

OpenStack communication mainly relies on two paradigms, which are used by all components and services. Following, APIs and Remote Procedure Calls (RPCs) will be briefly described.

2.4.2.1 APIs

All major OpenStack components contain an API service. These APIs expose one or more endpoints to the outside world to interact with. Other components and users themselves can use those endpoints to trigger actions or transfer data. The OpenStack command-line-tool and the Python Software Development Kit (SDK) make heavy use of the API services to enable users to interact with OpenStack.

2.4.2.2 RPCs

Apart from APIs, OpenStack heavily relies on RPCs for client-server intra-service communication. For example, if the creation of a new VM is requested, the Nova-Conductor (running on the controller) invokes Nova-Computer to build the VM on a compute host via an RPC. This is also indicated in Figure 2.5, where the communication inside the Nova component happens via RPCs, while the external communication happens via APIs. The implementation of RPCs is achieved through the Advanced Message Queuing Protocol AMQP. All RPCs are packed into messages, which are then sent to a message broker, who further distributes the messages to the proper destinations.

3. Host and Test Setup

To perform measurements and evaluate OpenStack, appropriate hardware is necessary. This hardware must also feature a certain OS, and the OpenStack components must be installed and configured. As embedded systems are not intended for Cloud Computing software like OpenStack, the installation and successful execution process includes difficulties. This chapter presents the hardware and software setup used for testing.

3.1 Hardware Setup

In order to study OpenStack on embedded systems, two different hardware platforms were chosen. To evaluate a broader spectrum of hardware, one platform was chosen to be x86-based, while the other was chosen to be an ARM-based platform. For the x86-based platform, an Intel Xeon Platinum CPU with 24 cores was chosen. Due to having much computational power, systems based on this CPU are currently used for developing autonomous driving. Therefore, this platform represents a powerful x86 embedded system intended for future use in automotive applications. The x86 platform comes further with 188 GB of memory and 960GB of NVMe secondary storage connected via a M.2 interface.

A Renesas R-Car Starter Kit Pro, featuring a Renesas R-Car H3 SoC, was chosen for the ARM-based platform. With eight cores in a big.LITTLE architecture, this SoC has four high-performance and four high-efficiency cores. It is specifically designed for the automotive industry and is currently used in automotive ECUs. The ARM platform comes with 4 GB of memory and a 64 GB SD-Card as secondary storage connected via an SD-Card interface.

Table 3.1 shows the specifications of the hardware used. Both platforms are equipped with 1 Gbit network interfaces. The x86 platforms additionally come with a 10 Gbit network interface.

Of each platform respectively, two systems were chosen to be used in the testing setup. As described in Section 2.4, an OpenStack cluster consists of different

Platform	Processor	Memory	Storage	Network
x86	Intel Xeon Platinum 8160T @ 3.60 GHz (24 Cores)	188 GB	960 GB	1x 1 Gbit 1x 10 Gbit
ARM	Renesas R-Car H3 4x Cortex-57 and 4x Cortex-53	4 GB	64 GB	1x 1 Gbit

Table 3.1: Hardware and specifications of the host systems used for testing

core services and additional separate components. Due to the high computational power and high memory, it was decided to configure one x86 platform as the controller node. The other three nodes (2x ARM and 1x x86) were chosen to be configured as dedicated compute nodes. Figure 3.1 shows the connection between all platforms, creating the OpenStack cluster. All platforms are connected via a Layer 2 1 Gbit Ethernet switch with Cat 5e Ethernet cables. However, the x86 platform configured as a compute node is in addition connected directly via a 10 Gbit Ethernet connection to the controller node. Further, the Ethernet switch is connected to a gateway via a 1 Gbit connection. The external network, though, is only used for installation and update purposes.
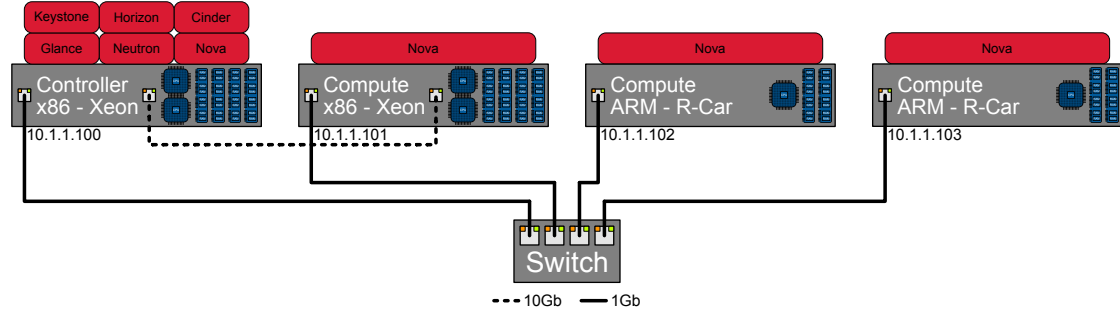
Figure 3.1: OpenStack hardware and network setup

3.2 Software Installation

The installation process of all software consists of two stages. As a first step, the host system must be prepared. This preparation includes the OS, the correct configuration of necessary settings, and the installation of necessary packages and dependencies. As a second step, the OpenStack follows, including the installation and configuration of the OpenStack cluster.

3.2.1 Host System Preparation

Before installing the OpenStack services, the host systems must be prepared accordingly. For both platforms, Ubuntu 18.04 was chosen as the host OS because of

two reasons. First, Ubuntu is a Linux distribution supported and recommended by OpenStack, ensuring compatibility and better reliability with less unforeseeable complications. Second, using an Ubuntu distribution enables a simplified installation of OpenStack through DevStack[1] and simplified interaction with the host system itself. For the x86 platform, an Ubuntu image already came prebuilt and preinstalled. Solely virtualization support had to be enabled in the BIOS settings. With the image being based on Ubuntu Server 18.04, a server OS on primarily for server usage intended Intel Xeon CPUs, problems were neither anticipated nor encountered.

Contrary to the Intel CPUs, the Renesas R-Car SoCs were neither intended to run Ubuntu as OS nor OpenStack or similar software. This presents the first difficulty in integrating such SoCs as OpenStack nodes. The following paragraphs will briefly describe the problems and decisions made to prepare the SoCs. Table 3.2 shows the final OSs installed and the according kernel on each platform.

Installation of Ubuntu Being an SoC intended for embedded usage, the OS usually is custom-built for the target system. Advantages of custom OSs lie in their small size by only including the necessary features, packages, and tools. To only include the necessary, all features and functionality in the form of packages must be known in advance. Together with a description of hardware features, a functional custom OS can be created. As such image creation can be a tedious task, the Yocto Project was chosen for image creation.

The Yocto Project[2] is an Open-Source project helping developers to create custom Linux-based systems [37]. Using the Yocto Project, packages needed in the image are specified through *recipes* and later *baked* into the final image. Similar, unique hardware properties are specified through the hardware manufacturer's board support packages and can be included in the image.

Due to OpenStack's massive size, knowledge about all necessary packages and dependency in advance was impossible as this also includes all indirectly used libraries and packages. Although a particular repository exists, which aims at providing all recipes needed, tedious trials showed that the repository was not an option. Due to version constraints, the recipes for OpenStack were highly inconsistent, which hindered a successful image creation.

For the final solution, it was decided to use a custom Yocto kernel along with a base version of Ubuntu. As Yocto generates all necessary outputs like kernel modules, they can be integrated and used by any OS. To perform a successful installation of Ubuntu, the guide from [33] was used and adapted accordingly. Summarized,

---

[1]DevStack: https://docs.openstack.org/devstack/victoria/
[2]Yocto Project: https://www.yoctoproject.org

both OSs, the Yocto generated and the Ubuntu one, were installed on separate partitions of an SD-Card. The Ubuntu OS was then configured to use the Yocto generated kernel and kernel modules. Although not having included the OpenStack packages into the image, the Ubuntu user space and especially aptitude enabled the standard way of installing any packages and other software.

Kernel Configuration Using the described method to install Ubuntu relies on the kernel and the kernel modules built with the Yocto Project. As previously said, custom OSs contain only a minimal set of features and packages. The same applies to the kernel build with Yocto. Since OpenStack relies on many packages, which again rely on specific kernel modules, these kernel modules must be added explicitly to the kernel configuration in advance. By creating a kernel configuration file containing all necessary kernel modules and features, the Yocto build system can read which features and modules should be incorporated into the kernel.

In addition to the basic Yocto kernel configuration, a few kernel modules have to be added. As OpenStack relies on network-based communication, *ebtables* and *netfilter* modules needed to be included. These modules enable the Linux firewall and packet filtering, which are substantial for OpenStack's internal communication. Further, *pseudo-filesystems* and *virtualization* had to be included. Pseudo-filesystems are necessary for OpenStack to keep track of the host systems' specific values and properties, while virtualization is necessary to virtualize resources. Last, the *device-mapper* module was necessary for OpenStack to manage storage as logical volumes for virtual machines.

Arm Trusted Firmware As the last step in preparing the SoC the ARM Trusted Firmware (ATF) had to be reconfigured. The ATF is a secure world software, that provides a trusted execution environment on a very low level [3]. The crucial aspect of the ATF for this thesis is that it must allow virtualization. Otherwise, despite the kernel support, no virtualization of hardware resources is possible. In order to achieve this, individual files with specific configurations need to be generated. Fortunately, Yocto also generates them automatically. Configuring the corresponding recipes to allow virtualization generates the according files. These files must then be flashed onto the SoC.

Platform	Kernel	OS
x86	5.0.0-23-generic (x86_64)	Ubuntu Server 18.04
ARM	5.4.0-yocto-preempt-rt (aarch64)	Ubuntu Base 18.04

Table 3.2: Software specifications used on host systems for testing

3.2.2 OpenStack Installation

Installing and configuring OpenStack can be a complicated, time-consuming, and error-prone task. Therefore, the DevStack scripts were chosen to install a complete OpenStack environment. OpenStack itself uses DevStack as a development environment and for functional testing. The future environment's configuration is made through a *local.conf* file, where parameters, settings, and services can be specified. In the following paragraphs, the configuration of the OpenStack cluster will be described and explained.

Installation The DevStack documentation includes multiple different guides for installing an OpenStack environment. For the test scenarios and this thesis's scope, a multi-node environment was desired, with the main focus on compute services. Therefore, a minimal OpenStack configuration consisting of the services presented in Section 2.4.1 was installed using the guide from [25]. Apart from Horizon, all services used represent the minimal set of services for an OpenStack environment. Horizon was also included for easier interaction with OpenStack and better visualization of current activities. As mentioned in Section 3.1, an Intel Xeon node was chosen as the controller node due to its high computational power. All necessary core services were decided to be installed on this node, along with the compute services. This configuration makes the controller node also an all-in-one node. Evaluating the R-Car SoCs as a separate or additional controller node was not possible due to limited resources on the R-Car, especially the low memory.
For the other nodes, only compute services and communication software was decided to be installed. The compute nodes were configured to communicate with the controller node if other services like network or storage services were needed. Due to later test purposes, a connection to the VMs was necessary, and the standard network configuration was not suitable. As OpenStack establishes a new internal network, there is no access from the external network possible. To make this possible, the OpenStack network's external interface needs to be connected to an actual physical interface. This can be achieved by specifying a public interface in the *local.conf* file, together with an exactly matching network configuration of the external network. Listings 3.1 and 3.2 show the final configuration files, which were used to install OpenStack.

```
1   [[local|localrc]]
2   PUBLIC_INTERFACE="eth0"
3   HOST_IP="10.1.1.100"
4   FLOATING_RANGE="10.1.1.0/24"
5   PUBLIC_NETWORK_GATEWAY="10.1.1.1"
6   Q_FLOATING_ALLOCATION_POOL="start='10.1.1.200',end='10.1.1.254'"
7
8   LOGFILE="/opt/stack/logs/stack.sh.log"
9
10  ADMIN_PASSWORD="adminpass"
11  DATABASE_PASSWORD="adminpass"
12  RABBIT_PASSWORD="adminpass"
13  SERVICE_PASSWORD="adminpass"
```

Listing 3.1: OpenStack controller node configuration file

```
1   [[local|localrc]]
2   PUBLIC_INTERFACE="eth0"
3   HOST_IP="10.1.1.101"
4   FLOATING_RANGE="10.1.1.0/24"
5   PUBLIC_NETWORK_GATEWAY="10.1.1.1"
6   Q_FLOATING_ALLOCATION_POOL="start='10.1.1.200',end='10.1.1.254'"
7
8   LOGFILE="/opt/stack/logs/stack.sh.log"
9
10  ADMIN_PASSWORD="adminpass"
11  DATABASE_PASSWORD="adminpass"
12  RABBIT_PASSWORD="adminpass"
13  SERVICE_PASSWORD="adminpass"
14  DATABASE_TYPE="mysql"
15  SERVICE_HOST="10.1.1.100"
16  MYSQL_HOST="$SERVICE_HOST"
17  RABBIT_HOST="$SERVICE_HOST"
18  GLANCE_HOSTPORT="$SERVICE_HOST:9292"
19  ENABLED_SERVICES="n-cpu,q-agt,c-vol,placement-client"
20  NOVA_VNC_ENABLED="True"
21  NOVNCPROXY_URL="http://$SERVICE_HOST:6080/vnc_lite.html"
22  VNCSERVER_LISTEN="$HOST_IP"
23  VNCSERVER_PROXYCLIENT_ADDRESS="$VNCSERVER_LISTEN"
```

Listing 3.2: OpenStack compute node configuration file

Post Installation Configuration After installing OpenStack, all services were configured with default settings and standard configurations. Although most of them were suitable and need not be changed, a few needed to be adjusted. The main configuration to adjust were the preconfigured flavors and images. DevStack automatically downloads and configures a Cirros x86 image for testing purposes. However, to mimic the host systems in the best way possible, Ubuntu 18.04 Cloud Images were chosen as guest images. Due to operating with two different CPU architectures, one image for the x86 and one for the ARM-based platform was necessary. Both could be configured and uploaded to the Glance image service either via the command-line or the web interface.

Next, DevStack also configures a couple of default flavors. A flavor in the context of OpenStack and overall Cloud Computing refers to the resource configuration of a VMs. Due to the intended testing of specific hardware in specific configurations, custom flavors had to be defined. Table 3.3 lists all created and used flavors throughout this thesis. Distinct flavors were defined for each platform, as both platforms differ in performance capabilities. For each platform, one *max*-flavor was defined, representing the highest possible VM configuration. The other flavors were derived from the minimal configuration on the ARM platform. Having to create VMs with at least one Virtual CPU (vCPU), this corresponds to one-eighth of the host's CPU resources on the ARM platform. Therefore, it was decided to create one flavor corresponding to one-eighth of all host's resources. The remaining flavors were derived by doubling the number of vCPUs, memory, and storage.

Additional to the difference in CPU architecture, the ARM platform further features a big.LITTLE architecture, consisting of four high-performance and four high-efficiency cores. At the time of writing this thesis, the KVM hypervisor cannot virtualize two different physical vCPUs simultaneously for one VM [31]. Trying to use high-efficiency and high-performance cores simultaneously results in an error of KVM. To avoid KVM choosing performance and efficiency cores simultaneously for one VM, the flavors were configured to always and only use the performance cores. Therefore, the maximum number of virtualizable cores on the ARM hosts is reduced to four instead of eight.

Finally, to access the VMs after creation, two additional settings needed to be changed. OpenStack, or rather Neutron, manages the traffic to the VMs using security rules. Based on these rules, only explicitly allowed traffic is forwarded from and to the VMs. In order to verify the connectivity of VMs, the internet control message protocol ICMP has to be allowed so that ping requests could reach the VMs. To further be able to log into the VM, Secure Shell (SSH) connections had to be allowed. As Ubuntu Cloud Images disable password authentication via

SSH by default, authentication had to be performed using public and private key pairs. Ubuntu's Cloud Images use the cloud-init package[3] to retrieve additional information at the first boot from specific files or servers. OpenStack through Neutron provides a metadata service, which among others, can insert a specified public key into the VMs to enable a password-less SSH authentication.

Name	rcar-01-256-06	rcar-02-256-06	rcar-01-512-06	rcar-01-256-12	rcar-max
vCPUs	1	2	1	1	4
Memory	256	256	512	256	2048
Disk	6	6	6	12	55

Name	xeon-06-21504-106	xeon-12-21504-106	xeon-06-43008-106	xeon-06-21504-212	xeon-max
vCPUs	6	12	6	6	48
Memory	21504	21504	43008	21504	174080
Disk	106	106	106	212	850

Table 3.3: OpenStack VM flavors

---

[3]Cloud-Init Package: https://cloud-init.io

4. Test Methodology

To examine and answer the research question, several different tests and measurements are performed. According to the research question, it was decided to perform measurements to gather insights on two topics. First, as resources in the embedded automotive environment are restricted, OpenStack's introduced overhead has to be assessed. Second, the performance of relevant resources should not be impacted too much, otherwise, OpenStack would not bring any additional value.

All measurements were decided to be performed using micro-benchmarks and Linux command-line utilities. The *Phoronix-Test-Suite*[1], together with the utilities *ping* and *sysstat*[2], were chosen to be used. As the measurements aim to observe OpenStack's impact, two measurements are obtained for each host configuration. The first measurement is performed on the freshly installed hardware natively, without any OpenStack services. The second measurement is performed inside an OpenStack VM running on the same host. The percentage overhead is calculated according to Equation 4.1. However, both, the percentage overhead and the absolute difference between the native and VM values are considered.

$$\text{Overhead} = \left| 1 - \frac{\text{VM Performance}}{\text{Native Performance}} \right| \qquad (4.1)$$

To achieve the best comparability between native and VM values, the VMs are configured in a way to match the host's system configuration as much as possible. This is achieved using the according *max*-flavor. Table 4.1 shows the native and the VM resources used. The R-Car VM uses only four cores due to KVM's limitations, as explained in Section 3.2.2. The chosen configuration of the *max*-flavor, although slightly lower than the native one, was found to function the most reliable. Additionally, Listing 4.1 shows the commands to obtain and install all necessary tools and utilities for testing.

---

[1] Phoronix Test Suite: https://www.phoronix-test-suite.com
[2] Sysstat Benchmark: https://github.com/sysstat/sysstat

Resource	Renesas R-Car		Intel Xeon	
	native	virtual	native	virtual
CPU	8 cores	4 cores	48 cores	48 cores
Disk	59GB	55GB	980GB	850GB
Memory	2GB	4GB	188GB	170GB

Table 4.1: Configuration of native and virtual machines for overhead measurements

```
1  # install phoronix-test-suite
2  wget http://phoronix-test-suite.com/releases/
↪    repo/pts.debian/files/phoronix-test-suite_10.0.1_all.deb
3  sudo dpkg -i phoronix-test-suite_10.0.1_all.deb
4
5  # install ping and sysstat utility
6  sudo apt install ping sysstat
```

Listing 4.1: Linux commands for installing all test utilities

4.1 Idle Resource Consumption

OpenStack naturally introduces an unavoidable overhead. First, OpenStack's fundamental overhead introduced to the CPU and memory is considered. While providing the OpenStack functionality, several additional processes are running. Consuming resources like CPU and memory, if the idle overhead of the OpenStack services is too high, OpenStack would be unreasonable on such hardware.

The hereby measured values are the CPU and memory usage. As a baseline measurement, the system's CPU and memory usage is measured and recorded every 5 seconds for a total of 500 seconds. The CPU usage values are gathered using the command "sar 1 5", which returns the average CPU usage during 5 seconds in percent, measured every second. The memory usage is gathered from the *meminfo*-file in the */proc*-directory. Containing the total memory value *MemTotal* and the currently free memory value *FreeMem*, the currently used memory can be calculated.

For comparison, the OpenStack services are installed, and the same measurements are repeated with a minor adjustment. In advance, a VM is created on the system under test and switched off. The measurements are taken in the same manner, except that after 200 seconds, the VM is started. These measurements first enable gathering insights on the overhead of OpenStack's services when running in an idle state. Secondly, insights on the overhead during a VM's startup and its provisioning in an idle state without any load are gathered. Listing 4.2 shows the bash-script used to gather the information with running OpenStack services. For gathering the baseline values, the same script without lines 8-11 is used.

```bash
1  #!/bin/bash
2  cnt=0
3  results="$HOME/benchmark/startup-test.txt"
4  cat /proc/meminfo | grep 'MemTotal'> "$results"
5
6  while [ $cnt -lt 100 ]; do
7      if [ $cnt -eq 40 ]; then
8          . $HOME/devstack/openrc admin demo && nova start test-compute-vm
9      fi
10     date +%s >> "$results"
11     cat /proc/meminfo | grep 'MemFree'>> "$results"
12     sar 1 5 | grep 'Average' | sed 's/^.* //' >> "$results"
13     echo " " >> $results
14     cnt=$((cnt + 1))
15  done
```

Listing 4.2: Bash-Script for gathering CPU and memory usage values

4.2   Resource Performance Measurements

To further determine whether OpenStack's overhead is substantial or not, the impact on four core resources is examined. Relevant resources in the context of embedded systems and Cloud Computing are the CPU, memory, storage, and the network [9, 12, 13, 32, 35].

4.2.1   CPU Performance

While gaining advantages through VMs, as little as possible computational performance should be sacrificed. Through virtualizing the CPU although, a particular overhead is introduced. Depending on the application, instructions from the VM have to be processed first by the hypervisor, and only then can they be executed on the hardware. The performance impact must not be high, otherwise, computations inside VMs would not be realizable or require enormous host resources.

To measure the CPU overhead, a compilation of the Linux kernel is performed, and the time needed is used as a performance indicator. The benchmark *Timed Linux Kernel Compilation*[3] from the Phoronix-Test-Suite is used to perform this measurement. Listing 4.3 shows the command to execute the test. The compilations' speed can be significantly increased by using multiple threads, for example, using one thread per core. This makes the test also suitable to examine multicore performance and parallel workloads.

To evaluate the platform as good as possible, the following decisions were made

---

[3]Timed Linux Kernel Compilation Benchmark: https://openbenchmarking.org/test/pts/build-linux-kernel

regarding the vCPU configuration. Having 24 identical cores with hyper-threading, the Intel Xeon CPU is perceived as a 48 core CPU. The cores' equality enables KVM to also manage VMs with up to 48 vCPUs, which was chosen for the test VM on the x86 platform.

The Renesas R-Car SoC being an ARM SoC with big.LITTLE architecture consists of eight cores. Due to KVMs limitation to only virtualize the same types of cores simultaneously, the highest number of vCPUs for a virtual machine is reduced to four cores. Therefore, the native compilation on the R-Car is also performed using four threads instead of eight to achieve comparable results. Using the same number of threads as CPU cores, the CPU is utilized to 100%, creating a maximum stress scenario.

```bash
#!/bin/bash
phoronix-test-suite benchmark build-linux-kernel
```

Listing 4.3: Command for executing the CPU performance test

4.2.2 Secondary Storage Performance

As storage and retrieval of data become more important, the secondary storage performance must not be a bottleneck. Again, through OpenStack as an additional middleware layer between the application inside a VM, and the actual storage device, data must pass through more instances, which affects performance. This could throttle the computation by the CPU or even prevent any computations in reasonable time frames.

Performance in the area of storage can be defined by the throughput parameter. The term throughput describes the amount of data that can be written or read from storage in a specific amount of time, for example, per second. While storage performance should not correlate to the storage's actual size, the VM storage was nevertheless chosen to be as close to the original size as possible. For measuring the throughput, the *FIO* benchmark[4] of the Phoronix-Test-Suite was chosen because of its high configurability and recommendation [23]. To evaluate the available storage speed, values for sequential read and write operations are measured, as these reflect the most common operations. Listing 4.4 shows the command for the test execution, which is performed on all systems without buffering and using the direct mode to avoid caching effects.

---

[4]FIO Benchmark: https://git.kernel.dk/cgit/fio/

```bash
#!/bin/bash
phoronix-test-suite benchmark fio
```

Listing 4.4: Command for executing the storage performance test

4.2.3   Memory Performance

Today's application execution takes mainly place from memory. Therefore, memory performance contributes a major part to the computational performance of a system. Similarly to the processor, as little as possible, performance sacrifices are desired, so OpenStack's impact on the memory should especially be as low as possible.

For evaluating the memory's performance, again, the throughput of the memory can be considered. Depending on the memory's throughput, more or fewer operations are possible, and more or fewer data can be processed. To measure the memory's throughput, the *Stream* benchmark[5] [19] from the Phoronix-Test-Suite is used. This benchmark was also chosen because of its recommendations [23]. Table 4.2 shows the operations performed by the benchmark to measure the memory's throughput. Although the test also evaluates the throughput of a memory copy operation, this operation is not considered in this thesis for two reasons. First, memory copy operations are less common in embedded programming and embedded applications because of memory wastage. The usage of pointers and value passing by reference is more common and preferred so that zero-copy memory can be achieved. Second, the computational performance is more in focus, which is represented by the chosen operations. Like the kernel compilation, this benchmark also uses multiple cores and executes one thread per core by default. As described previously, being able to equip the R-Car VM with only four cores, the benchmark is natively also executed using only four instead of eight threads. Listing 4.5 shows the according command to execute the test. Additionally, the VMs memory was chosen not to be 100% of the native hardware but a bit less. This should prevent the VM's from being killed by the host system due to using too much memory when it should become rare. However, this should not impact the memory's performance while enabling the host system to run with enough memory.

Kernel Name	Operation
Copy	$a(i) = b(i)$
Scale	$a(i) = q \cdot b(i)$
Add	$a(i) = b(i) + c(i)$

Table 4.2: Vector operations of the stream memory benchmark [19]

---

[5]Stream Benchmark: http://www.cs.virginia.edu/stream/

```
1  #!/bin/bash
2  phoronix-test-suite benchmark stream
```

Listing 4.5: Command for executing the memory performance test

4.2.4 Network Performance

Communication between different instances and ECUs takes place over the network, and therefore its performance as well must not be degraded too much. Performance in terms of networking can be defined by two values: throughput and latency. Throughput again refers to the amount of data that can be sent or received in a particular time interval. Latency, on the other hand, refers to the duration between the emit of a packet and its reception at the destination. All network tests are only performed inside the test network to receive reliable and uninfluenced results.

To measure both values, two different tests are performed. For throughput measurement the *Netperf* benchmark[6] from the Phoronix-Test-Suite is used. This benchmark measures the throughput during a TCP file send and TCP stream between client and server and vice versa. For latency measurement, the standard Linux command-line utility *Ping* is used. For 100 seconds, every second a ping is sent, and the average response time of all pings is considered. Both tests require a counterpart for the system under test to communicate with. Table 4.3 shows the systems under test and the hosts used as Netperf-Server and Ping-Partner. Listing 4.6 shows the commands for the network tests execution.

System	Netperf-Server	Ping-Partner
R-Car compute	Xeon compute	Gateway, Xeon controller
Xeon compute	Xeon compute	Gateway, Xeon controller
Xeon controller	Xeon controller	Gateway, Xeon compute

Table 4.3: Network test hosts and their counterparts

```
1  #!/bin/bash
2  phoronix-test-suite benchmark netperf
3  ping gateway -c 100
4  ping partner -c 100
```

Listing 4.6: Commands for executing the network performance tests

---

[6]Netperf Benchmark: https://github.com/HewlettPackard/netperf

5. Measurements

Executing the previously described tests, extensive results were gained on all tested platforms. The following chapter presents the results obtained. Apart from a description, causes for specific results are identified and their implications discussed.

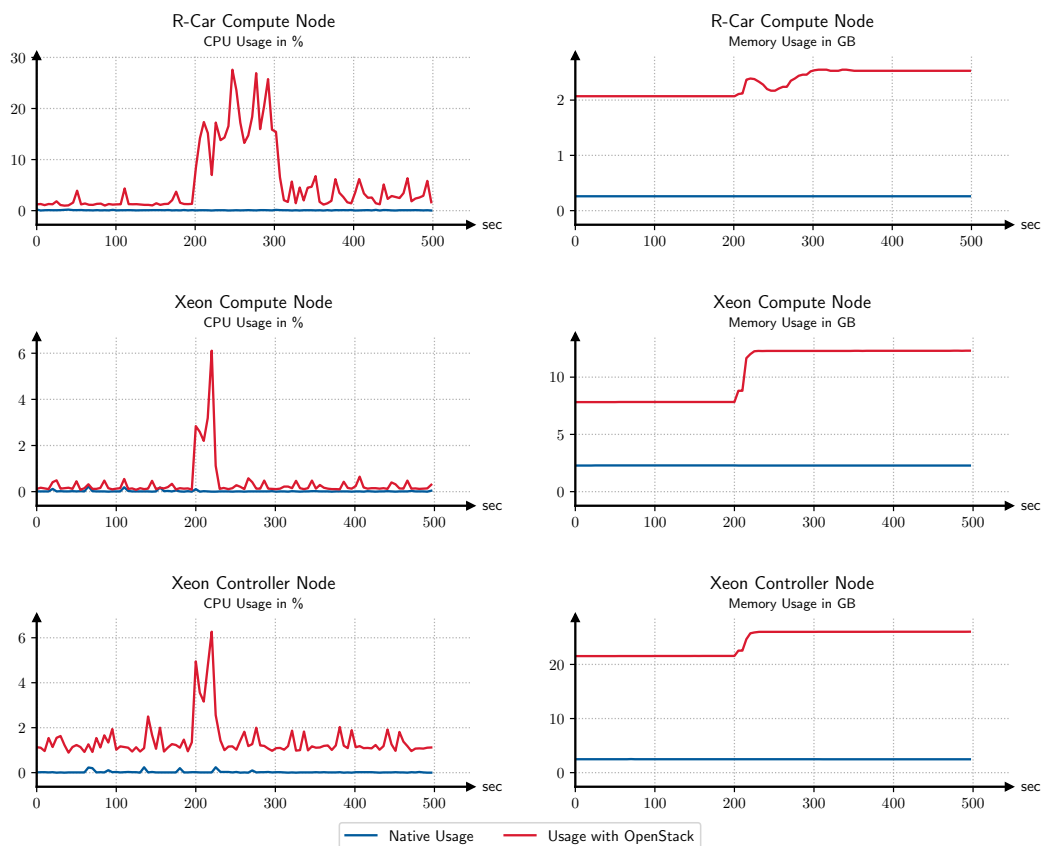## 5.1 OpenStack's Resource Consumption

Figure 5.1: Idle CPU and memory usage on all tested platforms

According to Section 4.1, the CPU and memory usage is measured on a system with and without active OpenStack services. The goal is to examine the overhead introduced by solely running OpenStack services and provisioning one VM. For all systems, a baseline measurement is performed to enable the comparison against the native system. Regarding the native CPU usage, all platforms have a nearly 0% CPU usage due to being a fresh installation with no active services. As for the native memory usage, with 400 MB the R-Car node has clearly a smaller memory usage than the Xeon nodes. This is expected because of the R-Car's custom kernel with an Ubuntu 18.04 Base OS, contrary to the Xeon's full Ubuntu 18.04 Server OS and standard Linux kernel. As expected, both Xeon nodes have similar memory usage of about 2.5 GB running without OpenStack.

Installing OpenStack on all R-Car and Xeon nodes (see Figure 3.1) indeed introduces overhead on the CPU and memory. Clearly visible, the OpenStack services introduce on all platforms small periodic peaks. These are due to OpenStacks's periodic services, which poll information from all compute hosts. Starting the scenario described in Section 4.1, the first 200 seconds in each plot in Figure 5.1 show the resource usage for all services running in an idle state and a configured but turned-off VM. On the R-Car compute node, the CPU usage increases by 2% on average, while on the Xeon compute node, it increases by only 0.3% compared to the native usage. This is due to the much higher core number and higher performance of the Xeon CPU, leading to an overall lower impact of the additional services. On the other hand, on the Xeon controller node, the CPU usage increases significantly by about 1%. The significant increase compared to the Xeon compute node depicts the additional core services running the controller node. This threefold increase in CPU usage versus the Xeon compute node indicates that all controller services are about two times as heavy as the compute services. Continuing with the VM's startup after 200 seconds, it is visible that the R-Car requires much more CPU power to perform the startup. Also, the longer-lasting CPU usage increase shows that the startup takes about four times longer on the R-Car node than on the Xeon nodes. Compared to the scenario with a turned-off VM (first 200 seconds), an active but idle VM only impacts the CPU usage on the R-Car node. After ca. 225 seconds on the Xeon nodes, the VM started successfully and runs in an idle state, introducing no visible overhead compared to a turned-off VM. After ca. 310 seconds, the VM also started successfully on the R-Car but influences the CPU more visibly compared to a turned-off VM. Again the lower impact on the Xeon nodes can be attributed to the overall higher performance of those CPUs and the therefore smaller impact of more load.

Considering memory usage, similar observations can be made. Compared to an

idle system, the OpenStack services and the turned-off VM introduce an overhead of 1.8 GB (2.0 GB total) on the R-Car and about 5 GB (7.5 GB total) on the Xeon compute node. Further, on the Xeon controller node, OpenStack requires additional 19 GB of memory (21,5 GB total) to provide all core and compute services. Assuming the compute services behave similarly as on the Xeon compute node, all controller services require about 14 GB of additional memory, making them about three times as heavy as the compute services. However, these memory values are influenced by the yet turned-off VM. Measuring the memory usage without any VM, the R-Car shows a total memory usage of 900 MB, while the Xeon compute and controller nodes show a total usage of 3.1 GB and 13 GB. This corresponds to an overhead of about 400-500 MB on the compute nodes and about 13 GB on the controller node solely through the OpenStack services.. Starting the VM increases the memory usage on all platforms. An increase of about 400 MB is measured on the R-Car node, and the Xeon nodes require an additional 4 GB for the VM provisioning. Once the VM is booted successfully, the memory usage does not change due to the VM also being in an idle state without any running services inside. The higher memory required by the Xeon nodes correlates to the higher VM memory configuration. Starting a VM with a lower memory configuration reduces this memory increase.

Overall, the results confirm the introduction of overhead through OpenStack. However, an CPU usage overhead of about 3-4% on the R-Car nodes and especially of about 0.2% on the Xeon compute node while provisioning VMs is lower than expected. An overhead in such dimensions does not rule OpenStack out on these systems. The same applies to memory usage. Especially the lower memory usage on the R-Car nodes is interesting and suggests a more in-depth examination. The controller node requires more memory and more CPU performance due to the additional core services. This advices to use a powerful enough system or dedicated network and storage nodes to provide the core services. The difference regarding both platforms' overhead originates from the high difference in computational performance and the chosen VM configurations. This makes the platforms not directly comparable but suggests considering them as possible low-end and high-end platforms.

5.2 OpenStack's Impact on Resources

Examining the in Chapter 4 defined resource's performance, the measurements show that all resources suffer some performance degradation. In all performed tests, the OpenStack VMs are outperformed by the native hardware. As expected, the

R-Car node performs overall weaker than the Xeon nodes due to not being intended for such a use case and representing a low-end hardware. However, the impact on none resource is substantial enough to entirely exclude OpenStack on either the ARM or the x86 platform. The following sections present the measurements on every resource.
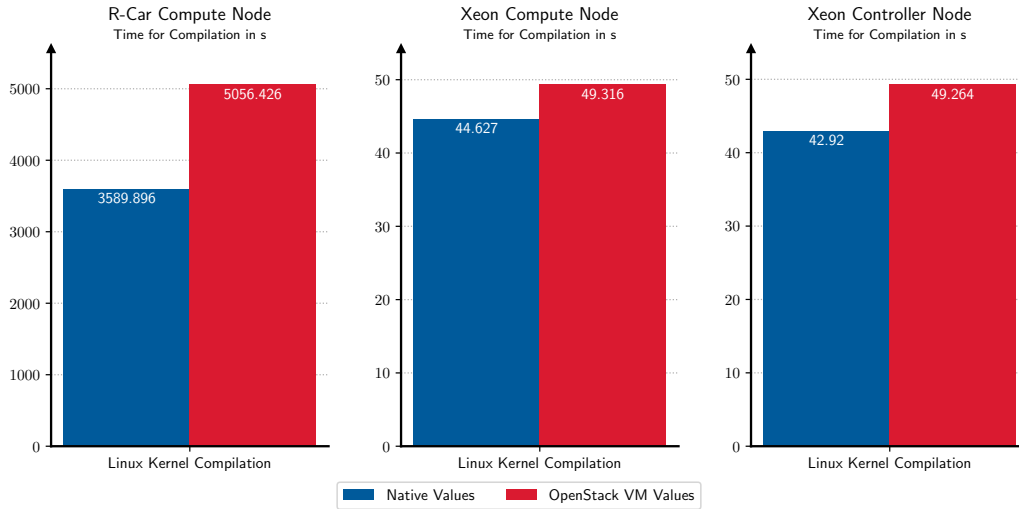
5.2.1 Effects on the CPU

Figure 5.2: Timed Linux Kernel compilation on all tested platforms

To evaluate the CPU performance degradation inside a VM, a Linux kernel compilation was chosen to stress the CPU, as explained in Section 4.2.1. Examining the difference in time between the native and the VM compilation, a bigger difference is found on the R-Car node. On the Xeon compute node, the VM is about 4.7 seconds or 9.5% slower, and on the Xeon controller node, it is about 6,3 seconds or 12.9% slower. The weaker performance on the controller node is due to the additional services running and the, therefore, higher CPU load. Interestingly, both Xeon VMs perform nearly the same, with the difference being the native value, which is better on the controller node. However, a performance overhead of 10% by the compute services on the compute node is tolerable. As the Xeon controller node in the here used configuration would represent the worst-case performance, an overhead of 13% is also acceptable.

The R-Car node, on the other hand, performs far worse. Comparing the VM compilation versus the native compilation, the VM takes about 1466 seconds or 29% longer to compile the kernel. Introducing such performance degradation, OpenStack would not be usable on this hardware. The R-Car's high overhead is, on the one hand, due to the low core number and its big.LITTLE architecture, on the

other hand, due to the R-Cars low secondary storage performance. As described in Chapter 3, the VM's cores are pinned to the host's high-performance cores. Through pinning the vCPUs to actual physical CPU cores, the virtual processes are executed on predefined CPUs cores. Contrary to the Xeon nodes, this takes the ability to reschedule these tasks to other cores by the host OS. Having enough cores to prevent a high utilization of one core, this should not impact the performance. However, the kernel compilation utilizes the full vCPU and, therefore, all physical high-performance cores by 100%. As the actual OpenStack compute services on the host also require computational power, they are simultaneously executed on the high-performance cores, leading to an over-utilization of those cores. This further leads to worse performance in general. Also, during the compilation process, many relatively small files need to be read from storage. Due to the low secondary storage performance on the R-Car, found and discussed in Section 5.2.2, the results are further influenced and not 100% representative. The R-Car results represent a worst-case scenario of a highly utilized CPU and low storage performance.

Figures 5.3a and 5.3b depict exemplary how a two-threaded instead of a four-threaded VM task stresses the cores inside the VM and on the actual host CPU. Inside the VM, cores 1 and 2 are fully loaded, leading to also fully loaded cores 1 and 2 on the host itself. As only two threads are being used, cores 3 and 4 inside the VM are idle and barely used. However, despite the non-utilized VM cores, on the host the cores 3 and 4 show a usage of 20% and 13%, indicating that apart from the VM application OpenStack's compute services also require a certain amount of resources.

(a) R-Car high performance cores utilization (b) VM cores utilization
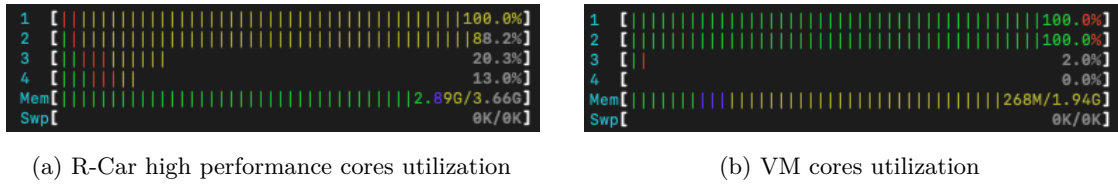
Figure 5.3: Host and guest CPU utilization during a 2 threaded VM task

To gather better and more reliable insights, this test should be repeated first, on a broader range of hardware, and second, with a more precise configuration. The big.LITTLE architecture in this configuration of four high-efficiency and four high-performance cores is not ideal. Through configuring, for example, the VM to used particular cores and the host to use other cores for the compute services, a mutual influence on each other could be prevented. Nevertheless, this would require hardware with sufficient computational power to provide sufficient resources to the host and the VM. Due to the over-utilization and deficient secondary storage performance, the results rather represent the worst-case. However, through the

better guest and host configuration regarding the utilized cores and better storage solutions, better and closer to reality results could be achieved.
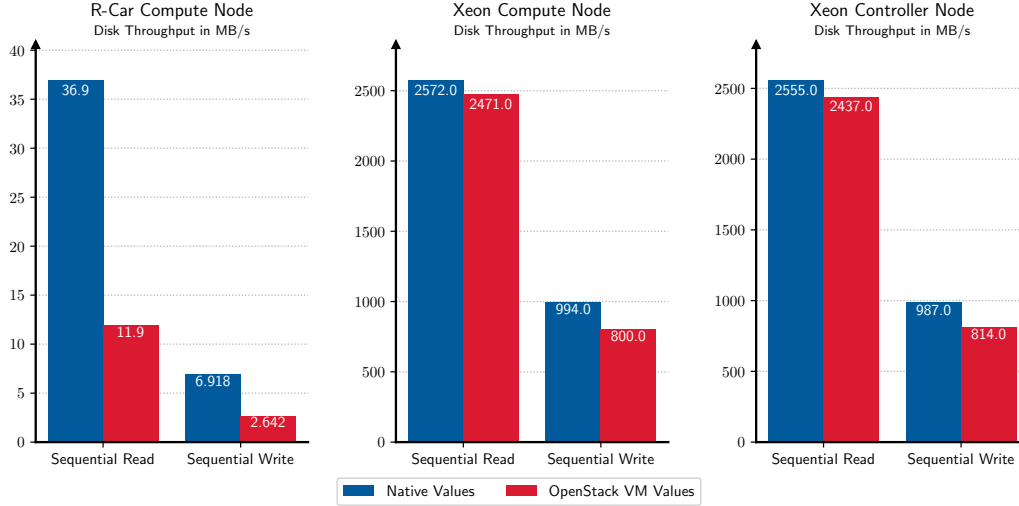
5.2.2 Effects on the Secondary Storage

Figure 5.4: Storage performance on all tested platforms

As described in Section 4.2.2, the platform's secondary storage's sequential read and write speeds were measured. Figure 5.4 shows the native and VM storage performance on all evaluated systems. Clearly visible, the R-Car has the worst performance in general and suffers the greatest performance impact. During testing, the SD-Card was found to achieve ambiguous results using different block sizes for testing. Despite the SD-Card's general low performance, a block size of 128kb was found to deliver stable and consistent results. Inside the VM, the storage suffers a 25 MB/s or 68% performance impact regarding sequential read speeds. Also, sequential write speeds are impacted by 4 MB/s or 62%. However, the measured values are not representative for two reasons. First, an SD-Card would not be used in a real-world scenario due to its low performance. Instead of SD-Cards, on-board storage like flash-storage or embedded multimedia cards would be used. The SD-Card was nevertheless chosen due to the easier handling regarding reflashing and reconfiguration of the host software. Second, because of the SD-Card's low performance, even little variation in performance yields a high percentage overhead. An impact of 25 MB/s on the Xeon platform would seem negligible, however, it makes a significant difference on the R-Car platform. Further tests using more suitable storage would very likely provide better and more accurate results. Especially the usage of the R-Cars' on-board flash storage would yield real-world values. However, this was not considered in this thesis due to the

limited storage capacity of the on-board flash, the installation complexity using two partitions, and to preserve the SoC's flash of numerous reflashes.

In contrast, the Xeon nodes feature an NVMe drive, enabling better insides on this platform. As the plots in Figure 5.4 show, the performance is significantly better on native and virtual hardware. Both Xeon nodes show similar performance impacts. Regarding sequential read speeds, the VMs suffer a 100-128 MB/s or 4-5% impact, while at sequential write speeds, the VMs suffer a 173-194 MB/s or 18-20% impact. Commonly, write speeds are lower than reading speeds due to the actual process of changing and physically writing data to the storage device. The low performance degradation regarding sequential read speed presents an acceptable overhead. However, the impact regarding the sequential write speeds is significant and not neglectable.

Despite the bad results on the ARM platform and the high overhead for sequential writes on the x86 platform, the results have to be relativized. In an automotive embedded environment, apart from research with many measurements creating lots of data points, the data to be stored and read is relatively small. This makes full utilization of the available bandwidth unlikely. Further, the read speeds are more relevant and ensure a fast execution of applications, which is desirable and achieved by the x86 platforms. The x86 platform also depicts that OpenStack is capable of utilizing high-speed storage. The performance could be further increased utilizing OpenStack's object storage service *Swift* or enhancing the configuration towards storage performance. Also, a real-word test scenario would provide better reliability regarding the results.

5.2.3 Effects on the Memory

Having measured the memory performance through mathematical operations according to Chapter 4.2.2, Figure 5.5 depicts the results. All VM measurements are lower than the native ones. Due to being processed by more software layers compared to a native operation, VM operations must be lower than native operations. The significantly higher throughputs on the Xeon systems reflect the higher thread count used on the Xeon nodes. However, dividing the values by the thread number yields values in pretty similar dimensions.

On the R-Car node, OpenStack introduces an overhead of 3% (add), 4% (triad) and 5% (scale). On the Xeon nodes, the introduced overhead lies in the same area at 3-6%. Contrary to the R-Car node, the Xeon nodes' scale operation introduces a lower overhead of 3% compared to the R-Car, where an overhead of 5% is introduced. However, the native performance of the Xeon nodes' scale operation
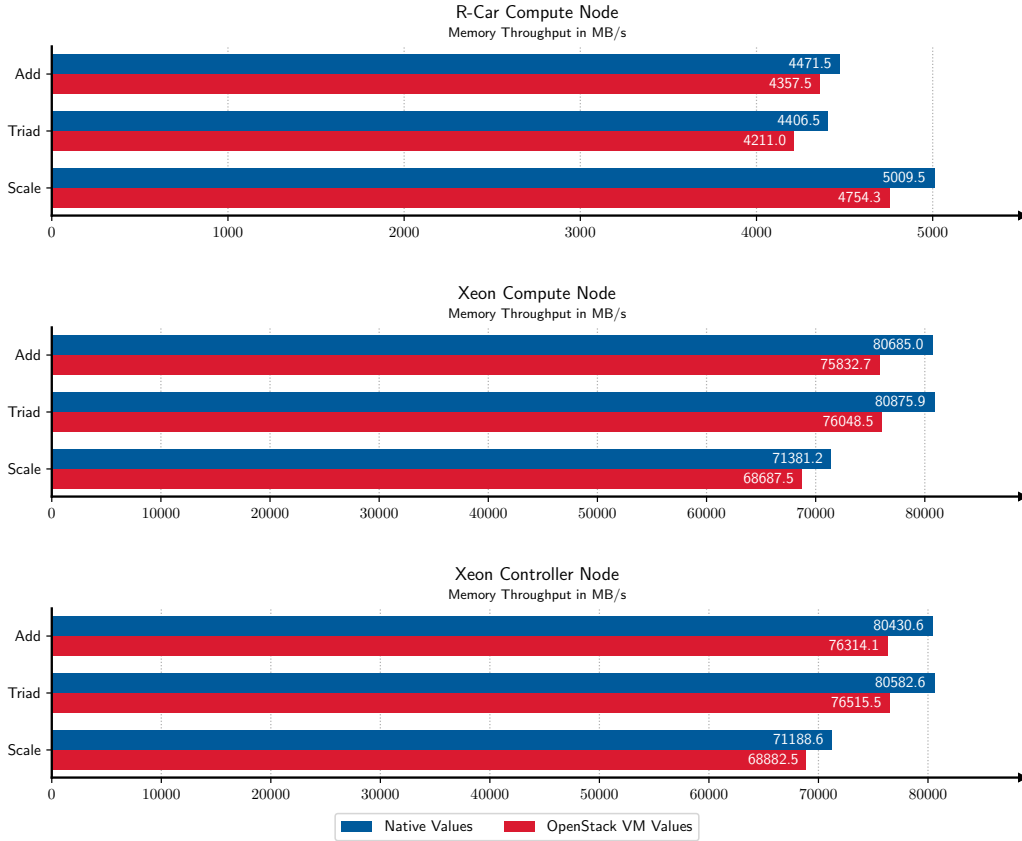
Figure 5.5: Memory throughput on all tested platforms

is in general lower compared to the two other operations, while on the R-Car, it is higher. This suggests that the ARM platform natively better supports such operations compared to the x86 platform.

Overall, an overhead of 3-6% on both platforms is plausible and acceptable. This indicates excellent computational performance with little performance degradation. Also, despite the R-Car's small memory, the performance is not impacted, further indicating good performance on systems with lower memory.

5.2.4 Effects on the Network

Studying the network throughput results in Figure 5.6, all three tested platforms yield different results. First, the Xeon nodes yield much higher general performance due to their 10 Gbit connection. This already is a relevant finding as modern vehicles make more and more use of 1 Gbit and, in the future, probably 10 Gbit connections. The performed measurements show that OpenStack can handle these connections. On the other hand, the R-Car node is only equipped with a 1 Gbit network interface, yielding significantly lower results.

The overhead lies at 20% for the stream tests on the R-Car node and at 17% for the
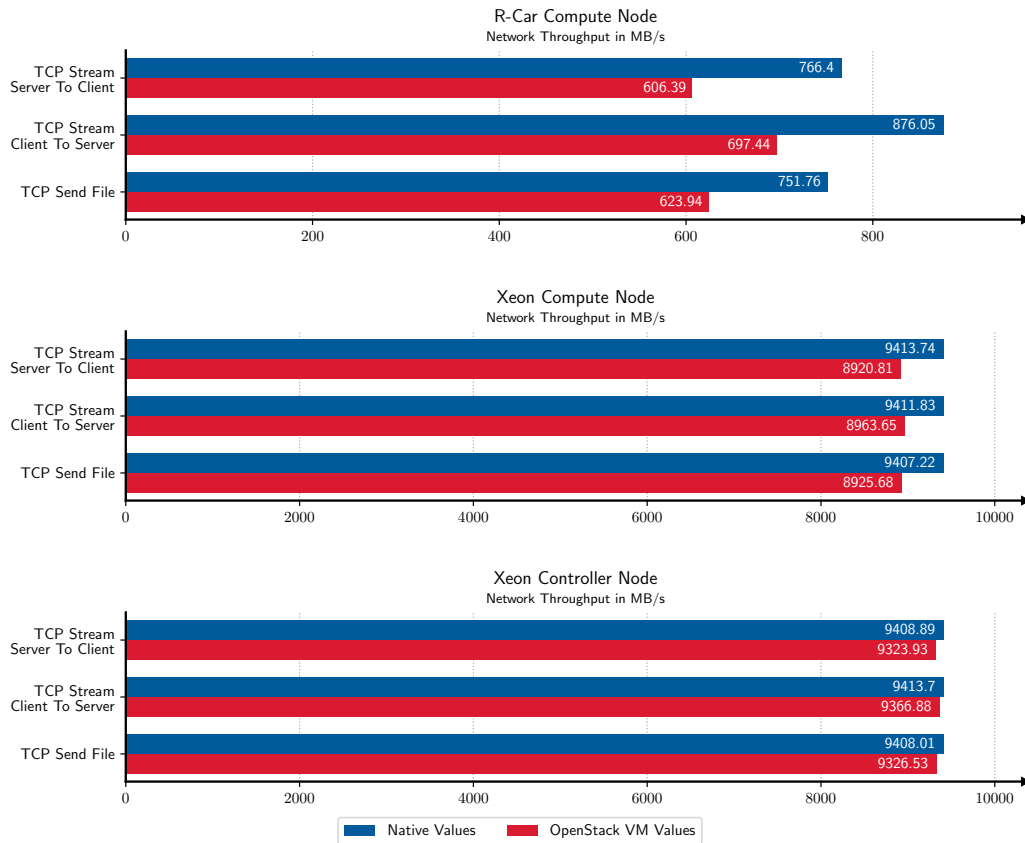
Figure 5.6: Network throughput on all tested platforms

file send test. In all tests, the VM suffers a similar impact, which could be related to the R-Car's overall lower computing capabilities during the simultaneous usage of the high-performance cores by the host and guest. As described in 5.2.1, the simultaneous resource usage by host and guest requires additional computational resources, as well as time, which reduces the throughput. In the context of network performance, additional services process data-packets before actually releasing them onto the network and vice versa.

Examining the Xeon compute node, results with much lower overhead are found. For all tested connections, the overhead stays at about 5%. This strengthens the indication that the host system's computational capabilities can influence network performance and lead to better results. Also, the results show that the VM utilizes the hardware and the 10 Gbit connection used.

Closing, the controller node delivers additional insides on the performance overhead. Regarding the test configuration, the environment differs in two points from the ones of the R-Car and the Xeon compute node. First, the compute services are now located on the same host as the OpenStack network service Neutron. This means that all requests can be directly processed by the same host and send out to the real world network without being first processed by a different node. Second,

the Netperf-Server is now not being run on the controller node but on the Xeon compute node, making the controller node less utilized. This leads to an even lower overhead of only 1% for all TCP connections.

Considering the low and very low overheads of 5% and 1% on the Xeon nodes, the performance is again satisfactory. The Xeon controller node's performance raises the possibility of always executing the Neutron service and the Nova services along with each other. At the expense of other resources, this could provide nearly native performance if necessary. The significant overhead of 20% on the R-Car, on the other hand, is not acceptable. However, in an automotive environment, the transferred data is relatively small, containing small pieces of information like signals. Therefore, high throughput rates might not always be the primary concern, but rather the latency. However, as with storage, OpenStack enables various network configurations and enhancements, which would improve network performance. In addition, the usage of an ARM platform with 10 Gbit Ethernet and higher computational power would most likely increase the comparability and reliability of the results.
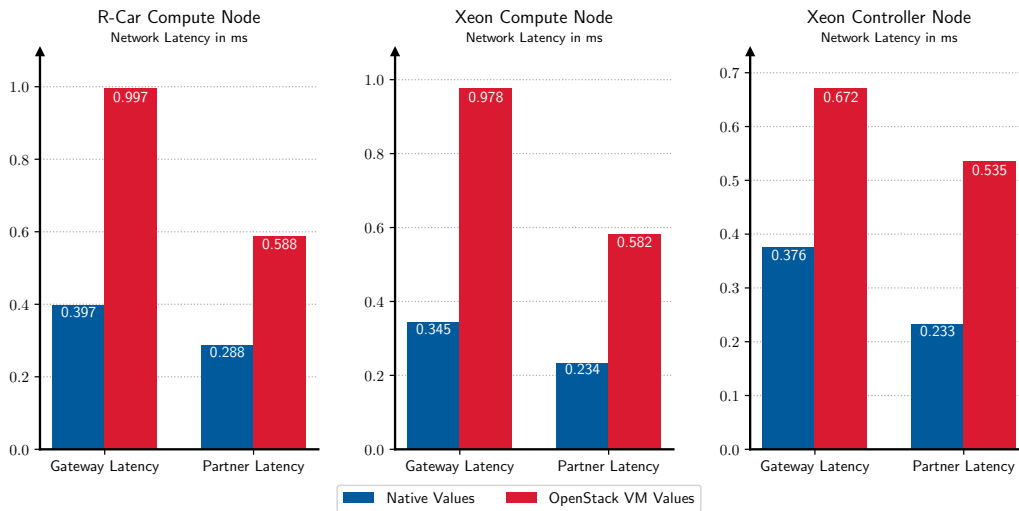
Figure 5.7: Network latency on all tested platforms

Despite the different hardware and platforms, all performed latency tests yield very similar results in Figure 5.7. Pinging the gateway, a host not part of the OpenStack cluster, results in a latency measurement between 0.35 and 0.4 milliseconds. Pinging the Xeon controller and the Xeon compute node, hosts part of the OpenStack cluster, yields times about one millisecond shorter. This is accountable to the higher performance of the nodes versus the lower performance of the gateway.

Regarding the measurements from inside the VM, a clear overhead can be identified. Pinging the gateway, the R-Car, and the Xeon compute node yield a latency of

about three times as high as natively. The Xeon controller node yields a latency of two times as high as natively. This overhead of about 200% and 100% can be attributed to the way OpenStack routes packets, in this case, ICMP packets, to their destination. Until the packets are released onto the real-world network, they are routed on an internal OpenStack network to the Neutron service host, in this case, the Xeon controller node. Arrived at the network node, the packets are processed by Neutron and only then released onto the real-wold network. The same situation applies to the replies in the backward direction.

Executing the ping from the Xeon controller node VM gives the possibility to identify OpenStack's actual overhead. Because the controller VM packets are already on the controller hardware, only an one-time processing of them is necessary to place the packets on the real-world network. Taking 0.3 milliseconds more on the controller node for both pings, this 0.3 milliseconds are the actual overhead introduced by OpenStack. This also matches the fact that the overhead is twice as high with 0.6 milliseconds on the two compute nodes, as on those nodes, the packets have to be processed twice, on the compute host itself and on the network node. Considering further that the ping utility measures the round trip time, meaning from source to destination and back, the actual overhead is even smaller. OpenStack introduces an overhead of about 0.15 milliseconds from the Neutron service host and 0.3 milliseconds from other compute hosts for packets to reach their destination.

Considering the native values, 100% and 200% longer latency values are not negligible. Nevertheless, the measurements were performed on a local network and within a close range. Real-world networks could introduce much longer latencies, making the native values much bigger. For example, a ping to the Google DNS server takes about 15 milliseconds. Considering an overhead of 0.3 or 0.6 milliseconds from an OpenStack VM, the percentage overhead is only 2% or 4%, being well reasonable and acceptable. Being an extreme example as the packets have to travel a long distance shows that the actual overhead compared to the native hardware can vary on a different network. To gather real-world values, the test should be performed on an actual automotive network with a real-word load.

5.3   Summary

Concluding from the performed results, OpenStack clearly introduces an overhead in all considered domains. However, on both evaluated platforms, the VMs function consistently and reliably.

The Xeon systems represent a significantly more powerful embedded platform

which delivers very high performance. OpenStack's impact on this platform is relatively small, and little performance degradation is measured. This suggests an actual in-vehicle use. The R-Car results, in contrast, do not show such good performance. Considering the high performance degradation of the CPU and secondary storage, an in-vehicle use seems not promising. However, the network and memory performance measurements show that good performance is achievable on the R-Car and the degradation does not apply generally. This promises better and improved performance on different hardware with better components like faster secondary storage and more CPU cores.

Despite the overhead, all functionality is continuously available, enabling the consideration of OpenStack's advantages in the following chapter. This chapter identified and evaluated the major negative impacts and disadvantages of OpenStack: the overhead and performance degradation. However, neither the better nor the worse performance of the platforms used do entirely prevent further tests or the in-vehicle usage in general.

6. Use Cases

Apart from a small impact and overhead on resources, OpenStack should also function reliably and as anticipated. Therefore, also advantages and new features have to be measured and evaluated. Considering the typical environment of OpenStack being, for example, a data center with usual hardware, embedded systems like automotive ECUs do not belong in this category. Nevertheless, OpenStack must perform reliable and consistent to benefit from its features and possibilities of Cloud Computing. As exemplary advantages of OpenStack in the context of automotive embedded systems, two scenarios are examined and measured. The following sections describe which value OpenStack can introduce through VM distribution and migration. The measurements are performed using Python scripts to interact with the OpenStack cluster using the OpenStack Python SDK. All scripts used can be found at [16].

6.1   VM Distribution

OpenStack enables the automatic distribution of VMs according to defined rules. While currently applications are bound to ECUs, they cannot be executed elsewhere. However, this would come in handy if, for example, an individual ECU is highly loaded while another does not have much to process. By pooling the available resources and distributing them, VMs can be executed independently on the most suitable hardware. The functionality should work as intended, otherwise, unforeseeable complications could occur, leading to, for example, overloaded ECUs.

The distribution on those platforms is evaluated separately because of specific instruction sets and, therefore, specific images for the ARM and x86 nodes. The Nova-Scheduler being responsible, among other things, for the assignment of VMs to hosts, can be configured by various parameters. For the evaluation of VM distribution, the following parameters are modified.

- *scheduler_driver_task_period*
  This parameter sets the interval of how often the scheduler updates its resource information. A smaller interval means better and more accurate values but also an increased overhead as the information must be polled more often from the compute nodes.

- *cpu_weight_multiplier*
  This parameter sets the importance of used CPU cores. A positive multiplier will try to achieve even distribution with an even number of cores used on every host. A negative multiplier will first use all available cores on one compute node before utilizing the next one.

- *ram_weight_multiplier*
  This parameter sets the importance of used memory. A positive number will try to achieve even memory utilization on all hosts. In contrast, a negative number will try to first utilize all available memory on one host before moving on to the next one.

- *disk_weight_multiplier*
  Similar to the cpu_weight_multiplier and ram_weight_multiplier, this parameter sets the importance of disk utilization. A positive number will again try to distribute VMs evenly based on the hosts' disk usage, while a negative number will first stack the VMs on one host.

- *build_failure_weight_multiplier*
  This parameter describes how much build failures of VMs on hosts influence the VM assignment. The higher the value specified, the more build failures weigh, resulting in hosts with recent build failures not being chosen for a new VM in the first place.

- *io_ops_weight_multiplier*
  Like the previous weight_multipliers, this parameter considers the CPU usage on the compute nodes. If a positive value is defined, compute nodes with high CPU utilization will be chosen over ones with low utilization. On the other hand, a negative value configures the scheduler to chose low-utilized nodes over high-utilized ones.

6.1.1 Distribution Scenarios

On both platforms, two different and contrary distribution strategies are tested. First, an equal distribution regarding the CPU cores is tried to be achieved. As the CPU cores are one source of computational performance, equal distribution

among all compute nodes would decrease each compute node's CPU load. This would prevent the VM performance from being degraded due to the host being too highly loaded. Second, a stacked distribution regarding the CPU cores is tried to be achieved. Configuring the scheduler to first use all available CPU cores on one compute node until utilizing a second node would enable to set these nodes into a standby or low power mode. Due to this, energy could be saved as not all nodes would be necessarily utilized all the time. Table 6.1 shows the parameter configurations for both distribution strategies.

Parameter	Stacked Distribution	Equal Distribution
scheduler_driver_task_period	5	5
cpu_weight_multiplier	5.0	-5.0
ram_weight_multiplier	5.0	-5.0
disk_weight_multiplier	5.0	-5.0
build_failure_weight_multiplier	5.0	0.0
io_ops_weight_multiplier	-1.0	5.0

Table 6.1: Nova-Scheduler parameters for VM distribution

The VM flavors used are the smallest defined, being the *rcar-01-128-06* and the *xeon-06-21504-106* flavor. In order to best evaluate the distribution, as many VMs as possible should be spawned and distributed. With the ARM cluster having only 16 cores in total, this was the highest number of machines possible. To keep the Xeon cluster comparable and the testing time reasonable, it was also decided to spawn 16 machines. Therefore the flavors representing one-eighth of the total resources were chosen.

The scenario measurements are performed in the following way: On each cluster, 16 VMs are created and started sequentially. For every VM, during its creation, the VM's compute host is checked. This way, the initially scheduled host by the Nova-Scheduler is captured, as well as any host changes due to build failures. The scenario is performed one time with each scheduler configuration.

6.1.2   Scenario Evaluation

Figure 6.1 shows the VM distribution across the R-Car cluster (top row) and the Xeon cluster (bottom row). First and foremost, no errors or host reschedules are visualized in the plots as no errors or reschedules occurred. For all distributions and hosts, the spawning was always successful. Further, it is clearly visible in the figure that the different distribution scenarios worked and were realized as expected by the Nova-Scheduler.
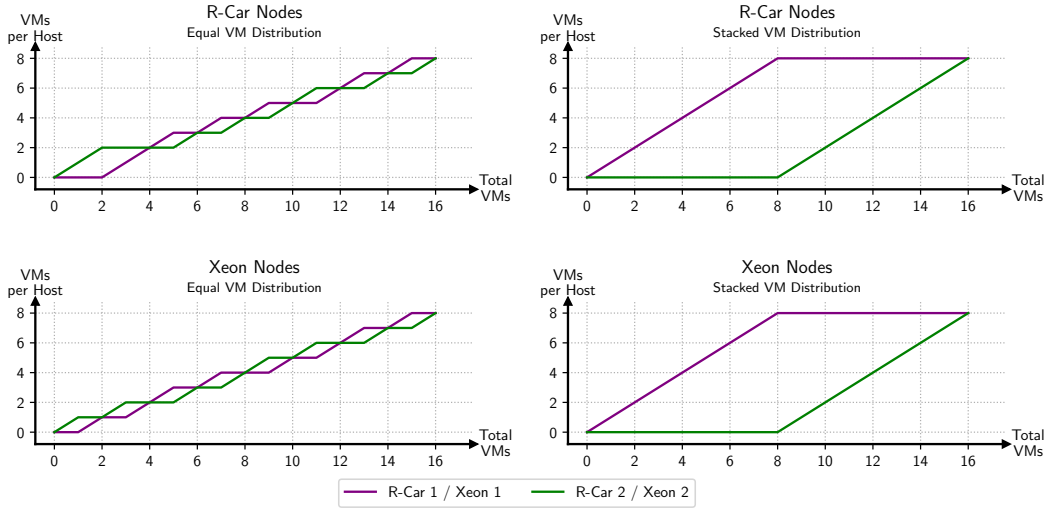
Figure 6.1: VM distribution across compute nodes

The right column of Figure 6.1 shows the stacked distribution. The first eight VMs are clearly assigned to the R-Car 1 / Xeon 1 compute node, while the last eight VMs are assigned to the other compute node. The left column shows the equal VM distribution. Obviously, over time the VMs are distributed equally across the two nodes. On the R-Car cluster as well as on the Xeon cluster, the distribution is not strictly alternating during the equal distribution, depicted by the first two VM assignations to the R-Car 1 node. Also, the 10th and 12th VM are assigned to R-Car 1 in a row. Similar behavior can be seen on the Xeon cluster: VMs 4 and 5, 8 and 9 and 12 and 13 are assigned to the same compute node in a row. However, as the number of running VMs per host is the same, a selection of the same hosts twice in a row is not false, as both hosts are equally utilized.

The correct functionality of the *io_ops_weight_multiplier* is captured in the first VM assignment on the Xeon cluster to the Xeon 2 node. Xeon 1, being the controller, is more utilized, and therefore Xeon 2 is chosen over Xeon 1. However, striking are the first two VM assignments to the R-Car 2 node using an equal distribution (Figure 6.1, top-left plot). Assigning the second VM also to the R-Car 2 node indicates that the CPU of R-Cars 1 is higher utilized at that time, as it has to outweigh the CPU, memory, and disk multiplier. However, the third and fourth VM are assigned to the R-Car 1 again, balancing the distribution again.

Concluding, both scenarios show that a distribution strategy on embedded hardware for VMs is possible. Also, the absence of reschedules and build errors demonstrates the reliable operation of OpenStack.

6.2   VM Migration

A large number of ECUs in today's vehicles brings two essential advantages. The presence of multiple ECUs brings redundancy and enables to group functionality on particular ECUs. In case of an ECU failure, another ECU can continue its work to avoid security and safety implications. Similarly, real-time software can be separated from less time-critical software to avoid interference or performance degradation. While the separation or grouping can be achieved easily and with various OpenStack methods, the aspect of redundancy can also be satisfied by OpenStack. Similarly, multiple VMs could calculate the same task on different hosts in parallel, realizing today's approach.

Yet, OpenStack further enables a different approach to deal with, for example, instance failures. By executing all functionality inside VMs, these VMs are decoupled from the hardware and can be migrated to other hosts. This can happen by transferring a switched-off instance but also a live, currently active instance. Therefore, if an ECU being a compute host fails or a problem is detected, its VMs can be migrated to other running hosts. As valuable as live-migration might sound, the performance during it must not be impacted too much to provide value, and the duration must be reasonable. While VMs are shifted to another compute host, the resources are in a state without one single defined host. As the resources are either partly on two hosts or for a short time on neither host, the performance could suffer drastically. Also, depending on the resources used by the VMs, the duration could take an unacceptable amount of time.

6.2.1   Time

To evaluate the live-migration duration of VMs, a Python script is executed. The script boots a VM on one host and flowingly migrates it to another host. Same as the VM distribution, the migration time is evaluated separately for the R-Car and the Xeon cluster. Using the OpenStack Python SDK, the VM's status is periodically polled, and the interval between entering and leaving the *migrating*-status is used for evaluation. To ensure the instances are migrated between the right hosts, only the compute services on the relevant hosts are active during the migration. The migrations are performed with all configured flavors in order to discover potential impacts of resources on time.
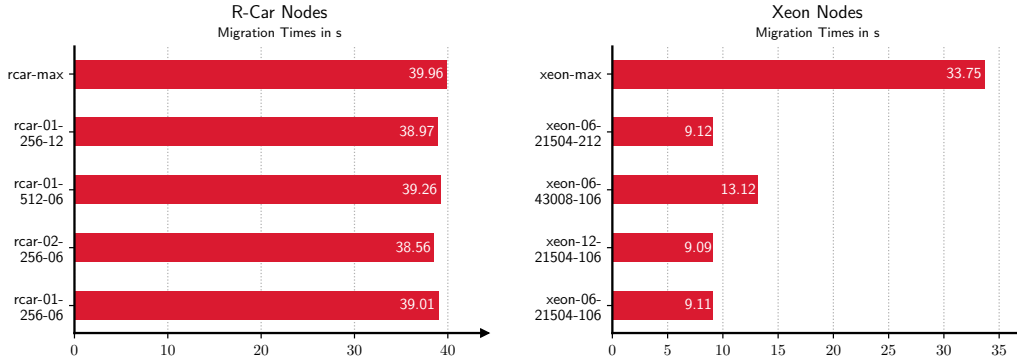
Time - Evaluation

Figure 6.2: VM migration times

Figure 6.2 shows the migration times for the R-Car and Xeon cluster. While on the R-Car nodes, all flavors take about 40 seconds to migrate, on the Xeon nodes, the flavors need about 10 seconds, except the *max*-flavor taking about 34 seconds. Considering the Xeon migration times, one resource can be identified as exceptionally time-consuming. All flavors except the *max*-flavor represent a twofold increase of one resource compared to the smallest flavor *xeon-06-21504-106*. Neither double the CPU cores nor double the disk size seems to affect the migration time significantly. Double the memory, however, extends the migration time by three seconds. Considering now the *max*-flavor needing 34 seconds together with the fact that a memory increase of 21504 MB increases the duration by three seconds, memory seems to have the most significant impact. Therefore, for the total migration time of the *max*-flavor, 24 seconds (8*3 seconds) are necessary only to migrate the memory, while the remaining resources need ten seconds in total.

Due to the VMs being in an idle state, neither the CPU nor the memory or the disk is utilized. This explains why neither resource increase is directly reflected in an increased migration time. As the actual amount of resources to migrate stays the same, no actual increase in resource usage is present, and therefore no change is introduced. However, while the memory utilization is also not differing from flavor to flavor, it nevertheless has an impact, leading to the conclusion that the memory has to be freed on the old host and allocated in some way on the new host.

Using these conclusions, the overall low performance and the little to no variation in migration time on the R-Car cluster can be explained. Striking, no substantial difference in migration time between the smallest, the biggest, or any other flavor can be identified. Also, a substantial influence of the memory size is absent.

This suggests that the migration time is mainly influenced by something else. Considering the fact that all VMs, on the Xeon and on the R-Car nodes, where started using local storage, everything is existing and stored locally on the current host. Flowingly, everything has to be transferred to the destination host accordingly during the migration. On the one hand, this explains the not changing migration times on the Xeon Cluster with increased storage because the actual data size stays the same. On the other hand, the Xeon nodes are connected via a 10 Gbit Ethernet connection, while the R-Car nodes have only a 1 Gbit connection. This leads to the fact that no variation in migration on the R-Car cluster can be identified because the storage migration always influences the migration time the most. Having a ten-times slower connection and less powerful hardware, the storage migration takes more time than anything else.

In current vehicles, 1 Gbit Ethernet connections are already present, and using 10 Gbit capable hardware will be the next step to follow. Together with powerful hardware, reasonable migration times can be achieved. Taking into account that memory requirements, as well as storage requirements, are kept as low as possible in embedded software, the performance would further increase. However, a utilized CPU and memory might additionally impact the migration time, as examined in the next section.

6.2.2   Performance

Considering live-migration in a redundancy use case, besides the reliability and the migration time, the performance during it is critical. Depending on the use case and its requirements, either continuation or performance or both may be favored. For example, in a time-critical real-time software, the performance must not be impacted, as this would make the application indeterministic and could lead to catastrophic events. In the case of an application with less strict timing requirements like an infotainment application, however, the functionality's continuation is more important, even if the performance is slightly reduced for a short amount of time.

The CPU being the primary computational resource, it was decided to gather values about the CPU performance. In order to achieve this, the *Sysbench* benchmark[1] is used. The benchmark uses prime number factorization to determine Events per Second (EPS) and latency values of the CPU. Considering these values over time during a live-migration, a statement about the performance is possible.
The values are collected as follows: First, a VM is created, and all software updates inside it are performed. Next, the Sysbench benchmark is installed using aptitude.

---

[1]Sysbench Benchmark: https://github.com/akopytov/sysbench

The Sysbench benchmark is configured to factorize as many prime numbers as possible for one second, every two seconds. The flavor with the doubled CPU core number is used on both clusters, and the number of threads used by Sysbench is specified to be half the number of available cores. This provides EPS and latency values every two seconds so that a complete visualization of the migration performance can be established without losing any significant values. The VM is run for 2 minutes on one compute host and flowingly migrated to the other host in its cluster. This is performed four times so that four migrations are captured.

Performance - Evaluation

Figure 6.3 shows the collected values. Visible in both clusters are the EPS drop and the latency spike during the migrations. However, the absence of a definite plateau in the EPS and latency plot indicates no measurement values were lost, and the impact only lasts a short amount of time. Further, both clusters' migration durations increase compared to the results in Figure 6.2. On the R-Car cluster, using the VM flavor *rcar-02-128-06*, the migration time increases from 40 seconds to 100 seconds (best case) or 150 seconds (worst case). Similarly, on the Xeon cluster, using the VM flavor *xeon-06-21504-106*, the migration time increases from 10 seconds to 17 seconds (best case) or 30 seconds (worst case). This clearly shows that a non-idle VM, having a utilized CPU takes longer to migrate than an idle VM. Additionally, Figure 6.3 shows that Xeon 1, in general, achieves lower EPS and higher latency values running the core services.
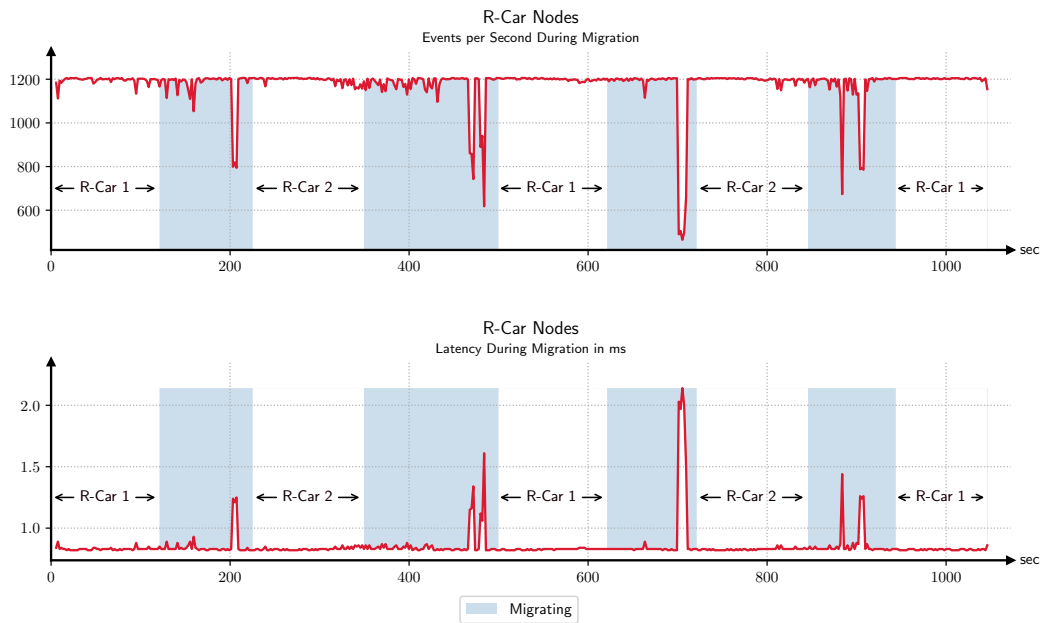

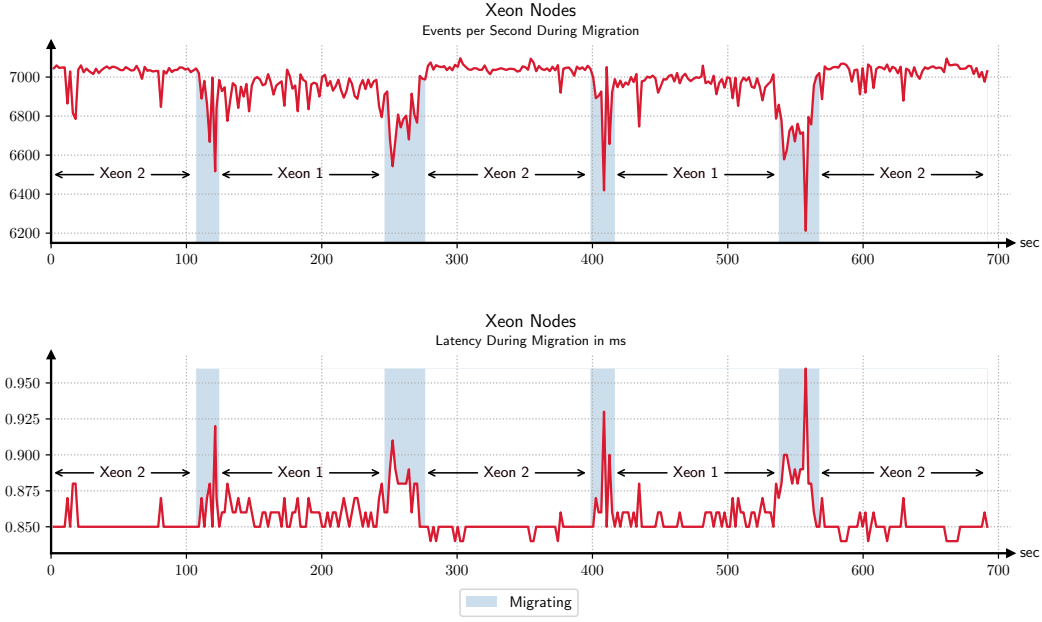Figure 6.3: VM migration performance on the R-Car and Xeon cluster

Figure 6.3: VM migration performance on the R-Car and Xeon cluster (cont.)

Examining the plot in detail, it can be seen that the performance degradation, in general, is only given for a concise amount of time, even when the migration itself takes much longer. The peak EPS drops only last for a couple of seconds on the R-Car and the Xeon nodes. This is very noticeable on the R-Car nodes. Due to only executing one thread on the R-Car nodes, only when this specific thread or rather CPU core is migrated, the EPS number drops. Nevertheless, the drop from 1200 EPS to 800-520 EPS corresponds to a performance degradation of 33-52%, being non-negligible even in short time frames if performance is critical.

On the Xeon nodes, the degradation is not as significant as on the R-Car. Due to executing three threads, the peak performance drops during the migration are less significant. On the Xeon nodes, the performance drops by 400-800 EPS, corresponding to a 5-10% performance impact. This leads to the finding that multi-threaded processes and parallelized applications suffer lower performance degradation during a live-migration compared to single-threaded processes. As threads seam to be migrated one after another, improved fundamental performance through more threads or higher single-core performance lowers the percentage degradation.

Considering the latency, the spikes correspond precisely to the EPS drops. Reasonably, the EPS number drops if the CPU has a higher latency, meaning that fewer requests can be carried out per time frame if the CPU takes longer to react to a request. However, the R-Car latency spikes are much higher (2 milliseconds) compared to the Xeon spikes (0.1 milliseconds). This can be attributed to the fact

that first, the Xeon CPU is more powerful, and second, executing three threads leads to a high probability of some thread always reacting quickly to a request. The influence of multiple threads can be seen in the fact that the time interval in which the EPS number drops is much longer compared to the R-Car cluster. While on the Xeon cluster the performance degradation is present for a longer duration, the R-Car cluster suffers a higher but shorter degradation. Nevertheless, on both clusters the rise in CPU latency inhibits a usage for real-time applications, as the CPUs' responsiveness becomes indeterministic.

Further, the graphical representation of the migration confirms the previous section's statement regarding the time share of different resources. The figure shows that the CPU is migrated last as the spikes and drops appear at the end of the migration. Especially, the R-Car plot shows that the CPU migration happens in a concise time frame compared to the total duration. As the total duration increases in contrast to the previous section, the memory impact on the migration time is further hardened. The CPU and memory utilization being the only difference to the measurements in Section 6.2.1, their utilization can be made responsible for the longer migration time.

6.2.3 Application Migration

Having examined the migration time and performance in the previous sections, an actual application shall be migrated. Since real-time and hard-safety applications would require further and more complex tests, these are out of this thesis's scope. Nevertheless, the previous measurements and scenarios suggest that an OpenStack cluster on automotive hardware is realizable. To demonstrate this, it was decided to evaluate the migration of a Tetris application. Being a small game, this represents the use case of playing a game on a vehicle's infotainment system. Not having hard real-time requirements but mainly regarding the quality of service, this reflects the previously described choice of a continuous function over high performance. However, the performance should still be as good as possible.

This use case is evaluated in a similar manner as the migration performance in the previous section. However, as the Tetris game requires a more sophisticated user interface, the VMs must be prepared accordingly. A desktop shall represent the user interface which would also be present in an actual infotainment system. As the Ubuntu Cloud Images do not initially include a desktop, it first has to be installed. Further, a Virtual Network Computing (VNC) server is installed inside the VM. This enables the remote access of the machine's desktop and its interaction. As

Tetris game, the game *Quadrapassel*[2] from the Ubuntu repository was chosen due to its easy installation via aptitude and low hardware requirements.

In order to evaluate this scenario, two different measurements are performed. First, using the resource usage measurements from Section 4.1, the overhead and resource usage with an active application inside the VM are examined. Second, the same measurements are performed during the live-migration. Only one migration is measured as multiple migrations do not provide more information but only extend the test time.

Application Resource Usage - Evaluation

Before the migration, Figure 6.4 shows the resource usage on the compute hosts R-Car 2 and Xeon 2. In addition to the tests from Section 5.1, in this measurement, the VMs are not left idle after creation, but the VNC server and the application are started. Figure 6.4 enables a clear distinction between the different actions performed while preparing the system before migration. In addition, to Figure 5.1, through measuring the CPU and memory usage inside the VM too, even better visualization of OpenStacks overhead is possible. Regarding the R-Car node, an overhead of 2% in CPU usage and an overhead of 100 MB in memory usage is found. For the Xeon node, the CPU overhead also lies between 1-2% during an active application, while the memory overhead lies at about 4 GB. Also visible, the CPU and memory usage only rise significantly if the usage inside the VM rises, apart from small periodic peaks due to periodic OpenStack processes. Further, the figure shows the difference between the VM startup and the OS's actual boot inside the VM. On the R-Car cluster, the duration between the VM startup and the actual VM boot is significantly longer than on the Xeon cluster. As explained in Section 5.1, this originates from the higher computational power of the Xeon platforms.

Application Migration - Evaluation

Figure 6.5 shows the CPU and memory usage on the R-Car and Xeon hosts during the live-migration. First and foremost, the live-migration succeeds. The player is able to continue to play the game throughout the whole migration without significant interruption. The game lags for a couple of seconds only towards the end of the migration, leading to inputs from the keyboard and outputs from the screen being slightly delayed. This fits together with the gained knowledge on the performance during a live-migration. As the performance drops towards the end,

---

[2]Quadrapassel Game: https://wiki.gnome.org/Apps/Quadrapassel

also the game performance drops. However, the game is not interrupted entirely and is available throughout the migration.
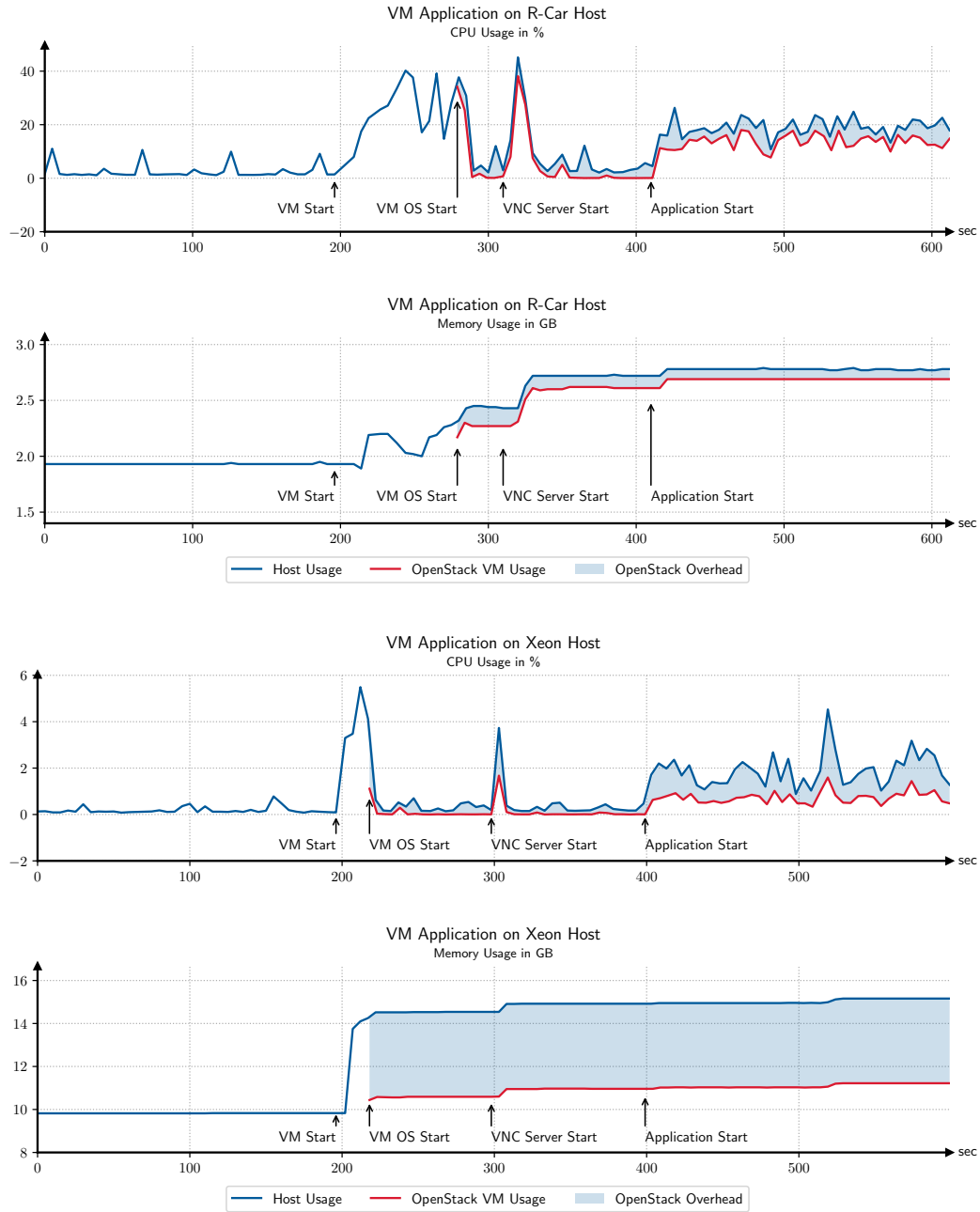
Figure 6.4: R-Car and Xeon resource usage with VM application

Examining the results in detail, the R-Car cluster duration shows as most striking. Taking about 1850 seconds or 31 minutes, it is significantly higher compared to the duration measured in Section 6.2.1 and 6.2.2. However, this is due to the limited resources on the R-Car. The measured memory values show that the use case requires 3.5-3.6 GB of memory, which corresponds to the R-Car's maximum available memory. As from this point on, the swap-memory is used, the operations

are much slower, resulting in a significantly longer migration duration. As the disk data has to be migrated too, it first has to be read into memory before transferring it over the network. Using the much slower secondary storage as swap-memory results in a significant performance impact leading to the long migration time. Also, zooming into the memory usage reveals that the memory is transferred towards the end, depicted by the destination host's continuously growing memory usage. Further, the CPU usage illustrates that only the CPU usage on the destination host increases while the source's CPU usage does not seem unusually affected by the migration. Clearly visible, the CPU usage of the destination host is around 0% in the beginning, while in the end, the source host shows a CPU usage of about 0%. The same behavior can be seen regarding memory usage.

Figure 6.5: R-Car and Xeon resource usage with VM application and migration

The Xeon cluster, on the other hand, shows noticeable better and more confident results. Comparing the migration duration with results from Section 6.2.1, the times only differ slightly. While the idle VM needs 34 seconds, the VM with an active application takes only 3 seconds or 8% longer. As on the R-Car cluster, Figure 6.5 shows the host's additional CPU and memory usage during the migration. However, contrary to the R-Car cluster, the live-migration affects the destination host and the source host on the Xeon cluster. As the migration happens significantly faster on the Xeon cluster, the CPU has more load, which explains the not existing CPU usage increase on the R-Car cluster. Due to the long duration, the R-Car CPU can process all instructions over a longer period of time, leading to a lower overall load. Also, the increased fundamental CPU and memory load through the control services is visible.
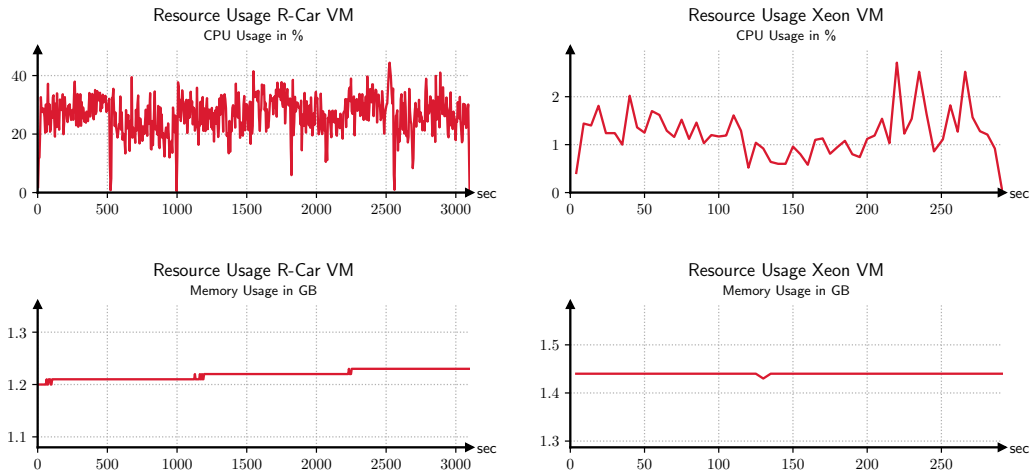

Figure 6.6: VM resource usage with VM application and migration

Finally, Figure 6.6 additionally shows the CPU and memory usage of the migrated VMs. The plots show no visible increase in CPU or memory usage. This underlines that the live-migration does not affect the resource usage inside the VM, but only on the host.

In summary, the R-Car cluster results clearly show the unsuitability for a live-migration in a host failure case. Due to the very long duration, the host would have most probably already failed before the complete VM would be fully migrated. On the Xeon cluster, on the other hand, the migration time is significantly lower. Also, the used VM flavor represents an improbable VM configuration, indicating even lower migration times using smaller VM flavors. Therefore, the results suggest actual usability of the live-migration functionality if the hardware is powerful enough.

6.3   Summary

Concluding from the two examined use cases, OpenStack's advantages can be exploited. On both platforms evaluated, the tested functionality is reliably and continuously available. However, the smaller resources on the R-Car nodes along with the not advantageous big.LITTLE architecture and slow-performing SD-Card, counteract a performable operation of OpenStack. While the performance doesn't influence the VM distribution, VM migration is greatly influenced. As migration times with idle VMs seem to be just in an acceptable range, further measurements show the influence of active VMs on the migration time. With migration times spanning multiple minutes up to half an hour, this is not practicable in a real-world scenario and therefore doesn't introduce additional value. Powerful hardware, in contrast, is able to perform the migration in reasonable time frames, even with very large VMs. This allows the assumption that also the ARM platform would achieve better results with more resources available. Figure 6.5 depicts that more memory, for example, 8 or 16 GB, could already suffice to achieve reasonable migration times. Together with more cores to provide adequate resources to the host and guest system and as well as faster storage better performance could most probably be achieved.

7. Conclusion

This thesis's main research objective was to determine whether the Cloud Computing advantages of OpenStack can be leveraged on embedded automotive hardware. The first challenge was the installation process and successful execution of OpenStack on such hardware. As systems like the Renesas R-Car SoC are not intended for such usage, the installation process poses difficulties. Due to custom kernels and OSs, crucial packages and kernel modules for OpenStacks's functionality might not be available, leading to the necessity of an extended kernel and OS, as described in Chapter 3.

After a successful installation of OpenStack, two aspects were examined in order to provide a comprehensive answer to the main research question: First, the overhead and the impact of OpenStack were examined. Through performing dedicated measurements on the CPU, memory, network, and storage as described in Chapter 4, the performance degradation through OpenStack as a middleware was examined. Summarizing the overhead results, OpenStack indeed introduces an overhead. However, the overhead is lower if the hardware is more powerful. This leads to low and acceptable overheads on the Xeon platform, while the R-Car platform suffers greater impacts. Especially the R-Car's big.LITTLE architecture, low memory, and low secondary storage performance, mainly lower OpenStacks performance. However, introducing hardware with minor changes like more CPU-cores, faster secondary storage, or more memory could presumably suffice to achieve acceptable and Xeon-like results.

Second, two advantages of Cloud Computing and OpenStack were examined in greater detail: VM distribution and VM migration. Both advantages can introduce benefits for the automotive area. Considering the VM distribution, it was found that OpenStack distributes VMs according to defined rules reliable and without errors on both tested platforms. As for the VM migration, also on both platforms, the live-migration was shown to happen reliably. However, the used ARM platform has too small resources to deliver a good performance and is therefore not suitable for such a use case, as found in Section 6.2.3 and Figure 6.5. Although all performed

tests succeeded and no wrong functionality was found, the performance was too low to enable the actual usage. The x86 platform, on the other hand, delivers excellent performance. Considering the live-migration, even using the highest possible VM configuration, it delivers very high performance, suggesting an actual in-vehicle usage.

Concluding from all measurements performed, OpenStack is basically executable on automotive embedded hardware. Despite the overhead on all examined resources, OpenStack runs as stable on the R-Car SoCs as on the Xeon CPUs. While the overhead in some cases is non-negligible, all measurements were performed on a DevStack environment without any optimization towards better performance. As this thesis's scope covers only the basic functionality of an OpenStack cluster, the chosen advantages of VM distribution and migration were reliably available. Considering the main research question, **Is it possible to leverage the advantages of Cloud Computing on embedded hardware?**, it can be answered with a **yes**. Although the ARM platform shows deficient performance, it also shows that OpenStack is functional on such hardware, even with a Preempt-RT kernel patch.

7.1 Further Work and Outlook

Gathered an answer, however, further tests are suggested to gain even more knowledge and confidence. Considering the performed tests, they should be extended regarding three areas:

- First and foremost, an installation without DevStack should be performed. Apart from unnecessary configurations and software, a better configuration of relevant settings could already improve the results.

- Second, the tests should be extended to a broader range of hardware, especially ARM hardware. Having compared a high-performance x86-based platform to an efficiency-focused big.LITTLE ARM-based SoC reduces both platforms' comparability and possibly introduces bottlenecks on the ARM platform due to insufficient resources. An interesting candidate would be the *NXP Layerscape LX2160A* processor[1] with 16 cores and no big.LITTLE architecture. Also, performing the test in a real-world environment with real automotive applications or benchmarks could further reveal undiscovered insights.

- Third and last, the OpenStack cluster should be optimized. The performed tests were executed on an out-of-the-box DevStack installation. Optimiz-

---

[1]NXP LX2160A Processor: https://tinyurl.com/yxe26f2t [22]

ing the cluster towards better network performance, higher computational performance, and faster storage solutions could significantly influence the results.

Besides extending the performed tests, also other, in this thesis not covered areas need to be studied. While the R-Car SoCs are running a kernel with a real-time patch, the specific real-time behavior was out of scope for this thesis. Proving that OpenStack and its VMs guarantee an in-time execution of tasks and therefore provide real-time capabilities would open the possibility the execute real-time applications inside VMs. Through this, OpenStack could introduce even more value as a middleware.

Besides handling computational tasks, today's ECUs must interact with various bus systems. Considering OpenStack as a middleware on multiple ECUs to execute crucial tasks inside VMs, some data exchange with, for example, sensors must happen. Although the Ethernet standard is already present in modern vehicles and the communication via IP-addresses is common, other bus systems will undoubtedly remain due to various factors like costs. As determinism, and therefore real-time capability, is also a crucial factor in some bus systems, real-time capable VMs would here as well be necessary, as well as a reliable and performant data exchange with other bus systems.

Further, a topic not considered by this thesis is the usage of hardware acceleration. Especially with, for example, image processing or machine learning, the necessity for graphical processing units is becoming increasingly critical. OpenStack enables VMs the usage of GPUs via PCI-passthrough. Here again, the usability in terms of reliability and performance would have to be evaluated.

Apart from the performance and possibilities of OpenStack, the actual usage and the structure of such a cluster would have to be evaluated. First, contrary to this thesis, in a real-world deployment of such a cluster, multiple VMs would be active concurrently. Due to an individual management overhead for each VM, the total overhead could be located in completely different dimensions. Second, countless different options exist how such a cluster could be organized. The major drawback of the cluster used throughout this thesis is the single point of failure, introduced by the core services' single presence. A malfunction or failure of one core service, like Neutron, or the whole controller node would lead to a non-functional cluster and, therefore, a complete failure of all VMs and their applications. Hence the question arises, what kind of architecture would be suitable in order to guarantee a reliable operation with efficient resource usage. As the core services are critical for a functional OpenStack cluster, a double presence of those services on the cluster would seem reasonable to eliminate the single point of failure.

Looking further into a future with vehicles commonly running local private clouds, also new topics arise. Having entirely decoupled the software from the hardware, VMs can be executed anywhere and perform any task. With bigger clouds in the backend, this introduces the possibility of offloading VMs into the backend and performing the actual computing there. Using technologies like 5G, calculations could happen in the more powerful backend, leaving the vehicle's ECUs with as little power consumption as possible. This enables the inspection of the vehicle as a participant in a multiaccess edge computing cluster. Instead of completely offloading VMs or applications to other clouds in the backend, the vehicle could act as an edge node, preprocessing specific data locally before sending them to other networks and clouds. This would genuinely enable to deal with the *how* to compute, instead of the *where* to compute.

7.2   Review

All initial goals of this thesis were achieved and implemented. An OpenStack cluster was successfully installed and executed on automotive embedded hardware. The knowledge gained will be used to further examine Cloud Computing and OpenStack's capabilities for in-vehicle usage.
Regarding the initial schedule, it could not be totally met. This is mainly due to the installation process of OpenStack on the ARM platform. The usage of Yocto was not known in advance and therefore not considered in the initial planning. Because of Yocto's complexity, the unsuitability of the OpenStack recipes was only detected after four weeks. The discovery of using Ubuntu on the R-Car nevertheless enabled the installation of OpenStack. Through the usage of DevStack, the overall installation process could be accelerated and simplified. Still, the according kernel and ATF configuration took more time than anticipated, leading to a delayed start of the measurements.

The execution of the measurements progressed as planned. However, the incomparability of the chosen platforms quickly emerged. The limited time frame and the unavailability of according hardware prevented the evaluation of further, more comparable platforms. Also, the again specific installation process on other hardware has additionally hindered other platforms' consideration. An earlier in-depth review, along with more time and initial tests, could have revealed this property, enabling the use of additional hardware for more confident results.

References

[1] Ahmed Alzahrani, Nasser Alalwan, and Mohamed Sarrab. "Mobile cloud computing: Advantage, disadvantage and open challenge". In: *ACM International Conference Proceeding Series* (2014), pp. 4–7. DOI: 10.1145/2590651.2590670.

[2] Amazon. *What is cloud computing?* [Online; accessed November, 2020]. 2020. URL: https://aws.amazon.com/what-is-cloud-computing/?nc1=f{_}cc.

[3] Arm. *Trusted Firmware Open Governance Project.* [Online; accessed January, 2021]. 2021. URL: https://developer.arm.com/tools-and-software/open-source-software/firmware/trusted-firmware.

[4] A. Binu and G. Santhosh Kumar. "Virtualization Techniques: A Methodical Review of XEN and KVM". In: *Advances in Computing and Communications.* Kochi, India: Springer, 2011, pp. 399–410. DOI: 10.1007/978-3-642-22709-7_40. URL: http://link.springer.com/10.1007/978-3-642-22709-7{_}40.

[5] Andreas Blenk et al. "Survey on network virtualization hypervisors for software defined networking". In: *IEEE Communications Surveys and Tutorials* 18.1 (2016), pp. 655–685. ISSN: 1553877X. DOI: 10.1109/COMST.2015.2489183. arXiv: 1506.07275.

[6] Drawingdatabase. *BMW 5 Series Limousine G30 2020 Blueprint.* [Online; accessed January, 2021]. URL: https://drawingdatabase.com/bmw-5-series-limousine-g30-2020/.

[7] OpenStack Foundation. *OpenStack Documentation.* [Online; accessed January, 2021]. 2021. URL: https://docs.openstack.org/victoria/.

[8] Pawel Garbacki and Vijay K. Naik. "Efficient Resource Virtualization and Sharing Strategies for Heterogeneous Grid Environments". In: *2007 10th IFIP/IEEE International Symposium on Integrated Network Management.* IEEE, 2007, pp. 40–49. ISBN: 1-4244-0798-2. DOI: 10.1109/INM.2007.374768. URL: http://ieeexplore.ieee.org/document/4258520/.

[9] Lee Gillam et al. "Fair benchmarking for cloud computing systems". In: *Journal of Cloud Computing* 2.1 (2013). ISSN: 2192113X. DOI: 10.1186/2192-113X-2-6.

[10] Daniel Hallmans et al. "Challenges and opportunities when introducing cloud computing into embedded systems". In: *Proceeding - 2015 IEEE International Conference on Industrial Informatics, INDIN 2015* (2015), pp. 454–459. DOI: 10.1109/INDIN.2015.7281777.

[11] Arne Holst. *Public IT cloud services market revenue worldwide from 2016 to 2019.* [Online; accessed November, 2020]. 2020. URL: https://www-statista-com.thi.idm.oclc.org/statistics/370305/global-public-it-cloud-services-spending-by-segment/.

[12]   Mikyung Kang et al. "A Comparison of System Performance on a Private OpenStack Cloud and Amazon EC2". In: *IEEE International Conference on Cloud Computing, CLOUD* 2017-June (2017), pp. 310–317. ISSN: 21596190. DOI: 10.1109/CLOUD.2017.47.

[13]   Charalampos Gavriil Kominos, Nicolas Seyvet, and Konstantinos Vandikas. "Bare-metal, virtual machines and containers in OpenStack". In: *Proceedings of the 2017 20th Conference on Innovations in Clouds, Internet and Networks, ICIN 2017* (2017), pp. 36–43. DOI: 10.1109/ICIN.2017.7899247.

[14]   KPMG. *Usage of cloud computing in companies in Germany from 2011 to 2019.* [Online; accessed January, 2021]. 2020. URL: https://www-statista-com.thi.idm.oclc.org/statistics/459998/usage-of-cloud-computing-by-companies-in-germany/.

[15]   Edward A. Lee. "Computing Foundations and Practice for Cyber- Physical Systems : A Preliminary Report". In: *Electrical Engineering* UCB/EECS-2007-72 (2007), pp. 1–27. URL: http://www.eecs.berkeley.edu/Pubs/TechRpts/2007/EECS-2007-72.pdf.

[16]   Max Malinowski. *Evaluation of OpenStack as Middleware for Automotive Embedded Systems.* Version v0.1. 2021. DOI: 10.5281/zenodo.4478425. URL: https://doi.org/10.5281/zenodo.4478425.

[17]   Dan C. Marinescu. "Cloud Resource Virtualization". In: *Cloud Computing.* Second. Cambridge: Elsevier, 2018. Chap. Cloud Reso, pp. 365–402. DOI: 10.1016/b978-0-12-812810-7.00013-3. URL: https://linkinghub.elsevier.com/retrieve/pii/B978012812810700011X.

[18]   Peter Marwedel. *Embedded Systems Foundations of Cyber-Physical Systems.* Thrid. Embedded Systems. Cham: Springer International Publishing, 2018, pp. 1–25. ISBN: 978-3-319-56043-4. DOI: 10.1007/978-3-319-56045-8. URL: http://link.springer.com/10.1007/978-3-319-56045-8{_}1http://link.springer.com/10.1007/978-3-319-56045-8.

[19]   John D McCalpin. "Memory Bandwidth and Machine Balance in Current High Performance Computers". In: *IEEE Computer Society Technical Committee on Computer Architecture (TCCA) Newsletter* May (1995), pp. 19–25.

[20]   P M Mell and T Grance. *The NIST definition of cloud computing.* Tech. rep. Gaithersburg, MD: National Institute of Standards and Technology, 2011. DOI: 10.6028/NIST.SP.800-145. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf.

[21]   Nerdio. *20 Cloud Computing Quotes You Can't Miss.* [Online; accessed January, 2021]. 2016. URL: https://www.slideshare.net/Nerdio/20-cloud-computing-quotes-you-cant-miss.

[22]   NXP Semiconductors N.V. *Layerscape® LX2160A Processor.* [Online; accessed January, 2021]. 2021. URL: https://www.nxp.com/products/processors-and-microcontrollers/arm-processors/layerscape-processors/layerscape-lx2160a-processor:LX2160A.

[23]   OpenStack. *Hypervisor Tuning Guide.* [Online; accessed January, 2021]. 2016. URL: https://wiki.openstack.org/wiki/Documentation/HypervisorTuningGuide.

[24]   OpenStack. *Introduction: A Bit of OpenStack History.* [Online; accessed January, 2021]. 2018. URL: https://docs.openstack.org/project-team-guide/introduction.html.

[25] OpenStack. *Multi-Node Lab.* [Online; accessed January, 2021]. 2020. URL: `https://docs.openstack.org/devstack/victoria/guides/multinode-lab.html`.

[26] OpenStack. *Nova System Architecture.* [Online; accessed January, 2021]. 2019. URL: `https://docs.openstack.org/nova/victoria/user/architecture.html`.

[27] OpenStack. *The OpenStack Landscape.* [Online; accessed January, 2021]. 2020. URL: `https://www.openstack.org/software/`.

[28] Gerald J. Popek and Robert P. Goldberg. "Formal requirements for virtualizable third generation architectures". In: *Communications of the ACM* 17.7 (1974), pp. 412–421. ISSN: 0001-0782. DOI: `10.1145/361011.361073`. URL: `https://dl.acm.org/doi/10.1145/361011.361073`.

[29] Jonas Repschläger, Danny Pannicke, and Rüdiger Zarnekow. "Cloud Computing: Definitionen, Geschäftsmodelle und Entwicklungspotenziale". In: *HMD Praxis der Wirtschaftsinformatik* 47.5 (2010), pp. 6–15. ISSN: 1436-3011. DOI: `10.1007/bf03340507`.

[30] Robert Bosch GmbH. *Bosch is actively shaping the transformation in automotive electronics.* [Online; accessed January, 2021]. 2020. URL: `https://www.bosch-presse.de/pressportal/de/en/bosch-is-actively-shaping-the-transformation-in-automotive-electronics-222214.html`.

[31] RussianNeuroMancer. *"kvm_init_vcpu failed: Invalid argument" error on ARM big.LITTLE boards.* [Online; accessed January, 2021]. 2019. URL: `https://bugs.launchpad.net/ubuntu/+source/libvirt/+bug/1832395`.

[32] Selome Kostentinos Tesfatsion, Cristian Klein, and Johan Tordsson. "Virtualization techniques compared: Performance, resource, and power usage overheads in clouds". In: *ICPE 2018 - Proceedings of the 2018 ACM/SPEC International Conference on Performance Engineering* 2018-March (2018), pp. 145–156. DOI: `10.1145/3184407.3184414`.

[33] Tnx94241. *Run Ubuntu 18 on the R-Car Starter Kit.* [Online; accessed January, 2021]. 2020. URL: `https://elinux.org/R-Car/Ubuntu`.

[34] R Lingeswaran (UnixArena. *OpenStack Architecture and components overview.* [Online; accessed January, 2021]. 2015. URL: `https://www.unixarena.com/2015/08/openstack-architecture-and-components-overview.html/`.

[35] Venu Vedam and Jayanti Vemulapati. "Demystifying cloud benchmarking paradigm, An in depth view". In: *Proceedings - International Computer Software and Applications Conference* (2012), pp. 416–421. ISSN: 07303157. DOI: `10.1109/COMPSAC.2012.61`.

[36] Philipp Weber et al. "Energy-Optimized Elastic Application Distribution for Automotive Systems in Hybrid Cloud Architectures". In: *Proceedings - Euromicro Conference on Digital System Design, DSD 2020* (2020), pp. 455–462. DOI: `10.1109/DSD51259.2020.00078`.

[37] Yocto Project. *The Yocto Project.* [Online; accessed January, 2021]. 2021. URL: `https://www.yoctoproject.org`.

[38] Detlef Zerfowski and Andreas Lock. "Functional architecture and E/E-Architecture – A challenge for the automotive industry". In: *19. Internationales Stuttgarter Symposium.* Ed. by Michael Bargende et al. Wiesbaden: Springer Fachmedien Wiesbaden, 2019, pp. 909–920. ISBN: 978-3-658-25939-6. DOI: `10.1007/978-3-658-25939-6_70`.