CHALMERS
UNIVERSITY OF TECHNOLOGY

Container Based Virtualisation for Software Deployment in Self-Driving Vehicles

Master's thesis in Software Engineering

Philip Masek

Magnus Thulin

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Gothenburg, Sweden 2016

Master's thesis 2016

Container Based Virtualisation for Software Deployment in Self-Driving Vehicles

Philip Masek

Magnus Thulin

CHALMERS
UNIVERSITY OF TECHNOLOGY

Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
Gothenburg, Sweden 2016

Container Based Virtualisation for Software Deployment in Self-Driving Vehicles
Philip Masek & Magnus Thulin

Supervisor: Hugo Sica de Andrade, Department of Computer Science and Engineering
Supervisor: Christian Berger, Department of Computer Science and Engineering
Examiner: Regina Hebig, Department of Computer Science and Engineering

Master's Thesis 2016
Department of Computer Science and Engineering
Division of Software Engineering
Chalmers University of Technology
SE-412 96 Gothenburg

Container Based Virtualisation for Software Deployment in Self-Driving Vehicles
Philip Masek
Magnus Thulin
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract

Delivering new software features in a continuous fashion has become a competitive advantage for organisations operating in the web domain. Being able to deliver new features to customers on a regular basis allows organisations to rapidly respond to change in customer requirements and to verify customer value. Software development in the domain of web applications differ greatly in comparison to embedded, cyber-physical systems which are tightly coupled to hardware, electronics and mechanics. A cyber-physical system (CPS) can benefit from a platform that enables the continuous deliver of new features. Virtual machines is a popular method for software deployment where applications are sand-boxed and pre-installed in a highly portable environment. This study contributes to the research community by understanding the performance overhead of using virtual containers as a deployment platform for CPSs which are highly sensitive to timing delays. Methods of experimentation are used to understand the timing behaviour of two sample applications realised with the development architecture for CPSs, OpenDaVINCI. Sample applications are run in various deployment and execution environments where a real-time enabled Linux kernel is used. Hypotheses testing and statistical analysis is performed on timestamps extracted from the sample applications, where results show that the virtual container manager Docker achieves near native performance when executing applications in a virtual environment in comparison to native execution. The experiment is executed in a controlled environment where the results are validated by adapting the experiment on a self-driving vehicle that participated in the Grand Cooperative Driving Challenge 2016 held in the Netherlands. This research concludes that Docker together with a real-time enabled kernel is a deployment platform good candidate for vehicular CPSs.

Keywords: Deployment, self-driving vehicles, cyber-physical systems, virtual containers, Docker, real-time systems, experiment.

Acknowledgements

We extend our gratitude to Christian Berger and Hugo Sica de Andrade for their dedicated support during the thesis project. We also thank Ola Benderius and the GCDC team at Chalmers Revere for accommodating and allowing us to experiment on the self-driving vehicle. We further extend our thank you to SAFER, the Vehicle and Traffic Safety Centre at Chalmers University for hosting us during the thesis project and allowing us to use their offices for running the experiments. We would like to thank the PhD students in the Department of Software Engineering for dedicating time into helping us throughout the project.

Philip Masek & Magnus Thulin, Gothenburg, April 2016

Contents

1
Introduction

In order for organisations to remain competitive, there is a need to continuously improve time to market for new products, features and services. The societal transformation of moving from a product economy to a service economy has affected the way organisations deliver products and services [1]. Products are required to move from business requirements to delivery as fast as possible. This societal transformation from a product to service economy has given rise to a number of tools and methodologies that bring more value to customers in a shorter amount of time. Continuous integration (CI) and continuous deployment (CD) are software engineering concepts that have become strongly adopted by organisations in order to deal with the need for more agility. More agility is required in order to rapidly respond to change in customer requirements in a market that is constantly evolving. With a well designed CI and CD process, organisations can deploy new features to customers in a short amount of time and on a regular basis. Being able to deploy multiple times a month or even multiple times per day has become a competitive advantage for companies operating in the web domain [2]. Companies can deploy new features quickly in order to verify customer value by carrying out practices such as A/B testing [3]. A/B testing is the procedure of comparing two software versions in order to identify which performs better. The popularity and success of CI and CD is strongly based on web applications. However, software development in the domain of web applications differ greatly when compared to embedded systems. Such systems, as described by Lwakatare et al. [4], are tightly coupled to hardware, electronics and mechanics that introduce complexities typically not seen for development in the web domain. Continuous deployment of features is becoming an important factor for applications in the domain of Internet of Things (IoT) and cyber-physical systems. The use of virtualization has also become a popular trend in real-time embedded systems within the automotive industry [5]. This is apparent in current literature [6, 7, 8], that experiment with virtual containers for deployment in IoT applications and cyber-physical systems.

A cyber-physical system (CPS) consists of computer systems collaborating and coordinating to control physical resources that interact with their surroundings [9]. CPSs are becoming an integral part of society and are already available to consumers in modern automotive vehicles. Collision avoidance, autonomous parking and autonomous highway driving are some examples of CPSs within the automotive domain. By nature, such systems are very complex in their design and development, involving multiple software and hardware components of different types and architectures. The academic discipline of CPS aims to help designers and developers with

the complexity of such systems, requiring expertise from three major disciplines: (1) communication in heterogeneous networks, (2) embedded and real-time systems and (3) control systems [6].

The continuous deployment of features for a vehicular CPS is challenging due to the fact that they are real-time systems with safety and timing requirements. Furthermore, vehicular CPSs are typically resource constrained [10], so scaling up hardware is limited to a physical capacity (such as the size of the vehicle). Since a CPS requires real-time operations, any additional overhead affecting performance should be handled with care. Overhead can introduce latency which may consequently impact timing requirements that can result in software malfunction. CPSs typically interact with their surroundings so safety is of a high concern.

Deploying any complex system to a production environment is a challenging procedure. Using virtual machines simplifies the deployment process by packaging the application in a isolated sand-boxed environment. Shipping an application preinstalled in a virtual machine (VM) has many benefits: it decouples the system into sand-boxed subsystems improving scalability, independent versioning can take place per VM, safe roll back can be applied in the event of buggy code, A/B testing can be performed and resources can be limited and controlled during runtime. As real-time requirements are crucial to autonomous vehicles, the cost to performance must be identified if considering virtualization as a deployment platform.

Virtualization can be done with a fully fledged virtual machine (e.g Oracle VirtualBox) or by using lightweight virtual containers. Virtual machines require more runtime resources and bring higher performance overhead since they require a full copy of a operating system as well as virtualizing the available hardware [11], so they are not suitable for a vehicular CPS. A more lightweight approach is to use virtual containers that share the host operating system (OS) rather than encapsulating an entire OS stack. Docker containerization is an open source technology that wraps applications into sand-boxed environments that are highly portable [11]. Docker containers are much faster to start up (typically less than a second) in comparison to a VM, since VMs carry extensive resource usage and so typically cannot be used on small computers or resource-constrained devices. Containers do not virtualize the available hardware, but rather act as a sand-box for applications that package and isolate in application improving scalability, security and reliability [6].

1.1 Problem Domain & Motivation

This study aims to uncover the performance impact of using virtual containers for software deployment in the context of self-driving vehicles. Self driving vehicles require minimal delays during runtime to allow real-time computations that enable safe autonomous driving. Minimal time-delay is a fundamental concern for allowing lane-following, decision making, and other computations utilised by the autonomous vehicle to interact with its surroundings. The software composed for self-driving vehicles can benefit by the use of virtualization, during development (such as safe

roll back and independent versioning) as well as post-development to ship updates and patches. In order to consider the use of virtual containers for vehicular CPSs, decision makers need to first understand the overhead that is introduced to ensure that the system meets the real-time requirements for safe driving.

There is a lack of evidence that present how state-of-art deployment strategies impact CPS systems in the domain of self-driving vehicles with their time sensitive performance requirements. There exists studies that explore the performance overhead of using virtual containers for general purpose software in the web domain. However, the requirements of such systems differ greatly to that of a self-driving vehicle. This creates a compelling gap in literature to explore virtual containers as a deployment platform in the domain of self-driving vehicles to build argumentation which decision makers can rely upon when determining for which deployment set-up is most suitable for the real-time application in question. The popularity of deployment strategies utilising containers is steadily increasing, thus making it important to understand the performance overhead introduced by containers such as Docker. While the implementation of virtualization technologies for deployment strategies brings many advantages, there still exists uncertainty to the disadvantage of how much, if any, performance overhead they carry.

It is of particular importance to understand the impact of using virtual containers for decision makers responsible for determining deployment strategies for time-critical systems utilised by self-driving vehicles. The rationale being that real-time systems are time sensitive and must guarantee responses within a specified time. If the system is to violate the required response-time it may lead to software failure, which can potentially be catastrophic as self-driving vehicles interact with their surroundings. As a consequence it is crucial to ensure that the execution environment and approach to software deployment in use will allow the real-time application to stay within its specified timing parameters. This is the gap in which the result of this research will seek to fulfil by recording specific measurement data from containerised sample-applications to identify the performance overhead.

In this study, an experiment is designed and executed to uncover the timing behaviour of two sample applications realised with the CPS open source development architecture, OpenDaVINCI [12]. Measurement points, in the form of nanoseconds, are extracted from the sample applications during runtime in order to measure the applications timing behaviour when executed in a native environment versus being executed within a virtual container to uncover the impact. Runtime latency of containerising two sample applications is mitigated by using a real-time enabled Linux kernel and comparing the performance to a stock Linux kernel. Docker [13] is the chosen technology for containerising the applications since Docker is an open source project that offers fast deployment of applications inside portable containers with a highly consistent environment [14]. Furthermore, the findings from the conducted experiment is replicated on a self-driving truck that participated in the 2016 Grand Cooperative Driving Challenge in The Netherlands [15]. The findings are replicated to further validate whether or not executing a CPS application in a Docker con-

tainer has an impact on the timing behaviour of the CPS application in a realistic environment.

1.2 Research Goal & Research Questions

This research seeks to systematically study the impact various execution environments have on two sample applications realised with OpenDaVINCI. The sample applications are tasked to measure scheduling precision and input/output performance respectively. A controlled experiment is performed to measure the timing behaviour (scheduling precision and I/O performance) of two sample applications running in the different execution environments in order to answer the following research questions:

RQ1 Does the respective execution environment influence the scheduling precision of the respective application?

RQ2 Does the respective execution environment influence the input/output performance of the respective application?

The execution environments is an alternation of the deployment context and an alternation of Linux kernels, in order to uncover the precise performance cost of using Docker as a deployment platform. For containerising the sample applications, Docker is the chosen technology due to its popularity in both academia and industry. The execution environments consist of a vanilla and a real-time enabled OS. The execution of the applications take place within a container where a comparison can be made relative to non-virtualized Linux. Similarly, a comparison can be made when using a vanilla OS relative to a real-time enabled OS. The following execution environments stem from the need to carry out a fair comparison by using a vanilla OS and native execution versus a containerised real-time enabled execution, as real-time enabled OS is required for time sensitive applications, to uncover the performance cost:

1. Executing the sample applications natively on Ubuntu Server LTS Linux.
2. Executing the sample applications inside a Docker container on Ubuntu Server LTS Linux.
3. Executing the sample applications natively on a real-time enabled Ubuntu Server LTS Linux.
4. Executing the sample applications inside a Docker container on a real-time enabled Ubuntu Server LTS Linux.

Answering the research questions will build argumentation on whether using Docker as a deployment platform in respect to the execution environment is viable for self-driving vehicles. The scheduling precision and input/output performance of the two sample applications is measured to uncover if using Docker will violate the timing requirements of the applications which are crucial for CPSs.

1.3 Contributions

The results of this research can be used for a broad audience within the research community as well as for organisations interested in adopting new technology to improve software deployment for cyber-physical systems. As more segments of today's society are becoming automated and reliant on software decision making, real-time systems play an integral part of this development. Financial, aviation, and vehicle systems are just a few examples of domains with systems that are highly sensitive to time delays. A self-driving vehicle has to interpret its surroundings in real-time where any delay can have a catastrophic effect. Similarly, applications in the financial domain have to react to market fluctuations within nanoseconds to avoid loss on investment. There exists research on the performance overhead of containerising applications in the context of cloud computing, however this study presents results on the performance overhead in the context of vehicular CPS. This study contributes with precise timing behaviour of two sample applications executed in various execution environments in the context of self-driving vehicles. The findings of the experiment build argumentation on which execution environment is desirable for implementation on the use-case of the self-driving truck. The use case validates the need for state-of-art deployment strategies for self-driving vehicles.

1.4 Scope

The scope of the study is in the domain of self-driving vehicles. The evaluation of two sample applications have been prioritised to evaluating the timing behaviour (scheduling precision and input/output performance) respectively. Schedulability analysis and input/output performance are crucial factors for self-driving vehicles and so have been prioritised over evaluating other factors such as networking and memory performance. Schedulability analysis is conducted as timing delays impact safe driving, and secondly, I/O performance such that storing information (such as recording a video stream and logging sensor data) is important for debugging and further development of algorithms. Furthermore, the study aims to implement the findings from the conducted experiment on the self-driving truck and measure the outcome to understand how the CPS application behaves in a realistic environment.

1.5 Structure of the thesis

The remainder of the thesis is structured as follows: Section 2 introduces the background of the paper, detailing the technology and concepts related to this study. Furthermore, related work and the process of gathering related work is specified in section 3. The research methodology is specified in section 4, detailing information on the experiment carried out to answer the research questions. Section 5 introduces the results from the experiment, where a discussion and conclusion is found in section 7 and section 9. Section 6 presents the conducted experiment on the use-case scenario of the self-driving truck.

2
Background

This section introduces further background information on cyber-physical systems, real-time scheduling concepts, software deployment and container-based virtualization.

2.1 Cyber Physical Systems

A cyber physical system (CPS) interacts with the world, sensing their surroundings by using embedded sensors, actuators and processors. These sensors, atuators and processors communicate and collaborate with each other to support real-time, guaranteed performance in safety critical applications [16]. Applications for CPSs span multiple domains including healthcare (robotic surgery), transportation (autonomous vehicles), agriculture, energy, and home automation to name a few. In the transportation domain for CPSs, autonomous vehicles, also called self-driving vehicles, are vehicles that operate without requiring human input. The intelligent behaviour of a self-driving vehicle include trajectory generation, lane following, lane keeping and intersection handling [16]. These are some of the computational responsibilities of a CPS for driver-less vehicles.

The academic discipline of CPSs was formed in the 2000s [17] to help designers and developers deal with the complexity of realising CPSs [6]. Even though forms of CPSs have been in industrial use for a long time, it is only recently that technology (such as wireless communication and processors) with impressive capabilities is available at low costs. This has led to the need for a more systematic approach to the development of reliable CPSs. Three main functional components of a CPS is control, computing and communications [17].

One key factor for the functional component of computing is the need for real-time computations. Cyber-physical systems require real-time computations to ensure a high degree of safety when interacting with the world. Comparing a real-time system with an ordinary application, the difference is seen in how the execution of code is made. For an ordinary application an algorithm is executed once to provide a resulting output from an input without any specified timing constraints. However, a real-time system is recognised by its time constrained characteristic as the system is configured to execute an algorithm within a specified time-slice, i.e. executing an algorithm continuously which for each iteration takes `10 milliseconds`. This is a time-triggered component, that after the passing a specific amount of time, the

component is once again executed.

Systems used for autonomous self-driving vehicles utilises a number of different algorithms to enable the self-driving functionality. Such algorithms may be responsible for processing a camera feed, detecting lanes within captured images or decision making such as steering, braking, or accelerating the vehicle depending on the content of input data. All these algorithms are embedded in a middle-ware application which sets the time constraints of the algorithms and handles the communication between them. This study utilises the open source middle-ware, OpenDaVINCI. OpenDaVINCI is compact middle-ware written in standard C++ that offers a high performing hardware and OS abstraction layers for concurrency, data storage and communication [12]. OpenDaVINCI has been used to realise a number of self-driving vehicles.

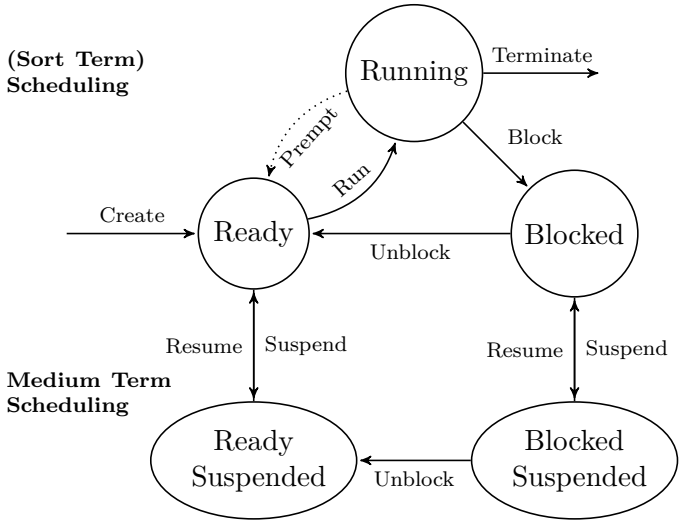
2.2 Real Time Scheduling

CPSs inherit the discipline of real-time systems requiring real-time computations so that the application exhibits the intended timing behaviour. A CPS is composed of multiple computer processes, each with their own demands for processing time. Orchestrating the demands of each process for processing time is the responsibility of the operating system scheduler. In this section we first introduce the concept of scheduling, describe scheduling precision and an introduce preemptive scheduling.

2.2.1 Scheduling Concepts

A unit of computation that requires the allocation of processing time is referred to as a *job*. Jobs can be event-driven (a specific event triggers activities in the system) or time-triggered (activities are initiated at predefined points in time). Time triggered events require strict timing behaviour and are necessary for hard real-time systems, thus being important for a CPS. When a time-triggered job is initiated, it communicates with the operating system scheduler to acquire processing time. The job enters a ready state assigned by the scheduler and starts processing. The job can then terminate (has reached the end of its processing) or become blocked which is what typically happens for time triggered jobs. A time-triggered job becomes blocked after processing the inner body of an algorithm, subsequently entering a sleep state to the scheduler. The job sleeps for a predefined amount of time: typically for how much time is left in the specified timing parameters. After the required time has elapsed, the job enters the state of being unblocked (waking the job after sleeping), hence ready to run for another execution cycle. A state machine of a scheduler, depicting all states is found in figure 2.1. In summary, a scheduler acts as a traffic police in a busy intersection, handling a queue of all the processes running on the system by prioritising some processes ahead of other processes. The scheduler may act and prioritise differently depending on what rules have been set for it to follow.

Figure 2.1: Operating System Scheduler [18].



Unblock is done by another task (a.k.a. wakeup, release, V)
Block is a.k.a. sleep, request

2.2.2 Scheduling Precision

The experiment conducted in this study analyses the scheduling precision and input/output performance of the automotive real-time application. The application executes time-triggered computations within elements referred to as time-slices. The time-slice is a specification of time allocated for the algorithm to execute and deliver a result. A real-time application running at `100 hertz` executes 100 time-slices per second, which results in one time-slice being `10 milliseconds` or `0.01 second`. The `10 milliseconds` time-slice is the time deadline set for the specific application, which is the maximum time allowed for the assigned algorithm to finish its computations. In scenarios where the algorithm utilises less than the assigned time-slice the application will sleep for the remaining time until it fires a new execution. The operating system scheduler is responsible for assuring that the application sleeps for the time specified by the real-time application. The scheduler is also responsible for waking the real-time process after the specified sleeping time has elapsed. Other than the assigned algorithm, the real-time application consists of code which is responsible for communicating with the OS scheduler that controls the sleep of the time-slices. Therefore a part of the time-slice has to be consumed to execute the required code.
Scheduling precision refers to how accurately the application executes the specified algorithm from the point of firing the time-slice until the algorithm begins its computation. Further accuracy can be measured between the point of where the algorithm finishes its computations until the real-time application sleeps. Lastly, measurements can done to see whether the `sleep` function of the system actually sleeps for the remainder of the time-slice or if it overstays the specified timing deadline. It is important to understand how much time each part of the required code occupies the time-slice. The less time required for executing the code outside of

the assigned algorithm the more deterministic a system is said to be. In a scenario where the assigned algorithm requires 80% of the time-slice to execute the code, it is assumed that the application will sleep for the remaining 20%. However, as there exists additional operations surrounding the algorithm the application might sleep 18% whereas 2% is required for the surrounding code to execute. If the application still sleeps 20%, executes the algorithm for 80%, and uses 2% for the required code it will overstay its time-slice by 2% thus rendering the application less deterministic (by a small amount). It is the time available for the algorithm the experiments of this study will seek to identify to inform software engineers of how much of the time-slice can be used for effective computations, i.e. time available for generating a result.

2.2.3 Preemptive Scheduling

A general purpose operating system scheduler implements a FIFO (first-in-first-out) approach with two process scheduler algorithms. Namely, a time-sharing algorithm and a real-time algorithm where the former is a scheduler that maintains fairness, distributing the system's resources equally over all processes in the queue ensuring that no process is completely starved. The later is an algorithm which prioritises the processes based on their set importance, where a higher prioritised process is provided more resources in comparison with a lower process. However, the generic Linux kernel version does not allow the scheduler to cancel all resources for processes already utilising the CPU. A higher prioritised process will therefore not be able to utilise 100% of the CPU's resources if there are other processes already using the CPU. For a general purpose operating system this approach is standard practise and is logical for non time critical processes. However for a real-time system it is crucial to ensure that the highest level process can interrupt any running processes at any point in time and occupy all resources available to ensure the process meets the time deadline. A real time enabled kernel using the RT_preempt kernel patch implements the preemptive approach thus allowing for such a behaviour of cancelling resources from lower prioritised processes to occupy all resources for a time-sensitive high priority process. Furthermore, the RT_preempt patch applies resource locking for RT prioritised processes so that lower prioritised processes cannot utilise the same resource simultaneously.

2.3 Software Deployment

Software deployment is a crucial part of the software development, it refers to the activities which makes the software system available for use [19]. The process contains a number of activities which all play into the life cycle of a software system with the goal to implement into the runtime environment where the system is set to operate live. Carzaniga et al. describe the following main activities of software deployment:

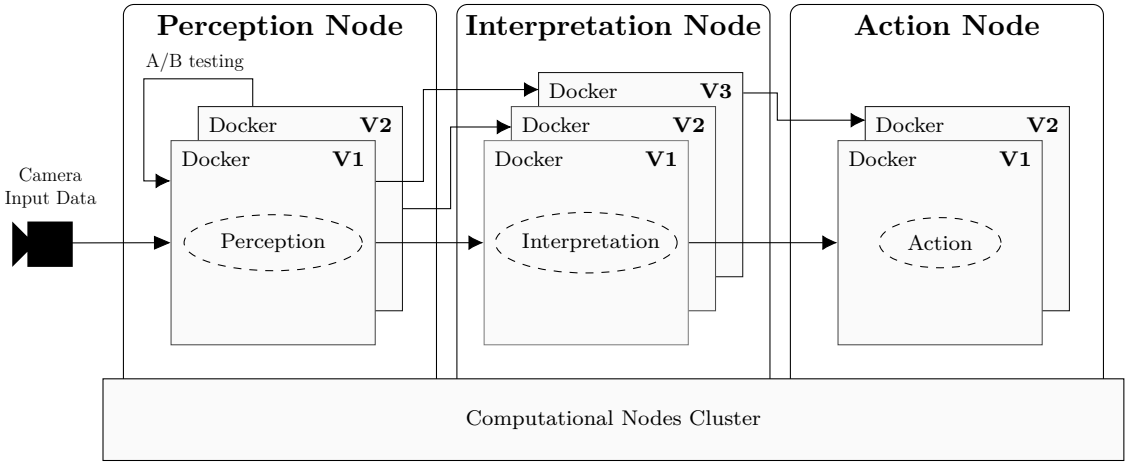
Release – is the activity of packaging the software for delivering it to the end user. This includes processes such as including the software's requirements and depen-

dencies to external components, such as libraries and applications. It also includes the process of advertising – the process of informing interested parties about the software being released.
Install – refers to the activities of assembling all required resources for the runtime environment. It consists of two specific process, namely *transfer* and *configuration*. Where the former is the process of transferring the software from the developer to the runtime environment and the later is the process of making the software ready for activation.
Activate – is the process of executing the software and all dependent applications in the runtime environment.
Deactivate – is the opposite of the *activate* activity.
Update – is the activity of updating the version of the running software which consists of similar activities of the *install* activity.
Adapt – refers to the process of ensuring that the updated version is running correctly in the runtime environment.
Deinstall – is the activity of decommissioning the running software and includes sub-activities such as removing the external libraries and components.
Derelease – is the final activity which includes the process of advertising the withdrawal of the software system.

All these activities differ in how they are executed depending on the software engineering paradigm utilised for the software project. Traditional software engineering practices, e.g. the waterfall model, seeks to execute the software deployment process at the end of the software's development cycle. Whereas more novel software engineering practices aim to execute the software deployment process continuously throughout the software's development cycle. In state-of-art software engineering practices such as continuous integration and continuous deployment, software is required to be deployed daily [20] for full adaptation. Such requirements can easily make the process of software deployment exhausting and complex, where software tools such as Docker would simplify these processes greatly. Docker simplifies processes found within each of the software deployment activities, as it provides the runtime environment before the software deployment process has begun. The development environment is a clone of the production environment thus transferring the deployment processes from the live production server to a confined secure location where the deployment process does not affect the usability of the current running software.

Figure 2.2 exemplifies a deployment strategy design utilising Docker in the context of self-driving vehicles. The responsibilities are broken down into three computational nodes in which Docker containers are running instances of the code base independently. Each Docker container can run different versions of the separate nodes, where the interpretation node has three versions running separately. Version one (denoted V1) in each node represents the latest working configuration while other versions are run to test code which is still under development. Having multiple versions of a running environment allows for safe and simple roll-backs in the event of buggy code or degraded performance. Furthermore, multiple versioning of

Figure 2.2: Run-time environment using Docker.



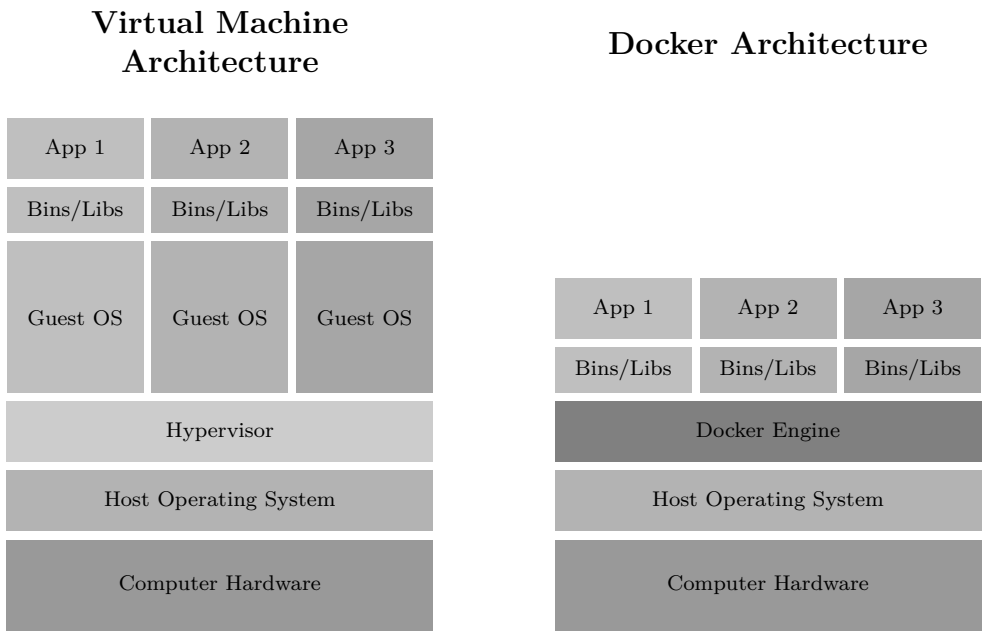
the same Docker container allows for split testing between different containers to take place. With an always functioning configuration, the development team can demonstrate a functioning version of the product to stakeholders at any point in the development cycle. The ability to demonstrate the product at any point in the development phase adds to the business value of the project, as possible investors or stakeholders can be presented a functioning product although the product is currently under development. This is possible with current deployment approaches, however the implementation of cloning systems or running virtual machines is more resource demanding [21] and not as flexible in comparison with utilising Docker for the deployment strategy.

2.4 Container-Based Virtualization

Docker is an open source light-weight container manager that launched in 2013 and has gained ground rapidly with its simplicity. The environment offered by Docker simplifies the process of software deployment by packaging all dependencies into a light-weight virtual container which ensures that all instances of the software is utilising the same dependent libraries. The functionality provided by Docker is comparable with virtual machines as both are virtualised environments where software applications can be executed with all dependent libraries and applications pre-installed. However, Docker, in contrast to virtual machines, is a light-weight alternative as it communicates directly to the host machine's kernel. A virtual machine applies the additional levels of a virtualising an operating system which adds complexity since the virtual OS does not speak directly with the host machine's kernel, as depicted in figure 2.3. With the benefit of packaging the dependent libraries and applications into a container, software developers can avoid uncertain deployments where library versions may differ between the developers' development environments and the live production environment. Docker presents further benefits such as safe roll-back between different software versions which provides projects the ability to always be able to fall back on application versions which are known to function correctly. By providing these benefits project managers can feel secure in

that there always exists a working runtime environment in the scenario of a new failing deployment.

Figure 2.3: Virtual Machines versus Docker [13]



The container in which Docker packages all dependent libraries and applications are referred to as a Docker image. This image contains everything which is required for an application to be executed. In the case of self-driving vehicles such dependencies may be image processing libraries such as OpenCV and the middle ware which enables the real-time application. When executing an application within Docker, a container is generated and executed based on the Docker image which consists of the installed libraries and applications. This software design allows for split testing of software as the same application can be executed multiple times without clashing with the other contained application. Thus being optimal for testing different versions of the same application simultaneously while knowing there is no interference between the executed applications.

3
Related Work

In this section we introduce (i) the process of identifying related work and (ii) discuss state-of-art on the scope of this study.

3.1 Methodology

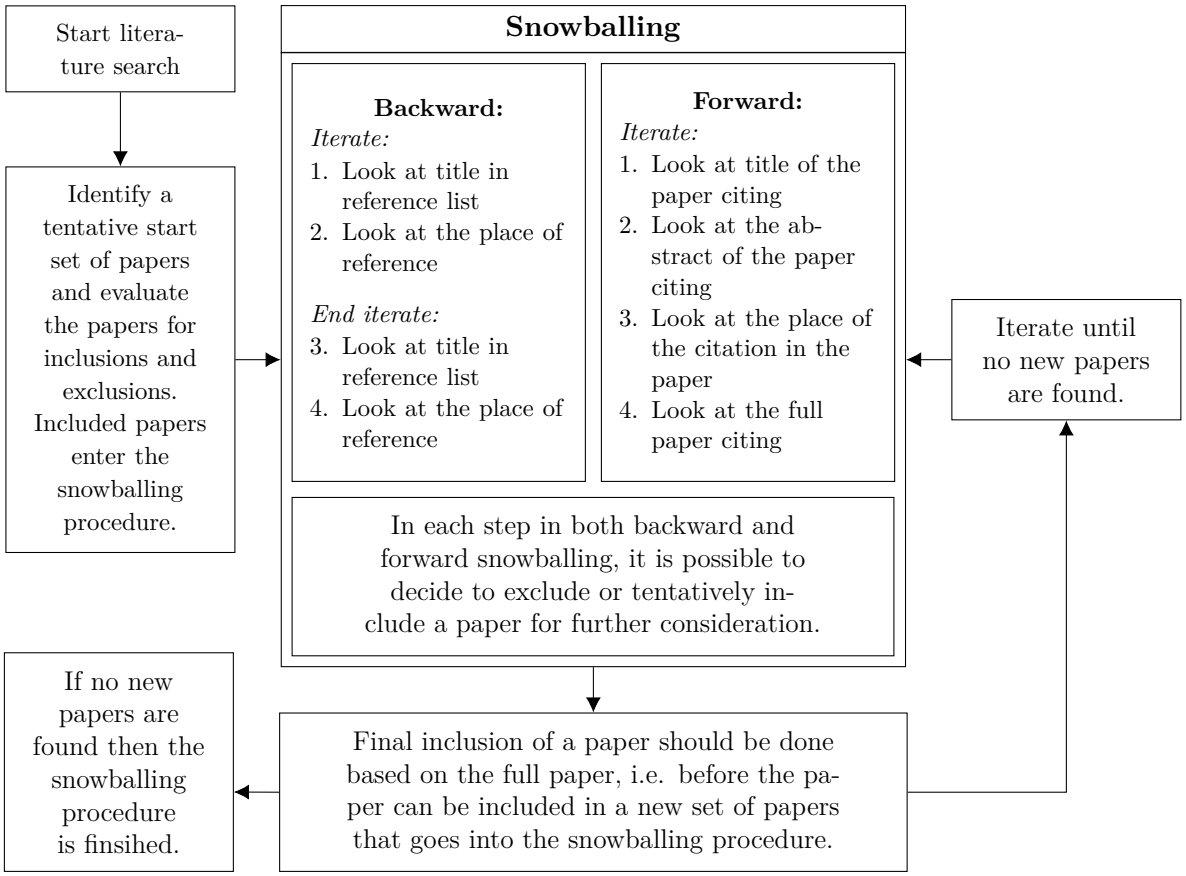
The snowballing search approach for systematic literature studies is used to find relevant literature on the topic of this paper. The snowballing search approach is used in order to perform a systematic approach to finding related work. The snowballing approach is complementary to a traditional database search. Specifically, the reference list and citations of a paper are studied in order to identify additional papers that are relevant to this particular review. The snowballing search approach is used to ensure good coverage of current literature in a systematic way.

The guidelines for conducting a snowballing search approach, presented by Wohlin [28] are followed for the search procedure. The steps to conduct a snowballing procedure, depicted in figure 3.1, involve (i) selecting a set of papers referred to as the start set, (ii) apply forward snowballing and (iii) apply backward snowballing on each paper identified in the set respectively. This process iterates until no new papers are found. To identify a start set of papers, keywords are extracted from the research questions, taking synonyms into account. Formulating a search-string from keywords that are broad and cover multiple areas of research may result in collecting large amounts of literature that span different subject domains. For that reason, broad keywords should be broken down into more specific and detailed keywords specific to the study. The search string is then applied to a database that preferably searches multiple publishers in order to avoid publisher bias. The papers resulting from the database search are then screened and included based on an inclusion and exclusion criteria. An exclusion criteria could state that all online material be excluded.

Backward and forward snowballing is then conducted on the start set. Backward snowballing is the process of studying the reference list to identify new papers. Looking at the place of reference and reading the title and abstract of the paper is a good starting point for inclusion, however, final inclusion states that the entire paper must be read [28]. Subsequently, forward snowballing is the process of identifying papers that cite the paper under inspection. The same process of reading the title, abstract and place of reference is applied in order to include new literature that is

found during the procedure.

Figure 3.1: Snowballing procedure [28].



3.2 Results

To gather a start set of papers, a database search on Scopus [29] is performed to identify papers from multiple publishers: the precise search string, in the required syntax for Scopus is:

TITLE-ABS-KEY(Performance OR Comparison OR Latency OR Evaluation OR Container-Based OR Linux Containers OR Lightweight Virtualization OR Container Cloud OR Docker) AND (LIMIT-TO(SUBJAREA,"COMP")

The search string is targeted at papers identifying the performance cost of using Docker. Applying the search string to the Scopus search function resulted in finding 215 papers from different publishers where no limitation to the search in respect to publication year is made. A screening process is then applied to the 215 papers, reading the title and abstract to see if the paper is related to this study. Papers from the set of 215 is then added to the start set upon meeting the inclusion and exclusion criteria. Papers were included if the data collected in the study is quantitative performance benchmarking. Papers were excluded if being reference to online

material or in a foreign language.

In total, eight candidates for inclusion were identified. The entire paper for each of the eight candidates is read as a requirement for final inclusion. The eight papers are listed in table 3.1, in order of reference. Backward snowballing is then applied to all papers in the start set. **P1** includes 26 references where one reference is already included in the start set (paper [21]). Based on passing the inclusion and exclusion criteria, one paper is read and passes for final inclusion. The paper identified and thus included is:

- [30] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange and C. A. F. De Rose, "Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments," 2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Belfast, 2013, pp. 233-240.

Table 3.2: Results from Backward Snowballing in Iteration 1

Start Set Paper	**No. References**	**Reference to Start Set**	**New Papers Identified**
[22]	26	[21]	[30]
[8]	21	[21], [24]	0
[23]	47	[21], [30]	0
[24]	42	[21], [8], [30]	0
[25]	46	[30]	0
[21]	50	[30]	0
[26]	19	[21], [30]	0
[27]	18		0

Table 3.2 lists the result of backward snowballing on the remaining seven papers. The results are presented in a table to avoid redundant text as the process and results of backward snowballing for each paper is very similar. In all papers, except [22], no new papers matched the inclusion criteria after reading the title and abstract respectively. Furthermore, most of the papers in the start set reference to each other, as seen in column three of table 3.2. All papers in the start set contain reference to [30], except papers [8] and [27]. Similarly, all papers in the start set reference to [21] except papers [25], [21] and [27].

Forward Snowballing is then applied to the papers in the start set. This step involves examining the citations towards all papers that are in the start set. All papers were searched for citations using Google Scholar. The Scopus database was chosen not to be used for the forward snowballing procedure since it was shown that Google Scholar was able to find more papers. Many of the papers in the start have been published recently, which may indicate the lack of citations for some of the papers. During the forward snowballing search, a relevant paper was found in regards to this study. It was then decided to include the paper as part of the related work upon

passing the inclusion and exclusion criteria. The specific paper included is:

- [31] Welch, James Matthew. Performance Optimization of Linux Networking for Latency-Sensitive Virtual Systems. Diss. ARIZONA STATE UNIVERSITY, 2015.

Table 3.3: Results from Forward Snowballing in Iteration 1

Start Set Paper	No. Citations	New Papers Identified
[22]	0	0
[8]	0	0
[23]	0	0
[24]	8	[31]
[25]	2	0
[21]	81	0
[26]	4	0
[27]	1	0

A second round of backwards and forwards snowballing is applied, as two additional papers to the start set were found in the first iteration.[30] has 32 references. From the 32 references, 16 references is excluded based on being references to online material. A large number of the resulting references have already been analysed in the previous iteration and no new papers were identified for inclusion. [31] has 69 references with reference to [30], [24], [21]. No additional candidates for inclusion were found when studying the reference list of the paper. A second round of forward snowballing is done. [30] has been cited by 104 papers. When reviewing the list of cited papers, many of them have already been reviewed during the snowballing procedure. Analysing the list of papers resulted in no additional candidates for inclusion. [31] has no citations. This is expected since [31] is not a published paper.

In total, ten papers is included for final inclusion for related work. Two additional papers are included but were not found during the snowballing procedure. The papers were found during the initial phase of the research project. One of the papers not have been found during the database search is a paper authored by C. Berger, which has been accepted for publication but has yet to be formally published. The second additional paper was found during the early stages of the research project and is relevant to this study. The search string includes the keywords present in the respective paper, however it was not found in the search. Since it is relevant to the study, it is included.

C. Berger [7] presents the exploration of a deployment strategy in the context of self-driving vehicles that utilises lightweight Docker containers. The deployment strategy makes use of a number of Docker containers that build and ship signed packages in a container ready for use. However, the paper does not look to identify the precise performance overhead when using Docker as part of the deployment

strategy but addresses the need for it in future work. The deployment of software in [7] exemplifies a possible deployment pipeline for resource constrained CPSs, that can be used as inspiration when investigating software deployment by using Docker.

The authors of [8] identify the challenge of deploying, maintaining and configuring software for IoT gateways and investigate using virtual containers to solve these challenges. The authors identify the performance overhead of using Docker as a deployment platform on two different versions of the Raspberry Pi System on Chip (SoC) computer [32]. They identify that there are clear benefits containerizing applications for deployment on resource constrained devices and further identify the need for such research is the field for IoT applications. This shows there exists a need for continuous deployment for embedded, resource contained and high performing applications. The authors of [8] recommend a case-by-case analysis when considering using Docker as a deployment platform for IoT devices. The case-by-case analysis is important as in [6] the study also experiment with Docker on a Raspberry Pi SoC computer but do not report any substantial degradation in performance. Further use of Docker as a deployment platform is exemplified in [6]. The authors of [6] explore using Docker as a deployment platform for a generic modular architecture for industrial automated cyber physical systems. A use case of the modular architecture is applied to the development of an automated guided vehicular CPS. The authors state that having a modular architecture, in the form of Docker containers, decouples the complexity of CPSs into simpler subsystems, that different teams can individually develop on. Development teams can work individually on separate subsystems and utilise Dockerhub, the team collaboration feature built into Docker. The authors of [6] conclude that using a real-time enabled Linux kernel is needed for further development of the architecture. There were only two papers found using the real-time enabled kernel when benchmarking Docker performance overhead. This study, through the controlled experiment, aims to provide measurement data when using a real-time enabled OS, fulfilling this gap.

In [21] the paper investigates the performance overhead of using virtual machines, Docker and compares the performance to native execution in the context of cloud computing. To analyse the processing overhead of using Docker, a lossless data compressions utility is used in various execution environments. The authors conclude containers have almost no overhead and recommend a case-by-case analysis as using network address translation and the AUFS storage driver are the only two factors that introduce considerable overhead. However, these factors are likely to become improved in the future.

In a study presented by Mao et. al [22], the authors investigate the next generation of Radio Access Networks (RANs) used by Telecom providers. Traditional RANs are moving towards software as a service in the cloud [22] due to being hardware dependent equipment which are expensive and lack scalability. Moving traditional RANs to the cloud as software RANs is a challenging task due to the latency requirements of cellular networks: a similar challenge for software with high hardware dependability and time-sensitivity face the domain of autonomous vehicles. To over-

come the time-sensitive requirements of software RANs, the authors use a real-time enabled Linux kernel, specifically the RT_preempt patch, and measure computational and networking latency when using Docker for real-time applications. The cyclic test [33] is used to measure computational latency and is a common tool to measure real-time capabilities of an operating system. However, the cyclic test is not comparable to a CPS as it lacks a supportive middleware. The worst case execution time in [22] is measured and reported in the study. The results of the study show that running real-time applications inside a Docker container achieves near-native performance, while the performance of virtual machines incur much higher latencies [22]. However, running multiple Docker containers on different hosts incur higher overhead [22]. Since CPSs may involve multiple computing nodes, measuring the performance of Docker on multiple hosts for real-time computations requires investigation if considering Docker as a deployment platform for multiple computing nodes.

In a study presented by Welch [31], the author identifies the performance overhead of using Docker for latency sensitive applications in the context of cloud computing. In [31], the RT_preempt kernel patch for Linux Kernel 3.18.20 is used to identify the networking performance of Docker containers in comparison to virtual machines. The results show that the Docker containers have generally higher bandwidth and lower latency than virtual machines. However, in both [31] and [21] the Network Address Translation (NAT) protocol introduces considerable overhead. NAT is needed when mapping a network to a single internet protocol (IP) address. Further research on this topic is required if the requirements for a CPS require independent IP address for multiple containers.

In papers [21, 31, 23, 24, 26, 30], chosen not to be further detailed due to their respective execution environments, state that the overhead by using Docker is negligible and CPU, memory, disk and network performance is near native. This confirms that Docker can be used for CPSs but further evidence is needed. However, [25] presents results on the performance overhead of Docker and recommend not combining disk and memory intensive workloads into different containers due in an apparent degradation of performance they observers. They suggest consolidating I/O and CPU intensive workloads to alleviate this problem.

In [27], the authors experiment with using multiple Docker containers for parallel processing to achieve real-time image processing. This is another benefit of using Docker as a deployment platform. If the resources of computing nodes allow for scaling up important processes in a CPS, one can run multiple Docker containers tasked with the same responsibility to produce an output faster (i.e parallel processing). The authors of [27] develop a drone tasked with tracking three mobile robots from a video stream. The video stream is sent to the cloud, in which multiple docker containers process the data in parallel. The authors were able to improve input frame rate by 158%, using two containers, to achieve real-time image processing.

From the papers searched and analysed our experience shows that there are not many papers that study the impact of lightweight containers in the domain of CPSs. Fur-

thermore, there are not many studies that use a real-time enabled Linux kernel when investigating the performance impact of container based virtualization. These areas are the focus point this study will contribute to the greater research community.

Table 3.1: Start Set

Paper No.	Citation
[22]	C. N. Mao, M. H. Huang, S. Padhy, S. T. Wang, W. C. Chung, Y. C. Chung, and C. H. Hsu, "Minimizing latency of real-time container cloud for software radio access networks," in 2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom), Nov 2015, pp. 611–616.
[8]	A. Krylovskiy, "Internet of things gateways meet linux containers: Performance evaluation and discussion," in Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on, Dec 2015, pp. 222–227.
[23]	M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing," in Information,Electronic and Electrical Engineering (AIEEE), 2015 IEEE 3rd Workshop onAdvances in, Nov 2015, pp. 1–8.
[24]	R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in Proceedings of the 2015 IEEE Interational Conference on Cloud Engineering, ser. IP10E '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 386–393.
[25]	M. G. Xavier, I. C. D. Oliveira, F. D. Rossi, R. D. D. Passos, K. J. Matteussi, and C. A. F. D. Rose, "A performance isolation analysis of disk-intensive workoads on container-based clouds," in 2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, March 2015, pp. 253–260.
[21]	W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on, March 2015, pp. 171–172.
[26]	C. Ruiz, E. Jeanvoine, and L. Nussbaum, "Performance evaluation of containers for hpc," in Euro-Par 2015: Parallel Processing Workshops: Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers. Cham: Springer International Publishing, 2015, pp. 813–824.
[27]	R. Wu, Y. Chen, E. Blasch, B. Liu, G. Chen, and D. Shen, "A container-based elastic cloud architecture for real-time full-motion video (fmv) target tracking," in 2014 IEEE Applied Imagery Pattern Recognition Workshop (AIPR), Oct 2014, pp. 1–8.

4
Research Methodology

In order to answer the research questions, an experiment is carried out following the guidelines for reporting experiments in software engineering [34]. The experiment is executed in a controlled environment, where parts of the outcome are applied to a self-driving truck. All material for the experiment is available online[1].

4.1 Goals

The aim of the experiment is to, through a sequence of controlled steps, systematically answer the two research questions. This is achieved through evaluating the scheduling precision and input-output performance of the experimental units while running in different execution environments with respect to the deployment context. Understanding how the respective execution environment (e.g running the experimental units in a Docker container with a real-time Linux kernel) influences scheduling precision and input-output performance will ultimately decide how deterministic - with respect to time - the system is. Uncovering how deterministic the system is will answer the research questions.

The goals of the experiment is described in table 4.1, where the experimental units, execution environment and system load is described further in this section.

Table 4.1: Experiment Goals

Goal	Description
1	Analyse the scheduling precision of the Pi Component (section 4.2.1) for the purpose of understanding how deterministic the system is with respect to the execution environment and system load.
2	Analyse the input and output performance of the Pi/IO Component (section 4.2.2) for the purpose of understanding how deterministic the system is with respect to the execution environment and system load.

The goals are set to identify the most deterministic execution environment, as it is crucial for CBSs to meets the specified time deadline requirements, i.e. to ensure that the CBS does not violate the designated time of its executed time-slices.

[1] https://github.com/docker-rt-research/experimental-material

4.1.1 Execution Environment

The execution environment describes the setting in which the execution of the experimental units take place. Specifically, the execution environment refers to all components that make up a complete system: the processors, operating system and so on. There are four execution environments used in this experiment. The execution environments are configured the same but with two main differences. The difference being (1) an alternation of two Linux kernels and (2) an alternation of the deployment context. In this case, the deployment context is running the experiment units natively on the target system or running the experimental units in a Docker container. The execution environments are precisely specified in table 4.2, where the configuration of the execution environment is later described in section 4.3.

Table 4.2: Execution Environments

Environment	Description
1	Running the experimental units natively on the target system running a vanilla Linux kernel.
2	Running the experimental units natively on the target system running a Linux kernel patched with RT_preempt.
3	Running the experimental units in a Docker container on the target system running a mainline vanilla Linux kernel.
4	Running the experimental units in a Docker container on the target system running a mainline Linux kernel patched with RT_preempt.

4.1.2 System Load

Submitting the target system to heavy load is required in order to (1) traverse as many code paths of the kernel as possible and (2) mimic the run-time load of a real-time system. Stress-ng [35] is a user-space application that interacts with many components of the kernel. It can spawn a number of worker threads that perform useless tasks in order to apply load to the target system. The type of worker thread can vary from processor load to disk intensive load. Applying heavy load to the target system enables the experiment to be executed in an environment that mimics a real-life scenario. One would expect a real-time system to consume 80% of the system's resources, leaving a 20% buffer for extreme circumstances to circumvent system overload.

The experiment is performed in two scenarios: applying no-load and apply high-load to the system. Having these two scenarios allows for a fair comparison when the experimental units are executed in a more realistic operational environment. Having

no load allows for the experimental units to be executed in isolated, controlled environment where a fair comparison can be made.

4.2 Experimental Units

The experimental units, codenamed Pi and Pi/IO, are realised by the open source middle-ware OpenDaVINCI. Using OpenDaVINCI as a development framework for the experimental units allows the experiment to mimic the middle-ware one can expect in a autonomous vehicle. The implementation of OpenDaVINCI has been altered to enable full measurement of the system during runtime. The measurement points are simply timestamps of when the application reaches specific points in the code base during run-time. The specific changes that were made to the OpenDaVINCI library are:

1. Included specific measurement points that captures the current time within OpenDaVINCI's source code.
2. The measurement of time is modified to nanosecond precision.
3. A function is included to write measurement data to a serial port.

4.2.1 Pi Component

The Pi Component is utilised for measuring scheduling precision in order to answer RQ1. The Pi Component is tasked with calculating the next digit of Pi until it reaches 80% of its' designated time-slice. The remaining 20% of the time-slice is spent sleeping. The rationale for why it is specifically 80% is for the ability to have a buffer that is reserved for sleeping. An alternative approach to the Pi Component was investigated. Namely, limiting the pi algorithm to only calculate x digits of pi instead of calculating pi for x amount of time. This approach made it difficult to predict whether the amount of pi digits calculated would occupy more than the specified timing requirement. The approach to limit the amount of pi calculations to 80% was chosen as it provides a more controlled environment. The algorithm that calculates pi, specifically the Leibniz formula [36], is implemented to simulate load on the system resources. In the actual development of a autonomous vehicle, the pi algorithm would be replaced with components which enable the self-driving capabilities of the vehicle. The source code for the Pi Component can be found in Appendix A.1, on page I.

4.2.2 Pi/IO Component

The Pi/IO Component is utilised for measuring input and output performance in order to answer RQ2. The component is an extension of the Pi Component with two additional factors. These factors are reading an image from a web camera (input) and storing the image to disk (output). Capturing the image and storing it will here on be referred to as input performance and output performance respectively. The code implementation of capturing an image from a camera is a direct copy from

the open-source project OpenDLV [37]. OpenDLV is open source software currently under development for the operation of an autonomous self-driving vehicle and is built upon OpenDaVINCI. Furthermore, the code implementation for storing an image to disk is also a direct copy from the OpenDLV project that is used for logging the system during runtime. In order to capture the specific performance measure of input and output, two measurement points are added to measure the overhead by capturing an image and storing it to disk. The source code for the Pi/IO Component can be found in Appendix A.2.

4.3 Experimental Material

In order for the experiment to begin, the target system needs to be prepared and configured with software. This section identifies the preparation of the target systems execution environment to ensure transparency and reproducibility of the experiment. This section introduces the hardware used in the experiment, the software environment and the runtime properties of the experimental units in respect to the deployment context.

4.3.1 Hardware Environment

The experiment presented in this report is executed on a AEC 6950[2] embedded personal computer manufactured by Aaeon. The AEC 6950 is an industrial computer marketed for data processing, machine control and fleet management applications [38]. The hardware specification for the target system can be found in table 4.3. The hardware used to capture measurement data from the Aaeon AEC 6950 is a Raspberry Pi SoC [32] computer Model 1 B+. The data is collected from the target system via RS-232 serial communication by using a serial to USB converter. The web camera used for the Pi/IO Component is a Logitech c930e.

Table 4.3: Target System Hardware Specification

Component	**Specification**
Processor	Intel Core i7 3517UE 1.7 GHz
Memory	4GB DDR3 1333/1600 SODIMM
Storage Device	2.5" SATA HDD x 1
Serial Interfaces	USB type A x 2 for USB 2.0 USB type A x 2 for USB 3.0 DB-9 x 2 for RS-232/422/485 x 2 DB-9 x 4 for RS-232 x 4 Isolated DB-9 x 2 for RS-232/422/485 x 2

[2]http://www.aaeon.com/en/p/fanless-embedded-computers-aec-6950/

4.3.2 Software Environment

The operating system used in the experiment is Ubuntu Server 14.04 Long Term Support. A long term supported version of Ubuntu Server is chosen due to its stability, performance and available features. The Linux 3.18.25 kernel is selected for the operating system due a number of factors involving performance and software dependencies. Namely, Docker requires a kernel version greater than 3.10, limiting a large number of potential kernels. Secondly, Docker currently ships with `aufs` as the default back-end storage driver. However, the `aufs` storage driver has become depreciated in the Linux kernel and is therefore no longer available in current versions. Other options for the Docker storage-driver include `devicemapper`, `zfs`, `btrfs` and `overlayfs`. The `devicemapper` driver is known to have significant performance issues and the storage drivers `zfs` and `btrfs` currently have stability issues, so they are all disqualified. The remaining choice for the storage driver boils down to `overlayfs`, which requires a kernel version greater than 3.18. At the time of writing, the latest RT_preempt patch for kernel 3.18 which is actively maintained and supported is 3.18.25-rt23. Therefore, the RT_preempt patch 3.18.25-rt23 and kernel 3.18.25-generic is selected for this experiment.

All software packages that are required for the experiment are installed via the Linux package tool. The precise versions for each software package installed is documented and replicated when perparing the Docker images.The installation of software packages and subsequent configuration is executed via a script to document the changes that are made to the target system and to ensure precise reproducibility.

4.3.2.1 Kernel Configuration

The vanilla Linux kernel 3.18.25-generic is downloaded from the official Ubuntu Kernel package archive [39]. In order prepare the real-time enabled kernel, a specific set of kernel configurations are applied during the kernel build process. The default configuration file (`.config`) for the vanilla kernel is extracted and used as a basis for configuring the real-time enabled kernel. The modifications made to the default configuration file for the real-time enabled kernel follows the guidelines from [40] and are specified in table 4.4.

Table 4.4: Kernel Configuration Modifications

Parameter	Setting	Description
PREEMPT_RT_FULL	*y*	Enable preempt-rt patch.
LOCK_DEBUGGING	*n*	Disable schedule debugging information.
HIGH_RES_TIMERS	*y*	Enable high resolution timers.
ACPI_DOCK	*n*	Disable ACPI management
ACPI_PROCESSOR	*n*	Disable processor power management

4.3.2.2 Docker Parameters

The back-end manager for controlling Docker containers is performed by the `docker daemon`. The `docker daemon` is in charge of setting up the software environment for the containers as well as managing and controlling the container's resources. As mentioned in section 4.3.2, the `docker daemon` configuration file is changed to persistently use the `overlayfs` storage driver.

In order to build Docker images, a recipe in the form of a `DockerFile` is used when executing the `docker build` command line tool. The recipe contains instructions that construct the image, where each instruction that is executed is stored in a new subsequent storage layer. The relationship between a Docker image and container is similar to the relationship between a program and process. The runtime execution of a Docker image is a container, where any changes made in the container do not affect the parent image. Rather, changes that are made are stored within the container itself. The Docker images built for the experiment are all based on the official Ubuntu 14.04 LTS Docker image. The Ubuntu 14.04 LTS image is chosen to ensure a consistent environment between the native execution of the experimental units and that of the Docker containers. The software packages installed in the Docker images are all of equal versions to the packages that exist on the host.

To run a image the Docker `run` command is used which instantiates a container from the image. The experimental units, run from within a image, are executed with real-time scheduling priority 49. The runtime properties for containers that execute the experimental units are specified in table 4.5.

Table 4.5: Docker Image Runtime Properties

Parameter	Description
`-d`	Run the container in detached mode.
`-net=host`	The networking configuration is derived from the host.
`-cap-add=sys_nice`	Allow access to devices such as the web camera and the serial port.
`-v`	Mount shared filesystems from the host into the container.
`-w`	The working directory to execute the experimental units.

4.3.2.3 Native Execution Parameters

The native execution of the experimental units is performed by running executable binary files in a `screen` [41] session. The experimental units are executed with real-time scheduling priority 49. Linux screen is a full-screen window manager that, when called, creates a single window with a interactive shell session in it that can be used to execute programs. Multiple windows can be created, destroyed, detached and activated. Screen is used for the native execution for being able to run processes in a shell session that can be detached.

4.3.2.4 Stress Parameters

When applying load to the target system, two processor worker threads are started that calculate pi with a set load of 80%. Since the experimental units are run with real-time scheduling priority 49, the stress worker threads a set at priority 48 to ensure the experimental units are not pre-empted by the operating systems' scheduler. Starting the stress worker threads is always executed using `screen` shell session regardless of the deployment context of a particular experiment run-scenario. The precise execution of the `stress-ng` application is:

```
stress-ng -cpu 2 -cpu-load 80 -cpu-method pi
-sched fifo -sched-prio 48
```

4.4 Tasks

The experimental units are executed in different execution environments with respect to the deployment context. The experimental units are tasked with capturing specified measurement points and sending the measurement data via serial communication to the data capturer where they are stored. A description of the experimental units is found in section 4.2 and the specific measurement points that are captured within these units are found in section 4.5.1.

4.5 Variables and Hypotheses

Two types of variables are defined for the experiment: dependent and independent variables. The variables are analysed to evaluate the hypotheses of the experiment which are defined in this section. The experiment is run with an alternation between the treatment variables presented in section 4.5.2. Table 4.6 presents the four separate execution environments. Each of the execution environment are run with load and without load where running the application with load is the main source for analysis as it is of highest interest to understand how the execution environments behave when load is introduced.

Table 4.6: Execution environment

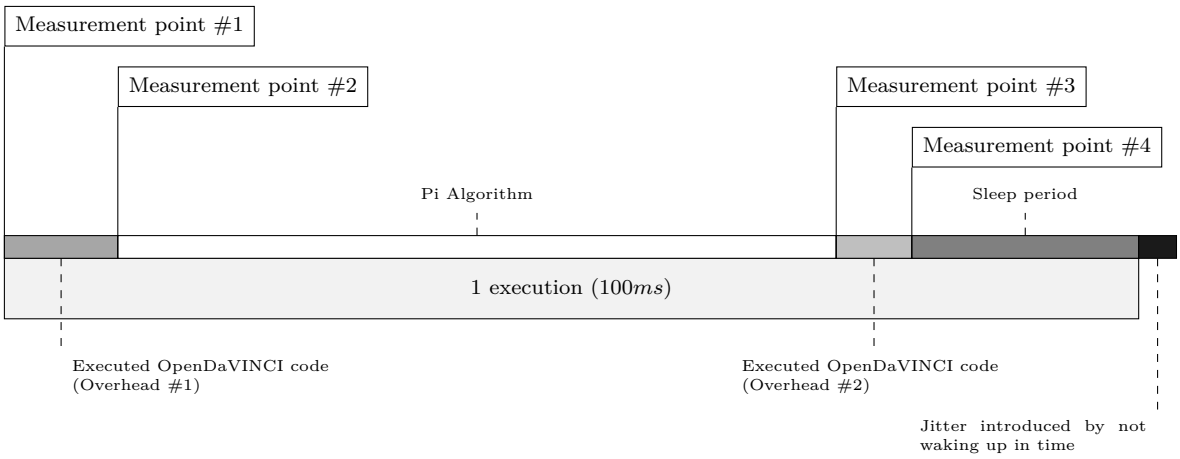
scenario	deployment	kernel	N with load	N without load
1	native	generic	5	5
2	native	rt	5	5
3	docker	generic	5	5
4	docker	rt	5	5

4.5.1 Dependent Variables

This research seeks to answer the two research questions which relates to the performance impact of the treatments on 1) scheduling precision^{RQ1}, 2) input performance^{RQ2},

and 3) output performance^{RQ2}. These three aggregated variables each consists of a set of dependent variables that construct the three dependent variables. Scheduling precision refers to how accurately, in terms of time, the system performs the necessary actions within a specified time-slice. The captured data points for measuring scheduling precision is the duration with nanosecond accuracy between four measurement points for the Pi Component. The location of the four measurement points within a time-slice is depicted in figure 4.1, which represent the Pi Components complete time-slice. The duration between measurement points one and two, and between measurement points three and four reveal the overhead of the middleware OpenDaVINCI. The duration between measurement points four and one (the next time slice) reveals the sleep precision of the Pi Component with respect to the execution environment.

Figure 4.1: Measurement points for Pi scheduling precision.

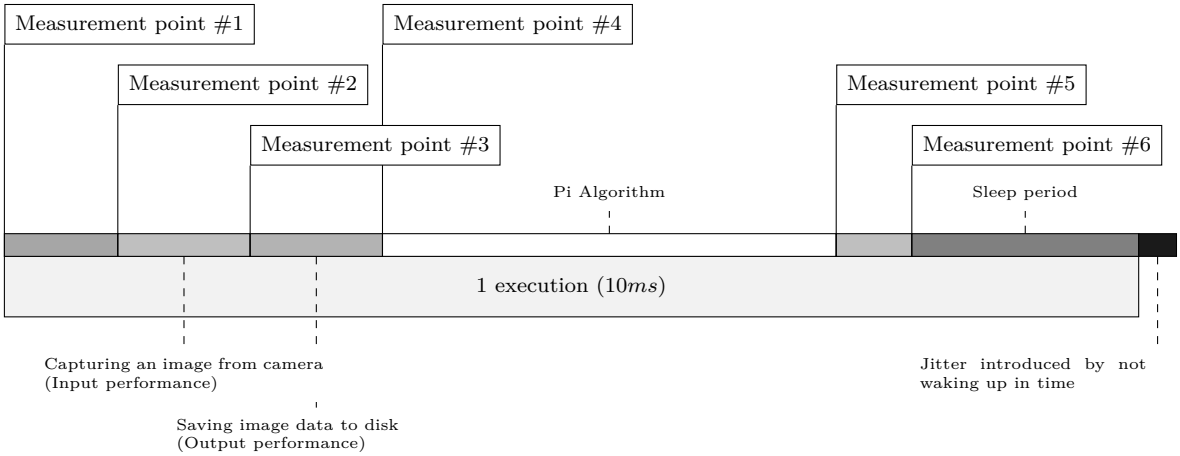


The Pi/IO Component is an extension of the Pi Component with the addition of two measurement points for measuring camera performance and disk write performance. The duration between measurement points two and three reveal the input performance of the Pi/IO component with respect to the execution environment. The duration between measurement points three and four reveals the output performance of saving an image to disk. The duration between measurement point two and four depict the total input-output performance of the executed time-slice. The remaining measurement points are the same as used in the Pi Component which are not analysed for answering research question 2.

4.5.2 Independent Variables

The treatments that are used for assessing the impact are factors specific to the execution environment, namely: 1) the Linux kernel (a vanilla kernel or RT_preempt patched kernel), 2) the deployment context (Docker or native) and 3) system load (no load or high load). Where the alternation of the two former treatments define the execution environment. It is of particular interest for this research to understand the impact of the second treatment, which is the deployment context. Deployment

Figure 4.2: Measurement points for IO performance.



context refers to how the application is executed, whether natively directly in the target machine's operating system, or if it is executed within a Docker container. The research questions are answered by analysing how each of the dependent variables are impacting the independent variables during runtime. Each of the treatments are stored as binary values ($1 : true/0 : false$) in the data table (table 4.8). The true and false value is set depending on which scenario the measurement points are collected from. Table 4.7 presents the binary alternation between the data scenarios The control group is having each of the treatment variables set to 0.

Table 4.7: Treatment alternation

	0	**1**
Deployment Context	Native	Docker
Kernel	Vanilla	RT_preempt
Load	No Load	CPU Load

4.5.3 Hypotheses

The dependent variables are analysed to evaluate the hypotheses of the experiment. The alternative hypotheses are stated below, denoted H_{1i_j}. The i corresponds to the goal identifier of the experiment where j is a counter when more than one hypothesis is formulated per goal.

The deployment context is executing the experimental units on the target system natively or within a Docker container. The execution environment is an alternating of the independent variables. Further details on the execution environments are specified in 4.1.1.

The hypotheses to achieve goal one of the experiment are:

H_{11_1}	The deployment context has an impact on scheduling precision.
H_{11_2}	The Linux kernel has an impact on scheduling precision.
H_{11_3}	The execution environment has an impact on scheduling precision.

The hypotheses to achieve goal two of the experiment are:

H_{12_1}	The deployment context has an impact on input performance.
H_{12_2}	The Linux kernel has an impact on input performance.
H_{12_3}	The execution environment has an impact on input performance.
H_{12_4}	The deployment context has an impact on output performance.
H_{12_5}	The Linux kernel has an impact on output performance.
H_{12_6}	The execution environment has an impact on output performance.

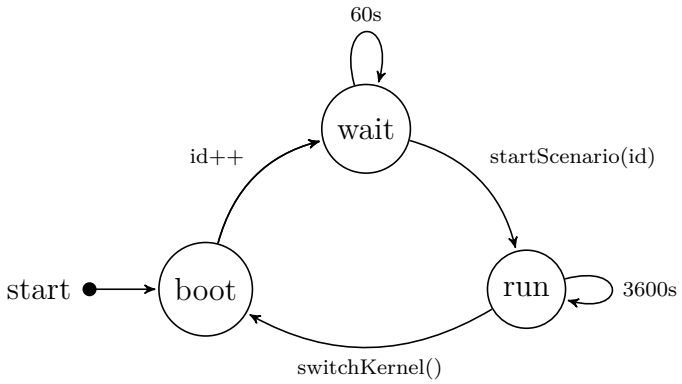
4.6 Design

In this controlled experiment a Quasi experiment design is used. A Quasi experiment design is used when a randomized experiment or when random assignment of study units to experimental groups is not possible [42]. Random assignment is not possible in this experiment since the experiment is a performance evaluation of the experiment units, being the Pi and Pi/IO components, with respect to the applied treatment. This study uncovers the cause and effect relationship between treatment and outcome.

4.7 Procedure

The overall procedure of the experiment is to execute the experimental units in 8 different scenarios. Each scenario is run for 3 600s with an execution frequency of 100 executions per second for the Pi Component experimental unit and 10 executions per second for the Pi/IO Component experimental unit. The scenarios consist of alternating two Linux kernels (vanilla and RT_preempt patched kernel), alternating two deployment contexts (executing in a Docker container or natively) and alternating between applying load and applying no-load to the target system. Executing the experiment run for each scenario is controlled from a script. The script is started automatically when booting the system and is responsible for selecting a scenario, configuring the system for that particular scenario and executing the experimental unit. The experimental units are executed five times per scenario, totalling 80 runs for the entire experiment: $2_e \times 2_d \times 2_k \times 2_l \times 5$ where $_e$ represents the two experiment units, $_d$ represents the deployment context, $_k$ represents the two kernels and $_l$ represents the two types of load. An overview of the procedure carried out by the execution script is depicted in figure 4.3.

Figure 4.3: State Machine Diagram of the Execution Script



4.8 Analysis Procedure

The dependent variables are the duration between time measurement points within all executed time-slices using a measurement unit of `nanoseconds`. There is a total of 14 399 960 data-points for the Pi Component experimental unit and 1 439 960 data-points for the Pi/IO Component experimental unit. 14 399 960 is derived from the experiment unit Pi Component running with 100 executions per second, 3 600 seconds per scenario, 8 scenarios, each scenario run 5 times. For each scenario the first time slice is trimmed of to allow for one execution before measuring thus resulting in 14 399 960 data points instead of 14 400 000 data points. The procedure for processing the raw data captured from the target system, is achieved using a script[3]. The script calculates the duration between two time-stamps using the equation shown in 4.1 and stores the specific duration into a comma-separated values (*.csv*) file together with the connected execution environment variables for the specific scenario run. The *.csv* table structure and data types for each of the experimental units is presented in table 4.8. All the results, data, and R scripts are available online[4].

$$duration = t_n - t_{n-1} \tag{4.1}$$

[3]https://github.com/docker-rt-research/experimental-material
[4]https://github.com/docker-rt-research/experimental-material/tree/master/data_analysis

Table 4.8: CSV table structure for both experimental units

Pi Component		Pi/IO Component	
Column names	**Measurement unit**	**Column names**	**Measurement unit**
deployment_context	`binary`	deployment_context	`binary`
kernel	`binary`	kernel	`binary`
load	`binary`	load	`binary`
overhead_1	`nanoseconds`	overhead_1	`nanoseconds`
pi_calculation	`nanoseconds`	camera_input	`nanoseconds`
overhead_2	`nanoseconds`	disk_output	`nanoseconds`
sleep	`nanoseconds`	pi_calculation	`nanoseconds`
		overhead_2	`nanoseconds`
		sleep	`nanoseconds`

Each of the subsections of section 5 begins by presenting the gathered descriptive statistics, that is intended for clarifying the variables used by the statistical analyses to answer the hypotheses. Bar charts are constructed to present an overview of the result. This structure is the same between the two experimental units.

To address research question 1 a three-way multivariate analysis (MANOVA) is conducted to understand the cause and effect relationship between the three treatments: 1) deployment context; 2) kernel; 3) load, which construct the execution environment and the four measurement points collectively referred to as the scheduling precision. The MANOVA provides P-values which are used to address the hypotheses connected to research question 1. The P-value explains if there exists a cause and effect relationship between the variables, however it does not disclose what that relationship is. To further understand what impact the treatments have on the dependent variables an η^2 value is calculated to reveal how large the impact is. Finally the cause and effect relationship is explained by analysing the coefficients between the treatments and dependent variables.

The second research question addresses two variables separately, namely input performance and output performance. Similar to the analysis procedure for research question 1, the second question seeks to understand the impact of the independent variables on the dependent variables. However the second research question have two separate dependent variables which are analysed separately to answer the first research question. Thus an three-way analysis of variance (ANOVA) is conducted for each of the dependent variable to understand the impact between the treatment and independent variables. The ANOVA will reveal if there exists a cause effect relationship between the treatments on the dependent variables of input and output performance separately. Lastly the relationship is analysed through calculated η^2 values which will disclose how large the impact is.

5

Results and Data Analysis

The conducted controlled experiments results in a series of datasets which are used to answer the research questions separately. This section seek to present the statistical analysis of all data captured. Each of the components are presented separately for the research question they individually seek to answer. Section 5.1 presents the processed data intended for answering research question 1 and its corresponding hypotheses. Section 5.2 aims to answer the second research question and its corresponding hypotheses. Initially the hypotheses will be answered by the conducted methods where further in depth analysis will be presented to uncover details not answered by the hypotheses. Throughout this chapter the three treatments will be presented in the tables and charts. These treatments are: 1) *Deployment Context* – running the experimental unit in Docker or natively; 2) *Kernel* – running the experimental unit using a generic kernel or an RT enabled kernel; 3) *Load* – running the experimental unit with simulated CPU load or without load. To extend the understanding of the impact between the treatments and dependent variables, an effect size (η^2) is presented. Cohen's D suggests that a large effect is attained when the η^2 value is above 0.80, medium above 0.50, and small above 0.20 [43].

5.1 Pi Component

The Pi Component aims to answer the first research question with a series of data points extracted during the experiment. The data presented in this section is extracted by running the experimental unit in $100hz$.

5.1.1 Descriptive Statistics

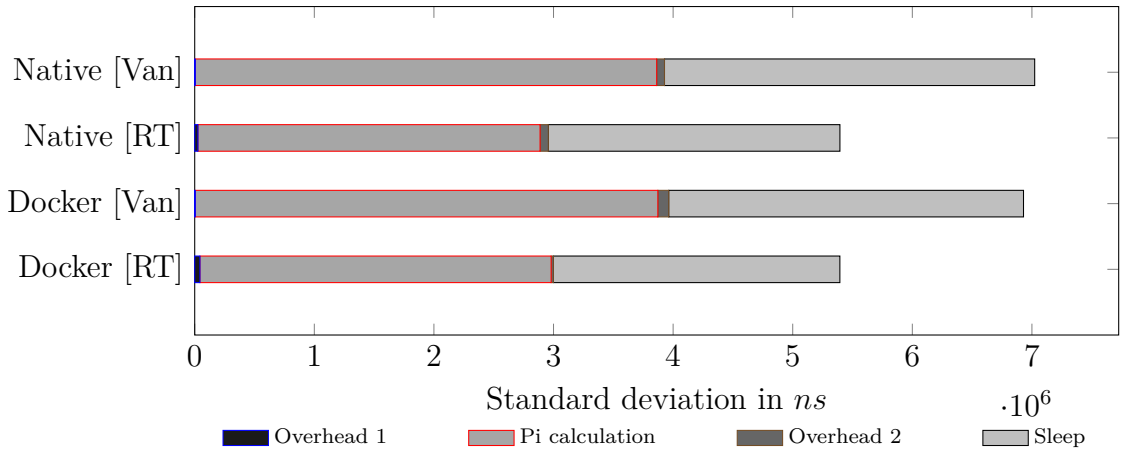
This section provides an overview of the collected data used for answering the hypotheses and research question 1. Table 5.1 presents the variables used for the statistical analysis conducted with their mean and standard deviation. `N` presents the amount of sample data collected. The dependent variables are categorical variables thus the lack of mean and standard deviation. Table 5.1 displays all standard deviations between the execution environments with load, while table 5.2 displays the same data however the execution environments are running on a system without load. It is worth noting that the x axis for both the charts have scientific notations in the bottom right corner. E.g. the execution environment of running the application natively on a vanilla kernel has a *Std.dev.* of $\sim 7\ 000\ 000ns$, more specifically where the sleep *Std.dev.* is $\sim 3\ 000\ 000ns$ and the pi calculation *Std.dev.* is $\sim 4\ 000\ 000ns$.

While the same environment without load has a *Std.dev.* of $\sim$ 16 000ns. A lower standard deviation imply a more deterministic execution environment. The charts indicate that the most deterministic environment is running the application on an RT kernel in Docker or natively. Comparing `Docker` with `Native` a similar result is shown between the two deployment contexts, whereas a large difference between the two load type groups is seen where `No Load` presents a very low standard deviation in comparison with running the application with `CPU Load`.

Table 5.1: Descriptive Statistics

Type of variable	Variable	N	Mean	Std.dev. (ns)	Scale
Independent variables	Deployment Context	14 399 960	N/A	N/A	Nominal
	Kernel	14 399 960	N/A	N/A	Nominal
	Load	14 399 960	N/A	N/A	Nominal
Dependent variables	Overhead #1	14 399 960	23 562	19 362	Ratio
	Pi Algorithm	14 399 960	8 157 746	2 419 989	Ratio
	Overhead #2	14 399 960	29 664	48 484	Ratio
	Sleep	14 399 960	2 004 799	1 960 327	Ratio

Figure 5.1: Std.dev. of execution environments with load *(lower is better)*



In figure 5.3 the scheduling precision for each respective execution environment is presented. The calculations are made by summing each of the dependent variables and dividing by the sum of all expected time-slice durations. The process is the application running at 100hz for 1 hour results in 3 600$s \times hz \times 5_{iterations}$ = 1 800 000$time\text{-}slices$ where each time-slice is 0.01s. This calculation results in a summed expected time-slice duration of 18 000s. The black dotted line seen in figure 5.3 displays the time deadline of a time-slice. Figure 5.3 shows that data extracted from running the application on a Vanilla kernel in both Docker and Natively violates the time-slice with an overall average violation of $\sim$ 10%. Where as figure 5.4 presents the same data for the environment contexts running on a system without load.

Figure 5.2: Std. dev. of execution environments without load *(lower is better)*

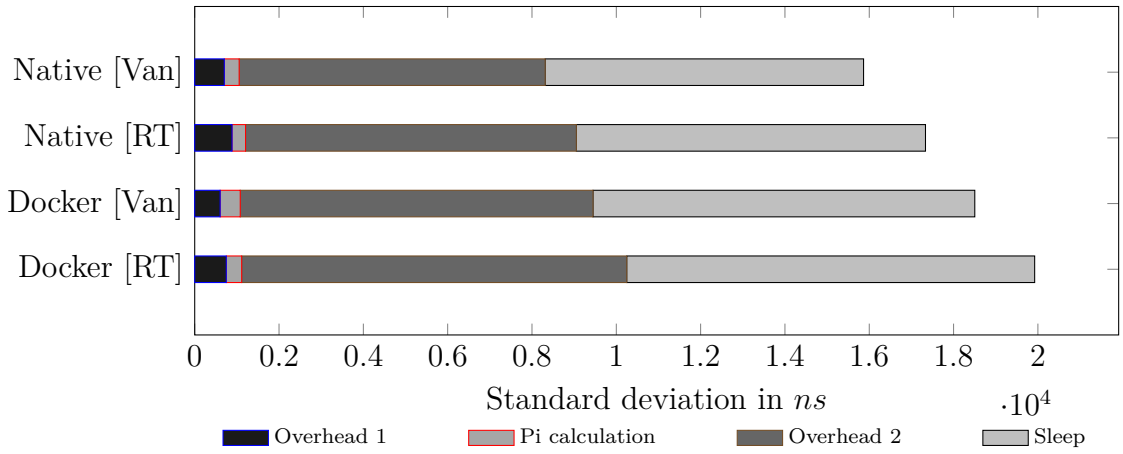
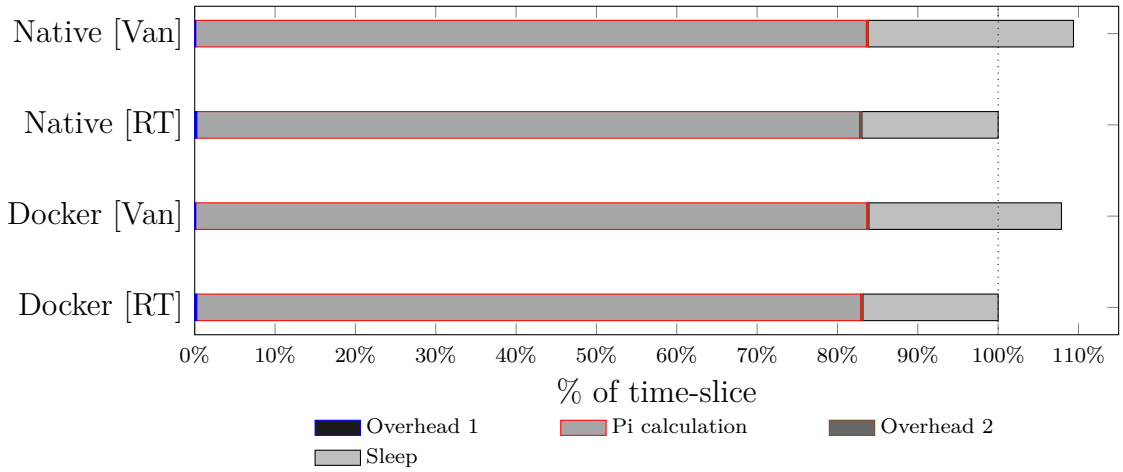


Figure 5.3: Execution environment mean consumed of time-slice with load *(closer to 100% is better)*



5.1.2 Hypothesis Testing

Table 5.2 presents the resulting P-values $Pr(>F)$ of the conducted MANOVA. Scheduling precision is referenced as all the collected dependent variables. The P-value gathered for each of the hypotheses is far below our significance level of $\alpha = 0.001$ and thus showing a significant impact on the scheduling precision from all of the treatments and rejecting the null hypothesis. To better understand what that impact is, an effect size value has been extracted. Table 5.3 is the result of running the MANOVA and a η^2 measurement on the data. While the P-value for all of the treatments indicate a significant impact on the dependent variables, the η^2, Pillai's trace, and Wilks λ indicate that there is a difference between the treatments impact on the dependent variables. As *deployment context* has a smaller value compared to the other treatments which suggests that the deployment context does not impact the scheduling precision to the same extent as the other two treatments.

Figure 5.4: Execution environment mean consumed of time-slice without load *(closer to 100% is better)*

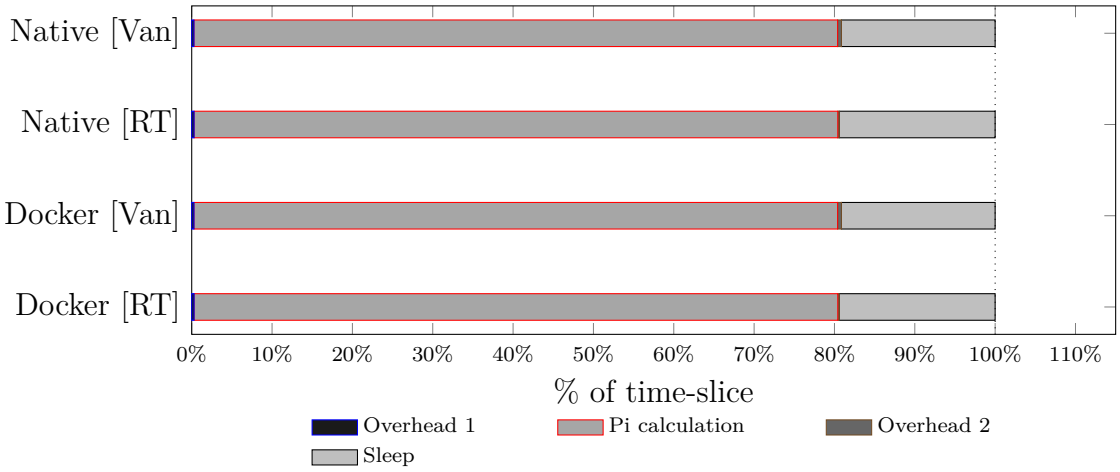


Table 5.2: Hypothesis results

	Hypothesis	**Pr(>F)**
H_{11_1}	Scheduling Precision ← Deployment Context	< 2.2e-16
H_{11_2}	Scheduling Precision ← Kernel	< 2.2e-16
H_{11_3}	Scheduling Precision ← Execution environment	< 2.2e-16

Table 5.3: MANOVA and Effect Size

	Df	**Pillai**	**Wilks**	**approx F**	**num Df**	**den Df**	**Pr(>F)**	η^2
deployment context	1	0.000	1.000	528	4	14 399 952	< 2.2e-16	0.0001
kernel	1	0.068	0.938	261 770	4	14 399 952	< 2.2e-16	0.0678
load	1	0.072	0.936	277 812	4	14 399 952	< 2.2e-16	0.0716
deployment context:kernel	1	0.000	1.000	370	4	14 399 952	< 2.2e-16	0.0001
Residuals	14 399 955							

Table 5.4: Coefficient between treatment and dependent variable (ns)

	Overhead #1		**Pi Algorithm**		**Overhead #2**		**Sleep**	
(Intercept)	24 675		8 034 093		38 997		2 155 372	
deployment context	119	(0,005)	2 121	(0,000)	987	(0,025)	-77 961	(-0,036)
kernel	7 358	(0,298)	-48 434	(-0,006)	-13 056	(-0,335)	-414 775	(-0,192)
load	-9 644	(-0,391)	291 274	(0,036)	-6 159	(-0,158)	156 070	(0,072)
deployment context:kernel	-118	(-0,005)	4 689	(0,001)	-877	(-0,022)	71 039	(0,033)

The MANOVA tests suggests that *deployment context* and the full *execution environment* has a smaller impact on scheduling precision compared to the other separate independent variables. This is further evident when analysing the coefficients captured. Table 5.4 displays the coefficients of each treatment onto each of the dependent variables. Intercept refers to the control variable where each of the treatments are set to default, e.g. having the application running natively with the generic kernel and without load. The values provided below the intercept values display the difference introduced in each of the dependent variable when switching the treatment variable. The figure in parentheses depicts how many percentage from intercept the treatment affects the particular independent variable. In table 5.4 it can be noted that the values displayed for the coefficients related to *deployment context* show a far smaller impact on the dependent variables which is inline with the η^2 value displayed in 5.3.

5.2 Pi/IO Component

This section provides an overview of the collected data and hypothesis test results for answering research question 2. The data presented in this section is extracted by running the experimental unit in $10hz$.

5.2.1 Descriptive Statistics

Table 5.5 provides an overview of the data gathered and implemented for understanding the impact of the execution environment and its treatments on the camera performance and disk performance. `N` presents the amount of sample data collected. The independent variables are the same as Pi Component however the dependent variables are focused on the camera and disk durations.

Table 5.5: Descriptive Statistics

Type of variable	**Variable**	**N**	**Mean**	**Std.dev.** (ns)	**Scale**
Independent variables	Deployment context	1 439 960	N/A	N/A	Nominal
	Kernel	1 439 960	N/A	N/A	Nominal
	Load Type	1 439 960	N/A	N/A	Nominal
Dependent variables	Camera Performance	1 439 960	6 023 777	3 178 647	Ratio
	Disk Performance	1 439 960	3 553 669	13 404 796	Ratio

Figure 5.5 presents the *Std.dev.* for each of the two dependent variables, namely input performance and output performance. Note that the x axis for both the charts have scientific notations in the bottom right corner indicating a multiplier to the value on the x axis. The figure show that the application running on a vanilla kernel results in more deterministic camera performance where the time it takes to capture an image fluctuate less. The graph further shows that the execution environment running the application in Docker utilising the RT kernel has a more

deterministic camera performance in comparison with the native equivalent. Lastly, the graph shows that the disk performance is not impacted by switching execution environment as the dark blue bar is consistent in size over each of the environments.

Figure 5.5: Std. dev. of execution environments with load *(lower is better)*

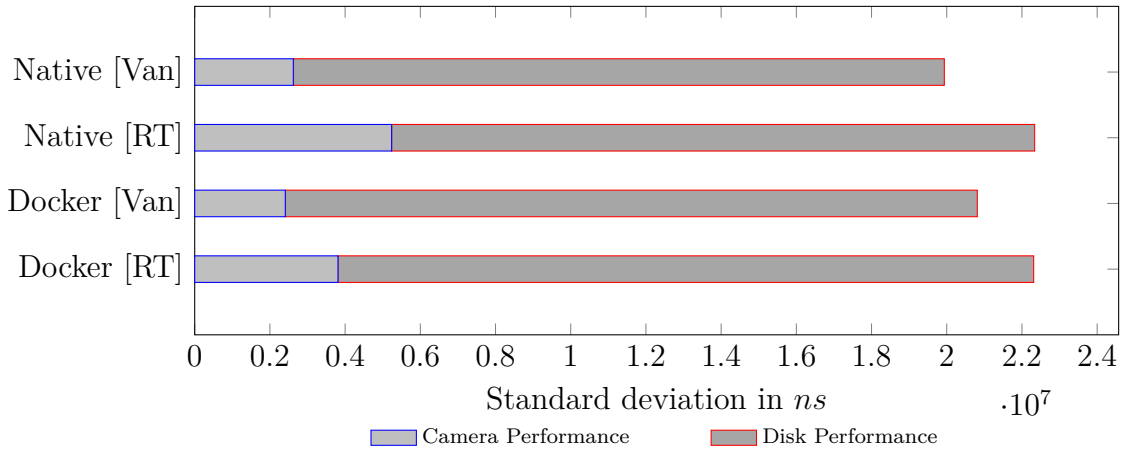
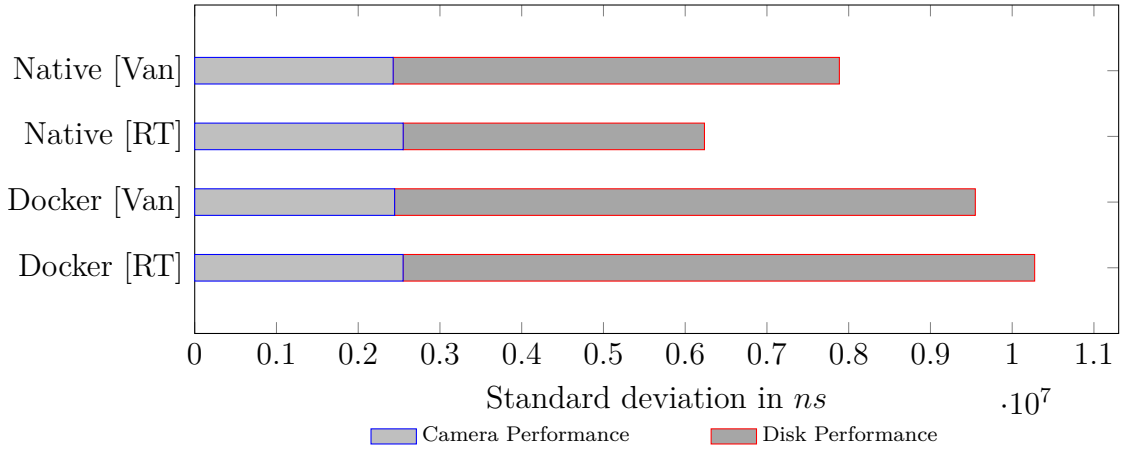


Figure 5.6 depicts the standard deviation of the input and output performance of each of the execution environments running on a system without load. The figure shows that the most deterministic execution environment is running the application natively utilising the RT kernel. The figure further displays that the worst deterministic execution environment for a no-load system is the Docker equivalent running with the RT kernel. This result show that docker has an impact on disk performance as seen by the dark blue bars being larger for both of the environments utilising Docker as its deployment context.

Figure 5.6: Std. dev. of execution environments without load *(lower is better)*



The input output performance results depicted in figure 5.7 show that the execution environments utilising the RT kernel impacts the camera and disk performance negatively, increasing the occupation time for performing the input and output tasks. This is seen by the larger size of both dependent variables spanning over a larger space of the time-slice.

Figure 5.7: Execution environment mean consumed of time-slice with load *(lower is better)*

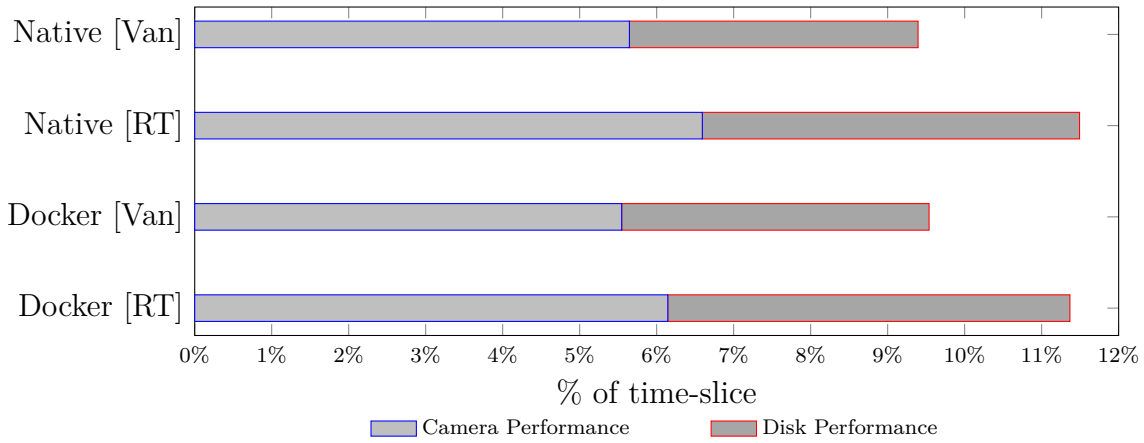
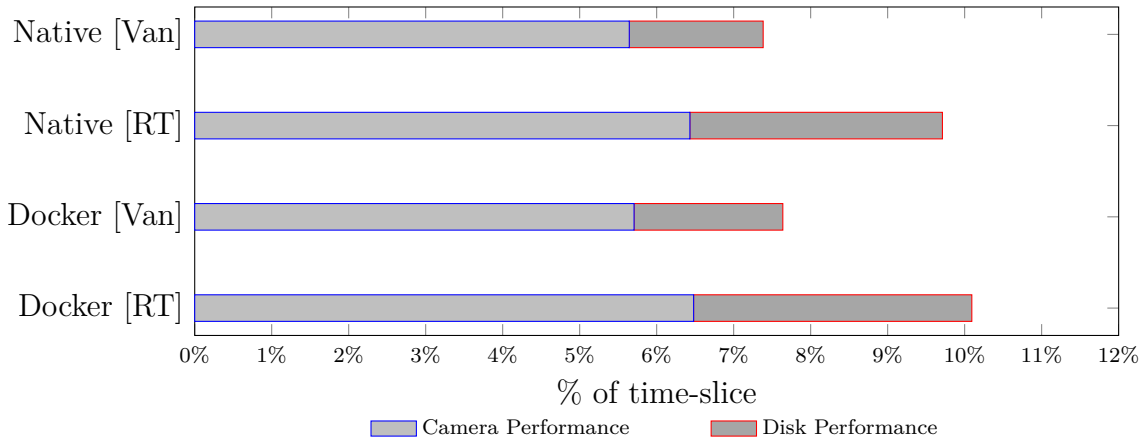


Figure 5.8: Execution environment mean consumed of time-slice without load *(lower is better)*



5.2.2 Hypothesis Testing

An three-way analysis of variance (ANOVA) is performed for each of the dependent variables to understand if there exists a cause and effect relationship between the independent variables and the dependent variables. Table 5.6 presents the results of the ANOVA test performed, displaying that each of the treatments has a significant impact on the dependent variables. The resulted P-value is far below the chosen $\alpha = 0.001$ which rejects the null hypothesis and report that there is significant impact on the camera performance and disk performance.

The η^2 and F values of the treatment groups in table 5.7 show that camera performance is mostly impacted on the chosen kernel. While load has a far lower effect on camera performance in comparison with the kernel.

Table 5.6: Hypothesis results

	Hypothesis	Pr(>F)
H_{12_1}	Camera Performance ← Deployment Context	< 2.2e-16
H_{12_2}	Camera Performance ← Kernel	< 2.2e-16
H_{12_3}	Camera Performance ← Execution environment	< 2.2e-16
H_{12_4}	Disk Performance ← Deployment Context	< 2.2e-16
H_{12_5}	Disk Performance ← Kernel	< 2.2e-16
H_{12_6}	Disk Performance ← Execution environment	0.00074

Table 5.7: ANOVA results Camera performance

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	Partial η^2
deployment context	1	4,329E+15	4,329E+15	435,65	<2E-16	0.0003
kernel	1	2,172E+17	2,172E+17	21 858,63	<2E-16	0.0150
load	1	2,487E+15	2,487E+15	250,24	<2E-16	0.0002
deployment:kernel	1	2,946E+15	2,946E+15	296,46	<2E-16	0.0002
Residuals	1 439 952	1,431E+19	9,938E+12			

Table 5.8: ANOVA results Disk performance

	Df	Sum Sq	Mean Sq	F value	Pr(>F)	Partial η^2
deployment	1	2,692E+16	2,692E+16	150,95	<2E-16	0.0001
kernel	1	7,063E+17	7,063E+17	3 960,47	<2E-16	0.0027
load	1	1,199E+18	1,199E+18	6 721,91	<2E-16	0.0046
deployment:kernel	1	1,090E+15	1,090E+15	6,11	0,0134	0.0000
Residuals	1 439 952	2,568E+20	1,783E+14			

The results from the conducted ANOVA depicted in table 5.8 suggests that the effect deployment context has on disk performance is below the effect of kernel and load type. This is seen on the η^2 and F values of the treatment groups.

Table 5.9 displays the coefficients of each treatment onto camera performance and disk performance. Intercept refers to the control variable where each of the treatments are set to default, e.g. having the application running natively with the generic kernel and without load. The values provided below the intercept values display the difference introduced in each of the dependent variable when switching the treatment variable. The figure in the column to the right of the nanosecond coefficients presents the percentage the dependent variable deviates from the intercept when the treatment is alternated from the intercept treatment.

Table 5.9: Coefficient between treatment and dependent variable (ns)

	Camera		**Disk**	
(Intercept)	5644123		1738978,8	
deployment context	61 896	(0,011)	193 572	(0,111)
kernel	787 322	(0,139)	1 540 975	(0,886)
load	1 710	(0,000)	2 010 246	(1,156)
deployment:kernel	-13 590	(-0,002)	140 025	(0,081)

6
Self-Driving Truck

This section introduces the result of executing parts of the experiment on a self-driving Volvo truck, depicted in figure 6.1, that participated in the Grand Co-operative Driving Challenge (GCDC) 2016 in the Netherlands. The experiment conducted on the use-case is carried out with the intention to further understand how the scheduling precision of the CPS application is impacted by the deployment context in a system with an operational full-scale vehicular CPS running simultaneously. The use-case experiment will be run on a system with a real-time enabled kernel, as the outcome from the controlled experiment has shown that an execution environment utilising a Vanilla kernel carries undesirable overhead.

Figure 6.1: Chalmers Revere GCDC Truck

6.1 Method

This section is aimed at providing the method of the experiment conducted on the uncontrolled environment of the use-case scenario.

6.1.1 Experimental Units

A decision to exclusively run the Pi Component is made as the self-driving truck implements a networked camera setup and hence does not have the same camera implementation as the controlled target system (section 4.3.1). The Pi Component

presented in section 4.2 is executed on the target system identical to the one used during the controlled experiments. The computer is connected to hardware components mounted to a Volvo FH16 truck. As the use-case data is extracted using the Pi Component no measurements are done to understand the camera input and disk output performance for the use-case due to the priority of building an understanding of how the scheduling precision behaves with the runtime load of the self-driving truck.

To understand how Docker impacts the scheduling precision in the context of the use-case, the experimental unit is executed first natively and secondly within a Dockerized container. The runtime properties of Docker are presented in table 6.1. Docker version 1.11.1 is used and the Docker daemon is set to use the `overlayfs` storage driver.

Table 6.1: Docker Image Runtime Properties

Parameter	Description
`-d`	Run the container in detached mode.
`-net=host`	The networking configuration is derived from the host.
`-cap-add=sys_nice`	Allow access to devices such as the web camera and the serial port.
`-v`	Mount shared filesystems from the host into the container.
`-w`	The working directory to execute the experimental units.

6.1.2 System Load

The major difference in comparison to the controlled experiment is that all software components required for the Volvo truck to operate in a driver-less manner is started and run in the background: this bring a more realistic operational load and consequently a less controlled environment in respect to the controlled experiment. Fifteen components are executed natively and within a Dockerized environment on the target system. These components are tasked with various responsibilities spanning from capturing satellite positioning data, reading data from the vehicle (e.g steering position), vehicle to vehicle communication and a number of camera operating components. Measurement points are captured via serial communication to measure the impact in a real use case.

6.1.3 Target System

Table 6.2 presents the hardware setup for the target machine upon which the experimental unit (section 6.1.1) is executed. The target system is running ArchLinux operating system with a real-time enabled kernel (RT_preempt), version 4.5.0-1. The decision to operate the system using ArchLinux is made by the REVERE development team due to their preferences and is not a variable which this research

can control. However, to ensure that the execution environment is in line with the outcome from the controlled experiment, an RT_preempt kernel is implemented.

Table 6.2: Target System Hardware Specification

Component	**Specification**
Processor	Intel Core i7 3517UE 1.7 GHz
Memory	4GB DDR3 1333/1600 SODIMM
Storage Device	2.5" SATA HDD x 1
Serial Interfaces	USB type A x 2 for USB 2.0 USB type A x 2 for USB 3.0 DB-9 x 2 for RS-232/422/485 x 2 DB-9 x 4 for RS-232 x 4 Isolated DB-9 x 2 for RS-232/422/485 x 2

6.1.4 Variables

The uncontrolled experiment is intended to understand the deployment context's impact on the scheduling precision. In comparison to the controlled environment the kernel is statically set to an RT_preempt kernel and thus not used as a treatment to the use-case experiment. The deployment context is a categorical variable which holds two values, namely 1) Native and 2) Docker. Where the former is the execution of the experimental unit on the host OS and the second is the execution of the experimental unit in a Dockerized container. As the use-case scenario is a replication of the controlled environment, the same dependent variables are used for measuring the scheduling precision of the experimental unit, namely measurement points: Overhead #1, Pi Algorithm, Overhead #2, and Sleep. Figure 4.1 presents the measurement points from where within the time-slice each of the dependent variables are extracted.

6.1.5 Procedure

The procedure is to execute the experimental unit in 2 different scenarios. The two scenarios is an alternation between two deployment contexts: executing natively on the host OS, and executing within a Docker environment. Each scenario is manually executed for $1h$ on the target machine with the Raspberry Pi SoC [32] hardware connected through a serial to USB device used for capturing measurement data via RS-232 serial communication.

6.1.6 Analysis Procedure

Analysing the data is made with a partly replicated analysis procedure used for understanding the scheduling precision of the controlled experiment. The purpose of running the uncontrolled experiment is to provide an insight to how the experimental unit's scheduling precision behave in an environment with realistic load.

Each of the measurement points presented in figure 4.1 is extracted and processed by calculating the duration between the points using a script. The durations is then analysed by building bar charts to illustrate the scheduling precision which is intended to provide an overview of all data points extracted. Furthermore, a table of sample size, means, and standard deviations is constructed to provide the descriptive statistics of the generated bar charts. Lastly, a one-way multivariate analysis (MANOVA) is conducted to understand how the deployment context impact the scheduling precision and an effect size (η^2) between the deployment context and scheduling precision is calculated to bring a more detailed understanding of what impact the deployment context has on the scheduling precision.

6.2 Results

This section presents the results extracted from the experiment conducted on the use-case of the self-driving truck in the Chalmers Revere Lab. The goal is to understand and provide an insight in how the deployment context impacts the experimental units within an environment which have realistic system load.

6.2.1 Descriptive Statistics

Table 6.3: Descriptive Statistics

Type of variable	**Variable**	**N**	**Mean**	**Std.dev. (ns)**	**Scale**
	Deployment Context	719 996	N/A	N/A	Nominal
Independent variables	Overhead #1	719 996	29 268	8 506	Ratio
Dependent variables	Pi Algorithm	719 996	8 018 460	5 452	Ratio
	Overhead #2	719 996	54 993	26 340	Ratio
	Sleep	719 996	1 897 278	31 649	Ratio

The results shown in figure 6.2 show that there exists a difference between the two deployment contexts in the load environment introduced by the other components running on the same system. The difference indicates that the deployment context has an impact on how deterministic a system is as native fluctuates roughly 40% less than the Docker deployment context. Running a statistical test to understand the cause and effect relationship between the deployment context and the dependent variables resulted in a $\eta^2 = 0.504$ which relates to a medium effect size between deployment context and the dependent variables according to Cohen's D ($\eta^2 > 0.5$). Figure 6.3 show that there exists no violations for the average time deadline for either of the two execution environments.

Figure 6.2: Standard deviation of execution environments with real-life load *(lower is better)*

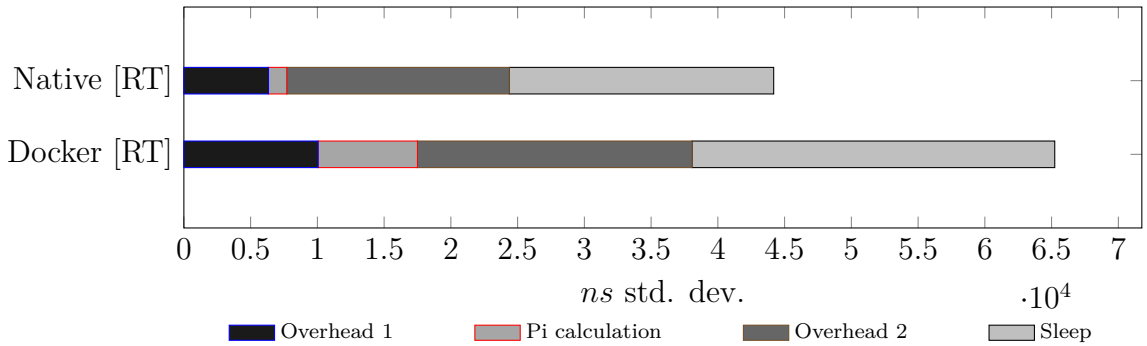
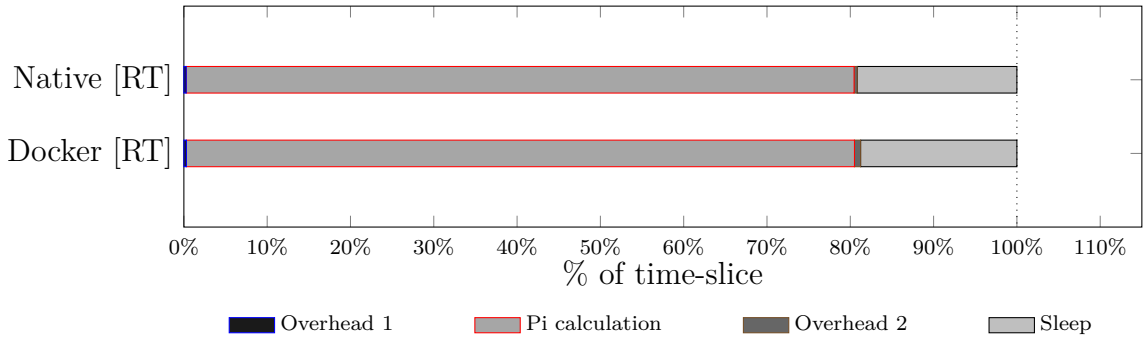


Figure 6.3: Mean consumed of time-slice with use-case load *(closer to 100% is better)*



6.2.2 Hypothesis Testing

Table 6.5 presents the resulted MANOVA and effect size calculated from the data derived from the uncontrolled experiment. As the P-value (Pr(>F)) is smaller than the chosen $\alpha = 0.001$ it indicate that the deployment context has a strong significant impact on the scheduling precision in the uncontrolled experiment. The η^2, Wilks λ, and Pillai's trace values all indicate that the effect of the deployment strategy is of medium size as the η^2 value is larger than the medium threshold provided by Cohen's D ($\eta^2 > 0.5$).

The coefficients between treatment and dependent variables are presented in table 6.6 to understand what the impact presented in table 6.5 of deployment context is. The right column for each dependent variable depicts the percentage of impact the deployment context has on each of the dependent variables. When executing the application inside a Docker container an increase of 102% in the duration of Overhead #2 is found.
As the uncontrolled experiment only has one of the three treatments used in the controlled experiment, only one of the three hypotheses is applicable. The P-value presented in table 6.5 indicates that H_{11_1} is significant and thus rejecting the null hypothesis.

Table 6.4: Hypothesis results

	Hypothesis	**Pr(>F)**
H_{11_1}	Scheduling Precision ← Deployment Context	< 2.2e-16

Table 6.5: MANOVA and Effect Size

	Df	**Pillai**	**Wilks**	**approx F**	**num Df**	**den Df**	**Pr(>F)**	η^2
deployment context	1	0.504	0.496	182 958	4	719 993	< 2.2e-16	0.5041
Residuals	719 996							

Table 6.6: Coefficient between treatment and dependent variable (ns)

	Overhead #1		**Pi Algorithm**		**Overhead #2**		**Sleep**	
(Intercept)	27 944		8 017 426		36 462		1 918 167	
deployment	2 648	(0,095)	2 067	(0,000)	37 062	(1,016)	-41 777	(-0,022)

7
Discussion

To fully comprehend the impact of utilising Docker for the deployment strategy of a self-driving vehicle the data analysis have provided data looking at many aspects of the environment in which the real-time system may be run. Simply understanding how an application performs while running inside a Docker container in comparison with running the same application natively on a system is not enough to provide a comprehensive understanding of whether or not Docker is suitable for a real-time system environment. Such an environment is comprised by more factors than the deployment context exclusively, factors such as choosing the correct kernel or mitigating unnecessary system load. The result has provided in depth understanding of how a real-time system behaves depending on whether there is load on the system or if the system is utilising a real-time enabled kernel. All null hypotheses for both research questions are rejected suggesting that there exists an cause effect relationship on applications performance with switching deployment contexts or kernels. However the data analysis show that although having significant impact, the effect sizes of each of the treatment groups were minimal in regards to Cohen's D. The results presented in all (M)ANOVA tables presents an η^2 value far below the threshold for a small effect size which brings further interesting information about the relationships.

Research question 1 asks whether the execution environment has an impact on the scheduling precision of a real-time application or not. The results suggests that the scheduling precision is impacted by the execution environment, however this impact is related to the chosen kernel for the execution environment and not the deployment context chosen. This is strengthened when analysing the differences in F-value, Pillai's trace value and η^2 provided in table 5.3. As seen, the values differ between deployment context and kernel which indicates that there is a smaller impact made by switching between running the application in Docker and natively in comparison with choosing the correct kernel for the execution environment. The charts presented in section 5.1.1 further discusses the importance choosing the correct kernel for the real-time application. This is seen in figure 5.3 where the two execution environments utilising a vanilla kernel has an average deadline violation of approximately 10% while comparing the Docker environments with the native environments show nearly no difference in scheduling precision.

Research question 2 aims to answer if there is an impact on input/output performance switching between execution environments. The results reveal that there is an impact however that impact is insignificant in relation to Cohen's D. Each of the η^2 values are far below than the threshold of having a small effect on the dependent

variables. An observation can be made on the η^2 for the impact kernel has on camera performance, which indicates that there exists a larger effect on camera performance switching between kernels in comparison with the other treatment variables. One interesting point can be made on the camera performance during load, which seems to perform better when there is load on the target system. Lastly, it is worth noting that the input and output performance is indicated to be negatively impacted by utilising the RT kernel, this could be the case where the preemptive scheduler works against the components used as the scheduler may preempt the system processes used for capturing the image and saving the data.

In relation to related work, the results of this study confirm that Docker carries near-native performance. However, it is important to use a real-time enabled kernel to meet the timing requirements and for Docker to achieve neglible overhead. This too confirms current literature, and enriches the current state of knowledge which requires further insight utilising a RT kernel for Docker.

Comparing the experiments conducted in a controlled environment with those extracted from the use case of the self-driving truck, it is evident that the load in the controlled environment is far more exhaustive than that of the realistic load. This is seen in the size of the *Std.dev.* between the different execution environments. The total *Std.dev.* presented by the simulated system load in the controlled environment had a maximum value of $\sim 7000000ns$ whereas the realistic load had a maximum total *Std.dev.* value of $\sim 65000ns$ which is approximately 0.9% of the controlled environment's value. While the figure and η^2 indicated that deployment context has an impact on the dependent variables for the use case of the self driving truck, this does not necessarily imply that Docker is not suitable as a deployment strategy for the use-case. It could simply indicate that Docker scales less efficiently when introducing load to the system where both execution environments may reach an equilibrium eventually when increasing the load. This equilibrium may be observed in figure 5.1 where both deployment contexts share roughly the same total standard deviation.

Lastly, during the project there was a process of porting the existing truck system from a native execution environment into a Dockerized execution environment. This experience was found to be far more simple than expected. The total time for transferring all software components into Dockerized containers was roughly 40 work hours, which argues for the minimal transferability cost of Docker. Therefore it is worth mentioning the porting experience for decision makers that are reluctant of porting their existing systems into a Dockerized environment due to possible transferability costs. The software quality benefits presented by Docker seems to surpass the performance and transfer costs involved in implementing Docker in an existing system. The benefits of a Docker implementation seem to span over all ISO software quality aspects. This is the experience of this research and the statements does not base in empirical findings but rather in the use case of the project. Further research is required to give a definite answer for how Docker impacts the quality factors of an existing system.

8

Threats to Validity

In this section, the four perspective of validity threats presented by Runeson and Höst [44] are discussed.

8.1 Construct Validity

Construct validity refers to the degree of which the findings and observations made in the study reflect what is actually investigated in respect to the research questions. Data is collected in order to understand if the execution environments and deployment context impact the timing behaviour of the sample applications (experimental units). The sample applications are realised with the OpenDaVINCI library by using a skeleton code for time-triggered components from the tutorial section of OpenDaVINCI's online documentation. Minor modifications were made to the overall structure of the sample applications code base. This ensure that the sample applications used in the experiment is correct and in-line with the design requirements for real-time systems. OpenDaVINCI is open source and has been used to realise a number of autonomous vehicles. This adds validity to the use of the library. Furthermore, the sample application Pi/IO Component, used for capturing images from a web camera and storing them to disk is composed of code inherited from the OpenDLV open source project. The adds further validity to the correctness of the applications used in the experiment. The data captured from the sample applications are simply timestamps which fire in specified parts of the code base. The durations between all measurement points are extracted and then compared over multiple runs in different execution and deployment contexts to uncover the impact.

8.2 Internal Validity

Internal validity refers to the extent to which an unknown factor has an effect on the factor under investigation or may refer to the issues that may affect the relationship between treatment and outcome. In order to mitigate the risk of an unknown factor interfering with the data being captured a number of strategies is used, namely:

1. During the experimental run, all execution environments are initiated through a set of scenario scripts. Every initiated scenario is automated to ensure that no external factors may impact the outcome of the data during run time. Having each scenario automated further strengthens the assurance that when each

scenario is re-initiated, the environment is a reproduction of the previous iteration of the same scenario. This mitigates any external factor interfering with the data being collected, however the scripts may include bugs not accounted for. To mitigate the risk of bugs manual validation runs were made with the intent to confirm that the execution environment is configured correctly as expected. Post validation was also carried out on inspecting the data collected which states the operating system configuration.
2. To mitigate any impact made on the performance on the system, the machine's network is disabled, as any traffic on the networking line may directly impact the operating system scheduler.
3. Data is captured through a serial port to bypass any performance impact made locally on the disk or through the USB kernel stack. The serial port is connected to an external device that handles the data capturing and processing it to a file. By writing all data through a serial port onto an external device any impact on the scheduler made by USB, memory, or disk is mitigated.

8.3 External Validity

External validity refers to how generalisable are the findings beyond the actual study, and to what extent are the findings of interest and relevance outside of the scope. Scheduling precisions and I/O performance are two important factors for a CPS and to systems that are highly sensitive to timing delays. The results of this study can be applied for any system sensitive to timing delays in respect to the execution environment, the hardware and the specific versions of software used. The hardware used in the experiment is consumer level hardware, so reproduction of the experiment is viable and a relation to the results can easily be made.

8.4 Conclusion Validity

Conclusion validity refers to validity of the measured effect of the treatments used in the experiment. There exists two apparent threats to conclusion validity, namely Type I and Type II statistical errors. Due to the samples size of this research the threat of Type I error is considerable. For the statistical phenomena where an increasing sample size will result in a decreasing P-value. This research have thus put less weight on the P-value results as there exists a large number of data samples for the hypothesis testing [45] which will result in highly significant results. This requires a more extensive analysis of the results, taking larger consideration on the effect size while evaluating the data.

9

Conclusion & Future Work

This research sought to understand the impact Docker brings when utilised in the deployment strategy for systems which enable self-driving capabilities in vehicles. The research was conducted in collaboration with research project held in collaboration between Chalmers and SAFER who are exploring the development of a self-driving truck. The results from this research acts as support in the decision making process of using Docker containers for software deployment for further software development on the self-driving truck project. Previous literature [21, 31, 23, 24, 26, 30] show promising results for containers as a deployment platform as their presented results show negligible overhead introduced by utilising Docker in the execution environment of the applications explored. However, previous research has not been able to build a case for the decision making specific for the domain of self-driving vehicles where the domain stresses the real-time requirements of vehicular cyber-physical systems. While the research conducted in [22] have explored how Docker behaves using cyclic test in a cloud computing environment with real-time requirements, it has not explored how Docker behaves in the context of a complete cyber-physical system. This is the gap this research has explored.

The findings from this research are in-line with previous literature and show that Docker introduces negligible overhead to the scheduling precision, input and output performance to the cyber-physical system in both the controlled environment as well as the environment of the self-driving truck at Chalmers Revere lab. Furthermore, the results convey that choosing a correct *kernel* for the system carry heavier importance in comparison to choosing Docker or native execution for the deployment strategy. As the results show that the kernel type has a larger impact on the scheduling precision. The results show a considerably more deterministic environment when utilising an RT_preempt kernel in comparison to a vanilla kernel. This argues that Docker is not the predominant factor impacting the scheduling or IO performance for when designing the correct execution environment for a self-driving vehicle.

This research has focused on the specific context of implementing Docker as a deployment strategy for self-driving vehicles. The design of this research has aimed to find an understanding of how Docker impacts scheduling precision and camera and disk performance for the contexts used. Analysis has been carried out to understand how these performance factors behave while switching between different treatments, such as system load and kernel version. The performance factors scheduling precision and camera input and disk output performance build a foundation of understand-

ing the impact a Docker implementation has on real-time systems, however, there exists other factors that are of interest to investigate. There exists a need for understanding the impact of using Docker on multiple hosts, as CPSs typically contain multiple computing nodes. The computational overhead of using multiple hosts is identified in [22] in the context of cloud computing. However, a similar approach is needed within the context of CPSs, using experimental units that utilise a full implementation of a CPS such as the one implemented in this research.

The software requirements for the specific use case play the primary role for deciding whether Docker is safe or unsafe to implement into the software deployment architecture. This research has found that for vehicular CPSs which have scheduling precision, camera input and disk output performance as critical software requirements, Docker is a favourable choice for the deployment strategy. Docker is favourable due to the added value it brings to a software development project in the form of saved time, added safety and functionality, and portability among other things. However, for vehicular CPS projects which introduce additional critical requirements such as network performance, further research is required to understand how Docker behaves when network performance between separate computer nodes is part of the critical requirements. This is particularly interesting to explore as there may exist additional scheduling precision overhead of the CPS when introducing significant networking interrupts onto a system executing the CPS. Therefore, it is required to understand the behaviour during such a scenario before drawing conclusions to whether Docker performs appropriately as a deployment strategy in a networked environment.

To conclude, Docker together with a real-time enabled kernel is a viable platform for deployment in the context of vehicular CPSs in respect to meeting the timing requirements of a CPS application. It is therefore recommended to use Docker in an environment with a: 1) real-time enabled kernel (RT_prempt), 2) modern storage driver (OverlayFS).

Bibliography

[1] O. Bosser and I. Ip, C. Starikova. (2015, September). [Online]. Available: http://www.mckinsey.com/business-functions/business-technology/our-insights/beyond-agile-reorganizing-it-for-faster-software-delivery

[2] D. G. Feitelson, E. Frachtenberg, and K. L. Beck, "Development and deployment at facebook," *IEEE Internet Computing*, vol. 17, no. 4, pp. 8–17, July 2013.

[3] R. Kohavi, R. Longbotham, D. Sommerfield, and R. M. Henne, "Controlled experiments on the web: survey and practical guide," *Data Mining and Knowledge Discovery"*, vol. 18, no. 1, pp. 140–181, 2009. [Online]. Available: http://dx.doi.org/10.1007/s10618-008-0114-1

[4] L. E. Lwakatare, T. Karvonen, T. Sauvola, P. Kuvaja, H. H. Olsson, J. Bosch, and M. Oivo, "Towards devops in the embedded systems domain: Why is it so hard?" in *2016 49th Hawaii International Conference on System Sciences (HICSS)*, Jan 2016, pp. 5437–5446.

[5] M. Aichouch, J. C. Prévotet, and F. Nouvel, "Evaluation of the overheads and latencies of a virtualized rtos," in *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, June 2013, pp. 81–84.

[6] P. González-Nalda, I. Etxeberria-Agiriano, I. Calvo, and M. C. Otero, "A modular cps architecture design based on ros and docker," *International Journal on Interactive Design and Manufacturing (IJIDeM)*, pp. 1–7, 2016. [Online]. Available: http://dx.doi.org/10.1007/s12008-016-0313-8

[7] C. Berger, "Testing continuous deployment with lightweight multi-platform throw-away containers," 2015.

[8] A. Krylovskiy, "Internet of things gateways meet linux containers: Performance evaluation and discussion," in *Internet of Things (WF-IoT), 2015 IEEE 2nd World Forum on*, Dec 2015, pp. 222–227.

[9] W. Wolf, "Cyber-physical systems," *Computer*, vol. 42, no. 3, pp. 88–89, Mar. 2009. [Online]. Available: http://dx.doi.org/10.1109/MC.2009.81

[10] J. Wan, H. Yan, H. Suo, and F. Li, "Advances in cyber-physical systems research." *TIIS*, vol. 5, no. 11, pp. 1891–1908, 2011.

[11] C. Anderson, "Docker [software engineering]," *IEEE Software*, no. 3, pp. 102–c3, 2015.

[12] Opendavinci - open source development architecture for virtual, networked, and cyber-physical system infrastructures. [Online]. Available: http://opendavinci.cse.chalmers.se/www/

[13] What is docker? [Online]. Available: http://www.docker.com/what-docker

[14] D. Bernstein, "Containers and cloud: From lxc to docker to kubernetes," *IEEE Cloud Computing*, no. 3, pp. 81–84, 2014.

[15] Grand cooperative driving challenege. [Online]. Available: http://www.gcdc.net/en/event-en

[16] G. Siddesh, G. Deka, K. Srinivasa, and L. Patnaik, *Cyber-Physical Systems: A Computational Perspective*. CRC Press, 2015.

[17] R. Alur, *Principles of Cyber-Physical Systems*. MIT Press, 2015.

[18] Os lecture 4. [Online]. Available: http://cs.nyu.edu/courses/spring11/G22.2250-001/lectures/lecture-04.html

[19] A. Carzaniga, A. Fuggetta, R. S. Hall, D. Heimbigner, A. Van Der Hoek, and A. L. Wolf, "A characterization framework for software deployment technologies," DTIC Document, Tech. Rep., 1998.

[20] M. Meyer, "Continuous integration and its tools," *IEEE Software*, vol. 31, no. 3, pp. 14–16, May 2014.

[21] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and linux containers," in *Performance Analysis of Systems and Software (ISPASS), 2015 IEEE International Symposium on*, March 2015, pp. 171–172.

[22] C. N. Mao, M. H. Huang, S. Padhy, S. T. Wang, W. C. Chung, Y. C. Chung, and C. H. Hsu, "Minimizing latency of real-time container cloud for software radio access networks," in *2015 IEEE 7th International Conference on Cloud Computing Technology and Science (CloudCom)*, Nov 2015, pp. 611–616.

[23] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "Kvm, xen and docker: A performance analysis for arm based nfv and cloud computing," in *Information, Electronic and Electrical Engineering (AIEEE), 2015 IEEE 3rd Workshop on Advances in*, Nov 2015, pp. 1–8.

[24] R. Morabito, J. Kjällman, and M. Komu, "Hypervisors vs. lightweight virtualization: A performance comparison," in *Proceedings of the 2015 IEEE International Conference on Cloud Engineering*, ser. IC2E '15. Washington, DC, USA: IEEE Computer Society, 2015, pp. 386–393.

[25] M. G. Xavier, I. C. D. Oliveira, F. D. Rossi, R. D. D. Passos, K. J. Matteussi, and C. A. F. D. Rose, "A performance isolation analysis of disk-intensive workloads on container-based clouds," in *2015 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, March 2015, pp. 253–260.

[26] C. Ruiz, E. Jeanvoine, and L. Nussbaum, "Performance evaluation of containers for hpc," in *Euro-Par 2015: Parallel Processing Workshops: Euro-Par 2015 International Workshops, Vienna, Austria, August 24-25, 2015, Revised Selected Papers*. Cham: Springer International Publishing, 2015, pp. 813–824. [Online]. Available: http://dx.doi.org/10.1007/978-3-319-27308-2_65

[27] R. Wu, Y. Chen, E. Blasch, B. Liu, G. Chen, and D. Shen, "A container-based elastic cloud architecture for real-time full-motion video (fmv) target tracking," in *2014 IEEE Applied Imagery Pattern Recognition Workshop (AIPR)*, Oct 2014, pp. 1–8.

[28] C. Wohlin, "Guidelines for snowballing in systematic literature studies and a replication in software engineering," in *Proceedings of the 18th International*

Conference on Evaluation and Assessment in Software Engineering, ser. EASE '14. New York, NY, USA: ACM, 2014, pp. 38:1–38:10. [Online]. Available: http://doi.acm.org/10.1145/2601248.2601268

[29] Scopus - document search. [Online]. Available: www.scopus.com

[30] M. G. Xavier, M. V. Neves, F. D. Rossi, T. C. Ferreto, T. Lange, and C. A. F. D. Rose, "Performance evaluation of container-based virtualization for high performance computing environments," in *2013 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, Feb 2013, pp. 233–240.

[31] W. J. Matthew, "Performance optimization of linux networking for latency-sensitive virtual systems," Ph.D. dissertation, ARIZONA STATE UNIVERSITY, 2015.

[32] Raspberry pi - teach, learn, and make with raspberry pi. [Online]. Available: https://www.raspberrypi.org/

[33] T. Gleixne. Cyclic test. [Online]. Available: https://rt.wiki.kernel.org/index.php/Cyclictest

[34] A. Jedlitschka, M. Ciolkowski, and D. Pfahl, "Reporting experiments in software engineering," in *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, and D. I. K. Sjøberg, Eds. London: Springer-Verlag London, 2008, ch. 8, pp. 201–228.

[35] Manual page for stress-ng - a tool to load and stress a computer system. [Online]. Available: http://manpages.ubuntu.com/manpages/wily/man1/stress-ng.1.html

[36] Leibniz formula for pi. [Online]. Available: https://en.wikipedia.org/wiki/Leibniz_formula_for_%CF%80

[37] Chalmers-revere/opendlv: A software for the operation of driver-less vehicles. [Online]. Available: https://github.com/chalmers-revere/opendlv

[38] Aec-6950 - embedded computers. [Online]. Available: www.aaeon.com/en/p/fanless-embedded-computers-aec-6950

[39] Index of /kernel-ppa/mainline/v3.18.25-vivid. [Online]. Available: http://kernel.ubuntu.com/~kernel-ppa/mainline/v3.18.25-vivid/

[40] L. Fu and R. Schwebel. Rt preempt howto. [Online]. Available: https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO

[41] W. Davison, J. Weigert, and M. Schroeder. Screen user's manual. [Online]. Available: https://www.gnu.org/software/screen/manual/screen.html

[42] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. K. Sjøberg, "A systematic review of quasi-experiments in software engineering," *Inf. Softw. Technol.*, vol. 51, no. 1, pp. 71–82, Jan 2009. [Online]. Available: http://dx.doi.org/10.1016/j.infsof.2008.04.006

[43] S. Jackson, *Statistics Plain and Simple*. Cengage Learning, 2013. [Online]. Available: https://books.google.se/books?id=pdNkCtMHYAYC

[44] P. Runeson and M. Höst, "Guidelines for conducting and reporting case study research in software engineering," *Empirical Softw. Engg.*, vol. 14, no. 2, pp. 131–164, Apr 2009. [Online]. Available: http://dx.doi.org/10.1007/s10664-008-9102-8

[45] V. B. Kampenes, T. Dybå, J. E. Hannay, and D. I. Sjøberg, "A systematic review of effect size in software engineering experiments," *Information and Software Technology*, vol. 49, no. 11, pp. 1073–1086, 2007.

A

Source Code

A.1 Pi Component

```
1  while (getModuleStateAndWaitForRemainingTimeInTimeslice() == odcore::data::dmcp::↩
       ModuleStateMessage::RUNNING) {
2      // Second measurement point
3      odcore::data::TimeStamp::writeNanoToSerial("2");
4      odcore::data::TimeStamp start, end;
5
6      // Pi algorithm variable are
7      // reset after each timeslice.
8      long double tempPi;
9      long double pi     = 4.0;
10     int i              = 1;
11     int j              = 3;
12     float oDuration    = 0;
13
14     while (true) {
15
16         // Calculate pi
17         tempPi = static_cast<double>(4)/j;
18         if (i%2 != 0) {
19             pi -= tempPi;
20         } else if (i%2 == 0) {
21             pi += tempPi;
22         }
23         i++;
24         j+=2;
25
26
27         // Occupy for a certain duration
28         end = odcore::data::TimeStamp();
29         oDuration = end.toNanoseconds()-start.toNanoseconds();
30         if (oDuration >= occupy)
31             break;
32     }
33     // Third measurement point
34     odcore::data::TimeStamp::writeNanoToSerial("3");
35     if (piTimes==duration)
36         return odcore::data::dmcp::ModuleExitCodeMessage::OKAY;
37 }
```

A.2 Pi/IO Component

```
while (getModuleStateAndWaitForRemainingTimeInTimeslice() == odcore::data::dmcp::↩
    ModuleStateMessage::RUNNING) {
    // Second measurement point
    odcore::data::TimeStamp::writeNanoToSerial("2");
    odcore::data::TimeStamp start, end;

    if (m_camera.get() != NULL) {
        odcore::data::image::SharedImage si = m_camera->capture();
        // Third measurement point
        odcore::data::TimeStamp::writeNanoToSerial("3");
        odcore::data::Container c(si);
        distribute(c);
        // Fourth measurement point
        odcore::data::TimeStamp::writeNanoToSerial("4");
    }



    // Pi algorithm variable are
    // reset after each timeslice.
    long double tempPi;
    long double pi     = 4.0;
    int i              = 1;
    int j              = 3;
    float oDuration    = 0;

    while (true) {

        // Calculate pi
        tempPi = static_cast<double>(4)/j;
        if (i%2 != 0) {
            pi -= tempPi;
        } else if (i%2 == 0) {
            pi += tempPi;
        }
        i++;
        j+=2;


        // Occupy for a certain duration
        end = odcore::data::TimeStamp();
        oDuration = end.toNanoseconds()-start.toNanoseconds();
        if (oDuration >= occupy)
            break;
    }

    // Fifth measurement point
    odcore::data::TimeStamp::writeNanoToSerial("5");
    if (piTimes==duration)
        return odcore::data::dmcp::ModuleExitCodeMessage::OKAY;
}
```

```
std::shared_ptr<odcore::wrapper::SerialPort> TimeStamp::m_serialPort = NULL;
void TimeStamp::setupSerial(const string port, uint32_t baud_rate) {
    std::shared_ptr<odcore::wrapper::SerialPort> serial(odcore::wrapper::↩
        SerialPortFactory::createSerialPort(port, baud_rate));
    TimeStamp::m_serialPort = serial;
}

const string TimeStamp::writeNanoToSerial(const char* measurementID) {
    TimeStamp ts = TimeStamp();
    try {
        stringstream s;
```

```
11          s << measurementID << ";" << ts.toNanoseconds() << "\r\n";
12          if (m_serialPort) {
13              m_serialPort->send(s.str());
14          } else {
15              return "Serial port not defined";
16          }
17          return "Successfully wrote toNanosecond() to serial";
18      }
19      catch(string &exception) {
20          return exception;
21      }
22  }
23
24  const string TimeStamp::writeMicroToSerial(const char* measurementID) {
25      TimeStamp ts = TimeStamp();
26      try {
27          stringstream s;
28          s << measurementID << ";" << ts.toMicroseconds() << "\r\n";
29          if (m_serialPort) {
30              m_serialPort->send(s.str());
31          } else {
32              return "Serial port not defined";
33          }
34          return "Successfully wrote toMicrosecond() to serial";
35      }
36      catch(string &exception) {
37          return exception;
38      }
39  }
40
41  const string TimeStamp::writeStringToSerial(const char* measurementID) {
42      TimeStamp ts = TimeStamp();
43      try {
44          stringstream s;
45          s << measurementID << ";" << ts.toString() << "\r\n";
46          if (m_serialPort) {
47              m_serialPort->send(s.str());
48          } else {
49              return "Serial port not defined";
50          }
51          return "Successfully wrote toString() to serial";
52      }
53      catch(string &exception) {
54          return exception;
55      }
56  }
57
58  const string TimeStamp::writeMessageToSerial(const string message) {
59      try {
60          if (m_serialPort) {
61              m_serialPort->send(message);
62          } else {
63              return "Serial port not defined";
64          }
65          return "Successfully wrote message to serial";
66      }
67      catch(string &exception) {
68          return exception;
69      }
70  }
```