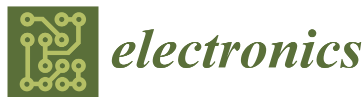


MDPI

Article

Container-Based Electronic Control Unit Virtualisation: A Paradigm Shift Towards a Centralised Automotive E/E Architecture

Nicholas Ayres [1,†], Lipika Deka [1,*,†] and Daniel Paluszczyszyn [1,2]

[1] Faculty of Computing, Engineering and Media, De Montfort University, Leicester LE1 9BH, UK; nick.ayres@dmu.ac.uk (N.A.); daniel.paluszczyszyn@horiba-mira.com (D.P.)
[2] HORIBA MIRA Ltd., Nuneaton, Warwickshire CV10 0TU, UK
* Correspondence: lipika.deka@dmu.ac.uk
† These authors contributed equally to this work.

Abstract: The past 40 years have seen automotive Electronic Control Units (ECUs) move from being purely mechanical controlled to being primarily digital controlled. While the safety of passengers and efficiency of vehicles has seen significant improvements, rising ECU numbers have resulted in increased vehicle weight, greater demands placed on power, more complex hardware and software, ad hoc methods for updating software, and subsequent increases in costs for both vehicle manufacturers and consumers. To address these issues, the research presented in this paper proposes that virtualisation technologies be applied within automotive electrical/electronic (E/E) architecture. The proposed approach is evaluated by comprehensively studying the CPU and memory resource requirements to support container-based ECU automotive functions. This comprehensive performance evaluation reveals that lightweight container virtualisation has the potential to welcome a paradigm shift in E/E architecture, promoting consolidation and enhancing the architecture by facilitating power, weight, and cost savings. Container-based virtualisation will also enable efficient and robust online dynamic software updates throughout a vehicle's lifetime.

Keywords: automotive E/E architecture; automotive ECU; containers; embedded system; performance and virtualisation

Citation: Ayres, N.; Deka, L.; Paluszczyszyn, D. Container-Based Electronic Control Unit Virtualisation: A Paradigm Shift Towards a Centralised Automotive E/E Architecture. *Electronics* **2024**, *13*, 4283. https://doi.org/10.3390/electronics13214283

Academic Editors: Sergio Trilles Oliver, Fabio Corti, Marco Del-Coco, Pierluigi Carcagni, Ditsuhi Iskandaryan, Xin Wang and Kai Fu

Received: 19 August 2024
Revised: 15 October 2024
Accepted: 25 October 2024
Published: 31 October 2024

Copyright: © 2024 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

The 21st-century automotive electrical and electronic (E/E) architecture is unable to manage the increasing demands being placed on it [1]. This arose as we witnessed the automotive technology move from a solely mechanically controlled device to a digitally connected system in the last four decades. These digital systems primarily consist of electronic control units (ECUs), which are small, often bespoke embedded systems that monitor and control the basic operations of the vehicle, as well as provide safety, efficiency, comfort, and infotainment functions and services [2]. The first single-function ECU was introduced in the late 1970s [3], and today, most premier vehicles incorporate in excess of 100 ECUs.

Current increased ECU numbers are fueled by technological advancements, together with increased demand for new consumer features and functions for occupant/vehicle interaction, as well as increased safety and vehicle operational efficiency. Increasing computerisation of the automobile requires complex computing hardware, application, and operating software, as well as multiple in-vehicle networking solutions, enabling communication between often disparate subsystems. Together with the number of ECUs and interconnection hardware, an exponential increase has been seen in the number of peripheral components, including sensors and actuators [4].

Non-standard ad hoc inclusion of new features packaged as niche ECUs, together with the requirement of safety and reliability, has resulted in the evolution of a more distributed

vehicular software architecture. Such a distributed architecture, though it has provided its benefits during the early development phase, has now evolved into a complex and intricate mesh of ECUs, incurring a number of disadvantages. These include the following:

- Increased cost: There is a marked increase in cost and overheads with each new addition of hardware and its associated software from development to maintenance [5].
- Increased complexity: The complexity of the modern automotive E/E architecture has arisen from the ever-increasing interactions between various components parts. The introduction of multiple in-vehicle networking solutions such as CANBus, LIN, and Flexray has aided in overall system complexity, as more automotive functions require data across multiple automotive domains of responsibility [5,6].
- Reduced scalability: embedded systems are not inherently scalable by design. They are bespoke pieces of hardware that have been highly optimised on an architectural level to ensure that only the necessary components are included [7].

To deal with the issues mentioned above, stemming from the growing complexity of the E/E architecture that has developed over the years, and to encourage system consolidation, multiple independent automotive functions must be combined with fewer hardware devices. Therefore, a new approach to automotive E/E architecture is needed. This empirical research investigates how a particular lightweight virtualisation technique can be used in the automotive industry to address many of the current hardware and software issues that have been identified.

2. Virtualisation in the Automotive Domain

The automotive E/E architecture has developed very similarly to the traditional data centre. Clients within a data centre access services and systems residing in dedicated servers. Drawing parallel to the E/E architecture, sensors (such as those of the anti-lock braking system (ABS)) and actuators (such as the ABS actuators that control the brake pressure to prevent wheel locking) are the clients accessing services, data, and systems from the ECUs, which are analogous to servers. Both systems have evolved through the need for new business functions being met by a new dedicated server, thus leading to the increase in hardware components that are underutilised [8,9]. Hence, conventional data centres have suffered from issues such as decentralisation of the hardware, consequent increases in the cost of their implementation and operation, and increased complexity and space constraints. The data centre community has successfully addressed these issues through the adoption of the virtualisation technology.

Virtualisation technology, dating back to the 1960s, abstracts physical resources like servers, storage, and networks into virtual machines (VMs) for efficiency. Full-system virtualisation can be Type 1 or Type 2, which is managed by a hypervisor or virtual machine monitor. The automotive ECU design lacks essential technologies for hypervisor support. Container virtualisation, a lightweight form of virtualisation, allows applications to run in isolated containers on a shared operating system, making it more efficient than traditional virtual machines. The container runtime, such as Docker Engine or containerd (as used in this research), is responsible for managing container operations and resource allocation.

To address the disadvantages highlighted in Section 1 within the automotive sector, there has been a move in recent years towards a more centralised automotive E/E architecture. To facilitate centralisation, multiple automotive functions must be combined into a lesser hardware platform, moving away from dedicated hardware for every function. Given the parallels with the traditional data centres, virtualisation has been seen as a mechanism for achieving a centralised and streamlined architecture [10] to consolidate hardware and reduce costs, such as in areas having client/server architecture [11,12]. In addition to cost reduction due to reduced hardware, virtualisation offers function isolation features, which can increase system redundancy, security, and productivity [13]. Architectural centralisation and virtualisation combine a group of physically separate hardware resources such as ECUs to facilitate the execution of multiple applications simultaneously on a single

system. Such ECU consolidation gives the impression of several separate physical systems or user environments.

Virtualisation allows dynamic load balancing and improved throughput and continuity, which is particularly necessary for system-critical applications. Another one of the key advantages is temporal and spatial separation [14]. However, full-system virtualisation and hypervisors can incur considerable system resource overheads [15]. To overcome such limitations, lightweight virtualisation solutions, such as containers, are being adopted [16–18].

Operating system virtualisation via containers offers a number of benefits to the automotive E/E architecture. Containerised applications can start almost instantly. With full-system virtualisation, a single hardware platform running multiple virtual machines requires an operating system for each virtual machine to support installed applications. In contrast, containers run within a single operating system environment, with each container sharing parts of the OS kernel. A containerised application includes in a single package all the required dependencies, libraries, binaries, and configuration files for that program to run. Containers offer a much-reduced footprint of megabytes rather than gigabytes. Similar to full-system virtualisation, each container's available resource is in isolation from other containers on the same host system. This resource isolation is achieved through Linux kernel namespaces, enabling different processes to utilise global system resources as though they were their own isolated set of resources.

An ECU utilising containers can operate in a "just-in-time" (JIT) mode, where it is activated as and when needed and then shut down when it is no longer required, thus freeing up the host's resources. Containers enable high modularity; large monolithic applications can be broken up into separate modules and run within their own individual containers. This microservice approach can facilitate changes to different application parts without rebuilding the entire application. Containers are ideally suited to over the air (OtA) software updates, which makes it possible to update software automatically, which is initiated by the vehicle owner without the need for domain experts [19–21].

Given the potential of a container-based virtualisation technique, this study proposes and evaluates container-based electronic control unit virtualisation by comprehensively studying the CPU and memory resource requirements needed to support container-based ECU automotive functions. It uses the newly proposed system performance metrics—the Utilisation, Saturation, and Duration (USD) methodology—to perform the evaluation.

3. Related Work

There have been some studies and articles relating specifically to ECU virtualisation. Refs. [10,22] discuss automotive E/E architecture scalability utilising virtualisation and the AUTOSAR standard. Embedded hypervisors are being introduced into the automotive arena to promote safety [22]. The authors in [23] looked at where virtualisation could be used within the motor car, focusing on less-critical solutions such as the vehicle head unit and passenger infotainment functionality. To promote system consolidation, ref. [24] investigated the reduction in overall cost and weight by dedicating specific cores to various virtualised automotive systems, including infotainment and telematic information. Containers have shown potential in non-automotive sectors that are both high performance [25] and embedded [17,26] in nature, hence holding promise for the E/E architecture.

Building on the work done by Soltesz et al. [27] and Feltes et al. [28], containers are a viable alternative to hypervisor technology that can provide increased scalability and higher performance workloads. Kugele et al. [29] proposed a data-centric service-oriented automotive E/E architecture to facilitate (among other higher-level goals) common messages needed by several functions in several ECUs as services and to address the issue around high bus load as a result of messages that have no receivers. With Kugele et al. [29], their primary contribution is in the area of using a shared middleware layer to provide a data-centric, publish–subscribe, service-oriented architecture. The architecture uses docker-based containerisation to harness its layered architecture approach to allow for

sharing and reusing layers among containers towards the higher-level goal of providing service-oriented architecture. The proposed architecture by Kugele et al. [29] assumes that lower layers do not change, while our proposed architecture is rooted in the fact that updates will be required throughout the software–firmware layers, thus allowing much needed adaptability.

Very little work or research has been published on container-based virtualisation in automotive E/E architecture. Examining containers and what they offer could provide major benefits to the modern motor car, which is not just limited to infotainment features and functions but adds to the entire automotive E/E. Given the aforementioned possibilities, an important direction for this research is to evaluate the efficiency of container-based ECU virtualisation whilst executing an automotive function. Our evaluation was performed on a bespoke hardware test bed, which modelled a number of different automotive ECUs that provided the automotive function of a central locking/unlocking mechanism of vehicle doors.

To fully test the container-based automotive architecture, it was decided that a mix of real-time and non-real-time systems would be required to represent a distributed automotive function. Several potential possibilities were researched, culminating in a vehicle door central locking mechanism incorporating real- and non-real-time inputs being selected as the test system to be simulated. A central locking mechanism is a simple vehicle-related function that locks and unlocks either all doors or selected doors based on some form of trigger mechanism.

4. Design of a Container-Based ECU

The automotive architecture has evolved historically in response to functional and customer needs, without a holistic blueprint, leading to a complex and error-prone architecture. Academia and industry have responded in recent times with possible standardised approaches to address these issues, and we present them within this section as appropriate in the context of the proposed work.

Automotive ECUs are generally simple computing devices, where power consumption, speed of operation, and efficiency are key aspects of their design. ECU hardware architectures are finely tuned to the software function they have been designed for, as observed in Figure 1. Their rigid design has little to no capacity for additional lines of code or functionality in any future software update. The microprocessor is an important feature of an ECU and is commonly an ARM (Advanced RISC Machine)-based microprocessor. ARM-based ECUs integrate many additional peripherals within their architecture, keeping the number of additional required components to complete a given function or task to a minimum. This type of architecture is ideal for an automotive ECU.

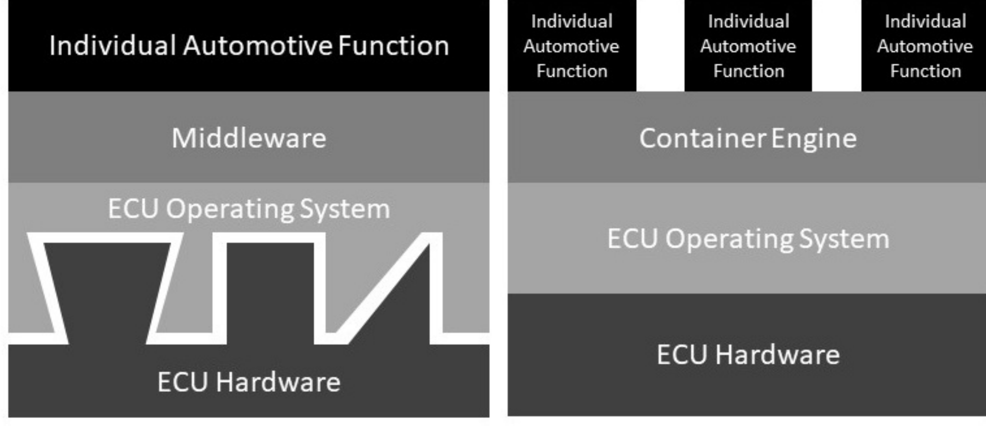


Figure 1. Traditional ECU Architecture (**left**) and Proposed Container-based ECU Architecture (**right**).

4.1. Automotive Central Locking Function

Figure 2 is an example of a classic door central locking/unlocking function. The modelled central locking mechanism has the main output of locking the vehicle doors through simulated relays. There are several triggering mechanisms to achieve this. These come from a simulated remote key fob when the "vehicle" achieves a specific speed or if the safety mechanism iss activated (simulated crash). The lock–unlock can be achieved manually via the fob, automatic locking is achieved through a specific vehicle speed, and automatic unlocking is achieved through triggering the safety mechanism. When recording the resource use/load on the individual ECUs, the mechanism is triggered periodically to look for spikes in use. Consequently, for this research, the modelled central locking function incorporated real and non-real-time conditions, triggering the locking mechanism, including the visual outputs when the system is activated. The central locking mechanism was activated via the following triggers:

- Infrared Remote (Simulated remote key fob);
- Transmission—Selection of a specific gear;
- Vehicle Speed—When vehicle achieves a specific speed.

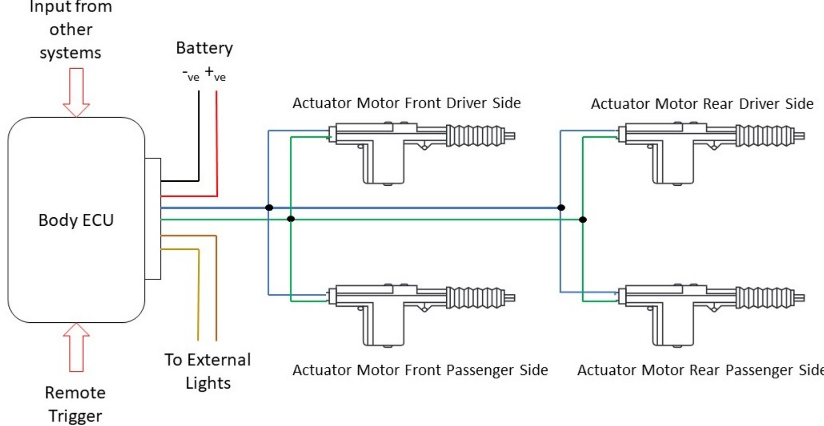


Figure 2. Diagrammatic Representation of a Central Car Door Locking System.

4.2. Testbed ECU and Sensor Hardware

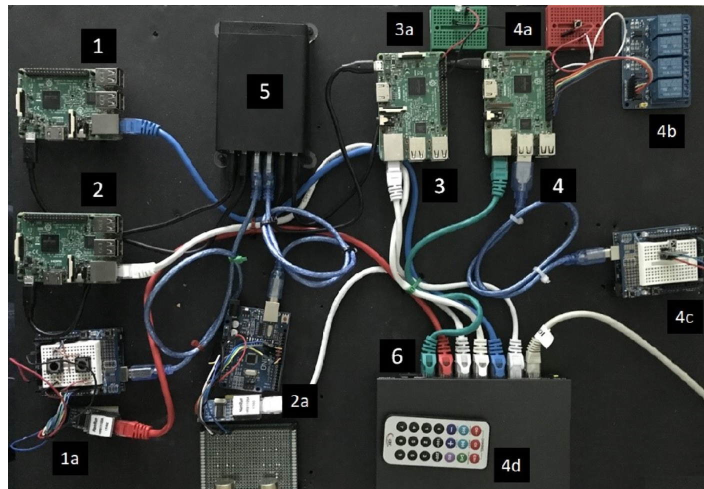
To assess how suitable containers are within an automotive context, an experimental test system was required to model an ECU/vehicle function utilising a generic ARM-based hardware. For the experimental test system, the ARM Raspberry Pi 3B was chosen to model an ECU, as seen in Figure 3. The Raspberry Pi (similar to an ECU) has many available GPIO pins, which can interact directly with hardware and other ECUs connected externally. Although the Raspberry Pi has limited potential within a commercial environment, it is ideally placed as a platform for research purposes [30].

The initial simulated data are provided by the sensors on the Raspberry Pi. The door-locking mechanism is controlled by the actuators, and the LEDs simulate the vehicle lights. The above-described system requires the following peripheral hardware, including the following:

- Central door-locking mechanism—implemented using a four-channel relay module, where every relay corresponds to a particular door, and a door is locked when its associated relay is closed.
- Remote door lock trigger—an infrared transmitter device operated remotely to alter the manual door states. The transmitter signals an IR receiver, including one of three possible door commands. This functionality is possible only when the gear is in a neutral state.
- External lighting—the vehicle headlights are simulated using an LED, and the lighting is adjusted according to the door state it receives.

- Gear position—the selected gear is indicated on one of the two push-button sensors through a series of depressions, indicating that the gear is going down or up.
- Vehicle acceleration—a potentiometer sets the acceleration of the simulated system measured in r/min with data ranging between 0 and 1024.
- The collision sensor—the airbag activation safety mechanism or collision sensor is simulated using a push-button sensor.

Certain peripheral devices are connected directly from the GPIO pins of an ECU, while Arduino programmable circuit boards connect others. These boards collect, process, and transmit raw sensor data to the pertinent ECU through an in-vehicle network.



System Number	System	System Number	System
1	Gear ECU	4a	Collision Sensor
1a	Gear Select Sensor	4b	Door Lock Relays
2	Engine ECU	4c	IR Receiver Sensor
2a	Accelarator Sensor	4d	IR Sender/Fob
3	Light ECU	5	System Power Supply
3a	Exterior light	6	In Vehicle Network Hub
4	Door ECU		

Figure 3. ARM-Raspberry-Pi-3B-based Central Car Door Locking Testbed.

5. Implementation

To accurately evaluate containers' performance for automotive E/E, the central door-locking mechanism modelled must closely mimic an existing vehicle's, thus involving ECU functions and control systems from several fields or domains on separate ECU platforms as seen below:

- The Transmission ECU is from within the transmission field;
- The Acceleration ECU is from within the engine field;
- The Door Control ECU is from within the body;
- The Light Control ECU is also from within the body domain.

Each ECU function identified was hosted on its personal Raspberry Pi platform and connected by a network to replicate functionality across domains. Each Raspberry Pi platform was responsible for collecting and processing specific data. Figure 3 shows the testbed design for the ECU hardware and peripherals.

The adopted operating system was Raspbian Lite, which includes all the necessary components to execute a virtual container environment, including the network software stack and tools to monitor performance.

6. Evaluation Methodology

The evaluation began by running the software within the ECU operating system to establish a baseline. This setup, termed as being in a native state, allowed for comparison when testing the software in a container. The CPU and memory resources needed for an ECU functionality were ascertained during the designing and testing phase. The paradigm shift proposed within this study to a container-based virtual ECU architecture will require supplementary software and an additional abstraction layer in between the application software and the hardware. To understand the overhead on system resources as a result of ECUs operating within a container compared to the current ECU environment, an evaluation was carried out, as explained below, detailing the metrics used for conducting performance evaluation, the various test modes, and the tools used.

6.1. System Performance Metrics

The USE Methodology (Utilisation, Saturation, Errors) can be used to evaluate the performance of hardware resources (such as CPU, memory, disk, and network) and ensure that systems operate efficiently [31,32]. It also supports identifying performance bottlenecks. In addition, the RED Method (Rate, Errors, Duration) is a methodology used primarily in the context of monitoring and analysing the performance of microservices and distributed systems [33,34]. A combination of USE and RED provides the required approach for monitoring the resources needed to support the execution of native and container-based ECUs. A new methodology was developed and implemented by integrating elements of both methods, focusing on the Utilisation, Saturation, and Duration (USD) components leading to the performance evaluation of the CPU and memory of the target system. The evaluation metrics in this study include the following:

- Utilisation—defined as the percentage of time a particular resource is busy as opposed to when the resource is free/available.
- Saturation—defined as the number of tasks waiting for a particular resource. It is identified as the length of the queue.
- Duration—defined as the total time spent processing a request.

6.2. ECU Test Mode of Operation

A series of test comparisons were conducted on three system-operational modes to ascertain what extra system resources a container-based ECU would need. These test modes included the following:

- Base system-operational mode—A baseline benchmark for the system resources is determined when the modelled ECU is idle in this test mode. These test results will allow for comparison when the ECU operates in the subsequent test modes to help determine the overheads of bringing in ECU-functional software.
- Native system-operational mode—The existing automotive E/E architectures and ECU function are modelled in this mode. Compared to the base system test, it will provide specific increases in system resource usage when each ECU functional software is executed.
- Container system-operational mode—Any resource overhead incurred while running the ECU functions in a container will be captured in this test mode. This overhead measure is against native operation.

The test criterion was defined to ensure consistent conditions for all resource usage tests in the operational modes defined above. Each metric was evaluated over several runs of the system, collecting several instances of data across different time frames to increase reliability and accuracy in understanding the system's resource usage. The test times were divided into three time frames (60 s, 600 s, and 1200 s), as indicated below:

- time_60—The 60-second tests analyse the resources needed during the initialisation of the ECU software. In the test setup, it was ascertained that 60 s allowed adequate time for the system to warm up before the execution of the scripts.

- time_600—The 600-second sample run tests comprehensively examined the entire life cycle of the functional software from initialization to termination. A 30-second system idle warm-up period was included to eliminate resource spikes associated with initial software execution. The timed tests involved a 500-second script execution time, after which the script terminated, and a 30-second cool-down period ensued. A critical focus of the investigation was the levels of CPU and memory resources once the scripts were executed, which determined if the system returned to a state similar to the pre-execution of the script. Understanding the amount of resources not given back to the system after script execution is particularly important for ECU systems that operate periodically to carry out distinct tasks.
- time_1200—This test runs for 1200 s. During this testing period, resource usage patterns were examined to determine whether there was any notable increase in usage during lengthy execution cycles. In order to present this data effectively, the published results should represent the entirety of the 1200-second test, particularly in cases where there was minimal to no observable activity.

6.3. Software Tools

To measure key CPU and memory metrics across all four ECUs, a variety of software monitoring tools were necessary. The specific areas of research within each system and the associated tools can be found in Figure 4. With the newly proposed USD methodology during each ECU run, we observed these specific resources. We used the tools described below to observe and report CPU and memory utilisation, saturation, and duration for each ECU, both when the ECU was idle and when the function it housed was running in the native/bare metal and container test modes.

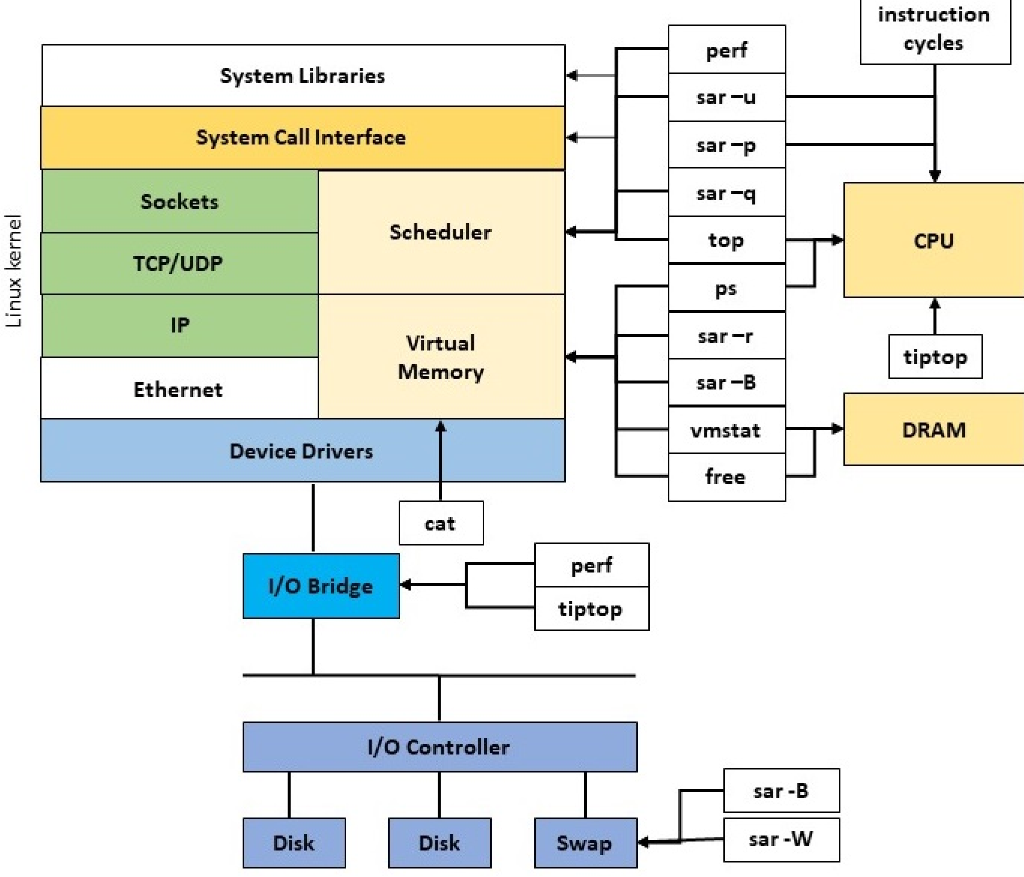


Figure 4. Software Monitoring Tools Used to Measure Key CPU and Memory Metrics.

6.3.1. Vmstat

Vmstat helps in ratifying the data gathered by other monitoring tools, and it is primarily used to determine resource bottlenecks. The metrics monitored included the following:

- The total processes or tasks waiting for runtime (R state);
- The total processes or tasks waiting in an uninterruptible sleep (D state);
- The total available system memory;
- The information on swap memory;
- CPU statistics, which include %us, %sy, and %id.

6.3.2. Free (Free and Used Memory)

This tool comprehensively displays statistics on memory, encompassing total available memory, swap usage, and physical memory. Additionally, it presents other key metrics, such as the allocation of memory in use (buffers column) and the status of memory that has been allocated but swapped to disk or not allocated based on resource requirements (cache column).

6.3.3. Schedstat (Scheduler Statistics)

The schedstat command provides OS kernel scheduler statistics for individual processes. These values represent the average CPU time of a process or time spent waiting for the CPU at the time of schedstat command execution. schedstat encompasses three statistics that characterise scheduling latency:

- The amount of time processes spend running on the CPU.
- The amount of time processes spend waiting to be scheduled.
- The number of times processes are switched on and off the CPU.

ECU functions typically have real-time requirements, as they operate as safety-critical systems. Understanding the scheduling latency of ECUs operating within containers instead of the bare metal or native mode is essential for this research.

6.3.4. Sar (System Activity Reporter)

This research extensively relies on the sar tool to monitor the use of ECU resources, including processor and memory utilisation and saturation. The tool can be configured with specific flags to monitor and record various metrics, such as paging information (sar—B); CPU load, run queue, and process list lengths (sar—q); memory usage values (sar—R); the percentage of used system memory (sar—r); and paging rates (sar—W).

6.3.5. Perf (Performance Analysis)

This system-wide profile tool provides statistical information to analyse the performance of an application during execution. perf provides data on schedulers such as perf sched latency and perf sched timehist, which measure the latency of processes and events. This includes the sched-in count, total run time, and average run time per sched-in count.

6.3.6. Pidstat (Process Statistics)

This tool reports statistics, including the time spent, executing kernel tasks, and associated child processes.

6.3.7. Top/Htop/Nmon

These tools give real-time data on system metrics, such as CPU usage, memory usage, disk I/O, and network activity, allowing users to analyse system behaviour and troubleshoot performance issues. They monitor the process and thread execution of the ECU and identify any supplementary processes initiated during native and container modes. These tools were used to verify the precision of the results obtained from other tools discussed above.

7. Experimental Results

This section presents the in-depth experimental results of how containers utilise system resources, especially processor and memory resources, which are essential in real-time systems where timing is linked to safety.

7.1. System Saturation

CPU saturation is a crucial metric for evaluating system performance. It depicts the additional work that the CPU cannot process immediately and must queue, thereby increasing latency [32]. The CPU is deemed saturated if the average system load exceeds the number of CPU cores and sustains a saturation level for a prolonged period, where the CPU load is the total number of processes executing or in the queue waiting for the CPU at any given time. An over-saturated system experiences overheads, including longer waiting times for *idle* or *waiting* processes, increased request response time, and a consequent rise in CPU utilisation.

The running program is CPU-bound when the number of processor-executable instructions exceeds the other types of instructions. Thus, an algorithm that has numerous calculations and uses the entire schedule-allocated time is an example of a CPU-bound process. The CPU saturation tests carried out during this study utilise the standard built-in OS scheduler (SCHED_OTHER), which is a traditional time-shared process.

7.1.1. System Load

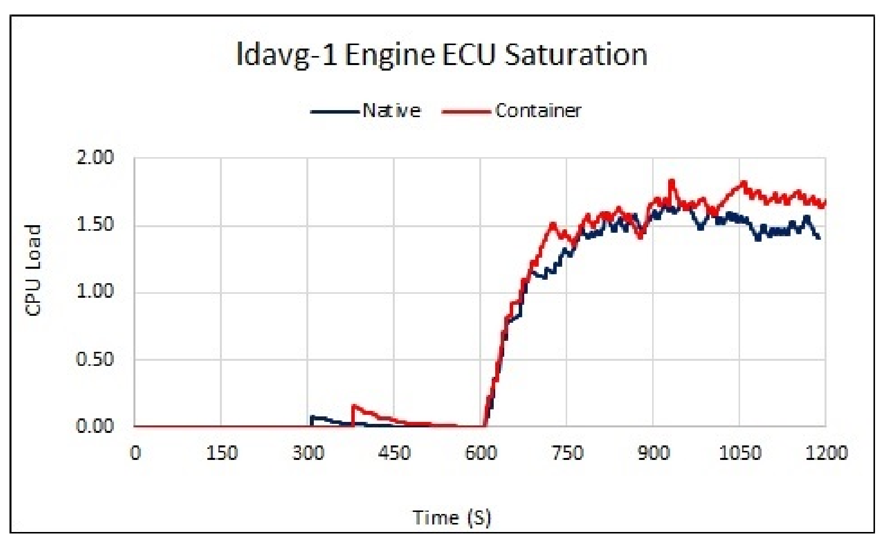
The system load is determined by averaging the number of running processes (i.e., in the R state) and the number of processes in uninterruptible sleep (i.e., in the D state). ldavg-1 represents the total of R and D state processes over a 60-second period. Meanwhile, ldavg-5 and ldavg-15 measure longer time frames of 300 and 900 s, respectively, and are not simply averages of the ldavg-1 results repeated over time. The following provides a clear understanding of the relationship between each load average:

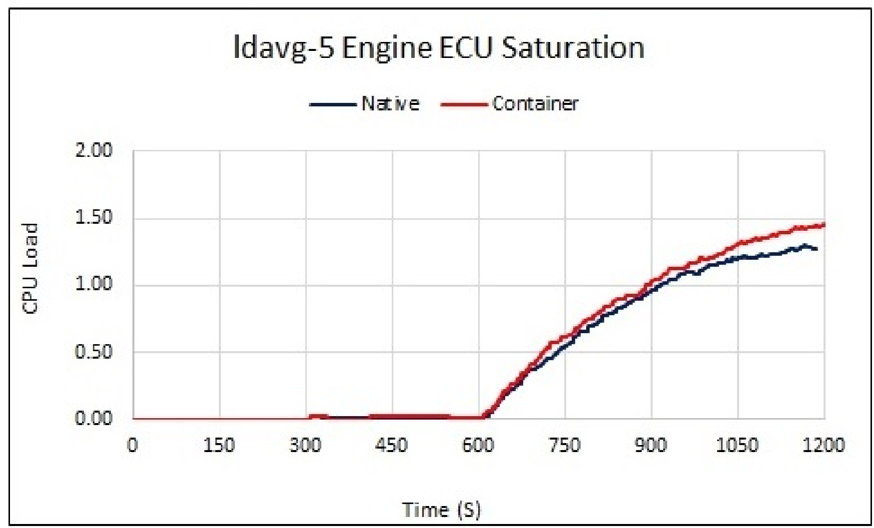
- The system is in an idle state if the average is 0.00.
- The system load is considered to be increasing if the ldavg-5 or ldavg-15- average is greater than the ldavg-1.
- The overall system load is decreasing if the ldavg-5 or ldavg-15- average is lesser than the ldavg-1.

It is important to remember that if the average load in any ldavg exceeds the number of CPU cores, it can impact system performance. A high value on the ldavg-1 metric may indicate brief spikes in the use of resources caused by the program itself or increased paging to the disk, providing a snapshot of what system activities took place recently. If the ldavg-15 metric consistently surpasses the CPU core count, it most often indicates a sustained increase in processes in the R and D states, potentially indicating a persistent issue affecting overall system performance.

In Figure 5, the system load average results are displayed. Throughout the base test run, all CPU load averages experienced prolonged periods of minimum load, with ldavg-1 reaching its peak at 0.17 (17%). While ldavg-5 registered at 0.11 (11%) and ldavg-15 at 0.05 (5%), both outputs were lower than anticipated. The average load for all the base test runs was 0.02 (2%). Additionally, the ldavg-5 and 15 exhibited a declining trend over the course of the test run, indicating that the CPU was primarily in an idle state.

The results shown in Table 1 provide details about the CPU load averages obtained from native and container environments. The total load for all the ECU functions, which combines the CPU ldavg-1, ldavg-5, and ldavg-15 load averages, was observed to be +9.50%, +4.33%, and +3.98%, respectively.





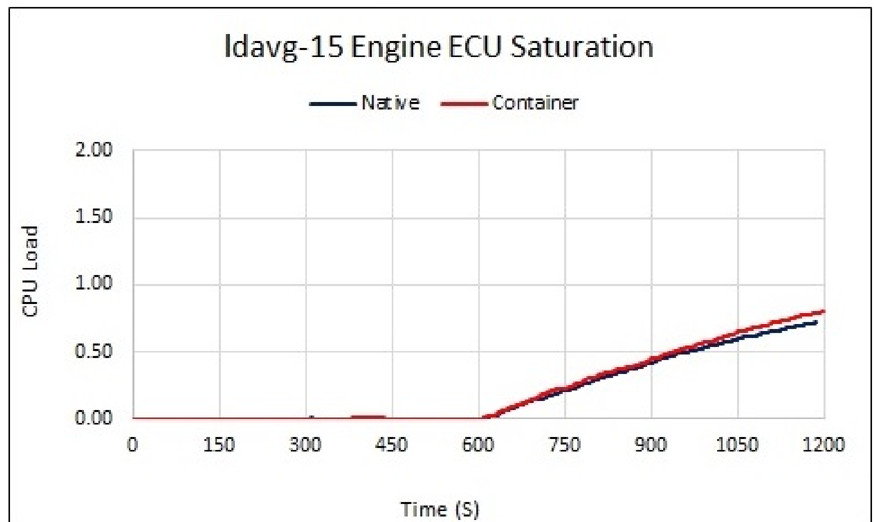


Figure 5. System Load Average Results. ldavg-1 (1st graph) represents the total of R (the total of processes in the running state) and D (the number of processes in uninterruptible sleep) state processes over a 60-second period, ldavg-5 represents the total of R and D state processes over a 300-second period, and ldavg-15 represents the total of R and D state processes over a 900-second period.

Table 1. CPU load averages of ECU functions in native and container environments.

All ECU Ldavg 1, 5, 15 (Automotive Function Load)			
System Load	**Test Mode**		**Increase in CPU Load**
	Native	**Container**	
ldavg-1	747.22	818.22	+9.50%
ldavg-5	482.04	502.83	+4.33%
ldavg-15	225.80	234.79	+3.98%

7.1.2. Scheduler Statistics

The scheduler is responsible for determining which processes will currently run and which will be placed in the waiting queue (runqueue). Upon creation, each process is issued a time slice, i.e., the amount of processor time it is allocated. The two metrics monitored are the time spent executing on the CPU and the time spent waiting on the run queue.

While operating in the container mode, more time was spent executing the ECU software than in the native/bare metal mode. The amount of time the software spent in the runqueus was longer in the bare metal mode than in the container mode. In the container mode, the software reserves a specific amount of CPU time for execution. To replicate this effect during the native mode, the priority of the native process must be increased, thus facilitating more CPU time. However, this can negatively affect other processes executing on the same system.

Tables 2 and 3 display the CPU utilisation and task wait times for CPU allocation, respectively . The total CPU saturation increased when all ECUs ran within containers. Yet, it did not lead to a significant rise in new tasks, processes, or waiting times of processes, nor did it impose an excessive overhead on the CPU. Notably, the average CPU load over all ECUs was 9.44% higher when operating in the container mode compared to the native execution. These findings illustrate that operating in the container mode did not lead to CPU saturation in any ECU.

Table 2. CPU Utilisation Time According to Each Task in Native and Container Mode.

Test Mode	Test on CPU			
	Door	**Engine**	**Gear**	**Light**
Native	0.124 s	0.146 s	0.269 s	0.124 s
Container	0.134 s	0.196 s	0.255 s	0.121 s

Table 3. Waiting Time on *runqueue* by Each Task in Native and Container mode.

Test Mode	Waiting Time on Runqueue			
	Door	**Engine**	**Gear**	**Light**
Native	0.052 ms	0.062 ms	0.115 ms	0.041 ms
Container	0.018 ms	0.032 ms	0.047 ms	0.011 ms

7.2. System Resource Utilisation

CPU saturation indicates the amount of work pending to be processed, while CPU utilization measures the workload demand relative to the capacity. Elevated CPU utilisation may suggest subpar application performance, as processes may need to wait for other processes in the queue to finish execution. The key CPU metrics observed during these experiments were the following:

- %user—refers to the percentage of CPU time used for executing user-level application processes and kernel activities such as handling interrupts and managing resources.
- %system—the percentage of time the CPU was used to execute system-level processes.

- %idle— the percentage of time the CPU was not used, with no pending requests for disk I/O.
- %nice—the percentage of CPU utilisation refers to the time spent executing higher-level user processes. A positive value of %nice indicates more higher-level processes that use the CPU more.
- %iowait—the percentage of time the CPU was idle, albeit with pending disk I/O requests.

The workload is deemed to be CPU-bound if the total of %user, %system ,and %nice is 100%. A long-term, constant high CPU utilisation can have detrimental effects, including the following:

- A rise in hardware temperatures, reducing the CPU's operational lifetime or leading to its untimely failure.
- An increase in power consumption (power is an essential factor to be considered and kept optimal in automotive systems).
- A resulting increase in CPU usage.

7.2.1. Overall CPU Utilisation

The monitored metrics of %user, %system, and % idle were collected from the iostat, mpstat, vmstat, and sar monitoring tools. These tools provided outputs akin to each other and were used to ensure data consistency and correctness over all the modelled ECU's CPU cores.

The tests showed minimal overall CPU utilisation, as depicted in Figure 6 and Table 4, similar to the CPU saturation baseline. Due to space restrictions, only one example of CPU utilisation is shown here, with Figure 6 demonstrating %user values of the Engine ECU in the native and container modes. Table 4 demonstrates the %utilisation values of all the ECUs in the native and container modes, including the overhead as a difference. Exhaustive results can be found in [16]. When the script was executed inside the container, there was an average increase of +4.01% in %user utilisation. The %system utilisation within the container indicated the supplementary processes and tasks required to support it, including the container shim, docker, and contained processes. This resulted in system function increasing by an average of 7.75%. Altogether, the extra %user and %system utilisation needed to support the central locking system was 4.86% higher than the native mode of operation.

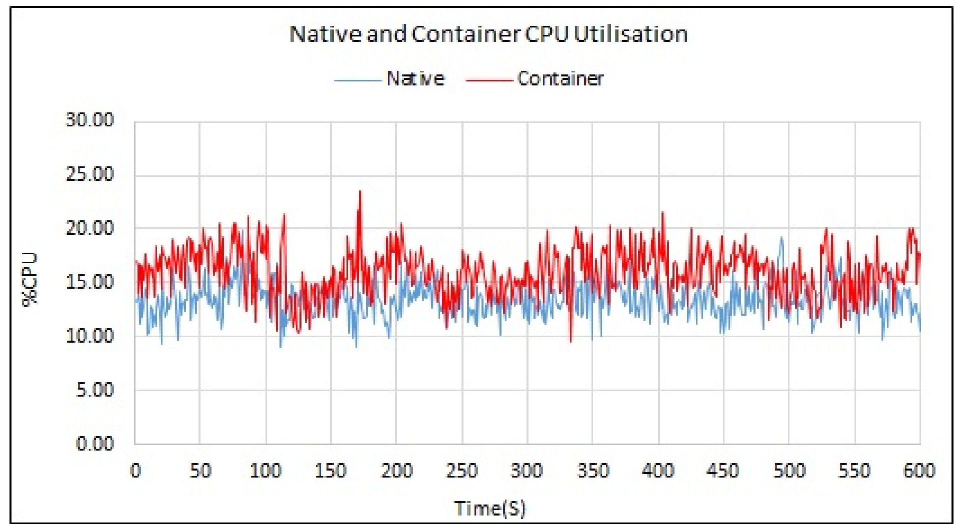


Figure 6. Engine ECU %user Native and Container Tests across all CPUs.

Table 4. All ECU %utilisation Values Across Native and Container Modes.

ECU	Average CPU Utilisation		
	Native	Container	Difference
Engine	35.91%	37.37%	+4.07%
Gear	41.66%	47.61%	+14.28%
Door	38.62%	38.85%	+0.60%
Light	25.23%	25.35%	+0.48%
All ECU	Function Utilisation		+19.43%

7.2.2. Instructions per Cycle

The IPC metric measures the number of instructions executed in each CPU clock cycle, with the stated IPC value for the ECU used in this study (ARM Cortex-A53 processor) being 2.0. The stated IPC value served as a benchmark for providing an overall perspective on CPU utilisation throughout the three test modes. The results are presented in Figure 7 and Table 5.

Table 5. Instructions Executed per CPU Cycle in Each Test Mode, Followed by The %difference Values Between Native and Container Modes.

60 s Sample	Instructions per Cycle (IPC)			
	Base	Native	Container	%Diff (Native and Container)
Door ECU	0.61	0.57	0.55	−3.57
Engine ECU	0.62	0.45	0.45	0.00
Gear ECU	0.57	0.43	0.43	0.00
Light ECU	0.59	0.58	0.58	0.00

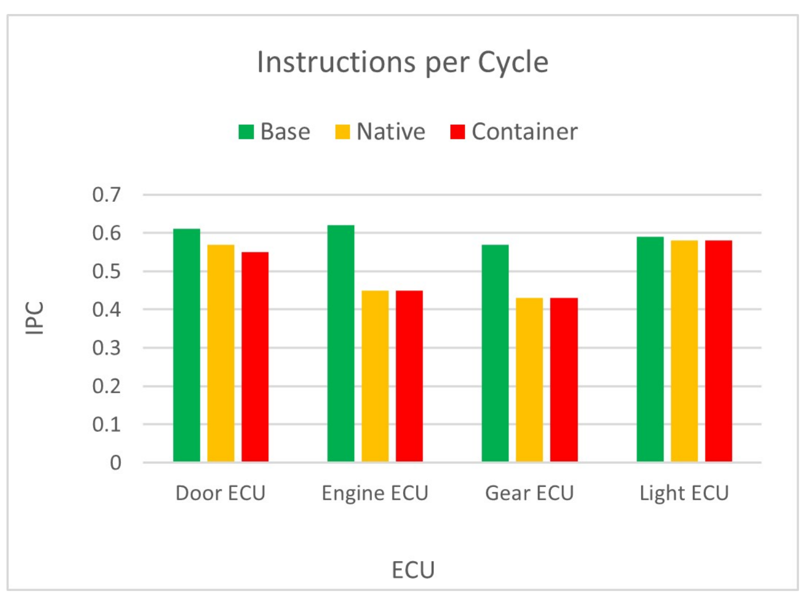


Figure 7. Instructions Executed per CPU Cycle.

As the CPU load increased, it impacted the overall IPC value. In the baseline test mode, the recorded IPC value ranged between 0.57 and 0.62 for all four ECUs modelled in this study. The IPC values obtained during the native test mode did not differ much from those obtained during the container test modes. The Door ECU exhibited a small decrease in its IPC value of −3.50% when operating within the container. This test results

emphasise that, regardless of the ECU operational mode, there is no significant difference in the number of instructions executed per second by the ECU processor.

7.2.3. Memory-Unique Set Size

Tools that monitor memory usage typically emphasise the Resident Set Size (RSS), representing the amount of physical memory that a process occupies. However, the RSS often overestimates the memory used by processes and provides limited information about the pages of memory shared among processes. In contrast, Proportional Set Size (PSS) indicates the main memory that a process is assigned, consisting of memory exclusively used by the process and that was shared with other processes. Understanding the proportional and exclusive memory use by comparing memory usage in bare metal and container test modes is crucial. The Unique Set Size (USS) represents the quantity of exclusive memory allocated by the kernel to a particular process.

The USS (Figure 8 and Table 6) shows the quantity of memory allocated to processes. The USS test results indicate that the memory required by the functional software script when running within a container environment is much higher than previously considered. On average, the functional script requires 70.93% more system memory compared to the native operation over the four ECUs. Additionally, taking into consideration the extra memory needed for container-specific processes, the average memory usage was seen to be 13.07%. Hence, even though all types of virtualisation will require more memory to house the virtual instances, this increased memory requirement is reasonable.

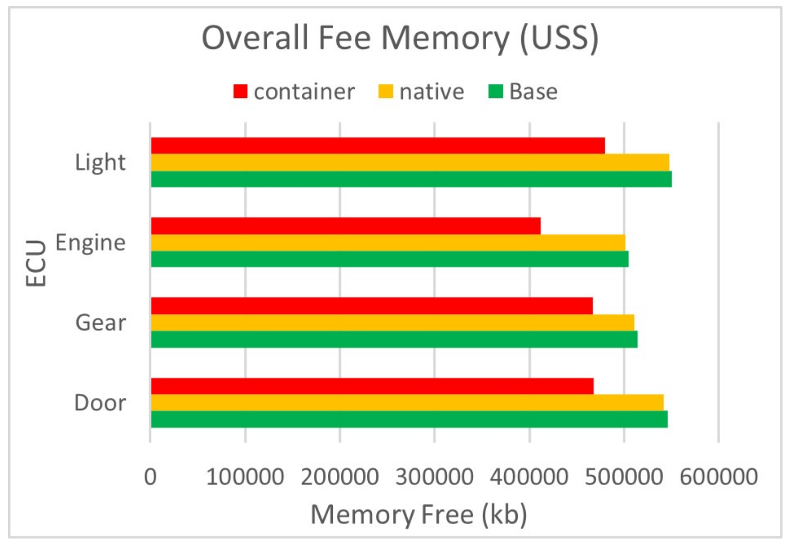


Figure 8. Memory-Unique Set Size—Quantity of Memory Allocated to Processes when Running in the Different Test Modes.

Table 6. Memory-Unique Set Size—Quantity of Memory Allocated to Processes when Running in the Different Test Modes, Followed by the Overhead of Running in Container Mode over the Native Test Mode.

ECU	Total System Available Free Memory (kb)				
	Base	Native	Container	Diff (Native and Container)	% Diff
Door	546,076	541,464	467,400	74,064	+13.56
Gear	514,048	510,588	466,884	43,704	+8.50
Engine	504,480	501,556	411,616	89,940	+17.83
Light	550,488	547,932	479,688	68,244	+12.40

7.3. Evaluation of Container Specific Resource Consumption

Several subprocesses were initiated and run in the background to incorporate the ECU into the container environment. Each container functions as a client in silo, being supported by the container daemon server. A container is built from an image of multiple layers of software that support process configuration and the container-encased function.

When the container is executed, the docker process executes first, using CPU resources to start up the container. The runc process is a lightweight container runtime process that contains code interacting with dedicated OS features related to the execution of container virtualisation. The runc process initialises the container and, upon completion, hands over control to the container shim. The container shim is responsible for managing the respective container. Its Python script is automatically executed after the runc process and container shim have completed the handover.

The shim and runc processes utilise minimal CPU resources. This has been indicated in the section above on CPU utilisation. The CPU utilisation overhead to support the container-based virtualisation was minimal overall.

Once the container is initialised and under container shim management, the observed memory usage of container-specific processes remains relatively static. The containerd process is a persistent process from the time the system is initiated to its shutdown. Functioning as an executor of containers, the containerd process manages the life cycle of containers. Figures 9 and 10 illustrate the specific CPU and memory usage rates by the individual container processes.

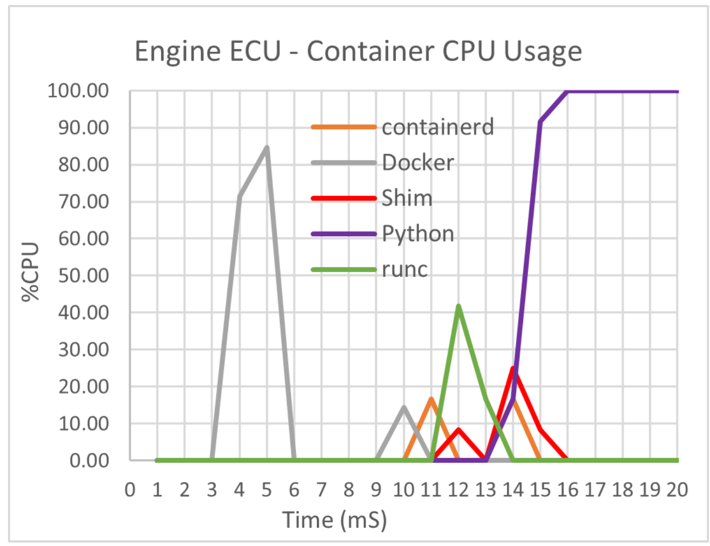


Figure 9. Engine ECU Container CPU Use According to Process Type.

The daemon and shim container processes were launched to manage and sustain the memory that the container shell required, starting from container initialisation. The daemon process monitors API requests and oversees container configuration tasks such as imaging, networking, and volumes, resulting in notably higher memory usage overall. Upon completion of initialisation by the shim process, the Python script executes and allocates memory appropriately.

The results depicted in this section show the individual ECU computation load, comparing performance when a specific automotive function ran within a container or directly on the hardware. The increase in load when the function was "containerised" was minimal, supporting the hypothesis of a container virtualization-based EE architecture that can address increasing complexity and aid in continual software updates throughout the vehicle's lifespan.

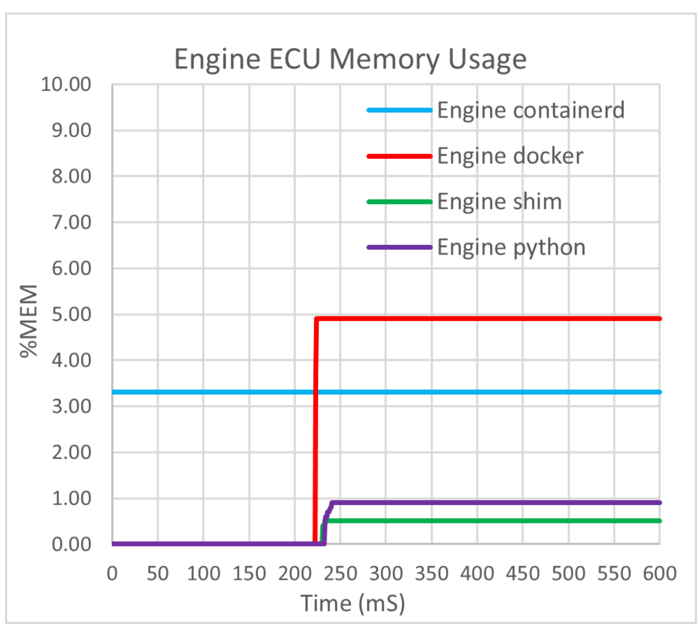


Figure 10. Engine ECU Container Memory Use According to Process Type.

8. Conclusions and Future Work

In the past, vehicle manufacturers have typically addressed the growing demand for new consumer features and functions by increasing the number of ECUs. However, this research suggests a departure from this approach and highlights several parallels between the data centre and automotive E/E architecture. It argues that virtualisation technologies, which have proven beneficial in data centres, can be successfully implemented in the automotive context. Specifically, container virtualisation can most importantly facilitate ECU unification, leading to cost reductions and improved vehicle efficiency.

8.1. Key Findings

During the initial start up of the ECU functional software, small spikes were observed in the CPU load. Still, the increase in the CPU load during container execution was just 6.73% over all triplets and ECUs. The CPU utilisation tests in the native and container modes showed only slight differences. The tests indicate that the average increase in required memory across all ECUs is 13.07%, which is equal to an overall average of 275 kb of memory, or about 68 kb per container. Hence, it has been shown that executing ECU software within a container has no significant negative impact on CPU load and memory. The results also indicate that container-based ECUs are well suited for ECU consolidation, whether processes are memory-bound, CPU-bound, or a combination of the two.

8.2. In Summary

The results of this study are significant for the automotive industry, as well as any industry using embedded systems. To begin with, the study shows that multiple ECU functions can be integrated into single containers on the same hardware platform, merging embedded software and hardware. This reduces physical hardware and overall weight, directly benefiting vehicle fuel economy and operational range and saving costs. For instance, savings are possible through reduced hardware development costs. This can have major benefits, as the costs needed to develop tailored ECU hardware have increased by 75% since 2000, with an indication that the cost will double by 2030. Savings can also be achieved through a decrease in ECU hardware components. Additionally, these savings can be passed on to consumers by cutting manufacturers' costs.

8.3. Going Forward

The research conducted utilised standard OS and container software. However, a more optimised and efficient container-based ECU could be achieved using optimised OS and container platforms, specifically by leveraging kernel scheduling algorithms. Future work will explore this aspect in greater detail. The security of container-based virtualisation systems may be lower than that of fully virtualised systems; hence, thorough investigation and development are essential. Container isolation is integrated into the OS kernel instead of being included as separate virtual machines. Another avenue for future research will involve studying the transfer rates of data and the advantages of container-based ECU consolidation. This has the potential to reduce the need for in-car networking, subsequently leading to reductions in associated hardware, weight, and complexity.

Author Contributions: Conceptualisation, N.A. and L.D.; methodology, N.A.; software, N.A.; validation, N.A., L.D. and D.P.; formal analysis, N.A.; investigation, N.A.; resources, N.A., L.D. and D.P.; data curation, N.A.; writing—original draft preparation, N.A.; writing—review and editing, N.A., L.D. and D.P.; visualisation, N.A.; supervision, L.D. and D.P.; project administration, L.D. and D.P.; funding acquisition, L.D. All authors have read and agreed to the published version of the manuscript.

Funding: This research received no external funding.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Our data will be made available upon request.

Acknowledgments: This work was part of the Ph.D. studies of the first author. The Ph.D. studentship was funded by De Montfort University. We are thankful for the support.

Conflicts of Interest: Author Daniel Paluszczyszyn is employed by HORIBA MIRA Company. The remaining authors declare that the research was conducted in the absence of any commercial or financial relationships that could be construed as a potential conflict of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ND	Nicholas Ayres
LD	Lipika Deka
DP	Daniel Paluszczyszyn

References

1. Saidi, S.; Steinhorst, S.; Hamann, A.; Ziegenbein, D.; Wolf, M. Future automotive systems design: Research challenges and opportunities. In Proceedings of the International Conference on Hardware/Software Codesign and System Synthesis, Torino, Italy, 30 September–5 October 2018.
2. Takada, H. *Introduction to Automotive Embedded Systems*; Nagoya University: Nagoya, Japan. Available online: https://cse.buffalo.edu/~bina/cse321/fall2015/Automotive-embedded-systems.pdf (accessed on 16 August 2024).
3. Bereisa, J. Applications of microcomputers in automotive electronics. *IEEE Trans. Ind. Electron.* **1983**, *2*, *87–96*. [CrossRef]
4. Breitschwerdt, D.; Cornet, A.; Kempf, S.; Michor, L.; Schmidt, M. *The Changing Aftermarket Game and How Automotive Suppliers Can Benefit from Arising Opportunities*; McKinsey: New York City, NY, USA, 2017 .
5. Quigley, C.P.; McMurran, R.; Jones, R.P.; Faithfull, P.T. An Investigation into Cost Modelling for Design of Distributed Automotive Electrical Architectures. In Proceedings of the 3rd Institution of Engineering and Technology Conference on Automotive Electronics, Dublin, Ireland, 22–25 April 2007; pp. 1–9.
6. Nolte, T.; Hansson, H.; Bello, L.L. Automotive communications-past, current and future. In Proceedings of the IEEE Conference on Emerging Technologies and Factory Automation, Catania, Italy, 19–22 September 2005 ; p. 8. [CrossRef]
7. Broy, M. Challenges in automotive software engineering. In Proceedings of the 28th International Conference on Software Engineering, Shanghai, China, 20–28 May 2006; pp. 33–42. [CrossRef]
8. Scheepers, M.J. Virtualization and containerisation of application infrastructure: A comparison. In Proceedings of the 21st Twente Student Conference on IT, Enschede, The Netherlands, 23 January 2014; Volume 21.

9. Rolik, O.; Zharikov, E.; Telenyk, S.; Samotyy, V. Dynamic virtual machine allocation based on adaptive genetic algorithm. In Proceedings of the Eighth International Conference on Cloud Computing, GRIDs, and Virtualization, Athens, Greece, 19–23 February 2017; pp. 108–114.
10. Reinhardt, D.; Kucera, M. Domain controlled architecture. In Proceedings of the 3rd International Conference on Pervasive and Embedded Computing and Communication Systems (PECCS 2013), Barcelona, Spain, 19–21 February 2013.
11. Chebiyyam, M.; Malviya, R.; Bose, S.K.; Sundarrajan, S. Server consolidation: Leveraging the benefits of virtualization. *Infosys Res. Setlabs Briefings* **2009** , *7*, 65–75.
12. Malhotra, L.; Agarwal, D.; Jaiswal, A. Virtualization in cloud computing. *J. Inf. Technol. Softw. Eng.* **2014**, *4*, 1–3.
13. Lombardi, F.; Pietro, R.D. Secure virtualization for cloud computing. *J. Netw. Comput. Appl.* **2011**, *34*, 1113–1122. [CrossRef]
14. Heiser, G. Hypervisors for consumer electronics. In Proceedings of the 6th IEEE Consumer Communications and Networking Conference, Las Vegas, NV, USA, 10–13 January 2009; pp. 1–5. [CrossRef]
15. Bermejo, B.; Juiz, C. Virtual machine consolidation: A systematic review of its overhead influencing factors. *J. Supercomput.* **2020**, *76*, 324–361. [CrossRef]
16. Ayres, N. Enhancing the Automotive E/E Architecture Utilising Container-Based Electronic Control Units. 2021. Available online: https://dora.dmu.ac.uk/server/api/core/bitstreams/694f85d2-3e57-4ffb-8ae2-a54485464bd6/content (accessed on 30 August 2024).
17. Noronha, V.; Lang, E.; Riegel, M.; Bauschert, T. Performance Evaluation of Container Based Virtualization on Embedded Microprocessors. In Proceedings of the 30th International Teletraffic Congress (ITC 30), Vienna, Austria, 3–7 September 2018; pp. 79–84. [CrossRef]
18. Morabito, R. Virtualization on Internet of Things Edge Devices With Container Technologies: A Performance Evaluation. *IEEE Access* **2017**, *5*, 8835–8850. [CrossRef]
19. Halder, S.; Ghosal, A.; Conti, M. Secure over-the-air software updates in connected vehicles: A survey. *Comput. Netw.* **2020**, *178*, 107343 . [CrossRef]
20. Ayres, N.; Deka, L.; Paluszczyszyn, D. Continuous Automotive Software Updates through Container Image Layers. *Electronics* **2021**, *10*, 739. [CrossRef]
21. Ayres, N.; Deka, L.; Passow, B. Virtualisation as a Means for Dynamic Software Update within the Automotive E/E Architecture. In Proceedings of the 2019 IEEE SmartWorld, Ubiquitous Intelligence and Computing, Advanced and Trusted Computing, Scalable Computing and Communications, Cloud and Big Data Computing, Internet of People and Smart City Innovation, Leicester, UK, 19–23 August 2019; pp. 154–157. [CrossRef]
22. Reinhardt, D.; Morgan, G. An embedded hypervisor for safety-relevant automotive E/E-systems. In Proceedings of the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014), Pisa, Italy, 18–20 June 2014; pp. 189–198. [CrossRef]
23. Pelzl, J.; Wolf, M.; Wollinger, T. Virtualization Technologies for Cars: Solutions to increase safety and security of vehicular ECUs. In Proceedings of the Embedded World Conference, Nuremberg, Germany, 26–28 February 2019. Available online: http://www.escrypt.com (accessed on 1 July 2024).
24. Dakroub, H.; Shaout, A.; Awajan, A. Connected Car Architecture and Virtualization. *SAE Int. J. Passeng. Cars—Electron. Electr. Syst.* **2016**, *9*, 153–159. [CrossRef]
25. Xavier, M.G.; Neves, M.V.; Rossi, F.D.; Ferreto, T.C.; Rose, C.A.F.D. Performance Evaluation of Container-Based Virtualization for High Performance Computing Environments. In Proceedings of the 21st Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, Belfast, UK, 27 February–1 March 2013; pp. 233–240. [CrossRef]
26. Morabito, R. A performance evaluation of container technologies on Internet of Things devices. In Proceedings of the IEEE Conference on Computer Communications Workshops (INFOCOM WKSHPS), San Francisco, CA, USA, 10–14 April 2016; pp. 999–1000. [CrossRef]
27. Soltesz, S.; Herbert, P.; Fiuczynski, M.E.; Bavier, A.; Peterson, L. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems, Lisbon, Portugal, 21–23 March 2007; pp. 275–287. [CrossRef]
28. Felter, W.; Ferreira, A.; Rajamony, R.; Rubio, J. An updated performance comparison of virtual machines and linux containers. In Proceedings of the 2015 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), Philadelphia, PA, USA, 29–31 March 2015; pp. 171–172. [CrossRef]
29. Kugele, S.; Hettler, D.; Peter, J. Data-Centric Communication and Containerization for Future Automotive Software Architectures. In Proceedings of the IEEE International Conference on Software Architecture (ICSA), Seattle, WA, USA, 30 April–4 May 2018; pp. 65–69. [CrossRef]
30. Walter, J.; Fakih, M.; Grüttner, K. Hardware-based real-time simulation on the Raspberry pi. In Proceedings of the 2nd Workshop on High Performance and Real-Time Embedded Systems, Vienna, Austria, 20 January 2014.
31. Hartmann, H. System Monitoring with the USE Dashboard. 2017. Available online: https://www.circonus.com/2017/08/system-monitoring-with-the-use-dashboard/ (accessed on 16 August 2024).
32. Gregg, B. Thinking methodically about performance. *Commun. ACM* **2013**, *56*, 45–51. [CrossRef]

33. Jackson, J. The RED Method: A New Approach to Monitoring Microservices. 2018. Available online: https://thenewstack.io/monitoring-microservices-red-method/ (accessed on 16 August 2024).
34. Wilke, T. The RED Method: Key Metrics for Microservices Architecture. 2017. Available online: https://www.weave.works/blog/the-red-method-key-metrics-for-microservices-architecture/ (accessed on 16 August 2024).

Disclaimer/Publisher's Note: The statements, opinions and data contained in all publications are solely those of the individual author(s) and contributor(s) and not of MDPI and/or the editor(s). MDPI and/or the editor(s) disclaim responsibility for any injury to people or property resulting from any ideas, methods, instructions or products referred to in the content.