Performance Isolation of Co-located Workload in a Container-based Vehicle Software Architecture

Johannes Büttner, Pere Bohigas Boladeras, Philipp Gottschalk, Markus Kucera and Thomas Waas
Faculty of Computer Science and Mathematics
Regensburg University of Applied Sciences
Regensburg, Germany
e-mail: {johannes2.buettner, pere.bohigas, philipp1.gottschalk, markus.kucera, thomas.waas}@oth-regensburg.de

***Abstract*—As the development in the automotive sector is facing upcoming challenges, the demand for in-vehicle computing power capacity increases and the need for flexible hardware and software structures arises, allowing dynamic managament of resources. In this new scenario, software components are to be added, removed, updated and migrated between computing units. To isolate the software components from each other and allow its orchestration, a container-based virtualization approach is being tested throughout this research. The analysis focuses on the question if this virtualization technology could be an option to ensure an interference-free operation. Four different sample applications from the automotive environment are tested for their susceptibility to resource contention. The research on the one hand shows that CPU and memory used by an application can be largely isolated with this technology, but on the other hand, it becomes apparent that support for I/O-heavy usage is currently not implemented sufficiently for container engines.**

***Keywords*—container-based virtualization; resource management; resource isolation; stress testing**

I. INTRODUCTION

The automotive sector is facing an upcoming unprecedented redesign of vehicles. Besides the introduction of new electrical propulsion systems, the achievement of higher levels of driving automation and the development of the connected car are driving the conversion of vehicles into computers on wheels. Thus, on the one side, the use of deep learning and AI to drive a vehicle requires a massive computer power combined with safety redundancy and high availability. On the other side, the use of a huge quantity of sensors and the interconnectivity with a variety of IoT-based devices demands new flexible and dynamic architectures.

The desired flexibility and processing capacity therefore requires a fundamental revision and redesign of the actual in-vehicle system architectures. In the research project A^{3}F (*Ausfallsichere Architekturen für Autonome Fahrzeuge* – fail-safe architectures for autonomous driving vehicles), in which the Regensburg University of Applied Sciences (*Ostbayerische Technische Hochschule Regensburg*) takes part, evaluates new concepts for future vehicle system architectures [1]. The focus is on well-known distributed computing architectures in current enterprise IT and data centers that have similar issues and face similar challenges. Dynamic and flexible application architectures in this area have been developed decades ago by means of technologies, such as virtual machines and containers.

Within the A^{3}F project, new architectures for the in-vehicle computation system are conceived through the analysis of multi-node homogeneous computer cluster structures. This new approach to a flexible and dynamically managed hardware platform, on which distributed performance-intensive applications can be executed and orchestrated, employs high performance server nodes and reconfigurable Ethernet switches, as shown in Figure 1. The generic server nodes execute

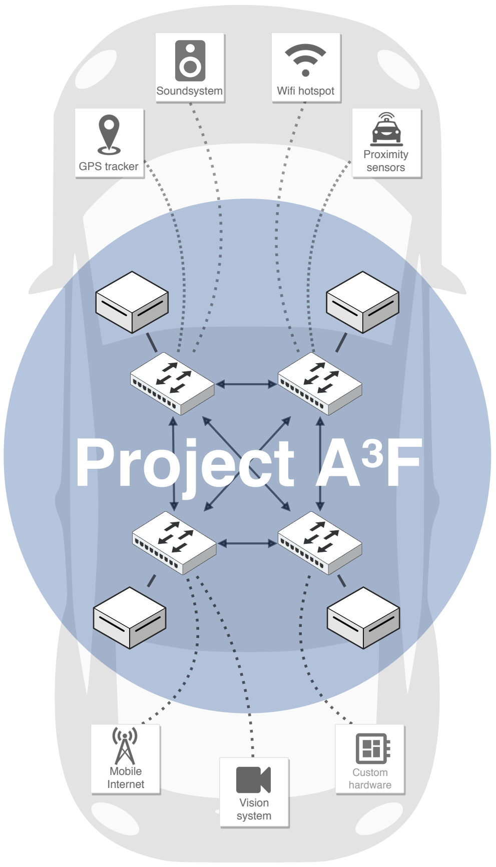


Figure 1. New architecture based on a computer cluster with redundant network connection, investigated within the project A^{3}F.

performance-demanding applications and provide for a flexible and extensible Software architecture. The reconfigurable Ethernet switches are responsible for maintaining and optimizing the network interconnectivity inside the cluster by rerouting the connections in real time. Other complementary hardware, such as real-time control functions, actuators, sensors and gateways for bus systems, are executed on their own dedicated, custom-tailored hardware, which are connected to the cluster and expose their signals via software services. This hardware architecture, however, falls out of the scope of this paper, and will thus not be discussed further. Instead, in the present pages the complementary designed software architecture, its framework and implementation challenges will be covered.

The rest of the paper is structured as follows: in Section II, the overall approach to the proposed system architecture is described and the formulation for its implementation is provided. The underlying important role of the isolation for the implementation of such architecture will be conferred in Section III. Section IV outlines the setup of the conducted tests, the results of which are depicted and analyzed in Section V. Finally, in Section VI, the findings of this investigation will be discussed.

II. Background

In recent decades, the size and complexity of the available software components in vehicles have been increasing dramatically [2]. So far, however, vehicles have been equipped with software configured in a static manner and dependent on dedicated hardware. Dependencies between software components are configured and validated at design-time. Subsequent changes, such as software updates or even new software components have therefore been cumbersome and expensive. The current, statically developed and configured ECU topology does not offer any practicable possibilities for dynamic changes at runtime.

In the years to come, in order to achieve the upcoming challenges of the automotive industry the size and complexity of both existing and new software components will have to grow exponentially. In addition, the safety in the vehicle will rely more and more on the efficient and uninterrupted performance of these components, whose role will be gaining importance increasingly. Thus, to be able to optimize the processing capacity available in hardware and to provide fail-safety features, the software architecture has to be reconceptualised.

A. Flexible Software Architecture

Today, many algorithms of modern in-vehicle functions primarily require high processing speeds, but are not dependent on specific surrounding hardware and can be executed on generic processors. Examples of these algorithms are multimedia applications, algorithms for image processing and geolocation services or calculations for optimal vehicle speeds and routes. Furthermore, dedicated hardware such as sensors and actuators may be exposed by a service-oriented architecture and are thus available to the software components on every computing unit.

In addition, today's users expect software in the vehicle to be easy to update and upgrade, the same way as in their mobile devices. A simpler software update capability, as illustrated in Figure 2, also ensures that vehicle manufacturers will no longer have to pay costly workshop visits or recalls, and enables them to integrate security-relevant, error-repairing or simply function-enhancing software updates with little effort. In the same way, software update capability would enable the possibility to have an application market, like those from the mobile world in which the user could download third party software, allowing also to import its business models.

To address the new demands, the project A^{3}F approaches the inclusion of flexibility and dynamism to the automotive software architecture by the adoption of available technologies used for distributed computing systems in data centers. On the one side, container-based virtualization is being tested to provide an encapsulation of the different software components and to introduce an abstraction layer between them and the hardware. Software components are thereby largely independent of the hardware used, being executed isolated inside a minimal virtualized operating system, called container, which in turn is run on a container engine. On the other hand, an orchestration tool is employed to provide container management, automating the deployment process and besides enabling the implementation of additional features that ensure fail-safe operation.

Within the research of the project A^{3}F the technologies *Docker* [3], for the container-virtualization, and *Kubernetes* [4], for the orchestration, are currently being tested. These have been chosen due to the widespread acceptance in the IT sector and the vast amount of compatible tools. Since Kubernetes is based on Docker, the adequacy test of these technologies have to be firstly and mainly focussed on the adequacy of Docker to meet the challenges. This work is

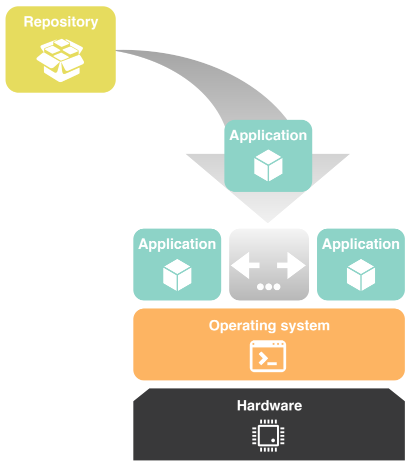


Figure 2. Layered diagram of a flexible software architecture with live software update capability.

focused in the analysis of one crucial aspect of the adequacy of the Docker technology for the automotive field.

B. Formal Description

One of the central aims of the research in the A^{3}F project is to determine which computing resources have to be taken into account when orchestrating multiple software components in a cluster-type architecture. In order to simplify the analysis, in this study every application is considered to be deployed once.

Below, a mathematical formulation for this problem (cf. [5]) is provided.

Given:

- A set of applications A:

  $A := \{a_0, a_1, \ldots, a_{N-1}\}, \, |A| = N$

- A set of computing nodes B:

  $B := \{b_0, b_1, \ldots, b_{M-1}\}, \, |B| = M$

- Each application requires a certain amount of resources:

  $a_i^{CPU}, \, a_i^{RAM}, \, \ldots \qquad \forall \, a_i \in A, \, 0 \leq i \leq N-1$

- Each computing node has a certain resource capacity:

  $b_j^{CPU}, \, b_j^{RAM}, \, \ldots \qquad \forall \, b_j \in B, \, 0 \leq j \leq M-1$

Find:

- Allocation matrix $M_{ij} \in [0, 1]$ in which $M_{ij} = 1$ if application a_i is allocated to computing node b_j.

Constraints:

- The application's resources allocated on each node may not exceed the node's capacity:

  $\sum_{i=0}^{N-1} (a_i^{CPU}) \cdot M_{ij} \leq b_j^{CPU}, \ldots \qquad \forall \, b_j \in B$

- Each application has to be allocated exactly once on each nodes:

  $\sum_{j=0}^{M-1} M_{ij} = 1 \qquad \forall \, a_i \in A$

This problem is known to be NP-hard [6]. There are several well-researched sub-optimal solutions, such as bin packing heuristics [7]. For a list and comparison of some of the algorithms, the reader is directed to [5]. However, one issue remains somewhat unanswered throughout the literature, namely which resource metrics need to be taken into account.

Most of the contributions in the literature dealing with the problem described above focus on only one type of resource (such as CPU [8] or memory [7]). In this sense, this contribution addresses the question of how strongly interferences between software components with respect to different resource types affect the runtime of these. And subsequently, the question will be examined as to how well mechanisms for performance isolation function.

III. Performance Isolation

When investigating the effectiveness of performance isolation mechanisms, the first thing to do is to find out which interferences already can be avoided with today's technologies. Therefore, two different available performance isolation mechanisms for contention of the CPU in container-based virtualization are analyzed and their impact with regard to five metrics (CPU, RAM, Cache, I/Os and Network) is measured. In the following section, first the theoretical background is explained, what we understand by interference, where it comes from and why and how to avoid it. Thereafter, the technologies that will be investigated in this research are presented together with our expectations and hypotheses for our experiments.

A. Interference

An essential aspect when operating multiple software components on shared hardware is to ensure that they are free of interferences. Interferences could be caused by, for example, contention among co-located workload for shared physical resources (such as CPU, network, and cache). It must be ensured at design-time that each application can access the required resources with the necessary frequency, with the necessary amount of time in order to guarantee an unobstructed operation. As for a single application, this might be ensured by isolating the application as if it were running on dedicated hardware. This is also known as performance isolation.

Efforts have been made to model and predict this interference by various means [7]–[9]. However, these approaches focus mainly on IT and data centre environments. For example, in [7], a distinction is made between "latency-sensitive" software components and "batch", whereby the "latency-sensitive" applications are directly user-facing and therefore have high QoS requirements, which manifests themselves in a low latency tolerance. This differentiation is mainly based on the consideration that there is a trade-off between QoS requirements and resource efficiency.

This trade-off must of course be assessed differently in vehicles than in IT due to the much more catastrophic effects that would result if QoS were not met. Although it is difficult to quantify these effects in general terms, especially at this early stage of development, one can qualitatively say that the effects of non-compliance with QoS are more catastrophic than in IT. The situation in vehicles is not foreseeable at this stage. At the same time, our development strategy is to first design a system that works conceptually. The system can then be optimized for resource utilization at a later point in time. These are reasons why this paper follows the approach that interferences between software components should never be tolerated. Instead, applications should be isolated in such a way that interferences cannot occur in the first place. In this way, latencies of software components can be reliably predicted.

To meet this goal, the use of a container engine is analyzed in this project. Container-based virtualization provides an easy way to limit the resource consumption of an application and also meets the requirements of our proposed flexible software architecture. The interference between applications running inside such a resource-constrained environment are measured. For this, we run the applications while the system is put under stress by overloading certain resources. Resource contention has to be deliberately brought in to see if the limitation works. This is done through an application that is referred to here as a "stressor".

B. Container-based Isolation

Within the research of the project A^{3}F the platform Docker is being used for isolation between software components. It allows different mechanisms for the limitation of the resources allocated to a container. As for resource isolation, Docker builds upon two mechanisms: *cgroups* and *namespaces*. Both are native Linux kernel services. Namespaces allows to create isolated virtualized system resources such as process IDs, network access, file system, etc. Cgroups provide a mechanism to limit processed resources. However, since all containers share the same host OS kernel and rely on functions within the kernel, the overhead for managing the containers increases with the number of containers. This affects and degrades the performance of the containers itself, especially for I/O-intensive workloads [10]. This problem is not limited to containers, but affects all general purpose operating systems [11]. This already highlights one big problem regarding the consolidation of containers on a computing node.

In addition, as stated in [10], any resources that do not support concurrent use will cause a major bottleneck in the host OS kernel, because concurrent access to these resources is enabled and managed by the mentioned host OS. Consequently the generation of a very large number of interrupts under such loads results in the other processes being more frequently preempted. This activity manifests itself in high CPU load and many context switches for the host OS in order to serve interrupts. Input-output (I/O) bound applications have significantly higher overhead, particularly network-intensive applications. With such workloads, the resource requirements of the host OS kernel must also be taken into account [10].

C. Hypotheses

Regarding the above effects, we had the following hypotheses for limitation:

1) The limitation should work well for applications that aren't very I/O-intensive and less well for applications with high I/O-requirements.
2) The limitation should work less well for applications that use a lot of resources that do not support concurrent access (I/O, CPU, Network).

In order to render the above mentioned effects quantifiable and thus to support the hypotheses with real data, a test battery is performed on real automotive applications. This is described in the following section.

IV. Experimental setup

To evaluate the interference of the test applications, mentioned in Section IV-B, their execution time is measured in different cases with respect to different resource metrics. The general idea is to measure the execution time of each application while overloading certain resources with our stressor applications. After the completion of the application's operations, the stressor is terminated and the execution time of the application is noted. Each of the different combinations between the four test cases and the five stressor configurations was tested 10 times by each of the four test applications, in order to obtain sufficient quantitative data.

A. Hardware Setup

For the experimentation and testing, several Intel NUC-Kits were used as generic server nodes, as well as Marvell Ethernet switches especially designed for automotive requirements. The NUCs are often employed as examples of homogeneous, powerful but generic hardware units. They are equipped with a recent processor (x86-64, 4 cores, 8 MB L3) and 32 GB of RAM and are connected to each other via redundant Ethernet network. As operating system they run a distribution of GNU/Linux, kernel version 4.15.0. This configuration shall allow individual applications to be run on any node in the cluster, independently to a great extent of specific hardware.

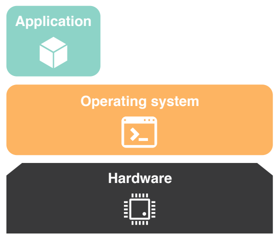
B. Application Types

In order to get an overview as close to reality as possible, four software modules from the open platform *Apollo* [12] employed to achieve the autonomous driving were tested:

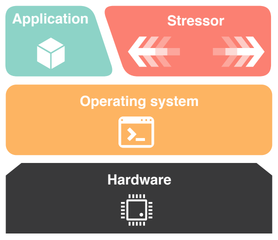
1) Perception: An image-processing application to identify road signs.
2) Planning: A GPS application to plan routes.
3) Prediction: An application predicting the trajectory of an object.
4) Controlling: An algorithm, which takes decisions based on a combination of the outputs of the above applications.

C. Test cases

The software stacks for the four different test cases are depicted in Figure 3 and 4. In case 1, as shown in Figure 3(a), the application is run without any limitations and without the stressor being executed in parallel. In case 2, the application is run without limitation, but with the stressor being executed in parallel. This is shown in Figure 3(b). In cases 3 and 4, a container virtualization layer is introduced, namely Docker, providing some means of limitation. The capabilities of limitation of workloads incorporated in the Docker engine at this time include only the CPU and the memory. However, there are two different CPU limitation mechanisms that are worth a distinct consideration. On the one side, one may specify the *CPU quota* which a container is allowed to use within one second. This kind of limitation is used in case 3, which is depicted in Figure 4(a). On the other side, one may bind a container to one or more specific *CPU core*, which is used in case 4 and shown in Figure 4(b).

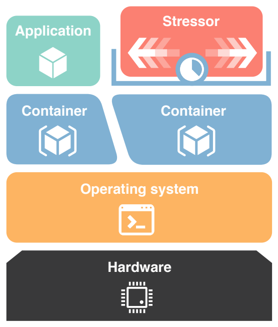


(a) Case 1: test application running without stressor.

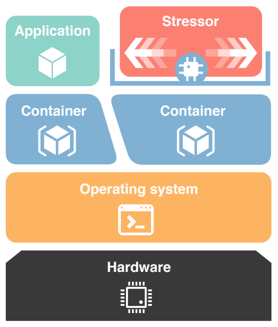


(b) Case 2: test application running with stressor.

Figure 3. Experimental cases without CPU resource limitations.



(a) Case 3: test application running besides stressor, with CPU quota limitation.



(b) Case 4: test application running besides stressor, with CPU core limitation.

Figure 4. Experimental cases with CPU resource limitations.

D. *Stress generation*

To evaluate the impact of the CPU containment into other resources, each test case was examined under five types of external stress, each one affecting a different resource between CPU, RAM, Cache, IO and Network. The application used to generate stress (the stressor) is based on *stress-ng* [13], with the exception of the one employed to generate network stress, which uses *iperf* [14]. In order to provoke the different stress environments, the stressor application was configured as it is shown in Table I.

E. *Source Code*

The source code for the four test applications can be found at [12]. The source code for all experiments conducted throughout this paper can be found at [15].

V. Results

The data collected in the tests is presented using box plots in Figure 5. The distribution of the measured time values is colored by each experimental case and classified by each stressor configuration.

The analysis of the data obtained in each of the case studies points towards the following statements:

a) Comparing the results obtained from case 1 and 2 it becomes evident that running with contention among shared resources has a severe impact on application performance.

TABLE I. CONFIGURATION OF THE STRESSOR WITH *stress-ng* AND *iperf*

Name	Stressor Configuration	Description
CPU	`stress-ng -c 25`	25 workers spinning on `sqrt(rand())`
RAM	`stress-ng --malloc 8 --malloc-bytes 2G`	8 workers exercising `malloc()/realloc()/free()`
Cache	`stress-ng -C 25`	25 workers trashing CPU Cache
I/O	`stress-ng -d 8`	8 workers spinning on `write()/unlink()`
Network	`iperf3 -c $IP -w 510M -n 510M`	send 510MB of data to a remote host (`$IP`)

b) Contrasting the non-limited cases with case 3 and 4, is clearly visible how the two analyzed limitation options offered by the platform Docker have a noticeable impact on the execution time of the applications (with the exception of the case 3 for the I/O metrics).

c) For all examined metrics, except for I/O, the CPU limitation is the first and foremost valuable kind of limitation, as all other metrics depend on the CPU resources.

d) Comparing the results obtained from case 3 and case 4 it becomes evident that running with CPU quota limitation can indeed mitigate the performance impact, but neither completely nor satisfactorily.

e) In contrast, the CPU core limitation (case 4) shows a better performance, but its execution values are still widely divergent from those of the single execution (case 1).

With regard to statement c), it is also noteworthy that I/O stress has hardly any performance impact. At this point there are two potential causes for the lower impact of the I/O stressor, which have to be considered.

- The stress generated for the I/O metric was not of the same range as the one caused by other resources. This may be due to a certain limitation of the *stress-ng* tool.
- The execution of the analyzed applications do not have intensive requirement in the I/O resources.

A preliminary analysis of the different impact into performance of the CPU quota limitation when compared to the CPU core limitation, as mentioned in the statement d), suggest two potential causes to be considered.

- The logic behind the functionality of the CPU quota limitation is probably creating in runtime a bottleneck, when the kernel / Host OS is managing CPU resources for many concurrent accesses. This confirms what we described in Section II.
- The weak performance of this CPU limitation option is caused by a bug in a quota option of the process scheduler *Completely Fair Scheduler (CFS)*, which is the default scheduler in the Linux kernel [16]–[18]. This bug appears

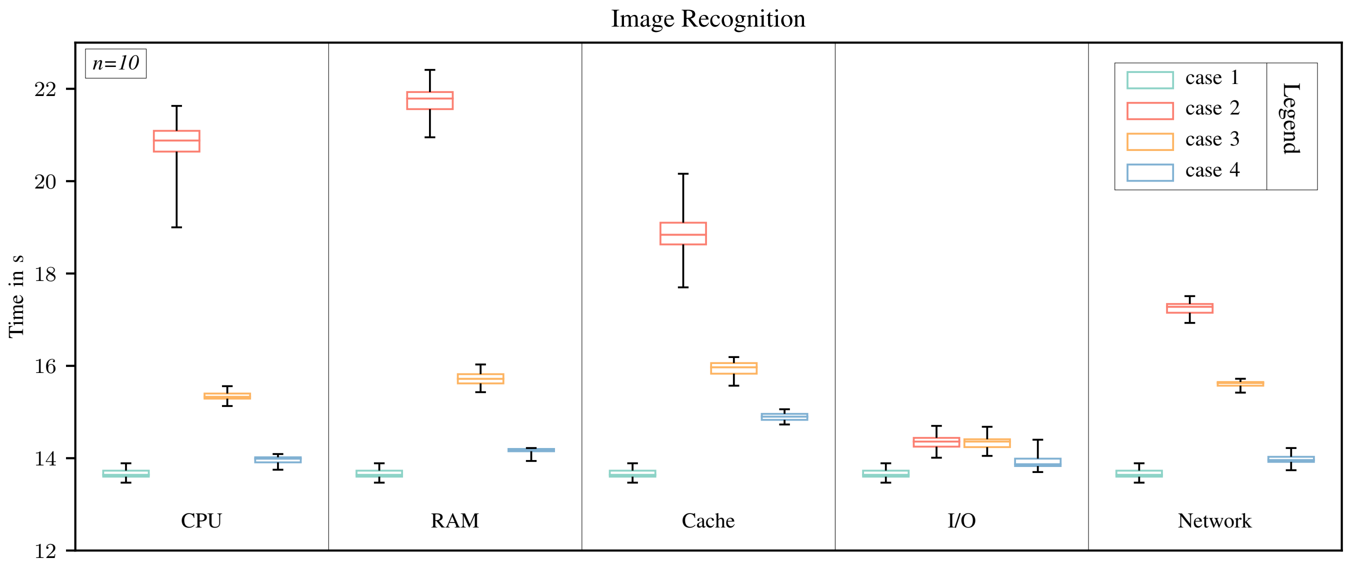


Figure 5. Test results for the Perception Application

to be fixed in more recent versions of the Linux kernel [19], [20].

These potential causes for the observed behaviour of the CPU quota limitation are uncorrelated between them, which means that despite the bug was fixed in recent kernel versions, the quota limitation could still not match the degree of isolation offered by the core limitation.

VI. Conclusions

The introduction of cluster-type architectures in vehicles demands an appropriate encapsulation of the different software components, which should abstract these components from the computing unit in which they are running on. This level of abstraction will enable flexibility and dynamism to the whole system, allowing to manage applications at runtime according to their needs. To provide this level of encapsulation the project A^{3}F studies the applicability of container-based virtualization, in particular the use of the technology Docker.

To consider the applicability of this type of virtualization in the automotive field, its fulfillment of the appropriate restrictive safety requirements has to be ensured. Safety critical systems have to be able to run uninterrupted getting access to all the resources they need.

In order to avoid having to distinguish between critical and non-critical applications and to have to proceed differently for each case, it was considered in this work that interferences between software components should never be tolerated.

As observed in test results, to be able to avoid interference when using container-based virtualization some sort of limitation of resources must be implemented. The limitation of the CPU seems to be a good choice, due to the fact that it has a direct impact on the other metrics. This appears not to be the case for the I/O resources, although it seems this could depend on test conditions. Nevertheless, a complete and fully effective containment of the available resources in the host OS cannot be provided by the Docker technology, or at least not by the time of this research.

Among the analyzed options, the usage of the CPU core limitation appears to be the best option to minimize the interferences between containers, nearing the execution times to the values obtained without stress. The CPU quota limitation, on its behalf, can only slightly reduce the interferences affecting the application but not even closely to the values obtained with the CPU core limitation. A first analysis points towards that this may be due to a bug in the internal scheduler of the Linux kernel, fixed in more recent versions. Therefore forthcoming researches in this direction to verify this hypothesis are suggested.

Although Docker seems not to be currently prepared to provide interference-free performance isolation, it becomes evident that the introduction of dynamical and distributed vehicle architectures requires container-based virtualization solutions to be implemented. Therefore, it will be necessary that the automotive industry develops its own container-based platform in order to ensure the complete isolation between the different software components, key aspect where the fulfilment of its particular safety requirements will rely on.

Acknowledgment

This research has taken place within the framework of the project A^{3}F, which is funded by the *Bayerisches Staatsministerium für Wirtschaft, Landesentwicklung und Energy* (Bavarian Ministry of Economic Affairs, Regional Development and Energy) under the funding programme *Informations- und Kommunikationstechnik Bayern* (Information and Comunication Technology Bavaria). The authors would like to acknowledge the colaboration of Ms. Cristina Alonso Villa for her work on editing this paper.

References

[1] J. Büttner, M. Kucera, and T. Waas, “Opening up new fail-safe Layers in distributed Vehicle Computer Systems,” in *Proceedings of the 8th International Joint Conference on Pervasive and Embedded Computing and Communication Systems, PECCS 2018, Porto, Portugal, July 29-30, 2018.*, 2018, pp. 236–240. [Online]. Available: https://doi.org/10.5220/0006903602360240

[2] D. Reinhardt, W. Kühnhauser, U. Baumgarten, and M. Kucera, *Virtualization of embedded real-time systems in multi-core operation for partitioning safety-relevant vehicle software (Virtualisierung eingebetteter Echtzeitsysteme im Mehrkernbetrieb zur Partitionierung sicherheitsrelevanter Fahrzeugsoftware).* Ilmenau: Universitätsverlag Ilmenau, 2016, oCLC: 951392623.

[3] “Docker,” [retrieved: July, 2019]. [Online]. Available: https://www.docker.com/

[4] “Kubernetes,” [retrieved: July, 2019]. [Online]. Available: https://kubernetes.io/

[5] J. Xu and J. A. B. Fortes, “Multi-objective Virtual Machine Placement in Virtualized Data Center Environments,” in *2010 IEEE/ACM Int'l Conference on Green Computing and Communications Int'l Conference on Cyber, Physical and Social Computing*, Dec. 2010, pp. 179–188.

[6] A. Hegde, R. Ghosh, T. Mukherjee, and V. Sharma, “SCoPe: A Decision System for Large Scale Container Provisioning Management,” in *2016 IEEE 9th International Conference on Cloud Computing (CLOUD)*, Jun. 2016, pp. 220–227.

[7] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, “Bubble-Up: increasing utilization in modern warehouse scale computers via sensible co-locations,” in *Proceedings of the 44th Annual IEEE/ACM International Symposium on Microarchitecture - MICRO-44 '11.* Porto Alegre, Brazil: ACM Press, 2011, p. 248.

[8] S. Votke, S. A. Javadi, and A. Gandhi, “Modeling and Analysis of Performance under Interference in the Cloud,” in *2017 IEEE 25th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS).* Banff, AB: IEEE, Sep. 2017, pp. 232–243.

[9] S. Govindan, J. Liu, A. Kansal, and A. Sivasubramaniam, “Cuanta: quantifying effects of shared on-chip resource interference for consolidated virtual machines,” in *Proceedings of the 2nd ACM Symposium on Cloud Computing - SOCC '11.* Cascais, Portugal: ACM Press, 2011, pp. 1–14.

[10] S. K. Garg and J. Lakshmi, “Workload performance and interference on containers,” in *2017 IEEE SmartWorld, Ubiquitous Intelligence Computing, Advanced Trusted Computed, Scalable Computing Communications, Cloud Big Data Computing, Internet of People and Smart City Innovation (SmartWorld/SCALCOM/UIC/ATC/CBDCom/IOP/SCI)*, Aug. 2017, pp. 1–6.

[11] G. Banga, P. Druschel, and J. C. Mogul, “Resource Containers: A new Facility for Resource Management in Server Systems,” in *Third Symposium on Operating System Design and Implementation (OSDI-III)*, New Orleans, LA, Feb. 1999, pp. 45–58.

[12] “Apollo Auto,” [retrieved: July, 2019]. [Online]. Available: http://apollo.auto/

[13] C. I. King, “stress-ng: a tool to load and stress a computer system,” [retrieved: July, 2019]. [Online]. Available: https://kernel.ubuntu.com/~cking/stress-ng/

[14] “iPerf,” [retrieved: July, 2019]. [Online]. Available: https://iperf.fr/

[15] “OTH Gitlab Projekt A3f,” [retrieved: July, 2019]. [Online]. Available: https://gitlab.oth-regensburg.de/IM/projekt_a3f/veroeffentlichungen/ambient2019-regular-paper

[16] I. Babrou, “Linux-Kernel Archive: Unexpected CFS thottling,” 2017, [retrieved: July, 2019]. [Online]. Available: http://lkml.iu.edu/hypermail/linux/kernel/1712.0/07072.html

[17] ——, “Overly aggressive CFS - GitHub Gist,” 2017, [retrieved: July, 2019]. [Online]. Available: https://gist.github.com/bobrik/2030ff040fad360327a5fab7a09c4ff1

[18] ——, “CFS quotas can lead to unnecessary throttling - GitHub,” 2018, [retrieved: July, 2019]. [Online]. Available: https://github.com/kubernetes/kubernetes/issues/67577

[19] D. Chiluk, “Fix low cpu usage with high throttling by removing expiration of cpu slices - LKML,” May 2019, [retrieved: July, 2019]. [Online]. Available: https://lkml.org/lkml/2019/5/17/581

[20] X. Pang, I. Molnar, P. Zijlstra, L. Torvalds, T. Gleixner, and B. Segall, “sched/fair: Fix bandwidth timer clock drift condition - GitHub,” Jun. 2019, [retrieved: July, 2019]. [Online]. Available: https://github.com/torvalds/linux/commit/512ac999