Bare-Metal vs. Hypervisors and Containers: Performance Evaluation of Virtualization Technologies for Software-Defined Vehicles

Long Wen, Markus Rickert, Fengjunjie Pan, Jianjie Lin, and Alois Knoll

***Abstract*— Software-defined vehicles (SDV) play an important role in future electrical and electronic (E&E) architectures. Their increased flexibility compared to traditional architectures is a crucial factor in the rapid development cycles of autonomous driving. Containerization and virtualization are two key technologies that enable rapid software installation and updates under the SDV framework. These two technologies have been widely adopted in cloud computing, but their performance and suitability in intelligent vehicles still has to be evaluated. In this work, we look at generic performance experiments of containerization and virtualization on both embedded and general-purpose computer systems regarding CPU, memory, network, and disk. We further investigate the impact of virtualization and containerization on the Autoware framework to evaluate scenarios that are close to real-world automotive applications. Additionally, we evaluate performance by splitting the Autoware framework into several dependent service parts, which are installed in separate containers. Extensive experimental results show that virtualization and containerization have no significant performance drop with 0-5% loss compared to a bare-metal setup in terms of CPU, memory, and network. However, both technologies suffer dramatic performance degradation on the disk side, losing 5-15% in containers and 35% in virtualization.**

I. Introduction

Due to the rapid development of automobiles, more and more functions are embedded in various electronic control units (ECUs) on current vehicles. Since software and hardware are optimized and coupled in embedded systems, the vehicle functions are difficult to be modified or updated on demand, as in mobile phones. Furthermore, the increasing number of recalls for cars is primarily caused by software issues [1] and only the original equipment manufacturer can fix these bugs [2]. The concept of software-defined vehicles (SDV) is introduced to increase the flexibility of software deployment and maintenance. The SDV provides a scalable and flexible solution utilizing domain, zone, or centralized architecture [3]. In such systems, automotive software is independent of the underlying hardware. Therefore, software installation and upgrades can happen frequently, changing the overall functionality of the system.

Virtualization and containerization are two fundamental technologies for achieving maximum flexibility and scalability of SDVs [4]. Fig. 1 illustrates the difference between virtualization and containerization. In cloud computing services, virtualization and containerization are heavily used in

L. Wen, M. Rickert, F. Pan, J. Lin, and A. Knoll are with Robotics, Artificial Intelligence and Real-Time Systems, School of Computation, Information and Technology, Technical University of Munich, Munich, Germany. {wenl, rickert, panf, jianjie.lin, knoll}@in.tum.de

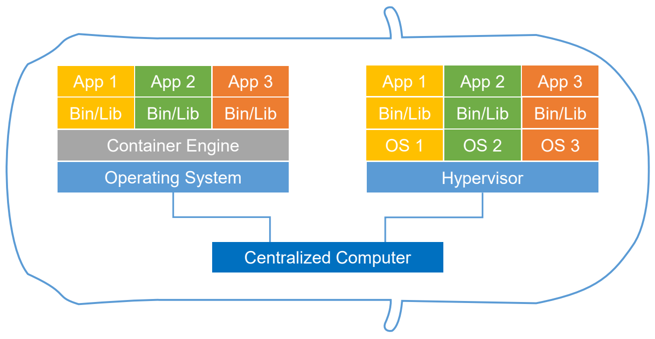


Fig. 1: Difference between virtualization (right) and containerization (left) in SDV.

server management due to the abundance of computing resources and the lack of significant difference in response time between 1 ms and 1 s for web pages. These two technologies can also be applied to software-defined vehicles, however cars cannot stack resources infinitely like servers due to the capacity of integrated chips. We therefore need to focus on the performance of these two technologies with limited resources, especially in relation to security-related software. Even though virtualization and containerization have been widely promoted in cloud computing, no benchmark reports are available for evaluating virtualization and containerization in an automotive context.

In this work, we aim to provide a reference for developers when searching for virtualization and container technologies with the best performance. We present several benchmarks of hypervisors and container engines in automotive-related scenarios. The experiments are conducted on platforms including an embedded system, a desktop computer, and a high-performance workstation. Our experiments start with a general performance evaluation with selected benchmark tools regarding CPU, memory, disk, and network. To further analyze automotive use cases, we investigate the performance of the Autoware framework (Autoware.Universe) in virtualized and containerized environments. Autoware is an open-source software stack built on top of the Robot Operating System (ROS) and offers various autonomous driving applications. In application-related tests, we recognize the application's startup time as the key performance indicator since automotive applications often demand short (re-)start times. Furthermore, we separate the complete Autoware framework into multiple dependent container segments to demonstrate the flexibility of container-based software maintenance. Any software segments can be (re-)started/updated in such a software structure. In our benchmark experiments, virtualization and containerization show a performance comparable to bare metal.

II. Related Work

Various research works have discussed the performance of virtualized/containerized environments. Giallorenzo et al. [5] presented an overview of state-of-the-art hypervisors and container engines. They also provide a collection of reference benchmarks on high-performance computers. Xavier et al. [6] conducted a performance evaluation of containerization for high-performance computing. They discussed the trade-off between performance and isolation in both containerized and virtualized environments. Felter et al. [7] focused on the hypervisor KVM and the container engine Docker for cloud computing. Morabito [8] evaluated the performance of Docker on single-board computers, including Raspberry Pi and ODROID systems, in the context of the Internet of Things. Raho et al. [9] showed the system and I/O-related performance of KVM, Docker, and Xen on an ARM-based platform. In these works, the performance of virtualization and containerization was discussed regarding four aspects: CPU, memory, network, and disk I/O performance. They showed that systems under virtualization and containerization had acceptable performance compared with the bare-metal scenarios.

Nowadays, the utilization of hypervisors and container engines in SDVs is actively discussed. According to Sundar et al. [10], hypervisors can allocate resources to separate virtual environments (VMs) and, thus, can be utilized for mix-critical applications in vehicles. With isolated VMs and containers, different applications can be deployed on the same physical machine without interfering with each other. While SOAFEE [11] is investigating the safety feature of VMs and containers, there is no benchmark report available discussing the runtime performance of virtualization/containerization technologies for automotive applications.

III. Generic Performance Benchmark

In this section, we present a general performance evaluation of virtualization and containerization tools regarding CPU, memory, disk, and network on different platforms.

A. Experimental Setup

We utilize the benchmark tools Whetstone [12], Dhrystone [13], and Kcbench [14] to perform CPU-related tests. The memory performance is tested via the tool RAMspeed and network-related experiments are conducted via iPerf3. Dbench and Bonnie++ are employed for Disk I/O evaluation. The functionality of each benchmark is described in the following section. Various hypervisors and container engines are evaluated, including Docker, KVM, Podman, and Systemd-Nspawn. The experiments are performed on three platforms with different disk configurations: Embedded (Raspberry Pi 4 Model B), Desktop (Dell Optiplex 7040 PC), and Workstation (a high-performance custom workstation). Table I and II present more details of platform configurations and software. Disk 1-4 in Table I is the list of disks we have used on the Workstation in the bus interface benchmark test, the Workstation is using a Western digital disk in the other tests. We selected different types of disks to achieve the best performance for each platform. Since our results are expressed as percentages compared to the bare-metal reference.

TABLE I: Configuration of platforms used in experiments.

	Embedded	Desktop	Workstation
	Raspberry Pi 4B	Dell OptiPlex 7040	Custom
Kernel	5.15.0-1013-raspi	5.15.0-46-generic	5.15.0-46-generic
CPU	ARM Cortex-A72	Intel i5-6500	Intel i9-12900K
Cores (HT)	4(4) 1.5 GHz	4(4) 3.2 GHz	8(16) 3.2 GHz 8(8) 2.4 GHz
RAM	4 GB LPDDR4 3200 MHz	8 GB DDR4 2133 MHz	32 GB DDR5 5200 MHz
Disk 1	Crucial MX500 SATA 250 GB	Crucial MX500 SATA 250 GB	Crucial MX500 SATA 250 GB
Disk 2	-	Samsung 980 Pro M.2 500 GB	Samsung 980 Pro M.2 500 GB
Disk 3	-	-	Intel D7 P5520 U.2 1.92 TB
Disk 4	-	-	WD Black SN850 M.2 1 TB
PCIe	-	3.0	4.0

TABLE II: Versions of all software used in experiments.

Benchmark tools	Version
Bonnie++	2.00
Dbench	4.00
Dhrystone	2.2a
iPerf3	3.9
Kcbench	0.9.5
RAMspeed	3.5.0
Sysbench	1.0.20
Whetstone	1.2

Virtualization tools	Version
Docker	20.10.17
k3s	v1.25.3
KVM	6.2.0
Podman	3.4.4
Systemd-Nspawn	249.11

The configurations of the KVM hypervisor are identical on all platforms. On each platform, we create a single VM allocated with all GPU cores and RAM. For the virtualization of disks, we select the “native” I/O mechanism and set the cache to “none”. In addition, the disk performance is influenced by the format of disk images. There are three formats of disk images: raw, qcow2, and direct use of a partition as a disk image. Our initial evaluation, visualized in Fig. 2, demonstrates that the partition format outperforms the others on all platforms.

Thus, we employ the partition disk format in the following experiments. During the evaluation of containerization technology, default settings of container engines are utilized.

B. CPU Performance

1) Benchmark tools: Whetstone can be used to evaluate float-point operations performance and was first proposed in Algol 60 in 1972. We evaluate Whetstone 50 000 000 times to acquire stable and meaningful results. Dhrystone is a synthetic computing benchmark program for emphasizing the integer performance of the CPU that outputs the number

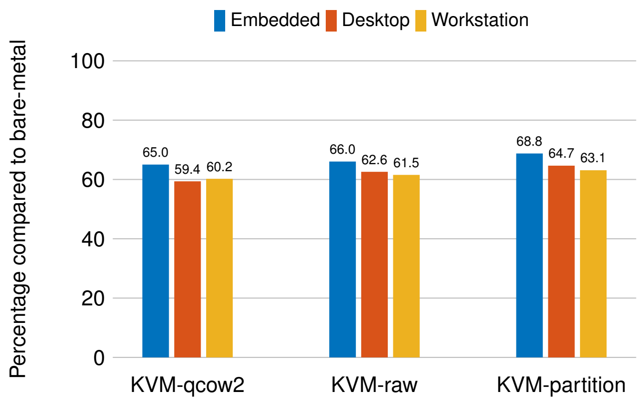


Fig. 2: Performance comparison between disk image formats.

of iterations of the main code loop per second. Kcbench is a benchmark script used to compile a Linux kernel a few times in a row and measure the compilation time. The results of Kcbench reflect the CPU's overall performance reasonably well. The source code of the Linux 4.11 kernel is compiled in the experiments. The default number of jobs is two times the number of CPU cores, however this needs to be adjusted according to different CPUs. In our case, the CPUs of the first two platforms have four cores, while the Workstation has eight performance cores (with 16 threads) and eight efficient cores. To achieve the best performance, the number of jobs is set to 4 and 24 in our experiments.

2) Benchmark results: The results are listed in Table III. The numbers in the table specify percentages compared to the bare-metal reference of the current platform. The close number on different platforms does not mean they have similar performance. As an example, the absolute numbers for Kcbench are: 2.68 kernels/hour for Embedded bare-metal, 17.58 kernels/hour for Desktop bare-metal, and 122.38 kernels/hour for Workstation bare-metal. The CPU performance gap between the three platforms is significant, as is the memory and disk I/O performance gap. These four technologies perform nearly identically in terms of float-point and integer performance compared to bare metal, with the Workstation suffering the least performance loss. One interesting aspect is that Podman performs even better than bare-metal in Whetstone on Embedded. The reason could be the different Debian images used in the container and bare-metal tests. There is no *Ubuntu for Raspberry Pi* container image, therefore a standard Ubuntu image is used instead. According to the Kcbench results, KVM has the most performance loss among these technologies on all three platforms. Overall, Embedded has a greater performance loss than the other two platforms, while the performance loss of Podman remains similar during the whole CPU experiment. Our conclusions are similar to that of Giallorenzo et al. [5] (using a high-performance computer) and Morabito [8] (using a single-board computer).

C. Memory Performance

1) Benchmark tool: RAMspeed measures a computer system's cache and memory performance. We use version 3.5.0 for multiprocessor machines running UNIX-like operating systems [15]. The read and write speed for integer and float-point operations are collected by averaging the result over five runs.

2) Benchmark results: Table IV shows that containerization has little performance loss on memory, while KVM results in more performance loss than container engines. On the Embedded platform, KVM loses more than 10% of bare-metal memory performance.

D. Network Performance

1) Benchmark tool: iPerf3 is a tool for active measurements of the maximum achievable bandwidth on IP networks [16]. Both the TCP send and receive performance should be tested; the whole duration of each test is set to 180 seconds for a stable outcome. The bitrate limit is disabled, and the first 10 seconds of the test are omitted to skip the TCP slow-start period for the best result.

2) Benchmark result: Table VI contains the results of KVM since the container engines show stable performance (100%) on three platforms. The KVM slightly influences the network performance on these platforms.

E. Disk Performance

1) Benchmark tools: Dbench is a tool to generate I/O workloads to a filesystem or network [17]. It uses a load file to generate workloads. A load file defines a sequence of operations, including open, read, close, etc. The default load file, which performs a list of create, write, and save work, is used in the experiments. Moreover, we apply a second tool Sysbench, which is used to enable and disable the fsync function. In addition, we also evaluate the tool Bonnie++, which is another file system benchmark tool that can test the input, output, delete and create performance [18]. The number of files for creation and deletion is set to 512, and we also set the random seed to a fixed one. Thus, Bonnie++ will perform the test identically on each run.

2) Benchmark results: Similar to Kcbench, the result of Dbench is influenced by the number of processes. We use 4 processes for Embedded and Desktop, and 24 processes for the Workstation. The results are listed in Table V. We notice that Nspawn has the best performance among these technologies, with nearly no difference (99%) compared to bare-metal. KVM shows the biggest performance loss (35%) while Podman and Docker are in the middle (6-15%), which matches the existing results.

In addition, we discuss the influence of synchronization (fsync) on disk performance. As shown in Table I, we test four disk types and visualize the results in Fig. 4. Sysbench is used in this experiment to get an intuitive result. The results show that applying fsync will significantly decrease the performance of the first three disks. Programs will use fsync to ensure that after a system crash, all data up to the time of the fsync call is recorded on the disk.

TABLE III: CPU performance of four technologies on three platforms. Numbers specify percentages compared to the bare-metal reference (100 %).

	Embedded				Desktop				Workstation			
	Podman	Docker	Nspawn	KVM	Podman	Docker	Nspawn	KVM	Podman	Docker	Nspawn	KVM
Dhrystone	99.77	99.74	99.76	93.72	99.58	96.97	97.10	99.24	99.19	96.28	97.20	99.61
Whetstone	100.63	100.00	100.00	99.38	100.00	100.00	100.00	100.00	100.00	100.00	100.00	99.26
Kcbench	99.25	90.67	91.42	89.55	99.49	96.81	96.41	93.96	99.66	97.73	97.35	94.12

TABLE IV: Memory performance of four technologies on three platforms. Numbers specify percentages compared to the bare-metal reference (100 %).

	Embedded				Desktop				Workstation			
	Podman	Docker	Nspawn	KVM	Podman	Docker	Nspawn	KVM	Podman	Docker	Nspawn	KVM
Integer writing	99.32	99.73	99.92	97.10	99.52	99.85	99.57	97.70	99.96	99.95	99.69	97.69
Integer reading	99.67	99.86	99.97	86.50	99.90	99.48	99.67	93.86	98.68	99.93	99.30	94.65
Float writing	99.61	99.99	99.97	97.33	99.63	99.55	99.43	97.74	99.70	99.29	99.43	98.88
Float reading	99.84	99.62	99.62	88.90	99.90	99.62	99.24	95.94	99.79	99.41	99.85	96.72

TABLE V: Bonnie++ performance of four technologies on three platforms. Numbers specify percentages compared to the bare-metal reference (100 %).

	Embedded				Desktop				Workstation			
	Podman	Docker	Nspawn	KVM	Podman	Docker	Nspawn	KVM	Podman	Docker	Nspawn	KVM
Bonnie++ output	85.85	85.85	96.23	92.45	89.62	88.08	96.73	95.58	89.14	87.29	96.94	94.76
Bonnie++ input	89.34	90.07	100.00	96.69	90.22	90.80	97.44	96.86	90.95	89.27	99.61	95.55
Bonnie++ create	62.72	59.21	97.04	90.59	65.71	63.40	94.63	93.73	67.18	63.08	96.42	90.27
Bonnie++ delete	88.27	84.79	97.71	66.66	76.76	71.80	98.82	63.01	76.13	72.02	97.47	64.32

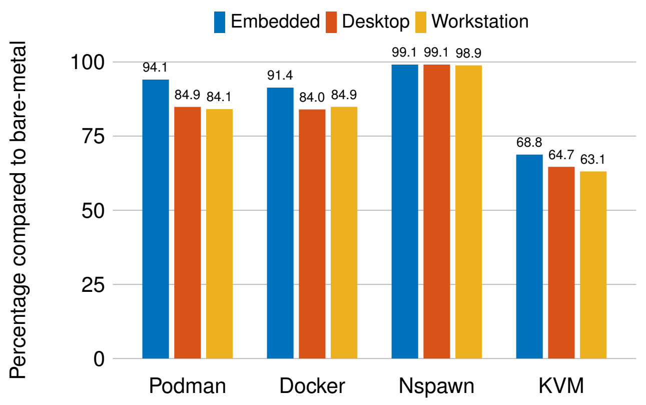


Fig. 3: Dbench results of four technologies compared to bare-metal on different platforms.

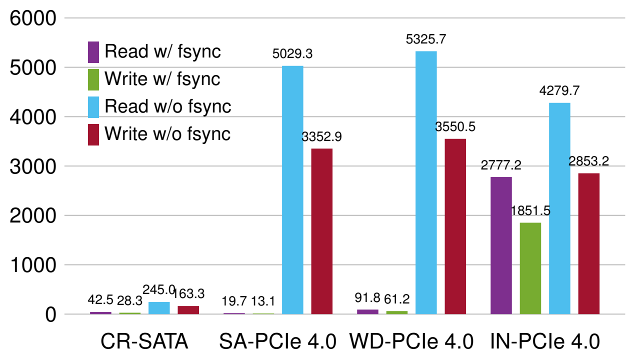


Fig. 4: Performance comparison between disks (MB/s)

Some disks, such as the enterprise-grade Intel SSD used in the experiment, have optimizations for fsync and their performance is far ahead while applying fsync. The result of the Intel disk without fsync is 15% worse than the Samsung and Western Digital disk, which means the fsync optimization is a drag on disk performance. Consequently, whether the chosen application applies fsync will affect the disk selection. Apart from fsync, the influence of the bus interface is also tested. The results in Fig. 3 are collected on Embedded (SATA), Desktop (PCIe 3.0), and Workstation (PCIe 4.0) with different bus interfaces. The performance loss on the Desktop and Workstation are nearly the same. Moreover, Embedded even has less performance loss with the SATA port except for Nspawn, although its speed is relatively slow. The conclusion is that the bus interface has few impact on virtualization and containerization; the bus interface and disk type can therefore be selected according to individual requirements.

We describe the results of Bonnie++ in Table V. Nspawn has the least performance loss among these technologies. KVM has a good performance in I/O and the create operation with only 5-10% of performance loss but experiences a significant drop in the delete operation on three platforms. Docker and Podman have the worst performance on Bonnie++, especially on the create operation (40% of loss).

TABLE VI: Network performance of KVM on three platforms. Numbers specify percentages compared to the bare-metal reference (100 %). Disk: Desktop (with Samsung disk); Workstation (with Western Digital disk).

	Embedded	Desktop	Workstation
iPerf3 send	96.67	95.31	97.13
iPerf3 receive	94.47	98.72	99.68

TABLE VII: Description of Autoware modules.

Module	Nodes	Function
Vehicle	1	Interface between Autoware and vehicle
System	12	Monitoring hardware performance and errors
Sensing	32	Collecting environment perception data
Planning	26	*Brain* of autonomous driving, decision-making
Perception	36	Processing data collected by sensing module
Map	7	Loading and broadcast maps
Location	12	Estimating vehicle pose/velocity/acceleration
Control	8	Generate motion signal to drive the vehicle
Api	62	Software interface, offering service
Rviz	1	Visualizing the application

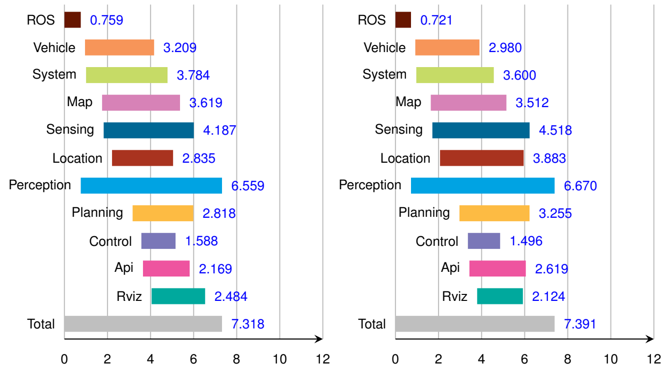
IV. Automotive Scenario Benchmark

The general performance evaluation in the previous section shows that systems utilizing hypervisor and container engines perform similarly to a bare-metal setup. This section extends our experiments with a realistic automotive scenario by utilizing the Autoware framework. In our experiments, the application startup time is considered as the performance indicator. Furthermore, we conduct experiments on mixed environments of VM and containers, e.g., multiple containers in a VM.

A. Experimental Setup

Autoware is an advanced autonomous driving framework based on ROS (Galactic) and contains 242 ROS nodes, which include all necessary functions for driving an autonomous vehicle. ROS allows communication across different devices, which makes it possible to separate Autoware into multiple containers. We select nine Autoware modules to simulate the automotive scenario, including vehicle, map, sensor, perception, location, planning, control, api, and rviz. These are split into ten containers. Additionally, we prepare a rosbag container responsible for playing back the point-cloud data recorded in a realistic scenario. Table VII describes the information and functions of these modules. The Workstation platform is selected to satisfy the high computational demand of Autoware during the experiments.

1) Choice of Disk: Applications using fsync can affect the choice of disk. However, it is unclear if Autoware utilizes fsync constantly. Hence, we select two disks, the Intel disk and the Western Digital disk, to show differences in the launch performance. As shown in Fig. 5, their performance in this scenario is identical. It means fsync optimization does not improve the startup time of the Autoware framework. The Western Digital disk therefore is a more appropriate selection, as it can achieve the highest speeds without fsync.



(a) Bare-metal+Western Digital (b) Bare-metal+Intel

Fig. 5: Comparison of Start-up time (in seconds) under disk w/ or w/o fsync optimization

2) Choice of Container: Among the container engines, Docker is chosen because Systemd-Nspawn currently does not offer GPU support and Docker has a more mature GPU support than Podman. We select k3s, a light-weighted version of Kubernetes, to orchestrate multiple containers. The version of k3s is listed in Table II, and the default container runtime is changed from containerd to Docker. We use the nvidia-docker2 package for Docker to manage the GPU and nvidia-device-plugin for k3s. KVM is configured in the following way: each VM is assigned with 24 vCPUs, 28 000 MB RAM, and 150 GB disk space. The "native" I/O mechanism and cache option "none" are set for the disk. The GPU pass-through mechanism is chosen to allow the VM to access the Nvidia GPU and the host machine uses the iGPU.

B. Start-Up Time Experiments

1) Start-Up Time in Container: This section aims to test how containerization and virtualization influence the start-up time of the Autoware framework. First, the start-up time of Autoware needs to be defined. As mentioned, the Autoware framework has 242 ROS nodes while running, including standard and composable nodes launched by Python scripts and C++ code. Using composable nodes aims to run multiple nodes in a single process with lower overhead and, optionally, more efficient communication. By editing the source code of Autoware and ROS, the start time and loaded time of different nodes can be logged. We regard the time when the "ros launch" command is executed as the start time and the time when the loading of the last node is completed as the end time. In this way, the start-up time of the whole application and that of different modules can be calculated.

Fig. 6 reports the start-up time of the whole application and different modules under four different conditions. Fig. 6c shows the results while the application is divided into ten containers and launched by k3s. As shown in the figure, it takes nearly 3.0 s for k3s to launch all containers. All modules except Rviz are launched almost simultaneously. In this figure, the bar with the k3s label stands for the time used by k3s to apply the deployments plus the time for creating containers. We will refer to it as k3s launch time in the following text. Rviz is a visualization tool of ROS. Preprocessings are needed to export visualized data in the Rviz container. Thus, the Rviz module is launched later than other modules. The perception module takes the longest time to launch, which is expected due to the GPU. It needs to load a large number of libraries for CUDA acceleration. In this case, the application takes 6.9 s to be launched.

Fig. 6a shows the results of the bare-metal setup. Unlike the results of k3s, modules are launched at different times on bare-metal. As the Autoware framework is not separated, individual modules are started one after the other instead of simultaneously. Surprisingly the startup time on bare-metal is 0.4 s longer than that on k3s, even including the time used by k3s. In order to explore this in more detail, additional experiments are needed: we put the whole application in one container and run it with Docker and k3s, as shown in Fig. 6e and Fig. 6f. The data shows that it takes 2.8 s for Docker and 2.4 s for k3s to launch one container this time. ROS's launch time is almost the same on bare-metal, Docker, and k3s for around 0.7 s. Moreover, the total time is longer than bare-metal, especially for Docker, meaning that container engines will slightly slow down the startup time. Consequently, it is the division into separate modules that makes the key difference. When the number of nodes increases, the performance of ROS will significantly decrease, and 242 is already a large amount [19]. Dividing the application into small modules with fewer nodes will reduce performance degradation. The benefit of dividing exceeds the container's overhead, thus resulting in a better performance compared to bare-metal.

By comparing Fig. 6a and Fig. 6c, a conclusion can be drawn that all modules are launched earlier with k3s. Furthermore, these modules take nearly half the time to be launched when divided into ten containers. This does not mean that k3s has much better performance than bare-metal. The main reason is that in k3s, each module has one separate container, which can focus on loading the nodes that belong to the module. We elaborate further on this with the vehicle module as an example. As shown in Table VII, the vehicle module only contains one node: robot-state-publisher, a native node ROS that can parse various parameters to link the vehicle and different sensors. In the Autoware framework, for all modules, some dependent nodes need to be loaded before the launch of core nodes. ROS is responsible for launching the vehicle module in the vehicle container. In this case, the only dependent node is the robot-state-publisher. After this node is loaded, the vehicle module begins to acquire vehicle information and links it with the relevant sensors. After this process is done, the launch of the vehicle module is complete. When it comes to the bare-metal, the task of ROS is to launch all of these ten nodes. Thus ROS has first to launch all the dependent nodes, which is a lot more than a single module has. In these experiments, we consider that the module begins to launch with its first dependent node. In other words, the launch time of modules under bare-metal also contains the launch time of dependent nodes needed by other modules. For the vehicle module under bare-metal, it has to wait for all 85 dependent nodes to be loaded after loading the robot-state-publisher node. Only after these nodes are loaded can the vehicle module begin parsing vehicle parameters. This is why modules take much longer to be launched under bare-metal, while there is no significant difference in the total launch time.

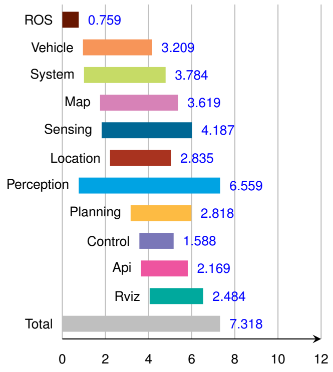
As mentioned before, automotive applications should have short (re-)start times to reduce boot time and recovery time after a software crash. It is meaningful to compare the influence of first-run and re-run. First-run is running the application for the first time after rebooting. Re-run means running the application after it is run multiple times. The results discussed before are all collected after reboot, therefore the results of the re-run are added in Figs. 6b, 6d, 6f, and 6h for comparison. Fig. 6d shows that the launch time for k3s decreased from 3.0 s to 1.5 s, and the launch time of all modules decreased. However, at this time, the ten containers no longer start at the same time, which is why the total launch time does not drop as much as the data mentioned before. The second difference is the launch time of ROS. The data in Figs. 6b, 6f, and 6h shows that the launch time of ROS decreased for around 0.3 s while utilizing all of these three technologies. This can also explain the decrease in launch time for all modules in Fig. 6d, since all containers need to launch ROS separetely inside. Docker also shows noticeable differences between first-run and re-run. Apart from the differences explained above, the launch time of the container significantly decreased for re-run, which is due to Docker's cache mechanism.

In conclusion, if the Autoware framework is not divided and runs in a single container, the start-up time for Docker and k3s will be longer than that on bare-metal. However, on first-run, Docker will be much slower. If the Autoware framework is divided into multiple containers and they are launched via k3s, the overall start-up time will be shorter than bare-metal. Our results show that containerization only has a slight influence on the application's start-up time, and it can even achieve better performance by dividing it properly.

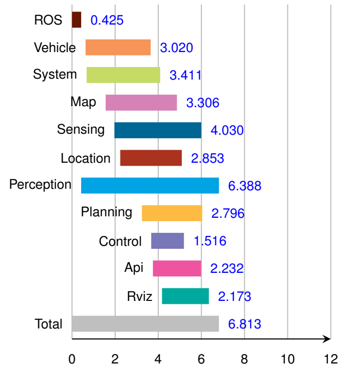
2) Start-Up Time in Virtualization: To evaluate the virtualization technology, we perform additional experiments under KVM. In addition to the previous tests, Docker and k3s performance in a VM are also evaluated. Fig. 7 lists the results under KVM, and the first discovery is that the performance of bare-metal, k3s, and Docker decreased under KVM. Table VIII listed the difference in launch time in three cases. For k3s and bare-metal, the performance loss is around 10.0%, and the re-run time is higher than first-run. The performance loss for Docker on the first-run is 19.1% under KVM, which is much higher than the other two cases.

TABLE VIII: Performance loss under KVM compared to bare-metal.

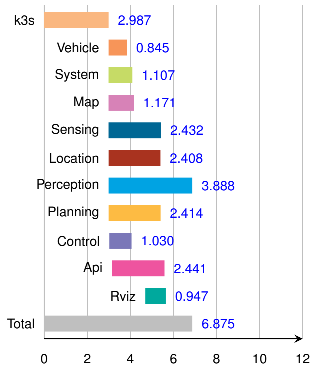
	Bare-metal	k3s	Docker
first-run	0.79 s (10.81%)	0.66 s (9.59%)	1.78 s (19.08%)
re-run	0.76 s (11.16%)	0.72 s (11.52%)	0.57 s (7.71%)



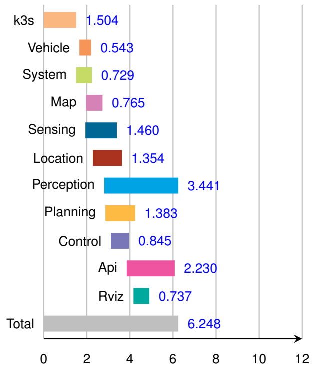
(a) Bare-metal first-run



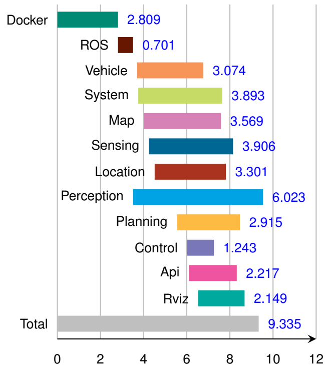
(b) Bare-metal re-run



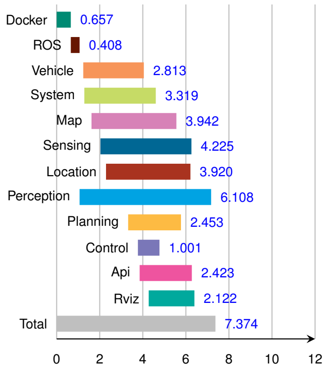
(c) k3s first-run



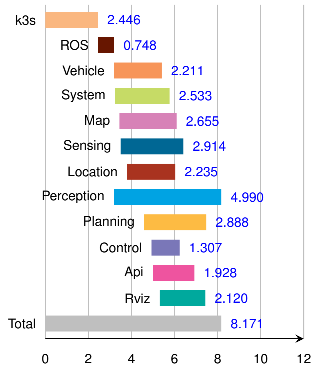
(d) k3s re-run



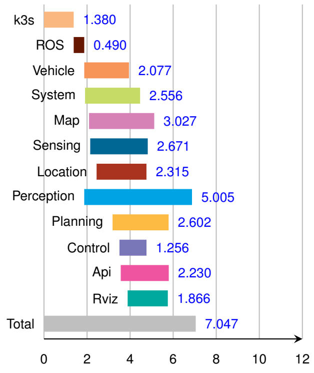
(e) Docker first-run



(f) Docker re-run

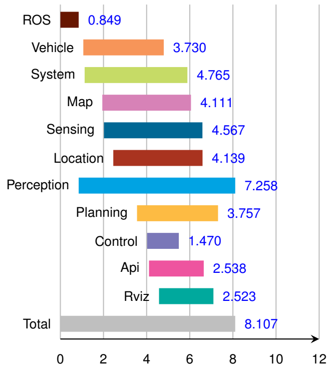


(g) k3s one container first-run

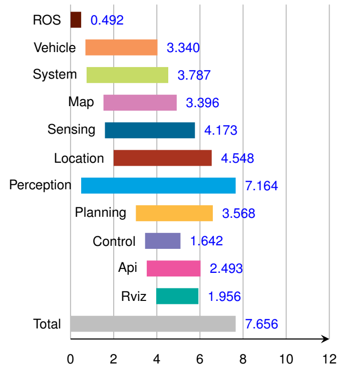


(h) k3s one container re-run

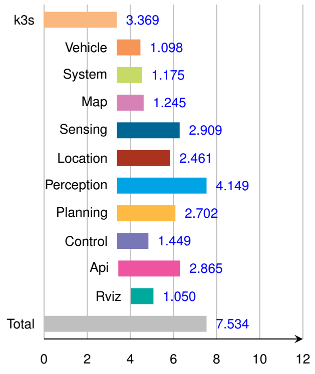
Fig. 6: Start-up time (in seconds) of modules and whole Autoware framework under bare-metal and container.



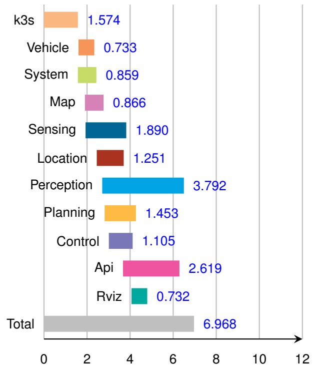
(a) KVM first-run



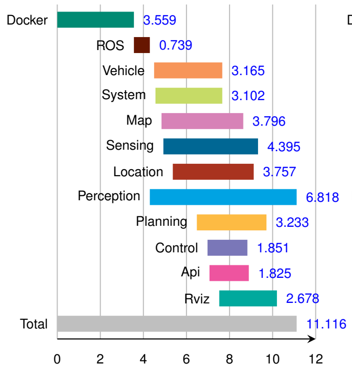
(b) KVM re-run



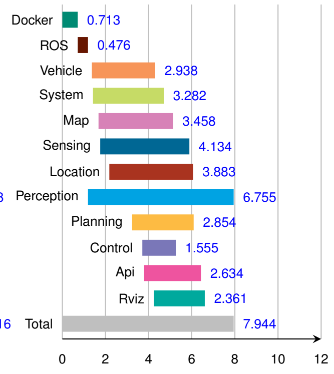
(c) KVM+k3s first-run



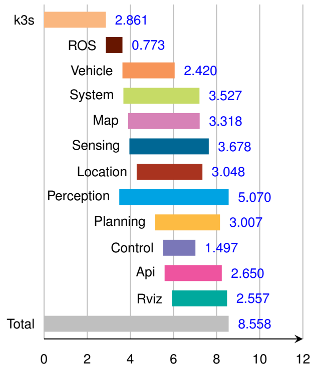
(d) KVM+k3s re-run



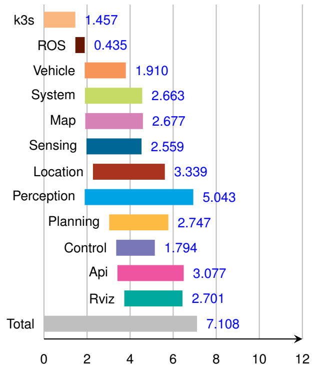
(e) KVM+Docker first-run



(f) KVM+Docker re-run



(g) KVM+k3s one cont. first-run



(h) KVM+k3s one cont. re-run

Fig. 7: Start-up time (in seconds) of modules and whole Autoware framework under KVM.

However, the performance loss on re-run is relatively low with 7.7%. Another thing worth comparing is the ROS and container launch time. Fig. 7 shows that the launch time of ROS and Docker container becomes longer on the first-run under KVM: 10.6% for ROS and 21.1% for Docker container. As for the re-run, it is 11.5% for ROS and 7.8% for the Docker container. These differences are mainly due to the disk performance of KVM. These data are similar to the total performance loss and are consistent with our conclusions in the previous section. The performance loss is acceptable (except for Docker on the first-run) if a VM outside a container is needed for better isolation and safety.

V. Conclusion

This paper presents various benchmark experiments of hypervisors and containers in the context of SDVs. Several virtualization and containerization technologies are evaluated, including KVM, Docker, Podman, and Nspawn. We consider the performance of bare-metal as the base line. Our experiments are conducted on different platforms, including a Raspberry Pi, a standard desktop, and a high-performance workstation. In the first part of our work, we perform a general performance test with selected benchmark tools regarding CPU, memory, network, and disk. Afterwards, we evaluate the performance in an automotive scenario with the Autoware framework in a VM, a container, and a container inside of a VM. Furthermore, we separate a single Autoware framework into different containers managed by the container orchestrator k3s and analyze its performance. The benchmark results show that the performance of running software in virtualized/containerized environments is comparable to the performance of bare-metal systems.

In general benchmark tests, software running in VMs and containers have approx. 0-5% decrease in the performance of CPU, memory, and network. Virtualization and containerization show a greater impact on disk performance. The container engines have approx. 5-15% performance loss on disk. The disk performance in KVM is 35% slower than that in a bare-metal setup. We further design automotive application-related experiments to measure the application's startup time in different software execution environments. The results show that startup times for KVM and Docker (excluding the first run) are 5 to 10% slower than those for bare-metal. Additionally, k3s performs better than bare-metal when Autoware is divided into nine containers. These results illustrate that virtualization and containerization are appropriate for automotive applications.

In future work, we plan to demonstrate the microservice-based architecture for a containerized Autoware framework and ROS applications and investigate the impacts on SDVs.

References

[1] N. Steinkamp, R. Levine, and R. Roth, "2021 automotive defect & recall report," Stout Risius Ross, accessed: 2023-04-14. [Online]. Available: https://www.stout.com/en/insights/report/2021-automotive-warranty-and-defect-report

[2] E. Sax, R. Reussner, H. Guissouma, and H. Klare, "A survey on the state and future of automotive software release and configuration management," Karlsruhe Institute of Technology, Karlsruhe Reports in Informatics, 2017.

[3] W. Haas and P. Langjahr, "Cross-domain vehicle control units in modern E/E architectures," in *Internationales Stuttgarter Symposium*, Apr. 2016, pp. 1619–1627.

[4] N. Jain and S. Choudhary, "Overview of virtualization in cloud computing," in *Proceedings of the Symposium on Colossal Data Analysis and Networking*, 2016, pp. 1–4.

[5] S. Giallorenzo, J. Mauro, M. G. Poulsen, and F. Siroky, "Virtualization costs: Benchmarking containers and virtual machines against bare-metal," *SN Computer Science*, vol. 2, no. 404, 2021.

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

[7] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," IBM Research Division, IBM Research Report RC25482, 2014.

[8] R. Morabito, "Virtualization on Internet of Things edge devices with container technologies: A performance evaluation," *IEEE Access*, vol. 5, pp. 8835–8850, 2017.

[9] M. Raho, A. Spyridakis, M. Paolino, and D. Raho, "KVM, Xen and Docker: A performance analysis for ARM based NFV and cloud computing," in *Proceedings of the IEEE Workshop on Advances in Information, Electronic and Electrical Engineering*, 2015.

[10] R. Sundar, F. Armin, G. Lothar, S. Idriz, and D. M. Nirmala, "Hypervisor for consolidating real-time automotive control units: Its procedure, implications and hidden pitfalls," *Journal of Systems Architecture*, vol. 82, pp. 37–48, 2018.

[11] "Development platform for SOAFEE," https://www.adlinktech.com/en/soafee, accessed: 2023-04-14.

[12] H. J. Curnow and B. A. Wichmann, "A synthetic benchmark," *Computer Journal*, vol. 19, no. 1, pp. 43–49, Jan. 1976.

[13] R. P. Weicker, "Dhrystone: A synthetic systems programming benchmark," *Communications of the ACM*, vol. 27, no. 10, pp. 1013–1030, Oct. 1984.

[14] T. Leemhuis, "Linux kernel compile benchmarks kcbench and kcbenchrate," accessed: 2023-04-14. [Online]. Available: https://gitlab.com/knurd42/kcbench

[15] Hollander, M. Rhett, and V. Bolotof, Paul, "RAMspeed, a cache and memory benchmarking tool," accessed: 2023-04-14. [Online]. Available: http://alasir.com/software/ramspeed/

[16] J. Dugan, S. Elliott, J. Poskanzer, and K. Prabhu, "iPerf - the ultimate speed test tool for TCP, UDP and SCTP," accessed: 2023-04-14. [Online]. Available: https://iperf.fr/

[17] A. Tridgell and R. Sahlberg, "DBENCH," accessed: 2023-04-14. [Online]. Available: https://dbench.samba.org

[18] R. Coker, "Bonnie++," accessed: 2023-04-14. [Online]. Available: https://www.coker.com.au/bonnie++/

[19] S. Profanter, A. Tekat, K. Dorofeev, M. Rickert, and A. Knoll, "OPC UA versus ROS, DDS, and MQTT: Performance evaluation of Industry 4.0 protocols," in *Proceedings of the IEEE International Conference on Industrial Technology*, 2019, pp. 955–962.