Testing the limits of general-purpose hypervisors for real-time control systems

Article *in* Microprocessors and Microsystems · May 2023

DOI: 10.1016/j.micpro.2023.104848

3 authors:

Rui Queiroz
University of Coimbra
5 PUBLICATIONS **16** CITATIONS
SEE PROFILE

Tiago J. Cruz
University of Coimbra
118 PUBLICATIONS **1,536** CITATIONS
SEE PROFILE

Paulo Simoes
University of Coimbra
199 PUBLICATIONS **1,616** CITATIONS
SEE PROFILE

Microprocessors and Microsystems 99 (2023) 104848

Contents lists available at ScienceDirect

Microprocessors and Microsystems

journal homepage: www.elsevier.com/locate/micpro

Testing the limits of general-purpose hypervisors for real-time control systems

Rui Queiroz, Tiago Cruz [*], Paulo Simões

University of Coimbra, CISUC, DEI, Departamento de Eng. Informática da FCTUC, Polo II da Universidade de Coimbra, 3030-290 Coimbra, Portugal

ARTICLE INFO

Keywords:
Industrial automation control systems
General-purpose off-the-shelf hypervisors
Real-time cyber–physical systems
Converged Infrastructure
Determinism
Latency

ABSTRACT

With the emergence of the Industry 4.0 paradigm, there is a need to introduce a significant degree of flexibility, security and resilience in automation infrastructures, while keeping up with real-time requirements that are characteristic of such domains. Interestingly, many of these driving principles are the same that encouraged the adoption of virtualization technologies on the IT domain, somehow suggesting that the same benefits could be realisable for Industrial and Automation Control Systems, allowing to virtualise servers and cyber–physical system control devices. However, the suitability of using off-the-shelf hypervisor technologies to address the specific real-time requirements of automation infrastructures remains unclear, due to their focus on maximising systems throughput and capacity, often at the expense of determinism and increased latency.

This work addresses this problem, presenting a discussion and an empirical evaluation on the feasibility of using general purpose off-the-shelf hypervisors to virtualise cyber–physical systems' servers and control devices. While the evaluation concludes that some of these hypervisors are already capable of dealing with typical real-time workloads, this cannot be generalised to all types of real-time systems.

1. Introduction

Real-time cyber–physical systems are embedded in the most diverse industrial environments, controlling a wide range of cyber–physical processes, such as production plants, railways, power generation and distribution, and water distribution. The emergence of Industry 4.0 paradigms such as smart factories has considerably raised the requirements in terms of resilience, security, scalability, flexibility and cost optimisation. Curiously, the same set of requirements was the main driver of the softwarization, virtualization and consolidation process that is taking place in the IT domain in general.

Actually, this virtualization trend already arrived at the Industrial Automation and Control Systems (IACS) domain, with the virtualization of less demanding components, such as Historians and other Supervisory Control and Data Acquisition (SCADA) stations with no real-time requirements. However, gradually extending this virtualization trend towards real-time components would enable considerable benefits. Cruz et al. [1], for instance, propose decoupling the execution environment of IACS control devices such as PLCs from their I/O modules, by means of virtualization. The internal I/O bus that previously established the connection between these components would be replaced by a low-latency and deterministic network using software-defined networking (SDN) enabled Ethernet fabric. All the components would be orchestrated based on the simplified programmability offered by software-defined networking (SDN), allowing for a dynamic creation and deployment of the virtualized execution environments (the virtual PLC, or vPLC), as well as the consequent routes and virtual channels necessary to accommodate the communication data flows between the different equipment (Fig. 1).

Hence, in order to validate the feasibility of such approaches, it is mandatory to perform an analysis of the distinct technologies required for their implementation, in order to identify possible developments that may prove instrumental. Among several technologies deemed crucial for this purpose, real-time hypervisor capabilities rank among the most important. While real-time hypervisors do exist, specific design limitations imposed by low-latency deterministic capabilities may prove too restrictive or inefficient for consolidation of less demanding workloads, thus making a case for studying and understanding the limits and capabilities of conventional hypervisors, whose main focus is oriented towards consolidation of IT workloads and throughput optimisation in private or public cloud data centre environments.

This paper addresses this topic by analysing the feasibility of using commercial off-the-shelf hypervisors in the context of real-time environments, namely for supporting the virtualization of cyber–physical control devices such as programmable logic controllers.

The rest of the paper is organised as follows. First, an analysis on cyber–physical systems is provided, with emphasis on their control

* Corresponding author.
E-mail address: tjcruz@dei.uc.pt (T. Cruz).

https://doi.org/10.1016/j.micpro.2023.104848
Received 6 December 2022; Received in revised form 19 March 2023; Accepted 28 April 2023
Available online 3 May 2023

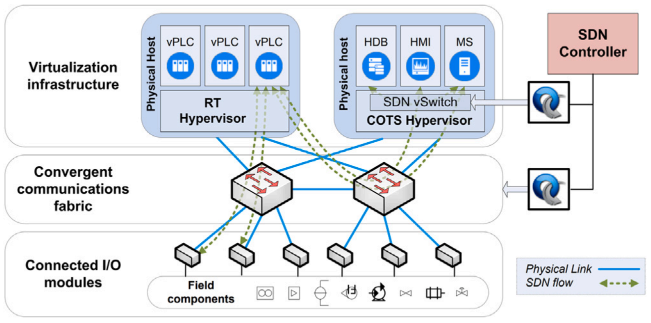

Fig. 1. The vPLC architecture — source [1,2].

devices, which are one of the main targets of this research. Next, an analysis of real-time systems is presented, highlighting their distinctive characteristics and influencing factors — within this scope, real-time operating systems (RTOS) and mixed-criticality systems are also covered. Following, a concise literary review on system virtualization is presented. Different types of virtualization techniques are discussed, as well as the differences between commercial off-the-shelf (COTS) hypervisors and real-time (RT) hypervisors, also encompassing examples of virtualization techniques applied in industrial environments. Finally, the results of an experimental analysis effort designed to assess the viability of virtualizing real-time workloads using COTS hypervisors are presented. The obtained tests results are then analysed, and the final conclusions are outlined.

2. Industrial and automation control systems

In this section, we provide a conceptual bridge between the automation and virtualization domains, in order to help the reader better grasp the challenges arising from virtualizing control equipment within the scope of cyber–physical systems (CPS) and, more specifically, within Industrial Automation and Control Systems (IACS).

First, an overview the nature and scope of IACS technologies, introducing a series of concepts, definitions and technologies which help the reader follow the rationale and evaluation efforts undertaken in this paper. Next, we discuss the specific concerns and requirements to be considered when virtualizing IACS control devices.

2.1. Cyber–physical systems

CPS [3] aim at monitoring and controlling the most diverse physical processes, by means of maintaining a close integration between those processes and the computing and communications technologies used to support the associated control systems (Fig. 2). Their presence in diverse environments, such as automotive, railways, aeronautic, smart grids, water supply, natural gas, fuel and oil transportation, manufacturing and healthcare, demonstrates the relevance of CPS in everyday life.

Among CPS, IACS play an important role in ensuring the availability and operation of essential services and critical infrastructures on a day-to-day basis. IACS manage all sorts of processes, allowing for decisions to be made based on real-time information, translating them into actions taken by the various physical actuators belonging to the manufacturing process. However, and despite being recognised for their robustness and reliability, the technologies used at the core of these systems have hardly evolved in the last decades. This is unsurprising considering that maturity was often considered synonymous for dependability and reliability. However, the emergence of the Industry 4.0 paradigm, as well as a series of security-related concerns and incidents, has pushed the industry to change the dominant mindset and become more open towards adopting evolved technologies and alternative solutions.

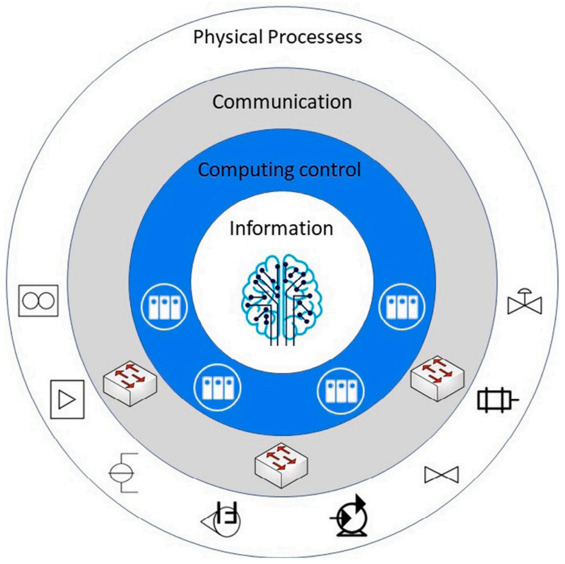

Fig. 2. Cyber–physical system.

2.2. Control devices

Control devices are one of the most important components commonly found in IACS. In general, these are embedded systems composed of a processing unit and I/O modules, and can operate in standalone mode or as part of a distributed topology. Their importance stems from the fact that they are responsible for processing data coming from (but not only) the physical part of the system and making decisions accordingly. These decisions define the CPS behaviour and determine the efficiency and stability of the entire system. Poor functioning or failure of a control device can have relevant consequences, ranging from system downtime to negative environmental impact or even loss of human life, among others. Therefore, it is of utmost importance that these control devices have a high level of availability and security.

Moreover, because they are physical devices and are often spatially dispersed, commissioning and maintenance are quite costly and time consuming. To scale the system or guarantee redundant solutions poses the same problems, since it is mandatory to acquire new equipment and perform new commissioning.

On the other hand, the concept of smart factory implies the existence of a flexible infrastructure, capable of adapting itself to different realities according to the needs of the moment. This may lead to a control device having to be dynamically deployed in the field, for reasons of criticality, at a given point in time. However, in the short term, its goal can be radically changed, having no reason to be deployed in the field and, on the contrary, benefiting from being located in a data centre where it may take advantage of having access to more computing resources. Also, the more physical elements are present in the CPS, the less flexibility it will have and the more complex it will be to adapt itself to a new reality.

Having these problems in mind, and noting the resemblance to some of the problems that existed in the data centres and which were solved through the virtualization of physical devices and network functions and, recently, complemented with the use of software-defined networking (SDN) technology, one wonders if it would be possible to apply similar techniques on cyber–physical system control devices, in order to deal with the aforementioned problems.

In real-time cyber–physical systems, regardless of whether they are mixed-criticality systems or not, there are typically one or more hardware control devices responsible for the automation processes. In IACS scenarios, such devices may include remote terminal units (RTU),

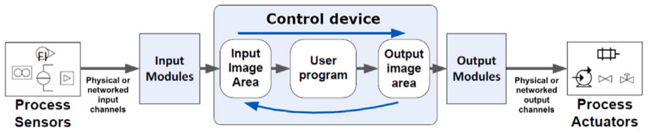

Fig. 3. Control device cycle.

programmable logic controllers, programmable automation controllers (PAC), and process control units, spread throughout a distributed control system (DCS). Often some terms are used interchangeably in the industry due to a lack of a common accepted definition and/or specification, as it is the case for the distinction between sophisticated RTUs and PLCs.

These are embedded systems, based on microcontrollers or microprocessors complemented by peripheral circuits, running real-time operating systems responsible for the execution environment of the main functions and services, with communication capabilities like serial point-to-point, bus topologies or Ethernet and TCP/IP. There are two common types of equipment: compact and modular. In compact equipment all the hardware is built-in and fixed. The modular equipment are installed in a special chassis which allows expansion modules (e.g. I/O modules) to be added *a posteriori* by means of an internal communication bus. Nonetheless, all of them share the same functional basis, as depicted in Fig. 3. Data is received from sensors, network communications or other data sources, to the control device. While performing the scan cycles, this event is detected and the control device processes the incoming data, according to the pre-programmed application present in the execution environment. A response to the event is then outputted, in the form of commands to physical actuators and/or network communications to other components of the CPS.

Determinism and predictability play an extremely important role throughout this process. Therefore, it is essential that both the hardware and the adopted internal mechanisms are able to meet the imposed real-time requirements. For this reason, all the control device internal data (application, input data and other auxiliary information) is kept at the memory level (RAM and ROM). Also, real-time operating systems are used to guarantee the best real-time performance. Normally, two types of coding are considered: cyclic code and interrupt code. The first one runs continuously in periodic cycles. Although the time to run the cyclic code from start to finish is theoretically constant, in practice it may vary due to the existence of other factors such as interrupts, whose number and frequency can greatly increase the cycle time. On the other hand, the interrupt code only runs when an interrupt is raised, allowing the system programmer to tie a routine or code section to a specific type of interrupt. This technique can be useful in order to guarantee a timely response to an event.

3. Real-time systems

A real-time system, by definition, is able to respond to an event or execute an action within a specified time boundary. This boundary is called a deadline, and usually comprises only a few milliseconds or even microseconds. Within that boundary, it is expected for such a system to be able to receive data from the surrounding environment, process it and, if needed, trigger some sort of response. The capability to respond correctly and in a timely manner is closely related to the system's determinism and predictability. By definition, one knows that a deterministic system involves no randomness: given the same initial state, with the same starting conditions, the produced output will always be the same.[1] A predictable system should always produce results in between the same timeframe. Often, these two concepts are merged and simply referred to as determinism.

Real-time systems can be divided into three distinct categories, according to the impact of failing to respond within a specific time boundary (Fig. 4):

- Hard real-time (HRT) — The inability to meet a deadline results in a system failure. Responses following the deadline are automatically devoid of value.
- Firm real-time (FRT) — Deadlines can be infrequently missed without causing a system failure, however, there may be a degradation in the quality of service. Responses following the deadline are automatically devoid of value.
- Soft real-time (SRT) — Deadlines can be infrequently missed without causing a system failure. Responses following the deadline are still considered, even though there may be a degradation in the quality of service.

The time set for the deadlines depends on the system and context in question, and it is the designer or programmer's responsibility to set it accordingly. While there is no "one size fits all" value, there are indicative values for the end-to-end latency [4], as shown in Table 1.

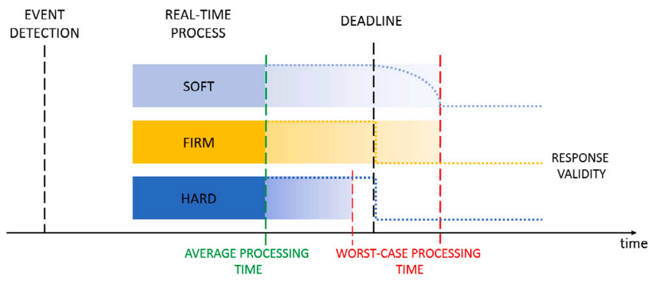

Fig. 4. Real-time systems categories.

Table 1
Industrial systems end-to-end latency.

Service	End-to-end latency	Jitter
Factory automation (motion control)	1 ms	1 μs
Factory automation	10 ms	100 μs
Process automation (remote control)	50 ms	20 μs
Process automation (monitoring)	50 ms	20 ms
Electricity distribution (medium voltage)	25 ms	10 ms
Electricity distribution (high voltage)	5 ms	1 ms
Intelligent transport systems (infrastructure backhaul)	10 ms	2 ms
Remote control	5 ms	1 ms

3.1. Latency

Several factors can contribute to latency. The designation "end-to-end latency" usually refers only to the time delay inherent in communicating data from its source to its destination. This metric is often used to define the maximum delay accepted by a certain service or system. This is so because it is often, and incorrectly, assumed that ensuring timely delivery of data packets is enough to guarantee that the systems' latency requirements will be achieved. However, when we analyse the delay tolerance of a real-time system, we must consider not only the delay related to the communication network, but also the delay at the processing level. This is due, not only but also, to the order of magnitude of the admitted latency being very low. As such, the processing latency cannot be discarded. The sum of both, the network and the processing latency, must be below the maximum value allowed by the system. Therefore, it is of utmost importance that the processing devices present in this type of systems withstand operating

[1] Which does not mean that, in practice, a deterministic system cannot run algorithms including random actions.

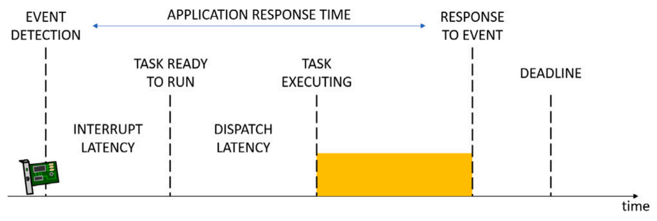

Fig. 5. Processing latency.

cycles that allow the fulfilment of these restrictions, as well as the communication network. Two sorts of latency must therefore be taken into consideration: network latency and processing latency.

Network delay comprises the total delay between a signal being sent from the point of origin to the point of destination. If we consider a packet-switched network, the network latency is a sum of multiple delays that may arise from different sources and are designated accordingly, such as:

- Propagation delay — the time required for a packet to travel from the sender to the receiver.
- Transmission delay — the time required to push an entire packet into the communication channel. Its value can be estimated considering a function of the packet's length over the transmission rate. Normally, the specifications of the hardware being used define the transmission rate.
- Network processing delay — the time required for a node to process a packet and be able to check for errors and determine its next destination.
- Queuing delay — the time that a packet spends in a queue waiting to be processed or transmitted. In addition to the specifications of the hardware and its QoS configurations, network overload is the one of the main influencing factors.

When the packet delivery takes place, the network device rises an interrupt so that the system becomes aware of this event. The real-time process (usually referred to as RT task) responsible for dealing with this event has first to be scheduled, then processed, and finally a response to the event is achieved. Thus, processing delay occurs after the packet is delivered to a real-time system control device. It is possible to identify different sources of delay in the processing latency (Fig. 5), among which two stand out:

- Interrupt latency — when the interrupt is raised, the system may not be available to handle this interrupt at this exact time due to circumstances that may be locking out interrupts. Also, actions like having the processor saving the state of execution, and the interrupt processing itself, add extra delay to the process.
- Dispatch latency — after the interrupt being handled, the RT task becomes ready to run and is scheduled for processing according to the scheduling policies. This dispatch process generates delay caused by context switching, scheduling, dispatching, among other conflicts that may arise in the process.

Both the interrupt and the dispatch latency depend on the application domain for which the operating system was developed for. For example, if the OS goal is to prioritise throughput, the kernel scheduler will probably apply a non-preemptive policy which will increase the dispatch latency. If the OS wants to deal with mutual exclusion problems and to assure that only one process is executed in a critical region at a time, it may increase the maximum time that interrupts can be disabled at the cost of increasing interrupt latency.

3.2. *Jitter*

As mentioned above, predictability implies producing results always within the same time boundary. Therefore, it is essential to ensure that the latency variation – also referred to as jitter – is as small as possible and within the RT task restrictions. The lower the jitter, the higher the predictability of the system.

In real-time systems, having a contained latency is just as (or even more) important than having a low latency. As presented in Table 1, the jitter value is always more restricted than the latency value. A response being produced earlier than expected can result in synchronisation problems, but a response being produced later than expected can result in its invalidation. Either case may lead to a critical system failure (depending on the type of RT system).

3.3. *Power management c-states, p-states, t-states*

One of the main factors that influences the mean time to failure (MTTF) of a microprocessor is its thermal operation envelope. Operation in excess of nominal characteristics may lead to problems such as electromigration and thermal runaway. To deal with this, modern CPUs incorporate efficient control over its power consumption in order to control its thermal load. It is also known that one of the great challenges related to electronic equipment, namely computers, is the reduction of its energy consumption. To face these challenges, processors started to include power management techniques such as Dynamic Voltage and Frequency Scaling (DVFS). For instance, for the Intel x86 family, such mechanisms encompass throttling states (T-states), idle or processor power states (C-states) and operational or performance states (P-states):

- T-states were the first to be used, specially to prevent damage to the processor. The power manager places the processor in different T-states according to its temperature — the higher the temperature the higher the t-state. This technique would slow down the execution of the running tasks by suspending them for one or more clock cycles, allowing the processor to de-stress and cool down.
- C-states enable the shutdown of the processor sub-systems to save energy, and are used mainly when the CPU is idle. The higher the state (C1, C2, …, Cn) the more circuits and signals are turned off and the greater is the power saving. Nevertheless, the time needed to return to the fully functional C0 state also increases. These states can be divided in sub-types like Core C-states (CC-states), Package C-states (PC-states) or logical C-states (OS C-states or LC-states).
- P-states allow the control of the thermal load and power consumption of the CPU by scaling the input voltage and the frequency at which the processor runs. The higher the state (P0, P1, …, Pn) the lower the voltage and the frequency, which translates into a lower processing performance but higher power saving and lower working temperatures.

The type and range of operation states depends on the processor, and even processors from the same family can have variations. However, the advantages associated with these techniques come at a cost: not only higher states imply response latency degradation, but also swapping between them can negatively affect determinism. As such, the same techniques that may extend the semiconductor lifespan, as well as its MTTF, will also negatively degrade the system's ability to fulfil real-time requirements. For this reason, this type of techniques is generally not recommended for use in real-time systems.

3.4. *Mixed-criticality systems*

The combination between the availability of multi-core microprocessors with reasonable processing power, with the need to reduce size, weight, and energy consumption of electronic systems, led to a new approach to critical systems. Referred to as mixed-criticality system (MCS), it optimises the use of hardware through the consolidation of distinct systems, with different criticality levels (e.g., real-time control tasks versus logging information), on the same hardware platform.

The criticality level of a system is normally set according to risk classification systems such as Safety Integrity Level (SIL), Automotive Safety and Integrity Level (ASIL) or Design Assurance Level (DAL), as seen in IEC 61508 and ISO 26262 standards or in DO-178C/ED-12C and DO178B/ED-12B guidelines. The risk classification defines the level of assurance that must be provided against system failures.

MCSs are often seen in avionics, aerospace, and automotive systems. Although a MCS does not necessarily imply the existence of real-time requirements, RT requirements are typically present. The merging of distinct requirements under the same hardware raises several challenges, such as the balance between sharing resources (to optimise the hardware usage) and the partitioning for security reasons, or the ability to comply with RT requirements. Above all, such a system must be able to decide, at each moment, which task should be executed based on its criticality level.

4. Virtualization of real-time systems

In this Section we address the virtualization of real-time systems. First, we present and discuss the specific RTOS requirements and characteristics, followed by a review of several system virtualization technologies, with a focus on the challenges introduced by virtualization of RT execution environments. Finally, the application of virtualization techniques to cyber–physical systems will be addressed, also identifying relevant use cases.

4.1. RTOS

To guarantee system determinism and predictability, real-time operating systems are commonly used across CPSs' control devices. This type of OS continuously delivers a high level of consistency which strongly depends on factors like interrupt latency, thread switching latency, memory allocation and scheduling mechanism. These assume a relevant role in the architecture of RTOS and allow them to present stable bounded output latencies.

Well-known open source RTOS include eCOS, FreeRTOS, μC/OS-II, Erika3, as well as some descendants of the L4 microkernel family (such as FIASCO/L4Re or seL4). Also, Xenomai (Cobalt) allows a general-purpose Linux distribution to be transformed into a RTOS by means of a co-kernel. The same happens with the Real-Time Linux project when applying the Preempt_RT kernel patch, which is often mentioned in the literature. For instance, Wang et al. [5] showed how to improve the latency of a vanilla Linux by using RT-Linux, making it suitable for real-time performance. After a brief description on the architecture of RT-Linux kernel principles, the authors explained how to patch a general-purpose Linux distribution — Ubuntu 18.04.1. Using the cyclictest tool, some tests were performed to measure the CPU latency while having a different number of threads running in parallel. By comparing the results obtained from a vanilla installation and from a RT-Linux patched installation, it was concluded that the real-time performance can be improved by applying the RT-Linux patch.

Within commercial RTOSs, one can mention QNX and VxWorks as examples of a long list of existing products [6]. Serino and Cheng [7] presented a brief theoretical comparison between some of these RTOSs. Special attention is paid to the main differences between general purpose operating systems (GPOS) and RTOS. Kernel differences are discussed, focusing on the preemption capability, and distinct scheduling techniques used by the two types of systems are explained (e.g., round robin, first-in-first-out, rate-monotonic and earliest-deadline-first). The priority inversion issue is also addressed and some solutions are presented, such as priority inheritance, priority ceiling, priority remapping and priority exchange. The merge between GPOS and RTOS using techniques like patched kernels or multi-kernels is also discussed. Lastly, an introduction on worst-case execution time tools is provided, followed by a suggestion to further explore multicore RTOS.

The evolution of microprocessors brought multicore architectures to real-time devices, which raised another challenge for RTOSs: how to get the most out of all available cores in a multicore environment without affecting RT requirements, especially in HRT scenarios. Static allocation of tasks to specific cores is often used to guarantee that HRT requirements are achieved. However, this technique may reduce performance when it comes to taking advantage of the overall installed processing power. On the other hand, full core migration allows the scheduler to relocate tasks at arbitrary times between cores, thus taking full advantage of all the available cores. Yet, core migration can also induce a considerable overhead, thus affecting the predictability of the system and leading to potential failure of HRT deadlines. This challenge is the subject of multiple research efforts, such as those carried out by Raffeck et al. [8] and Gsänger [9], that have concluded that adequate techniques schedulers help dealing with multicore environments in an efficient way.

Another essential factor to ponder when dealing with RTOS is that, regardless of how good the OS is, if the software being used on top of it has not been programmed according to the strict specifications to correctly function in a real-time mode, the whole system will likely perform worst than expected.

4.2. Virtualization

One of the most relevant aspects of virtualization is the ability to abstract from the underlying hardware, thus allowing to share physical resources among different systems, in order to create more secure, dynamic, and flexible infrastructures. Another relevant feature is how easy it is to replicate virtualized instances. In a context of digital industry transformation, virtualization assumes an even greater preponderance for leveraging the creation and usage of digital twins by accelerating the replication of the manufacturing process, in part or as a whole.

Virtualization technologies encompass several domains [10], of which system virtualization is the more relevant to the scope of this paper. System virtualization is often seen as the ability to create multiple virtual systems within a single physical system. These virtual systems are independent from each other, although hosted by the same hardware. Hardware resources – such as CPUs, memory, disks or network interfaces – can be shared among or allocated to specific virtual systems, according to the desired cost-performance ratio. Hardware-assisted virtualization support in commodity architectures – such as Intel VT and AMD-V for x86, or ARM virtualization extensions – have become commonplace in modern CPUs. Such extensions optimise the operation of hypervisors and virtualized operating systems, eliminating the associated overhead of running a guest OS in a non-privileged ring [10]. Even the emerging open standard RISC-V architecture contemplates hardware-assisted virtualization through its hypervisor extension (H-Extension) [11]. The next subsection will present the topic of virtualized execution environments into more detail.

4.3. From hypervisors to containers

Regarding software-based support for virtualized execution environments, there are three distinct approaches that can be used: type 1 hypervisor, type 2 hypervisor, and container-based.

As observed in Fig. 6, the biggest difference between *type 1 hypervisors* and *type 2 hypervisors* is the existence, in the latter, of an extra abstraction layer (host operating system) between the hardware and the hypervisor. Consequently, some resources become unavailable to the virtualized systems (guest OSs) since they are allocated to the host OS. In some processor architectures, it may also cause virtualized systems to run in a ring farther away. This implies that there are larger overheads derived from the existence of that extra layer. These first two approaches already achieved a considerable level of maturity

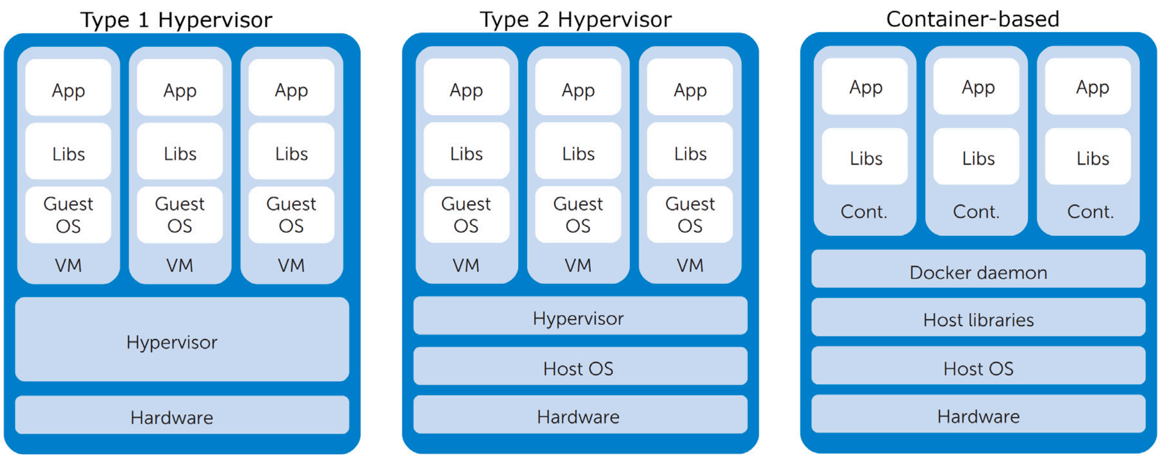

Fig. 6. Software-based virtualization — source [12].

and are widely used. While type 2 hypervisors are commonly used in personal computers where factors such as performance are neglected at the expense of other valences, type 1 hypervisors are commonly used in the infrastructure backbone, where performance requirements are higher and remote access is standard. In this paper we specifically focus on type 1 hypervisors.

General-purpose off-the-shelf type 1 hypervisors are usually designed to favour overall throughput, favouring techniques such as over-commitment at the cost of determinism. On the other hand, specialised real-time hypervisors focus on the determinism and predictability of the entire system — for instance limiting the number of supported virtual machines to existing physical resources, often allocated by means of static partitioning.

Several type 1 solutions for real-time virtualization have been proposed by the research community, such as BlueVisor [13], which aims at the virtualization for many-core embedded systems, and RTA-HVR [14], that is tailor-made to fit hard real-time automotive requirements and for hardware without virtualization support. ACRN [15] is also a noteworthy open source hypervisor that focuses on the requirements of the embedded IoT development. Nevertheless, at least some of the general-purpose hypervisors, like Xen Project [16] and KVM [17], a Linux kernel component system, are also able to accomplish some RT requirements when properly configured, as shown by Zhang et al. [18] and Abeni and Faggioli [19].

Despite being the most recent approach, *container-based virtualization* has its roots in the creation of the chroot system call in the late 1970s. This may have been the first step regarding process isolation by means of changing the root directory of a process and its children to a new location in the filesystem. In early 2000 the FreeBSD Jails feature was introduced, to enable the compartmentalisation of a FreeBSD system in independent micro-systems and the assignment of a different IP address to each one. Later, it was transposed to the Linux OS through a kernel patch. In 2006, Google introduced “process containers” that allowed some isolation, limitation, and accounting of resources. Later, this would be renamed as “control groups” (cgroups). Two years later, LXC appears as the first container manager, and in 2013 the Docker framework, for creating and managing containers, was presented.

Containers have the potential of significant performance improvements in terms of boot time, scalability and footprint [20], by using kernel features to create isolated environments. Unlike hypervisors, this approach shares the same underlying OS kernel among all containers, which makes it more lightweight and optimises the allocation of resources. However, in doing so, it does not benefit from the virtualization capabilities offered by type 1 hypervisors. To counter this limitation, its application in the IT environment is usually done in a second layer, that is, on top of a first layer consisting of a type 1 hypervisor. This solution has demonstrated an interesting potential in the IT environment. However, in a real-time context, it implies an extra layer that may add extra latency. Also, studies such as those carried out by Manco et al. [21] argue that this vision is not so straightforward, since it is possible to use a type 1 hypervisor (and benefit from all its advantages, such as hardware isolation) and still be able to have lightweight VMs capable of achieving considerably better performances than containers — this approach was adopted by Amazon with its Firecracker hypervisor for microVMs [22].

4.4. *The challenges of virtualizing RT workloads*

This section discusses the specific challenges involved in terms of RT workload virtualization, with a focus on isolation, scheduling mechanisms and their impact on determinism and predictability.

4.4.1. *Isolation*

Any virtualization process applied to real-time systems has to ensure that predictable deterministic real-time performance will be kept. This implies guaranteeing not only spatial isolation but also temporal isolation. As already mentioned, techniques like static partitioning virtualization are used for this purpose. Hardware resources such as RAM and CPU are isolated and assigned to a single system, opposed to over-commitment techniques, thus avoiding unnecessary overheads.

However, ensuring complete isolation of resources in increasingly complex systems is not trivial. For example, multi-core systems tend to share the memory among all CPUs and, as discussed by Capodieci et al. [23], interference may happen at all levels of the memory hierarchy. This means that although it is possible to statically partition RAM, there are still shared elements that may be susceptible to contention when exposed to heavy loads, such as the memory cache or bus. The existence of such contention is also shown by Danielsson et al. [24], that tested the Jailhouse RT hypervisor [25] in a multi-core environment to evaluate its CPU, cache and memory bus isolation capabilities. Although the authors concluded that there is in fact contention at the memory bus and L2 cache level, they also mention that the overall performance, considering a heavily loaded shared resource environment, was at least as good as a bare-metal Linux installation.

Multiple solutions, based on hardware or software, have been proposed to improve those weaknesses [26]. Cache colouring seems to be the one gaining more traction in the context hypervisors’ performance. Xen Project and Jailhouse ARM versions are some of the hypervisors that are already compatible with such technique, as successfully shown in the European projects I-MECH [27] and Hercules [28], respectively. Xvisor [29] and Bao [30] also presented positive results while using cache colouring. Other approaches were also proposed to improve isolation and guarantee real-time requirements. Pinto et al. [31] presented a lightweight hypervisor (LTZVizor) based on ARM TrustZone, a hardware-assisted security extension for ARM processors. This technology was exploited to leverage real-time virtualization by enabling the virtualization of a physical core in two distinct execution domains – one designated as secure world, which is a trusted execution environment (TEE) and was used to run time sensitive tasks, and the other designated as non-secure world, to run general-purpose tasks – and providing time and space isolation between these. This approach was also taken into account by Hua et al. [32]. However, in this case, the authors complemented their approach by also applying ARM virtualization extensions (VE), thus increasing the virtualization capabilities and enabling each guest with an isolated guest TEE with the same isolation capabilities and security as the physical secure world.

I/O sharing between multiple VMs cannot be left out when addressing isolation to maintain a predictable deterministic real-time performance, since it also presents considerable challenges, especially when global memory is being used in the path to I/O access. Specific hardware-assisted mechanisms can also be used in this case for controlling the I/O-related memory contention. For example, Borgioli et al. [33] addressed this problem by means of hardware regulators that control the number of memory transactions permitted from each device in a given period, thus controlling I/O-related memory interference. Extensive evaluation showed the proposed solution may perform up to eight times better. Like other hardware-assisted solutions, this one is dependent on the type of hardware being used (ARM QoS-400 regulators). However, the trend among the main players in this market is that they all end up providing equivalent hardware solutions to support virtualization and its various challenges. Therefore, the important thing to remember is that there are several ways to reinforce the isolation of virtualized systems, including hardware-assisted solutions which tend to deliver better performance for real-time requirements.

It must be stressed that, although static partitioning is one of the most used techniques to help ensure compliance with real-time requirements, it is not mandatory. In [34], Li et al. present a real-time virtualization platform based on a type 1 hypervisor that supports dynamic reallocation of CPU resources among VMs in runtime, without over-provisioning resources and guaranteeing SRT performance levels.

4.4.2. *Scheduling*

Scheduling policies also assume a relevant role in real-time applications. Partitioning-based scheduling, where each partition gets assigned strict time frames to ensure the temporal behaviour of each process, is often applied. Other techniques use a combination of priority and time-driven scheduling (adaptive time-partitioning scheduling), where the priority of each thread is also considered [35]. Some hypervisors that exploit these techniques to provide real-time assurances are PikeOS (rooted in the L4 [36] family of micro-kernels) [37], QNX [38] and XtratuM [39].

A hypervisor may support several schedulers, just one or none. For example:

- the Xen Project currently supports both general purpose and real-time schedulers: Credit, Credit2, Real-Time-Deferrable-Server (RTDS), ARINC 653 and NULL;
- KVM supports the Completely Fair Scheduler (CFS), as well as the SCHED_FIFO, SCHED_DEADLINE, and SCHED_RT;
- the VMware vSphere hypervisor supports two distinct versions of the Side Channel Aware scheduler, in addition to its default one;
- XtratuM only supports a fixed cyclic scheduler based on ARINC 653 specifications;
- Jailhouse and BAO do not include any scheduler, since there is no support for over-commitment. Both implement a one-to-one mapping of virtual to physical core, meaning that each VM uses its own physical core, thus nullifying the need for a scheduler.

Since there are several schedulers, choosing one that best suits the intended function is essential. Moreover, as shown in [40], it is also possible to optimise a scheduler taking into account the architecture of the processor on which it will be used. Specific configurations, like the scheduling time per slice, may also be tuned to achieve better RT performance. However, as shown by Tellabi and Ruland [41], the results cannot be extrapolated between different hypervisors or schedulers.

4.5. *Virtualization applied to cyber–physical systems*

Recent research on the topic of real-time workload virtualization applied to cyber–physical systems has validated the claim that both technologies can be reconciled in an effective way.

Hofer et al. [42] analysed the possibility of migrating real-time industrial control applications from dedicated hardware to virtualized servers with shared resources, aiming for an IaaS approach. Three distinct analysis were made. First, offline tests take in consideration different configuration options. Using a type 1 hypervisor and three distinct host profiles – a standard Linux installation, another with the PREEMPT_RT kernel patch and the other with Xenomai 3 kernel patch – multiple tests were executed to measure the CPU latency. Results are presented for each of the twelve different tweaks made in each of the hosts. Second, a hardware comparison was performed using the most favourable configuration previously identified, comparing CPU latency while using multiple hardware options offered by Amazon Web Services. Observed results lead the authors to conclude it is viable to migrate real-time applications to an IaaS solution. Finally, using a test-run of a Balena container, latency tests were executed inside a container. Observed results further demonstrate the feasibility of such migration.

Cinque et al. [43] presented an architecture for allowing industrial mixed-criticality systems to coexist on the same hardware and for enabling large-scale scalability based on container virtualization. They implemented real-time containers with distinct criticality levels. These are meant to run fixed-priority hard-real-time periodic tasks with temporal separation. This solution combines Docker over an Ubuntu Linux patched with the RTAI real-time co-kernel extension. After carrying out a set of exhaustive tests using representative task sets, the authors confirm the feasibility of the proposed concept. It is noteworthy that a preemptive fixed priority task scheduling was chosen.

Bock et al. [44] discuss the merging of CPS and IoT in the context of Industry 4.0, proposing Xvisor-RT, a real-time embedded hypervisor that supports multiple cores and multiple VMs with real-time tasks. Xvisor-RT is based on the Xvisor type 1 hypervisor. By focusing on enabling a scheduling mechanism to deal with a set of virtual processors (vCPU) with real-time constraints on a multi-core embedded system, it is able to choose the best candidate task and vCPU at each moment. To determine the optimal scheduling mechanism, the authors analysed multiple combinations of system-level schedulers with task-level schedulers and took into consideration the load of the vCPUs and the number of physical CPUs scheduled. The authors were able to select a set of scheduling algorithms, which allowed them to execute two distinct real-time applications without no deadline misses, thus validating the usage of virtualization in this context. As stated in the paper, the presented results may help others to shorten the path to choose the best set of scheduling algorithms to other scenarios.

Yang et al. [45] discussed how to achieve hard real-time performance while keeping the rich Linux feature set. The authors created a compounded real-time operating system (cRTOS) to enable richer features in real-time operating systems, and to increase the efficiency, usability, and maintainability of the development process. The system consists of a real-time type 1 hypervisor (Jailhouse) that hosts two distinct realms, a normal Linux realm referred to as general purpose operating systems (GPOS) and a hard-real-time realm of a swift RTOS (sRTOS). The first runs an Ubuntu Linux with a vanilla or a PREEMPT_RT patched kernel and is responsible for the rich features, while the latter runs a Nuttx x86 ported version and is responsible for real-time tasks. It is concluded that the proposed system can achieve hard-real-time performance while delivering Linux rich features without requiring any kernel modification. Thus, it is possible to reduce the complexity and cost of developing real-time applications incorporating rich features since developers can use the same toolchains and executables as with Linux. The outcome of this research not only validates the usage of virtualization in hard real-time environments, but also goes a step further by introducing new capabilities that may be explored in the context of mixed-critical systems and Industry 4.0.

Scordino et al. [46], under the I-MECH project [27], presented a modular real-time platform for IACS based on a type 1 hypervisor and using only open-source components. According to the authors, the developed architecture complies with requirements such as: working on COTS x86 platforms; supporting EtherCat master functionalities and communications; and being capable of running multiple OSs in parallel, including unmodified versions of Microsoft Windows and RTOSs and supporting resources partitioning. After describing with some detail the options taken regarding each component, a general evaluation based on experimental results is presented. This highlights multiple parameters regarding the hypervisor, such as introduced overhead and cache isolation capabilities. Techniques to improve and prevent interference between shared resources are also discussed. It is concluded that type 1 hypervisor-based virtualization can be used to create innovative and flexible infrastructures while still achieving real-time requirements as demanded by IACS.

Since I/O is one of the essential components of cyber–physical systems, its virtualization is of utmost relevance, namely in multi-/many-core environments where complicated I/O access paths and resource management is conducive to increase latency and timing variance of I/O operations. Jiang et al. [47] use an automotive case study to test their virtualization framework (I/O-Guard), which introduces a novel system architecture focused on multi-/many-core I/O virtualization. This modular framework relies on a new hypervisor micro-architecture, mostly supported by hardware-assisted virtualization, that enables random accesses of I/O operations and task prioritisation. It also relies on a two-layer scheduler (local and global) supporting preemptive scheduling methods (earliest deadline first policy) with guaranteed real-time performance, and on device-specific low-level I/O drivers integrated into the hardware. The presented architecture introduces several improvements when empirically compared with a real-time patched version of Xen hypervisor. It achieved good results in terms of I/O performance and throughput, with low levels of missed safety and function tasks due to the capability of optimising I/O access paths and resource management throughout the system.

Casini et al. [48] also highlighted the importance of I/O virtualization for achieving certification requirements of safety–critical systems. After identifying the need to analytically bound the virtualization latencies for providing off-line guarantees, the authors focused on three different I/O virtualization modelling techniques that are not dependent on specific hardware nor rely on custom scheduling algorithms and, as so, can be applied in standard inter-VM communications present in most hypervisors: Pass-Through I/O, I/O Paravirtualization with I/O VM, and I/O Para-Virtualization with I/O VM and Shared Buffers. These are presented in detail and evaluated by means of two experimental studies that take into consideration the following metrics: response time, data delivery latency and input processing latency. The outcome results and the trace that is made to the sources of delay allow to bound the latency coming from the I/O virtualization and provide a deep insight on how to comply with timing constraints.

5. Evaluation of COTS hypervisors for real-time scenarios

Considering the aforementioned developments in terms of COTS and RT-specific virtualization technologies, it still remains the question of understanding to which extent general-purpose off-the-shelf hypervisors are capable of hosting RT-constrained guest VMs. While not being designed to deal with HRT workloads, the introduction of specific VM profiles and/or specific fine-tuning may fulfil requirements of several RT targets, whose extent can only be assessed by means of experimental evaluation.

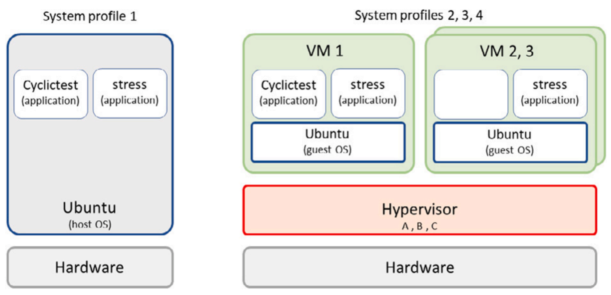

Fig. 7. System profiles architecture.

Therefore, this section presents the results of the experimental evaluation that was conducted to understand the performance impact when using COTS hypervisors and how feasible the usage of this type of hypervisors is in scenarios with real-time constraints. For this purpose, the evaluation effort will focus on assessing the latency and efficiency penalties associated to these environments, in comparison with partitioning techniques. First, the analysis of a GPOS use case is introduced to provide a reference baseline for comparison with RTOS payloads, which are to be next evaluated. The rationale for this approach is to consider both GPOS and RTOS payloads in COTS hypervisors, in order to understand how the same hypervisor platform can provide RT guarantees for RT guests, with some eventual optimisations. To better clarify this, several modifications were introduced in the paper.

5.1. *Testing scenarios*

Four system profiles have been defined for experimental analysis purposes (Fig. 7). The first, based on a bare metal Ubuntu Server 20.04 OS environment, with kernel version 5.4.0-86-generic, was used as baseline. Each of the other three profiles consisted of distinct type 1 hypervisor hosts — selected from the top four list according to the 2019 market share [49]. The hypervisors used in this study will be designated by a generic reference: hypervisor A, B, and C, instead of their name. This option is due to legal constraints, since these commercial hypervisors have product EULAs that restrict or strongly limit the publication of benchmark studies. Nevertheless, we believe this is not a limiting factor, since the purpose of this article is not to benchmark specific implementations but rather to assess the general panorama of commercial off-the-shelf hypervisors regarding real-time workload support.

Three virtual machines running similar versions of the above-mentioned Ubuntu Linux were installed as guests in each hypervisor. The first hosted the RT task emulation, and the second and third ones were used for parallel workload (as explained later). Two vCPUs were allocated to each VM, as well as 4 GB of RAM for the first VM and 6 GB for the two other VMs, thus using all the resources available in the hosting machine. Also, configuration settings to enable the static allocation of resources, like the CPU cores and RAM, were applied to the first VM, as well as the configuration of the VM as highly sensitive to latency, thus granting each virtual CPU exclusive access to a physical core and optimising the schedule delay for latency sensitive applications (when such options were made available by the hypervisor). Furthermore, the usage of multiple C-states was disabled through both the BIOS options and OS's kernel configurations, as well as hyperthreading and other dynamic power management options, due to their negative impact on latency and determinism. The underlying hardware consisted of an Asus PRIME H410M-A motherboard, with an Intel® Core™ i5-10400F 2.9 GHz CPU with 6 cores, 16 GB DDR4 RAM running at 2666 MHz, and two SATA3 solid-state drives.

Table 2
Testing scenarios.

System profile	Testing profile	1st VM	Cyclictest	Stress	RT tasks core affinity	2nd and 3rd VM
Baseline 1	1	✘	✓	✘	✘	✘
	2	✘	✓	✘	✓	✘
	3	✘	✓	✓	✘	✘
	4	✘	✓	✓	✓	✘
Hypervisor 2, 3, 4	1	✓	✓	✘	✘	✘
	2	✓	✓	✘	✓	✘
	3	✓	✓	✓	✘	✘
	4	✓	✓	✓	✓	✘
	5	✓	✓	✓	✘	✓
	6	✓	✓	✓	✓	✓

5.2. Testing procedures

Conducted tests consisted in emulating a RT task and measuring the response latency of its thread by means of the high-resolution testing tool Cyclictest [50]. The thread was clocked at 10 ms, and a FIFO scheduling policy was used, with the thread being assigned the highest priority.

Measurements were performed in distinct testing environments, some of which had best effort concurrent threads competing for the machine resources. For this purpose, the workload generator tool stress [51] was used. A total of 20 simultaneous workers were instantiated (10 exercising the CPU by spinning on square root function, and 10 spinning on malloc/free operations with 64 MB blocks), which was enough to overload the available cores. Each test had a duration of 8 h, which led to over 2.5 million samples per test. The dataset generated by this work is available in [52].

The profiles of the testing environments – which were applied to each of the system profiles as detailed in Table 2 – are described below:

- Profile 1 — RT task executed without core affinity or concurrent workload;
- Profile 2 — RT task executed with core affinity and without concurrent workload;
- Profile 3 — RT task executed without core affinity and with concurrent workload;
- Profile 4 — RT task executed with core affinity and with concurrent workload (RT task and concurrent workload running in distinct cores);
- Profile 5 — Three VMs running simultaneously. The first executing the RT task and the others running concurrent workloads. No core affinity was used for the RT task.
- Profile 6 — Three VMs running simultaneously. The first executing the RT task and the others running concurrent workloads. Core affinity was used for the RT task.

Notice that, when mentioning core affinity one is referring to the task being allocated to a specific vCPU inside the VM. In turn, as previously indicated, the physical cores are statically allocated to the VM.

The first four testing profiles (TP) were defined in a way which allowed analysing the performance impact of each hypervisor when directly compared to the bare metal baseline system profile. This can be seen as a representation of the vPLC. TP1 and TP2 represent the scenario where a vPLC only executes an RT task. In TP3 and TP4 the RT task must compete for resources against other workloads, representing a scenario that optimises the use of resources just like a mixed-criticality system. TP2 and TP4 intend to provide some information about the effectiveness of using the core affinity technique to achieve a more deterministic system. Profiles 5 and 6 allowed assessing the efficiency of the isolation techniques used by the hypervisors. This should clarify if it is possible to run multiple VMs in the same hypervisor without any sort of interference between them, which could compromise the RT task.

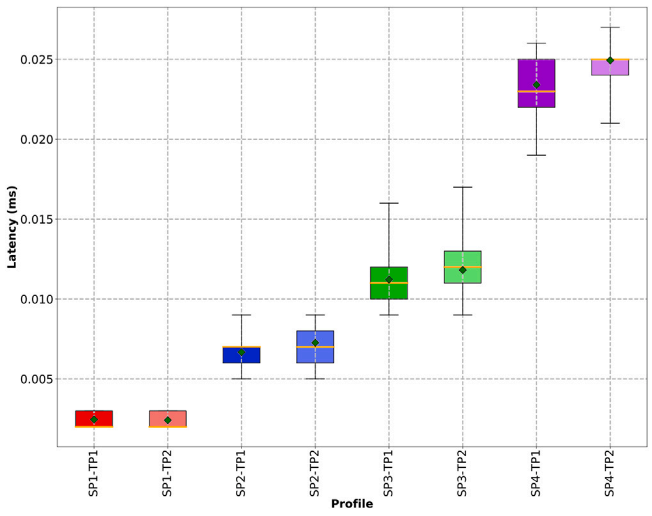

Fig. 8. Hypervisor overhead — TP1&2.

5.3. Obtained results and discussion

This subsection presents and discusses the results of the conducted experiments. First, we discuss results obtained with general-purpose operating systems (GPOS), followed by the results measured with Real-time Operating Systems.

5.3.1. GPOS

The results obtained in the various tests are presented below. Table 3 depicts obtained results, within a 95% confidence interval.

The testing profiles TP1 and TP2 present the best results throughout all system profiles (SP). Although it is possible to identify a slight overhead when using any of the hypervisors (Fig. 8), the measured latency is quite low, as well as its standard deviation.

The best results are achieved by hypervisor A (SP1), which presented a 0.007 ms average latency, a 0.001 ms standard deviation and a maximum jitter of 0.004 ms. The worst results were observed with hypervisor C (SP3), with average latency of 0.025 ms and a maximum jitter of 0.007 ms. Despite these results being quite interesting and relatively close to the baseline system, one must bear in mind that these testing profiles did not include concurrent workloads, which is the most likely scenario when using hypervisors.

Testing profiles TP3 and TP4 already incorporate parallel workload. Profile TP3 obtained the worst performance. Not only is the mean latency the highest (for three out of four system profiles), but also the dispersion of results is the more pronounced among all of them, which translates into the highest maximum jitter. These results are most likely due to the core swapping of the RT task since there was no sort of core affinity defined and multiple workers of stress were struggling for the same resources at the same time.

Table 3
Laboratory results with GPOS.

System profile	Testing profile	Lower wisker	Lower quartile	Median	μ	σ	Upper quartile	Upper wisker	Max jitter
#1 Baseline	1	0.002	0.002	0.002	0.002	0.000	0.003	0.003	0.001
	2	0.002	0.002	0.002	0.002	0.000	0.003	0.003	0.001
	3	0.005	0.007	0.009	0.037	0.146	0.012	0.572	0.567
	4	0.002	0.004	0.005	0.005	0.002	0.006	0.007	0.005
#2 Hypervisor A	1	0.005	0.006	0.007	0.007	0.001	0.007	0.009	0.004
	2	0.005	0.006	0.007	0.007	0.001	0.008	0.009	0.004
	3	0.006	0.008	0.009	0.030	0.111	0.011	0.464	0.458
	4	0.005	0.007	0.007	0.008	0.001	0.008	0.010	0.005
	5	0.007	0.009	0.011	0.011	0.003	0.013	0.020	0.013
	6	0.007	0.010	0.012	0.013	0.004	0.014	0.022	0.014
#3 Hypervisor B	1	0.009	0.010	0.011	0.011	0.002	0.012	0.016	0.007
	2	0.009	0.011	0.012	0.012	0.003	0.013	0.017	0.008
	3	0.009	0.013	0.016	0.040	0.120	0.018	0.552	0.542
	4	0.011	0.013	0.016	0.016	0.013	0.018	0.024	0.013
	5	0.013	0.023	0.030	0.037	0.214	0.038	0.051	0.038
	6	0.018	0.025	0.031	0.029	0.187	0.040	0.055	0.037
#4 Hypervisor C	1	0.019	0.022	0.023	0.023	0.005	0.025	0.026	0.007
	2	0.021	0.024	0.025	0.025	0.015	0.015	0.027	0.006
	3	0.013	0.017	0.019	0.039	0.105	0.022	0.372	0.359
	4	0.023	0.027	0.030	0.031	0.006	0.034	0.040	0.018
	5	0.026	0.040	0.051	0.053	0.095	0.057	0.074	0.049
	6	0.032	0.042	0.053	0.056	0.090	0.059	0.073	0.041

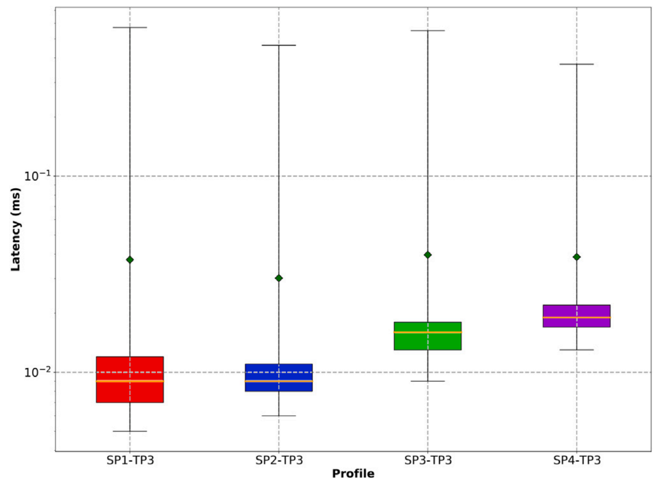

Fig. 9. Hypervisor performance — TP3.

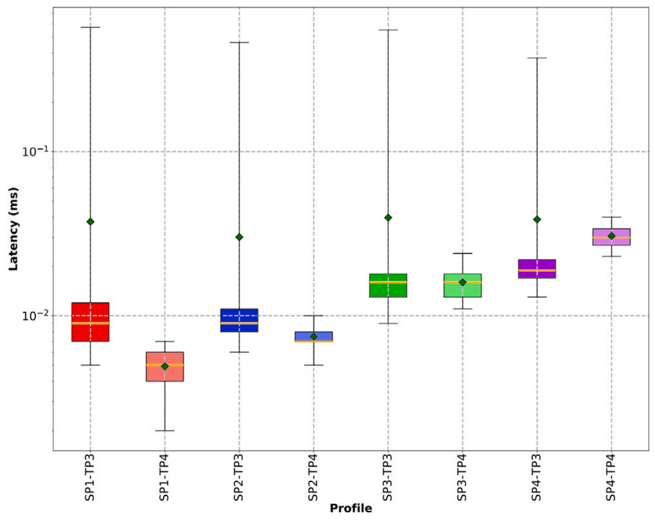

Fig. 10. Hypervisor performance — TP3&4.

Comparing the results obtained when using hypervisors with those with the baseline system (Fig. 9), it is possible to conclude that the decrease of performance identified is similar for both cases. Moreover, the system profiles using hypervisors presented a better overall performance in these tests than the baseline profile itself, which had higher maximum jitter and σ. Hypervisor A (SP2) was even able to achieve not only lower jitter and σ, but also lower mean latency.

The absence of core affinity meant that the baseline system had at its disposal all the six cores. It is possible that the cost of continuously swapping the threads between all the cores resulted in an overall worst performance. Although the existence of more cores is normally a positive factor (specially regarding throughput capacity), when considering processing latency it may actually lead to lower performance. Each virtual machine was limited to two cores.

When analysing the results of TP4 tests (Fig. 10), an improvement across all systems is noticeable. In TP4 distinct tasks were assigned to distinct cores, which means the RT task and the concurrent workload were executed in distinct cores, leading to better results. Not only the latency dropped consistently in all the tests (between 20% and 80%), achieving values in the order of 0.008 ms, but also σ improved considerably, decreasing more than 89% in all tests and reaching 0.001 ms. In line with these values, the maximum jitter also went down (between 94% and 99%), staying as low as 0.005 ms.

As in the previous testing profiles, the measured latency when using COTS hypervisors and when using the baseline system followed a similar trend. That is, when the baseline profile latency results increased, decreased, or remained similar (comparing to the previous testing) so did the results obtained with the hypervisors. This means that the trend of the results has not been negatively impacted by the usage of hypervisors. However, in most cases, the values themselves reveal a slight overhead originated by the usage of the COTS hypervisors. Yet, the outcome results of TP3 and TP4 shown that a COTS hypervisor can be used in some situations without a negative impact.

The goal for the testing profiles TP5 and TP6 was to analyse the effectiveness of isolating multiple virtual machines running at the same time. Just like in the previous testing profiles, a VM executed the RT task. However, in these cases, the concurrent workload was executed in two other VMs that ran in parallel. Just like TP4 (although with a different approach), this should guarantee the isolation of the cores running the RT task and the remaining workload.

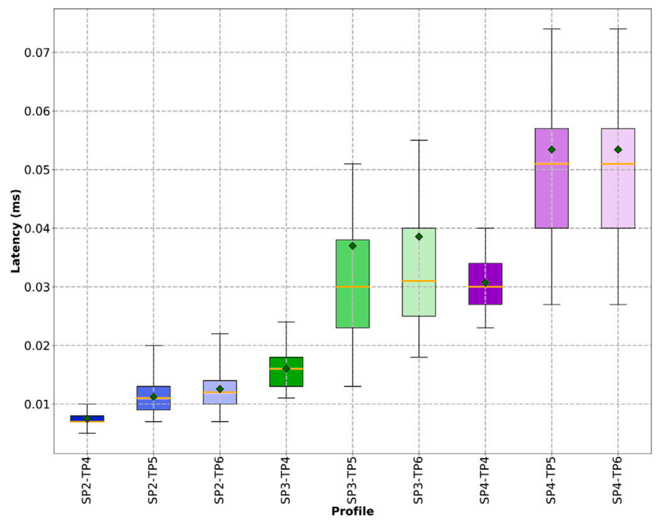

Fig. 11. Hypervisor isolation effectiveness — TP4&5&6.

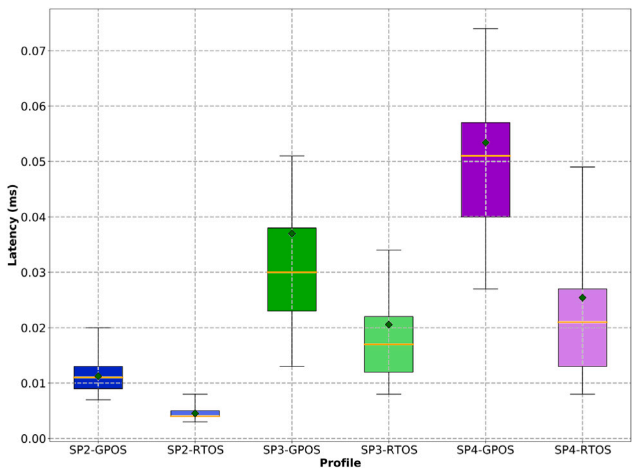

Fig. 12. Hypervisor performance with RTOS — TP5.

If observed latency was similar to that of TP4, one could conclude that there was a total isolation of the VMs. However, that is not the case. The obtained measurements show that the isolation techniques of the COTS hypervisors are not completely effective, as the mean latency increased in both testing profiles (Fig. 11). Even though a static allocation of the CPU cores and RAM was made, this was already to be expected, since there are always shared components such as the cache memory of the CPU and the internal communication buses.

Summing up, the tested hypervisors show distinct results in terms of latency and dispersion. The results of Hypervisor A stand out since they are very close to the non-virtualized baseline system, achieving even better results in some cases. Nevertheless, the remaining hypervisors also attained very interesting results. Despite the variability of some results, according to different scenarios and the limited effectiveness of the isolation techniques used by COTS hypervisors, when observing the order of magnitude of the results it is concluded that they are compatible with a wide range of IACS application scenarios. Considering the results from the testing profile TP5 – the one that comes closer to a realistic scenario when using a hypervisor – it is observed that the average latency stays between 0.011 ms and 0.053 ms, and the maximum jitter between 0.013 ms and 0.049 ms. Such results fit most of the industrial systems latency and jitter requirements presented in Table 1. However, it is necessary to consider that this table presents the end-to-end values, that is, the sum of all latencies present throughout all the process. In our tests we only consider the processing latency, meaning that a safety margin has to be considered to accommodate other sources of delay.

5.3.2. RTOS

Despite the previous tests showing very promising results, it is not clear if the observed results were also conditioned by using a GPOS, in addition to the impact of using COTS hypervisors. Thus, similar tests were conducted using a RTOS instead. Xenomai and Preempt_RT were considered for performance and compatibility reasons. Both have their pros and cons, as discussed in [53]. Therefore, the chosen RTOS was Xenomai 3.1 (Cobalt variant), which was installed over the same version of Ubuntu being used in the previous tests. As seen in Table 4, the latency values dropped significantly. This means that previously obtained results were indeed affected by the usage of a GPOS.

Focusing again on TP5 (Fig. 12), one can identify a noticeable performance improvement throughout all the system profiles when using a RTOS. Nonetheless, the results when using a RTOS still consolidate what was concluded above.

In order to determine the predictability of the system over time, the distribution of results after 1 h, 2 h, 4 h and 8 h was also analysed. As shown in Table 5, all system profiles presented a very stable behaviour over time, regarding average latency. The standard deviation also remained constant in SP2 and had a maximum fluctuation of 0.007 ms and 0.013 ms in SP3 and SP4, respectively. These results show that the predictability of such systems is quite high and reinforces the feasibility of using COTS hypervisors in some RT environments.

6. Conclusion and future work

COTS hypervisors have shown considerable evolution over the last years. The existence of services with stricter latency requirements is today a factor that the virtualization players are aware. This is confirmed by the presence of specific settings for low latency VMs in some of the analysed COTS hypervisors.

Considering the conducted experiments regarding the CPU processing latency, we confirmed the ability of type 1 COTS hypervisors to fulfil the requirements of a wide range of real-time systems. However, the use of such hypervisors in hard real-time systems, with stricter latency and jitter requirements, may not be feasible. Also, one cannot assume that any COTS hypervisor can be used in such environments. As shown by our experimental evaluation, not all hypervisors behave the same way when exposed to the same conditions. A small behaviour difference between them can result in a significant negative performance impact in a context of hard real-time systems which, in turn, may lead to serious cascade effects in the physical processes those control.

One of the tested hypervisors showed very homogeneous results, apparently confirming its ability to handle even some of the most demanding systems. However, they all showed that their VMs isolation capabilities are not completely reliable. The degree of disturbance that one VM can cause to the performance of another has not been analysed in detail. In a future study, it would be interesting to quantify the maximum degree of disturbance one VM can cause in the entire system, and its cause. For that, it would be necessary not only to use the RAM and CPU memory overload, but also to generate intensive peripheral traffic (e.g. network interface, disk drive) to understand to what extent the interrupts generated by those components can affect the performance of adjacent VMs.

It would also be interesting to determine the best performance that can be achieved by type 1 hypervisors, namely using RT-specific hypervisors. Unlike a COTS hypervisor, RT-specific hypervisors have as main objective to guarantee the determinism and predictability of the entire system.

Table 4
Laboratory results with RTOS.

System profile	Testing profile	Lower wisker	Lower quartile	Median	μ	σ	Upper quartile	Upper wisker	Max jitter
#1 Baseline	1	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	2	0.000	0.000	0.000	0.000	0.000	0.000	0.000	0.000
	3	0.001	0.002	0.003	0.003	0.001	0.004	0.005	0.004
	4	0.000	0.001	0.001	0.001	0.001	0.002	0.003	0.002
#2 Hypervisor A	1	0.002	0.002	0.002	0.002	0.000	0.002	0.003	0.001
	2	0.002	0.002	0.002	0.002	0.000	0.002	0.003	0.001
	3	0.002	0.003	0.003	0.003	0.001	0.004	0.005	0.003
	4	0.002	0.002	0.002	0.003	0.000	0.003	0.004	0.002
	5	0.003	0.004	0.004	0.005	0.001	0.005	0.008	0.005
	6	0.003	0.004	0.005	0.005	0.002	0.006	0.009	0.005
#3 Hypervisor B	1	0.005	0.005	0.006	0.006	0.017	0.007	0.008	0.003
	2	0.004	0.007	0.007	0.007	0.001	0.008	0.009	0.005
	3	0.004	0.005	0.006	0.007	0.005	0.008	0.010	0.006
	4	0.005	0.007	0.008	0.008	0.024	0.009	0.012	0.007
	5	0.008	0.012	0.017	0.021	0.139	0.022	0.034	0.027
	6	0.012	0.018	0.022	0.024	0.106	0.027	0.037	0.025
#4 Hypervisor C	1	0.005	0.006	0.006	0.007	0.005	0.006	0.009	0.004
	2	0.005	0.006	0.006	0.006	0.002	0.006	0.007	0.002
	3	0.005	0.006	0.007	0.011	0.011	0.009	0.049	0.045
	4	0.006	0.007	0.007	0.008	0.004	0.008	0.012	0.006
	5	0.008	0.013	0.021	0.025	0.091	0.027	0.049	0.041
	6	0.014	0.020	0.026	0.031	0.088	0.032	0.049	0.036

Table 5
Latency over time (TP5 in ms).

System profile	Hours	Lower wisker	Lower quartile	Median	μ	σ	Upper quartile	Upper wisker	Max jitter
#2 Hypervisor A	1	0.003	0.004	0.004	0.005	0.001	0.005	0.008	0.005
	2	0.003	0.004	0.004	0.005	0.001	0.005	0.008	0.005
	4	0.003	0.004	0.004	0.005	0.001	0.005	0.008	0.005
	8	0.003	0.004	0.004	0.005	0.001	0.005	0.008	0.005
#3 Hypervisor B	1	0.007	0.013	0.017	0.021	0.129	0.022	0.034	0.027
	2	0.007	0.012	0.017	0.021	0.131	0.022	0.034	0.027
	4	0.007	0.012	0.017	0.021	0.142	0.022	0.034	0.027
	8	0.007	0.012	0.017	0.021	0.138	0.022	0.034	0.027
#4 Hypervisor C	1	0.008	0.013	0.021	0.025	0.085	0.027	0.049	0.041
	2	0.008	0.013	0.021	0.025	0.089	0.027	0.049	0.041
	4	0.008	0.013	0.021	0.025	0.092	0.027	0.049	0.041
	8	0.008	0.013	0.021	0.025	0.090	0.027	0.049	0.041

Declaration of competing interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

Data availability

Data available at doi:10.21227/h7ye-x037

Acknowledgements

This work was partially funded by the National Funds through the FCT—Foundation for Science and Technology, I.P., Portugal, and the European Social Fund, through the Regional Operational Program Centro 2020, within the scope of the project CISUC UID/CEC/00326/2020. It was also partially co-funded by FEDER, in the context of the Competitiveness and Internationalization Operational Program (COMPETE 2020) of the Portugal 2020 framework, in the scope of Project Smart5Grid (POCI-01-0247-FEDER-047226).

References

[1] T. Cruz, P. Simões, E. Monteiro, Virtualizing programmable logic controllers: Toward a convergent approach, IEEE Embed. Syst. Lett. 8 (4) (2016) 69–72, http://dx.doi.org/10.1109/LES.2016.2608418.
[2] T. Cruz, R. Queiroz, P. Simões, E. Monteiro, Security implications of SCADA ICS virtualization: Survey and future trends, in: European Conference on Information Warfare and Security, ECCWS, Vol. 15, 2016, pp. 81–100.
[3] T. Bauernhansl, S. Kondoh, S. Kumara, L. Monostori, B. Ka, Cyber-physical systems in manufacturing, CIRP Ann. - Manuf. Technol. 65 (2016) 621–641, http://dx.doi.org/10.1016/j.cirp.2016.06.005.
[4] S. Figueroa-Lorenzo, J. Añorga, S. Arrizabalaga, A role-based access control model in modbus SCADA systems. a centralized model approach, Sensors (Switzerland) 19 (20) (2019) http://dx.doi.org/10.3390/s19204455.
[5] C. Wang, F. Yang, H. Wang, P. Guo, J. Hou, Improving real time performance of linux system using RT-linux, J. Phys. Conf. Ser. 1237 (5) (2019) http://dx.doi.org/10.1088/1742-6596/1237/5/052017.
[6] Real-time operating systems, 2021, https://www.osrtos.com/ Accessed: 2021-04-01.
[7] A. Serino, L. Cheng, A survey of real-time operating systems, 2019, pp. 27–65.
[8] P. Raffeck, P. Ulbrich, W. Schroder-Preikschat, Work-in-progress: Migration hints in real-time operating systems, in: Proceedings - Real-Time Systems Symposium, Vol. 2019-Decem, 2019, pp. 528–531, http://dx.doi.org/10.1109/RTSS46320.2019.00056.
[9] H. Gsänger, Dynamic Migration Decisions in Multicore Systems (Ph.D. thesis), Friedrich-Alexander-Universität, 2020, pp. 27–65.
[10] O. Nagesh, T. Kumar, V. Venkateswararao, A survey on security aspects of server virtualization in cloud computing, Int. J. Electr. Comput. Eng. 7 (3) (2017) 1326–1336, http://dx.doi.org/10.11591/ijece.v7i3.pp1326-1336.
[11] B. S'a, J. Martins, S. Pinto, A first look at RISC-v virtualization from an embedded systems perspective, IEEE Trans. Comput. 71 (2021) 2177–2190.
[12] T. Combe, A. Martin, R. Di Pietro, To docker or not to docker: A security perspective, IEEE Cloud Comput. 3 (5) (2016) 54–62, http://dx.doi.org/10.1109/MCC.2016.100.
[13] Z. Jiang, N.C. Audsley, P. Dong, BlueVisor: A scalable real-time hardware hypervisor for many-core embedded systems, in: Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, IEEE, 2018, pp. 75–84, http://dx.doi.org/10.1109/RTAS.2018.00013.
[14] A.K. Sundar Rajan, A. Feucht, L. Gamer, I. Smaili, N.D. M., Hypervisor for consolidating real-time automotive control units: Its procedure, implications and hidden pitfalls, J. Syst. Archit. 82 (June 2017) (2018) 37–48, http://dx.doi.org/10.1016/j.sysarc.2018.01.001.

[15] T.L. Foundation, ACRN hypervisor, 2021, https://projectacrn.org/ Accessed: 2021-04-01.

[16] Xen project, 2021, https://xenproject.org/ Accessed: 2021-04-01.

[17] R. Hat, Kernel virtual machine, 2021, https://www.linux-kvm.org Accessed: 2021-04-01.

[18] J. Zhang, K. Chen, B. Zuo, R. Ma, Y. Dong, H. Guan, Performance analysis towards a KVM-based embedded real-time virtualization architecture, in: 5th International Conference on Computer Sciences and Convergence Information Technology, no. May 2016 in 1, 2011, pp. 421–426, http://dx.doi.org/10.1109/ICCIT.2010.5711095.

[19] L. Abeni, D. Faggioli, Using xen and KVM as real-time hypervisors, J. Syst. Archit. 106 (January) (2020) http://dx.doi.org/10.1016/j.sysarc.2020.101709.

[20] Y. Brikman, Terraform: Up & Running: Writing Infrastructure as Code, second ed., O'Reilly Media, 2017.

[21] F. Manco, J. Mendes, K. Yasukata, C. Lupu, S. Kuenzer, C. Raiciu, F. Schmidt, S. Sati, F. Huici, My VM is lighter (and safer) than your container, in: SOSP 2017 - Proceedings of the 26th ACM Symposium on Operating Systems Principles, 2017, pp. 218–233, http://dx.doi.org/10.1145/3132747.3132763.

[22] A. Agache, M. Brooker, A. Florescu, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, D.-M. Popa, Firecracker: Lightweight virtualization for serverless applications, in: NSDI 2020, 2020, pp. 419—434.

[23] N. Capodieci, P. Burgio, R. Cavicchioli, I.S. Olmedo, M. Solieri, M. Bertogna, Real-time requirements for ADAS platforms featuring shared memory hierarchies, IEEE Des. Test 14 (8) (2020) 1, http://dx.doi.org/10.1109/mdat.2020.3013828.

[24] J. Danielsson, T. Seceleanu, M. Jagemar, M. Behnam, M. Sjodin, Testing performance-isolation in multi-core systems, in: Proceedings - International Computer Software and Applications Conference, Vol. 1, IEEE, 2019, pp. 604–609, http://dx.doi.org/10.1109/COMPSAC.2019.00092.

[25] Siemens, Jailhouse hypervisor, 2021, https://github.com/siemens/jailhouse Accessed: 2021-04-01.

[26] T. Kloda, M. Solieri, R. Mancuso, N. Capodieci, P. Valente, M. Bertogna, Deterministic memory hierarchy and virtualization for modern multi-core embedded systems, in: Proceedings of the IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, Vol. 2019-April, IEEE, 2019, pp. 1–14, http://dx.doi.org/10.1109/RTAS.2019.00009.

[27] I-MECH project, 2021, https://www.i-mech.eu/ Accessed: 2021-04-01.

[28] Hercules Project, Hercules project, 2021, http://hercules2020.eu/ Accessed: 2021-04-01.

[29] P. Modica, A. Biondi, G. Buttazzo, A. Patel, Supporting temporal and spatial isolation in a hypervisor for arm multicore platforms, in: Proceedings of the IEEE International Conference on Industrial Technology, Vol. 2018-Febru, IEEE, 2018, pp. 1651–1657, http://dx.doi.org/10.1109/ICIT.2018.8352429.

[30] J. Martins, A. Tavares, M. Solieri, M. Bertogna, S. Pinto, BAO: A lightweight static partitioning hypervisor for modern multi-core embedded systems, OpenAccess Ser. Inform. 77 (3) (2020) 1–3, http://dx.doi.org/10.4230/OASIcs.NG-RES.2020.3.

[31] S. Pinto, J. Pereira, T. Gomes, A. Tavares, J. Cabral, Ltzvisor: TrustZone is the key, in: Euromicro Conference on Real-Time Systems, 2017, pp. 117–128.

[32] Z. Hua, J. Gu, Y. Xia, H. Chen, B. Zang, H. Guan, VTZ: Virtualizing ARM trustzone, in: Proceedings of the 26th USENIX Conference on Security Symposium, SEC '17, USENIX Association, USA, 2017, pp. 541–556.

[33] N. Borgioli, M. Zini, D. Casini, G. Cicero, A. Biondi, G. Buttazzo, An I/O virtualization framework with I/O-related memory contention control for real-time systems, IEEE Trans. Comput.-Aided Des. Integr. Circuits Syst. 41 (11) (2022) 4469–4480, http://dx.doi.org/10.1109/TCAD.2022.3202434.

[34] H. Li, M. Xu, C. Li, C. Lu, C. Gill, L. Phan, I. Lee, O. Sokolsky, Multi-mode virtualization for soft real-time systems, in: 2018 IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS, 2018, pp. 117–128, http://dx.doi.org/10.1109/RTAS.2018.00022.

[35] M.-c. Platforms, S. Whitepaper, PikeOS Safe Real-Time Scheduling Adaptive Time-Partitioning Scheduler for EN 50128 certified Multi-Core Platforms PikeOS Safe-Real Time Scheduling, Tech. rep., SYSGO, 2016, pp. 1–6.

[36] G. Klein, K. Elphinstone, G. Heiser, J. Andronick, D. Cock, et al., seL4: formal verification of an OS kernel, in: Proc. of the ACM SIGOPS 22nd Symposium on Operating Systems Principles (SOSP '09), 2009, pp. 207–220.

[37] SYSGO, Pikeos RTOS & hypervisor, 2021, https://www.sysgo.com/pikeos Accessed: 2021-04-01.

[38] Blackberry, QNX hypervisor, 2021, https://blackberry.qnx.com/en/software-solutions/embedded-software/qnx-hypervisor Accessed: 2021-04-01.

[39] FentISS, Xtratum, 2021, https://fentiss.com/products/hypervisor/ Accessed: 2021-04-01.

[40] VMware, Performance Optimizations in VMware vSphere 7 . 0 U2 CPU Scheduler for AMD EPYC Processors, Tech. rep., VMware, 2021, pp. 8–26.

[41] A. Tellabi, C. Ruland, Empirical study of real-time hypervisors for industrial systems, in: Proceedings - 6th Annual Conference on Computational Science and Computational Intelligence, CSCI 2019, 2019, pp. 208–213, http://dx.doi.org/10.1109/CSCI49370.2019.00042.

[42] F. Hofer, M.A. Sehr, A. Iannopollo, I. Ugalde, A. Sangiovanni-Vincentelli, B. Russo, Industrial control via application containers: Migrating from bare-metal to IAAS, in: 2019 IEEE International Conference on Cloud Computing Technology and Science (CloudCom), IEEE, 2019, pp. 62–69, http://dx.doi.org/10.1109/CloudCom.2019.00021, arXiv:1908.04465.

[43] M. Cinque, R.D. Corte, A. Eliso, A. Pecchia, RT-cases: Container-based virtualization for temporally separated mixed-criticality task sets, Leibniz Int. Proc. Inform. LIPIcs 133 (5) (2019) 1–5, http://dx.doi.org/10.4230/LIPIcs.ECRTS.2019.5.

[44] Y. De Bock, S. Mercelis, J. Broeckhove, P. Hellinckx, Real-time virtualization with xvisor, Internet Things 11 (2020) 100238, http://dx.doi.org/10.1016/j.iot.2020.100238.

[45] C.F. Yang, Y. Shinjo, Obtaining hard real-time performance and rich linux features in a compounded real-time operating system by a partitioning hypervisor, in: VEE 2020 - Proceedings of the 16th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, 2020, pp. 59–72, http://dx.doi.org/10.1145/3381052.3381323.

[46] C. Scordino, I.M. Savino, L. Cuomo, L. Miccio, A. Tagliavini, M. Bertogna, M. Solieri, Real-time virtualization for industrial automation, in: IEEE International Conference on Emerging Technologies and Factory Automation, ETFA, Vol. 2020-Septe, 2020, pp. 353–360, http://dx.doi.org/10.1109/ETFA46521.2020.9211890.

[47] Z. Jiang, K. Yang, Y. Ma, N. Fisher, N. Audsley, Z. Dong, I/o-GUARD: Hardware/software co-design for I/O virtualization with guaranteed real-time performance, in: 2021 58th ACM/IEEE Design Automation Conference, DAC, IEEE Press, 2021, pp. 1159–1164, http://dx.doi.org/10.1109/DAC18074.2021.9586156.

[48] D. Casini, A. Biondi, G. Cicero, G. Buttazzo, Latency analysis of I/O virtualization techniques in hypervisor-based real-time systems, in: 2021 IEEE 27th Real-Time and Embedded Technology and Applications Symposium, RTAS, 2021, pp. 306–319, http://dx.doi.org/10.1109/RTAS52030.2021.00032.

[49] ITCandor, Share of the global server market in the first half of 2018 and 2019, 2020, https://www.statista.com/statistics/915091/global-server-share-physical-virtual/.

[50] Cyclictest, 2021, https://wiki.linuxfoundation.org/realtime/documentation/howto/tools/cyclictest/start Accessed: 2021-02-01.

[51] Stress, 2021, https://packages.debian.org/stretch/devel/stress Accessed: 2021-02-01.

[52] R. Queiroz, T. Cruz, P. Simões, Real-time task latency in general-purpose hypervisors - testing the limits, 2022, http://dx.doi.org/10.21227/h7ye-x037.

[53] R. Delgado, B.-J. You, B.W. Choi, Real-time control architecture based on xenomai using ROS packages for a service robot, J. Syst. Softw. 151 (2019) 8–19, http://dx.doi.org/10.1016/j.jss.2019.01.052.

Rui Queiroz is an Ph.D student at the Department of Informatics Engineering of the University of Coimbra (UC), also being a researcher at the Centre for Informatics and Systems of the UC. His research interests encompass areas as diverse as communications infrastructures management, software defined networking and critical infrastructure security, being currently focused on the topics of virtualized realtime execution environments and communications infrastructures.

Tiago Cruz received the Ph.D. degree in informatics engineering from the University of Coimbra, Coimbra, Portugal, in 2012. He has been an Assistant Professor in the Department of Informatics Engineering, University of Coimbra, since December 2013. His research interests include areas such as management systems for communications infrastructures and services, critical infrastructure security, broadband access network device and service management, Internet of Things, software defined networking, and network function virtualization (among others). He is the author of more than 80 publications, including chapters in books, journal articles, and conference papers. Dr. Cruz is a member of the IEEE Communications Society.

Paulo Simões received the Doctoral degree in informatics engineering from the University of Coimbra, Coimbra, Portugal, in 2002. He is an Assistant Professor in the Department of Informatics Engineering, University of Coimbra, where he regularly leads technology transfer projects for industry partners such as telecommunications operators and energy utilities. His research interests include future Internet, network and infrastructure management, security, critical infrastructure protection, and virtualization of networking and computing resources. He has more than 150 publications in refereed journals and conferences. Dr. Simões is a member of the IEEE Communications Society.