Integrating Containers and Partitioning Hypervisors for Dependable Real-time Industrial Clouds

Marco Barletta*, Francesco Boccola†, Marcello Cinque*, Luigi De Simone*, Raffaele Della Corte*, Daniele Ottaviano*
DIETI - Università degli Studi di Napoli Federico II, Via Claudio 21, 80125 Napoli, Italy
*{marco.barletta, macinque, luigi.desimone, raffaele.dellacorte2, daniele.ottaviano}@unina.it
†f.boccola@studenti.unina.it

***Abstract*—A major challenge in Industry 4.0/5.0 is to meet stringent real-time and dependability requirements for cloud-native components, which are starting to be adopted to streamline the development, update, and reconfiguration of industrial applications. This paper addresses this challenge by proposing the notion of partitioned containers: virtual machines running on partitioning hypervisors but seen as containers from the orchestration perspective. Partitioning containers pave the way to novel orchestration primitives that take advantage of the heterogeneity of industrial infrastructures and ease the management of real-time critical applications. We implement a Jailhouse-based framework to build and seamlessly deploy partitioned containers across heterogeneous industrial platforms. We use a probabilistic model to drive orchestration decisions by considering factors such as fault tolerance, replication, and the unpredictability of complex hardware. Preliminary experiments show the feasibility of partitioned containers in industrial settings.**

***Index Terms*—Industrial applications, Cloud, Orchestration, Containers, Virtualization, Isolation**

I. Introduction

The Industry 4.0/5.0 vision relies on the replacement of hardware elements with cloud-native components to simplify the development and management of industrial applications and reach a sustainable, flexible, and resilient environment.

Traditional cloud-native applications use virtualization to simplify the management of services and guarantee elasticity, flexibility, resiliency, and cost-effective resource usage. Hence, in cloud environments mainly used virtualization technologies prioritize consolidation over isolation, like OS-level virtualization (i.e., containers) integrated with orchestrators (e.g., Kubernetes). However, cloud-native components running in containers and virtual machines (VM) [1] cannot yet guarantee low and predictable response times, availability, and reliability required by industrial settings.

Conversely, the recent adoption of virtualization for industrial applications (e.g., automotive and avionic) is pushed by the need to achieve isolation while consolidating systems to meet Size, Weight, Power, and Cost (SWaP-C) requirements [2]–[4]. Thus, simplicity is prioritized over features to cope with certification (e.g., ARINC-653, DO-178C, ISO 26262).

Partitioning hypervisors are gaining the limelight in this perspective. They are a small software layer that exploits hardware-assisted virtualization (e.g., Intel VT-x, ARM VHE) to statically allocate hardware resources to a reduced set of VMs [5]–[9]. Such VMs generally include applications compiled with minimal Real-Time Operating Systems (RTOS) and run bare-metal on the resources allotted to them.

Nevertheless, partitioning hypervisors require a remarkable manual configuration effort, and are not supported by cloud orchestration tools modified to support real-time and critical applications, like our previous proposal [10]. Hence, real-time critical applications cannot benefit from easy and automated management at a scale, which includes deployment, networking, scaling, and migration. The complexity is exacerbated by the heterogeneity of nodes in industrial edge settings, which may span from low-end multi-core embedded boards (even with asymmetric cores, such as application and real-time processing units) to fully-fledged server machines. Currently, such heterogeneity is seen as an obstacle for cloud technologies, as the same application must be recompiled or rewritten for a different operating system or instruction set architecture.

In this paper, we aim instead to take advantage of industrial nodes' heterogeneity by introducing the notion of *partitioned containers*, as the convergence between container orchestrators and partitioning hypervisors. A partitioned container is an application running bare-metal in an isolated partition of a board managed by a partitioning hypervisor, but seen and managed as any other container by cloud orchestrators. Hence, partitioned containers provide a way to pack industrial applications, which can run on diverse platforms of a cluster featuring diverse implementations.

The inherent diversity allows for meeting stringent dependability requirements. In particular, we introduce unprecedented orchestration primitives to leverage diversity, namely *Diverse Replication*, *Seamless Migration*, and *Diversified Rolling Update*. For instance, a controller can be replicated on a fully-fledged edge cloud server and on an embedded board to avoid common mode failures through the diversity of the implementation and platform. If a replica fails, it could be migrated to yet another different platform, seamlessly.

We design and implement a framework to build and deploy partitioned containers. Furthermore, we design a probabilistic model for real-time tasks run in standard or partitioned containers deployed over a cluster of heterogeneous nodes. The model unifies the management of fault tolerance and timing unpredictability, and it can be used by the orchestration system when applying our orchestration primitives to plan the deployment of tasks implemented with diversity to meet the

required dependability level.

In summary, our contributions are: *i)* the introduction of partitioned containers and a set of novel orchestration primitives; *ii)* a framework that implements partitioned containers; *iii)* a probabilistic model that accounts for containers implemented with diversity to drive orchestration decisions; *iv)* preliminary experiments to show the feasibility of partitioned containers.

II. Partitioned Containers

Containers are commonly defined as a standard way of packing applications together with their required libraries, forming a *container image* [11]. Following this definition, a real-time application (e.g., a periodic POSIX thread) could be either compiled against Linux libraries to become a Linux container or compiled against a library RTOS (e.g., Zephyr, NuttX), to be packed as a (bare-metal) container.

In our vision, partitioned containers can be regarded as single binary real-time applications, which include their library RTOS (i.e., libOS + RTOS, the ancestors of modern unikernels [12]) to run in an isolated VM on a partitioning hypervisor. Cloud orchestrators can manage partitioned containers as any other container.

Although the Open Container Initiative (OCI) specifies [13] how to divide a container image into layers to reuse the layers containing libraries, an image containing only a single executable file is still compliant with the OCI specification.

We define a platform as the hardware/software determining the binaries within a container image. Thus, different platforms require different container images for the same application.

For standard containers, which are based on general-purpose OSes (e.g., Linux Docker containers), the platform is composed of the OS and the instruction set architecture (ISA). For example, the same application must provide two different container images for Linux x86-64 and ARM64 platforms.

On the other hand, partitioned containers use bare-metal images, which assume direct hardware control without any virtualization layer. Hence, the device drivers used are specific to a class of hardware within nodes, and a different partitioned container image must be provided for each platform willing to host the partitioned container. Although an image works only for a subset of nodes, the variety of boards in an environment is usually limited.

The availability of the same container for multiple platforms enables unseen orchestration primitives, particularly relevant for industrial settings. Such primitives are defined as follows:

• **Diverse Replication:** When an application should be replicated for either redundancy (often dictated by safety-related standards) or scalability reasons, this primitive allows generating replicas considering additional constraints. The two main considered constraints are (i) *criticality*, indicating the criticality of the replicas, and (ii) *replication mode*, indicating how to configure the replicas, e.g., in Triple Modular Redundancy (TMR), 2-out-of-2 configuration using different networks. Given the number and criticality of replicas, along with the replication mode, the primitive enables easy deployment of diverse replicas by selecting the nodes to spawn the containers to comply with the constraints. For example, an orchestrator can spawn a Linux container plus two partitioned containers across three different platforms for an application requiring three replicas. Diversity is guaranteed thanks to the different container images used.

• **Seamless Migration:** This primitive allows the seamless migration of containers between potentially heterogeneous platforms while accounting for the isolation guarantees provided by the nodes. When a migration is required (e.g., upon a node failure), the primitive allows respawning a container onto a different platform with the same isolation guarantees of the previous one. For example, the primitive transparently migrates a container from a Xenomai-based node (popular co-kernel for real-time) to another real-time Linux-based one to achieve similar timing guarantees. If no node providing comparable isolation guarantees is available, the primitive can select a node with lower guarantees to provide a possibly degraded service. For instance, a complex controller runs on a powerful Linux-based edge server to leverage large computing resources. Upon a failure, the controller can be migrated as a partitioned container to a different platform to run a simpler implementation, which can only provide minimal functionality or bring the system to a safe state

• **Diversified Rolling Update:** Rolling updates gradually replace the running application containers with updated ones, guaranteeing a minimum number of running containers to avoid downtime. In this regard, the primitive allows performing a diversified rolling update, where the containers to update for each round are selected according to the platform. The idea is to always keep diverse containers running on different platforms to prevent common mode failures due to regression faults affecting the same platform. Thus, during the update, the primitive selects a number of diverse (if possible) containers to stop, starts their updated version, and when the spawned containers are ready a new update round starts.

III. Proposed Framework

In the proposed framework, as in standard orchestration systems, the user defines the desired state for a cluster of nodes. The desired state includes not only information about running containers, but also details about their requirements in terms of isolation levels, physical resources, real-time constraints, and other specific needs.

Each cluster node declares the maximum level of isolation that can be provided to a container. While Linux-based (both single and co-kernel architectures, see §VI) containers can guarantee isolation only to a certain extent [10], a node supporting partitioned containers can offer the maximum level of isolation available.

The orchestration system assigns each container to a node that can host it, i.e., the node *i)* has an image of the container available (see §III-A), *ii)* provides an isolation level that matches the container's requirements, *iii)* has the necessary resources available (see §III-B).

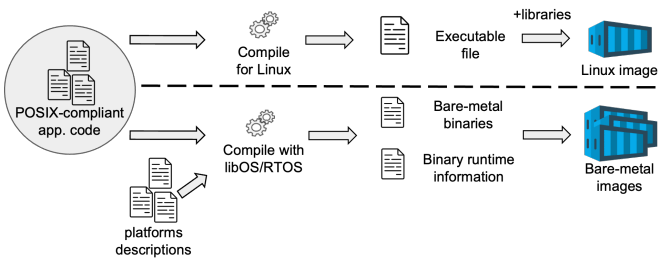

Fig. 1. Image building workflow for partitioned containers.

A. Partitioned Containers Image Building

To have an image available for a particular node, the building process must integrate the creation of the images used for partitioned containers.

Provided that a POSIX-compliant application must be packaged in a container, the same source code can be used to build both a standard Linux container and bare-metal images. The image building process is depicted in Fig. 1.

When building for Linux, the application is compiled using the application toolchain, and then an image containing both the application and its required libraries is created. The OCI specifies a standard for describing these images, which are divided into layers that can be shared and reused by multiple containers. For example, the base layer of an image might be an Ubuntu image, containing Ubuntu libraries. A second layer includes the libraries required by the application, and a third layer contains the application binary. When the application changes and a new image is built, only the last layer needs to be modified and downloaded.

When building for a bare-metal image, a POSIX-compliant RTOS or libOS (e.g., Zephyr, VxWorks, NuttX) can be used to pack the application and a minimal OS into a single bare-metal binary. In this context, the POSIX-compliant libOS RTOS acts as a minimal bare-metal runtime that abstracts hardware resources. The building process targets a specific node/boards, and relies on the platform descriptor typically shipped with the RTOS. During the building stage, in addition to producing bare-metal binaries, the process outputs *binary runtime information* files. These files contain the necessary information for the hypervisor to start a partitioned container. For example, the binary runtime information file may include the address space layout of the bare-metal binary, including the virtual start address of the binary, as described in §III-B2.

We evaluated other designs, such as compiling an application to WebAssembly (WASM) code and running it on a minimal WASM runtime. However, we found that support for real-time applications in WASM is limited and immature at the time of writing. Additionally, the technology readiness level of bare-metal WASM runtimes is still in its early stages.

B. Partitioned Containers Execution

Here, we depict the node stack needed for implementing partitioned containers, comparing it with the architecture of a Linux container stack (see Fig. 2).

1) Standard Linux Container Stack: An orchestration agent present on each node of the cluster interacts with the container manager through the Container Runtime Interface (CRI) API. This interface exposes functions to manage sandboxes (i.e., a group of application containers in an isolated environment with resource constraints) and their respective containers. The container manager tracks the state and lifecycle of containers, downloads or updates the container images, and manages virtual networks for the containers. Specifically, the container manager selects a suitable image version to download, according to the target OS and ISA. Next, the container manager relies on a shim daemon (typically one per container) to interact with the low-level container runtime and the container itself. The shim layer abstracts low-level runtimes and it exists as long as the controlled containers. The shim daemon interacts with the low-level container runtime through an API defined by the OCI standard, which mandates exposing at least a minimal set of functions to create, start, kill, delete, and get the status of a container. The low-level OCI runtime performs the system calls to configure the required resources for the container. For example, in the case of Linux containers, the low-level runtime performs system calls to configure the *cgroup* and *namespaces* for the container, creates the processes, and moves them into the container. Once a process runs in a container, its access to hardware resources is regulated by the kernel.

2) Partitioned Containers Stack: In the partitioned container architecture, the container manager is responsible for downloading images suitable for the specific platform.

The core of our proposed framework is *runPHI* [14]. runPHI is a low-level OCI runtime, responsible for creating, starting, killing, and deleting partitioned containers. Instead of relying on the containerization subsystem of the kernel, runPHI configures a static partitioning hypervisor to create a sandbox for the containers.

The static partitioning hypervisor statically allocates resources to VMs operating on application processing units (APUs), with APUs referring to the cores within an MPSoC equipped with MMUs for memory access protection. Our vision extends to the utilization of an *Omnivisor* [15], which expands upon the static partitioning hypervisor model. It enables partitioning to encompass co-processors lacking MMU protection, such as Real-Time Processing Units (RPUs) or soft cores on FPGA. This capability is essential in critical systems, where predictable cores, like RPUs, are required for high-criticality tasks. For instance, the Omnivisor can establish a partition featuring an RPU, a sensor, and an actuator, while leveraging paravirtualization to facilitate the exchange of network traffic through a management VM. Once the static partitioning through the Omnivisor is configured and started, runPHI relies on custom daemons to create a bridge with the partitioned containers. In particular, the daemons are in charge of maintaining the status consistency and the communication channels concerning the partitioned containers. The current network management in partitioning hypervisors lacks real-time guarantees. However, several approaches can enhance this

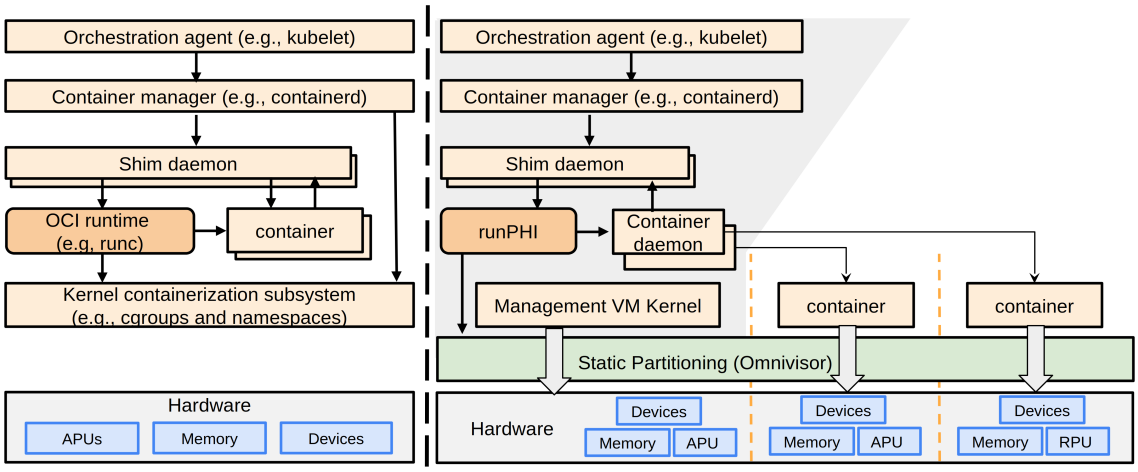

Fig. 2. Linux container stack (on the left) vs. partitioned container stack (on the right).

aspect. One approach is to use a broker to manage the network [16]. Another method involves using a board with multiple network interfaces partitioned and assigned to different VMs [17]. Additionally, hardware support such as SR-IOV with real-time capabilities can be used [18].

In Fig. 3, a high-level architectural description of runPHI internal is depicted. The topmost layer maintains compliance with the OCI APIs. Below that, an intermediate layer addresses the semantic mismatch between parameters and data structures belonging to containers and concepts associated with VMs. The intermediate layer relies on the bottom layer, which is hypervisor-specific. This hypervisor-specific layer contains commands to manage the partition lifecycle. Most partitioning hypervisors, as well as Omnivisors, rely on a configuration file to specify the resources assigned to a partition. Hence, a component that automatically generates the configuration file is necessary. This component must adhere to the indications of a resource manager, which tracks available resources on nodes over time. The output of the configuration generator is a file containing all details for running partitioned containers on the target platform. To achieve this, it relies on the information provided in a binary runtime information file shipped with the bare-metal image (see Fig. 1) and the container configuration. For instance, the image description contains the virtual start address where the binary must be loaded in memory, which is utilized to program the virtual-to-physical address translation provided by the Omnivisor. If necessary, the configuration generator may fill the gaps with predicted values for missing resources in the container description, according to requests from the orchestration platforms and the current usage of hardware.

IV. Model

We here introduce a model to show how the orchestration of partitioned containers can be considered in the system design. The main idea is to create a general framework that unifies the management of fault-tolerance and non-deterministic timing effects under a joint probabilistic model.

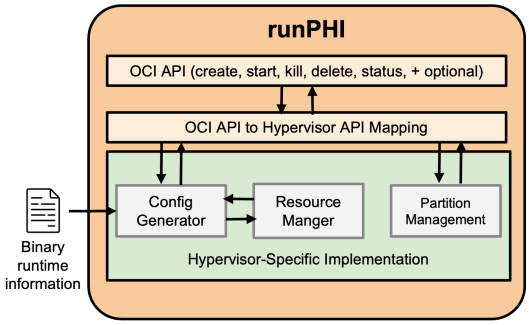

Fig. 3. High-level architectural description of runPHI, highlighting the interactions of the configuration generator and partition management.

Indeed, on one hand, in a distributed edge system, nodes have different hardware/software characteristics and hence timing guarantees. In some cases, it is either barely impossible or overly pessimistic to precisely define the Worst Case Execution Time (WCET) [19]. For example, a real-time Linux-based container running on complex Commercial Off-The-Shelf (COTS) hardware presents fewer timing guarantees due to interferences than industrial machinery equipped with a minimal real-time system.

On the other hand, simplistic assumptions are often used in fault tolerance techniques for real-time systems [20]. For example, assumptions include a well-defined $WCET$, perfect error detection, guaranteed time between faults/failures, limited fault/error model, etc. For example, in [20], the authors acknowledge that dealing with permanent failures different from total system failure is complex because they also include fail-partial and fail-slow behaviors.

The timing nondeterminism (due to complex hardware) and the fault tolerance are generally addressed separately. With the proposed model, we aim to deal with different node hardware/software determinism characteristics, schedulers, and fault tolerance techniques, with the same theoretic framework. This enables a holistic view of the system, in which we only care about a task Worst Case Response Time

(WCRT). The WCRT hides the complexity of models and assumptions local to a node, including failure behaviors, fault tolerance techniques, and timing determinism.

To this aim, we use the probabilistic-$WCET$ ($pWCET$ [21], [22]) random variable, to model both the inherent hardware/software unpredictability and the possible longer duration due to fault/failures handling or fail-slow behaviors. In this sense, different errors like crashes, unexpected longer executions, livelocks, and deadlocks, are all modeled like a tail in the $pWCET$ distribution.

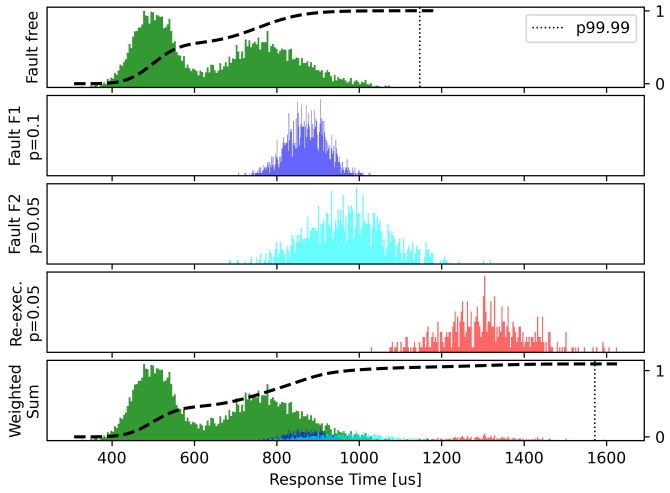

Fig. 4. Example of combined $WCRT$. The fault-free execution has a long tail due to interference caused by complex hardware. F1 and F2 are two different faults that cause a fail-slow behavior. For example, F1 is a fault affecting the task itself, while F2 is a fault affecting a higher priority task affecting the preemption time. A single task re-execution may be accounted for in the $WCET$, assuming a single transient fault. With a single $WCRT$ we can account for all of this weighted by its occurrence probability.

We model our edge cloud environment as composed of a set of microservices Γ^g deployed across the cluster, which provides functionalities to clients. A microservice is a set Λ_i of k task replicas $\tau_{i,j}$ (with $j \in [1,k]$) deployed across one or more nodes. Depending on the replication scheme, the replicas can be either identical for load-balancing purposes, or diverse for fault tolerance purposes.

Without a lack of generality, we assume a fixed-priority periodic task model, but any other task model using $pWCET$ would work. A task is defined as $\tau = (pW\vec{C}ET, T, D, p)$, where $pW\vec{C}ET$ is the vector of $pWCET$ defined for the cluster nodes that can host the task, T is the period, D is the relative deadline, and p the priority level. A task τ_i is assigned to only one cluster node. The $pWCET$ depends on the hardware/software characteristics of the node. In particular, the assurance provided by the node in terms of resource isolation, defined as $A = f(\alpha, \beta, \gamma)$ in [10], [23] determines the skewness of the $pWCET$. In this sense, the $pWCET$ accounts for nondeterminism in hardware contention and resource interference, but also fail-slow behavior, recovery blocks, and forward error recovery.

Multiple tasks $\tau_i, ..., \tau_k$ may be assigned to the same node, competing for hardware resources and scheduled by the algorithms characterizing a node. Thus, a stochastic response time analysis determines how, from the task set on the node, a $pWCRT$ (probabilistic $WCRT$ [24]) can be derived for each task. We leave undefined the response time analysis formula used because it could be different for each node. In this sense, the $pWCRT$ accounts for the scheduling algorithms along with task re-execution, dropping, degradation, etc. that may be implemented on the node. An example is shown in Fig. 4

A task fails if it is not able to respect its deadline (timing failure) or it does not produce a correct output (including a wrong result, or omission and crash failures). The failure probability of a task can thus be defined as shown in Equation 1, where *"correct val."* means that the task produces an output and the output is correct. Note that the event that a task does not produce a correct output can be modeled as an infinite $WCRT$, lying in the $pWCRT$ distribution.

$$\begin{aligned} P(Failure_{\tau_i}) = P(pWCRT_{\tau_i} > D_{\tau_i}) = \\ = P(pWCRT_{\tau_i} > D_{\tau_i} | \tau_i \text{ correct val.}) * P(\tau_i \text{ correct val.}) + \\ + P(\neg \tau_i \text{ correct val.}) \end{aligned} \tag{1}$$

A platform guaranteeing a high degree of isolation therefore has a lower failure probability compared to a platform that prefers workload consolidation to isolation.

The task replicas composing a microservice may present implementation diversity, and the nodes on which they are deployed may have different assurance levels. This translates to a deeply different $pWCET$, and consequently $pWCRT$, distribution for each task replica. Thus, each task $\tau_{i,j} \in \Lambda_i$ (with $j \in [0,k]$) has its own $P(Failure_{\tau_{i,j}})$.

The failure probability of a microservice depends on the fault tolerance scheme adopted. For example, assuming that the *"at least one"* scheme is adopted (i.e., the request is sent to all the task replicas, and the first response is used), the microservice fails to respond if all the task replicas in Λ_i fail to respond, as expressed in Equation 2, which assumes independent failures.

$$\begin{aligned} P(Failure_{\Lambda_i}) = \prod_{j \in [1,k]} P(Failure_{\tau_{i,j}}) = \\ = \prod_{j \in [1,k]} P(pWCRT_{\tau_{i,j}} > D_{\tau_i}) \end{aligned} \tag{2}$$

We plan to extend the model to include applications represented by Direct Acyclic Graphs (DAG) of microservices and consider the graph response time, similarly to [25].

V. Preliminary Experiments

We implemented a Jailhouse-based prototype of the proposed framework to show the feasibility and potential of partitioned containers. The runPHI implementation (see §3) compiles the partition configuration file, loads the binary into the partition, and finally starts it. We measured two metrics

commonly of interest for containers: *boot times* and *isolation from interference*. We run the experiments on two ARM-based boards commonly used in embedded edge computing scenarios [26]: a Raspberry Pi 4B and a Xilinx Zynq UltraScale+ MPSoC ZCU104 (hereafter ZCU104).

The Raspberry Pi software setup included Docker v24.0.5 using runc v1.1.11, a Linux kernel v5.15 patched with `PREEMPT_RT`, and configured for real-time (debug options disabled, no frequency scaling), `CONFIG_RT_GROUP_SCHED` enabled, and cgroups v1 configured. The Docker daemon was configured to have all the cpu-rt-runtime available. Inside the container, tasks run at 95 FIFO priority. The ZCU104 software setup included the Jailhouse hypervisor v0.12 patched with the Omnivisor extension [15] to run VMs on heterogeneous cores. The privileged VM in Jailhouse (i.e., root-cell) runs Linux kernel v5.15 patched with `PREEMPT_RT`. In the RPUs, Zephyr (a popular lib-OS RTOS) v3.22.1 runs as a VM.

To compare partitioned containers against Linux containers, we utilized POSIX-compliant applications. We compiled the same application source code twice: once for a Linux process running within an Ubuntu Linux container, and once against Zephyr to run bare-metal as a partitioned container. Note that the code of the partitioned container is moved to the Tightly Coupled Memory (TCM) (i.e., programmable on-chip memories) of the RPU before running.

A. Boot Times

In each experiment, we measured boot times by sampling the container creation timing. The start time was recorded right before the execution of the low-level container runtime (i.e., runc for Linux-based container, runPHI for partitioned containers), while the end time was sampled as the first instruction executed within the container. At the end of each experiment, we cleaned the caches in both testbeds to avoid dependencies. The image used in the experiment for the Linux container is an Ubuntu image (76MB) containing our executable file. The image used for the partitioned container is a micro-ROS (16.8MB) compiled to run over an RPU of the ZCU104.

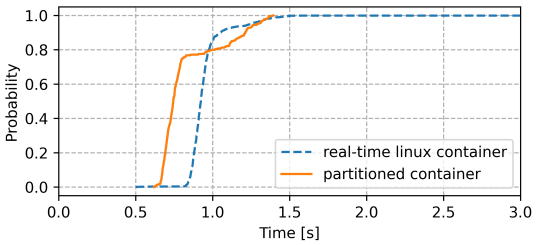

Fig. 5. Cumulative distribution function of the boot times of the solutions compared. The solutions show comparable boot times.

Fig. 5 shows the results. The time to boot a partitioned container is comparable to a standard Linux container. Breaking down its boot times, loading and starting the application on the RPU takes $0.142 \pm 0.001s$. The rest of the time, which contributes to almost the totality of the variability, is spent compiling the Jailhouse partition configuration file.

As shown in [15], the application loading times with the Omnivisor depend linearly on the image size. On the other hand, Linux container boot times are comparable with the ones measured in [27], where authors showed that multiple factors can impact Linux containers' boot times.

B. Isolation from Interference

We compared the isolation from interference between a partitioned container and a Linux real-time container. The aim is to show the potential of integrating the Omnivisor into container orchestration, rather than evaluating the isolation of partitioning hypervisor, already explored in-depth in [28]. This allows leveraging the flexibility provided by container orchestration tools while keeping the isolation.

We compiled a periodic POSIX task performing matrix calculations for both Linux and Zephyr OS (targeting the ZCU104 RPU). We used the same compilation flags in both building processes. In each experiment, we run the task for 1000 iterations and measure the execution time of each iteration. We run the experiments in varied stress conditions, including an experiment with no co-located stress on the node and experiments with co-located stress generated through *stress-ng*. In particular, for each experiment, we apply one of two stress types between *memcpy* and *udp*, and we repeat the experiment with increasing stress intensities, i.e., 1, 2, 4, and 8 threads. These tests were chosen because they stress memory and interrupt handling subsystems, known to be the two major sources of interference, even in partitioning systems [28].

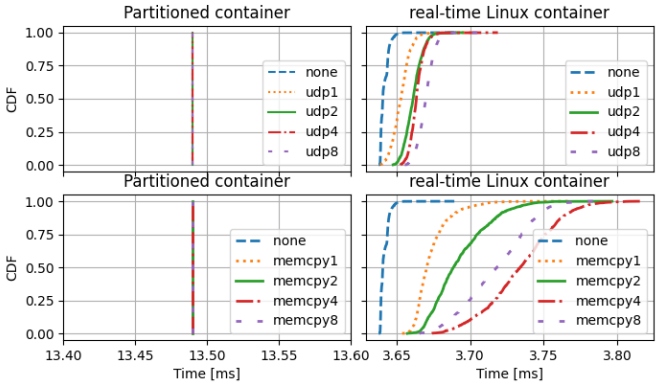

Fig. 6. Cumulative distribution function of the execution times in varied stress conditions. Partitioned containers show constant times.

Fig. 6 shows the results. The partitioned container is about 3 times slower than the real-time Linux container: this is due to the RPU clock frequency of the ZCU104, which is one-third of the clock frequency of the Raspberry Pi. However, the partitioned container outperforms the real-time Linux container in terms of predictability. Despite the co-located stress, the partitioned container exhibits the same execution time (with a μs resolution) across all test repetitions. Indeed, the code loaded in the TCM allows the application in the partitioned container to avoid resource contention regarding cache and RAM.

VI. Related Work

A. Real-time Containers

Consolidated real-time operating systems like VxWorks by WindRiver recently introduced the support of an OCI-compliant container engine [29]. Besides industry products, research studies also explored the use of containers to run real-time tasks. Interesting surveys of those studies can be found in [30], [31]. The most popular solutions include i) the use of Linux with the `PREEMPT_RT` patch [32], and a patched real-time group scheduling (rt-cgroups) [33]; and ii) the use of real-time Linux co-kernels, such as RTAI and Xenomai [34], [35], and hierarchical schedulers or distributed monitors to manage the hard real-time tasks belonging to containers.

The drawback of real-time containers proposed to date is the bloated code base on which they rely, which prevents any sound theoretical analysis and possible certification. Furthermore, Linux-based containers usually do not support advanced isolation mechanisms (e.g., cache and bank partitioning, and hardware virtualization support).

B. Hypervisor-based Containers

Sandboxed containers integrate the advantages of containers while inheriting hypervisor-based isolation. They allow running the containerized applications inside minimal VMs or unikernels in order to exhibit overhead and startup times comparable to containers relying on OS-level virtualization. *IBM Nabla* [36] and *LightVM* [37] build containers on top of unikernels. *Google gVisor* [38] creates a dedicated guest kernel for running containers. Amazon Firecracker is a hypervisor designed to run *miroVMs* hosting sandboxed applications [39]. *Kata Containers* [40] are placed in a dedicated VM with a prebuilt kernel optimized for orchestration. Xilinx proposed *RunX* [41], which exploits Xen to run containers as VMs.

All the analyzed solutions are based on general-purpose or type-2 hypervisors, which can provide isolation in terms of security but do not guarantee a level of resource and performance isolation suitable for hard real-time systems. Finally, some proposals explored the use of paravirtualized unikernels on top Xen for real-time contexts [42].

C. Partitioning Hypervisors

Partitioning hypervisors are designed specifically for industrial environments, due to the high-level of resource isolation and the reduced trusted code base, which enables certification. An example is *Jailhouse* [43], a Linux-based partitioning hypervisor developed by Siemens. *Jailhouse* enables asymmetric multiprocessing to assign hardware resources to isolated partitions at runtime. *Bao* [5] is a lightweight bare-metal hypervisor for mixed-criticality IoT systems, providing strong isolation, fault-containment, and real-time features. It does not rely on Linux, but the partitions are statically defined at boot time. *Xtratum* [44] is a paravirtualized partitioning hypervisor providing strong temporal and memory isolation. *PikeOS* [45] is a commercial hypervisor from SYSGO, used in several industrial domains. It was proposed in conjunction with an RTOS and Linux to consolidate both hard real-time workloads and distributed processing scenarios [46]. In [15] the authors propose the Omnivisor model. The model extends the capability of static partitioning hypervisors to run VMs on co-processors such as RPU or soft-cores while keeping the partition isolated from a temporal and spatial point of view. Partitioning hypervisors do not provide any means to automate the application deployment, reconfiguration, orchestration of partitions via cloud tools and DevOps workflows.

D. Fault-tolerance in Real-time Systems

In [20] common fault tolerance techniques in real-time systems are surveyed, while in [24] methods for integrating fault tolerance into probabilistic schedulability analyses are surveyed. The surveyed papers are divided by fault tolerance technique adopted: replicas/standby, re-execution, recovery blocks, and checkpoint/restart. In some papers, like [47], a combination of both spatial and temporal redundancy is used for critical and non-critical code sections respectively. Those papers have different assumptions/fault models. The assumptions include, for example, deterministic $WCET$, minimum fault interarrival times, or maximum fault duration. In [24] algorithms for probabilistic response time analysis are also surveyed. They mainly rely on convolutions of probability distributions. In [48] the $pWCET$ is used to account for the penalty caused by faulty cache banks, weighted for the fault probability. The topic of fault tolerance for real-time tasks in edge and cloud environments has been investigated [49], [50]. In [25] the authors model an application as a graph of microservice and determine the deployment that can guarantee the response time while minimizing the number of replicated tasks. In [51] authors proposed a scheduling for executing cloud real-time applications that meet the reliability required.

These works mainly regard the scheduling of tasks and do not consider tasks implemented with diversity as we do.

VII. Conclusion

We presented a novel framework for the orchestration of partitioned containers, bridging the gap between partitioning hypervisors and cloud orchestration systems. The framework enables new orchestration primitives, i.e., diverse replication, seamless migration, and diversified rolling update. Those primitives mainly rely on the heterogeneity of architectures and operating systems, typical of industrial scenarios. Additionally, we proposed a probabilistic model of the system that uses the primitives, jointly accounting for hardware nondeterminism and faults/errors. Preliminary experiments demonstrated the efficacy of our proposal. Partitioned containers exhibited comparable boot times with regard to Linux containers, while maintaining superior isolation, thereby reducing the likelihood of deadline misses and system failures.

Acknowledgment

This work is partially supported within the MICS (Made in Italy – Circular and Sustainable) Extended Partnership and received funding from the European Union Next-GenerationEU (PIANO NAZIONALE DI RIPRESA E RESILIENZA (PNRR) – MISSIONE 4 COMPONENTE 2, INVESTIMENTO 1.3 – D.D. 1551.11-10-2022, PE00000004).

REFERENCES

[1] D. C. Van Moolenbroek, R. Appuswamy, and A. S. Tanenbaum, "Towards a flexible, lightweight virtualization alternative," in *Proc. SYSTOR*, 2014, pp. 1–7.

[2] Bloomberg. (2022) Automotive Chip-Shortage Cost Estimate Surges to $110 Billion. [Online]. Available: https://tinyurl.com/2p9akbhu

[3] BlackBerry Limited. (2021) Are Hypervisors the Answer to the Coming Silicon Shortages? (White Paper). [Online]. Available: https://blackberry.qnx.com/content/dam/blackberry-com/Documents/pdf/BlackBerry_QNX_Hypervisor_WhitePaper_22April2021_FINAL.pdf

[4] M. Cinque, D. Cotroneo, L. De Simone, and S. Rosiello, "Virtualizing mixed-criticality systems: A survey on industrial trends and issues," *Elsevier FGCS*, 2021.

[5] J. Martins, A. Tavares, M. Solieri, M. Bertogna, and S. Pinto, "Bao: A lightweight static partitioning hypervisor for modern multi-core embedded systems," in *Proc. NG-RES*. Schloss Dagstuhl - LZI, 2020.

[6] Siemens AG. Jailhouse hypervisor source code. [Online]. Available: https://github.com/siemens/jailhouse

[7] P. Lucas, K. Chappuis, B. Boutin, J. Vetter, and D. Raho, "VOSYS-monitor, a TrustZone-based Hypervisor for ISO 26262 Mixed-critical System," in *Proc. FRUCT*. IEEE, 2018.

[8] H. Li, X. Xu, J. Ren, and Y. Dong, "Acrn: a big little hypervisor for iot development," in *Proc. VEE*. ACM, 2019, pp. 31–44.

[9] M. Masmano, I. Ripoll, A. Crespo, and J. Metge, "Xtratum: a hypervisor for safety critical embedded systems," in *Proc. RTLWS*. Citeseer, 2009, pp. 263–272.

[10] M. Barletta, M. Cinque, L. De Simone, and R. Della Corte, "Criticality-aware monitoring and orchestration for containerized industry 4.0 environments," *ACM TECS*, vol. 23, no. 1, pp. 1–28, 2024.

[11] A. Randal, "The ideal versus the real: Revisiting the history of virtual machines and containers," *ACM CSUR*, 2020.

[12] A. Madhavapeddy and D. J. Scott, "Unikernels: the rise of the virtual library operating system," *Communications of the ACM*, vol. 57, no. 1, pp. 61–69, 2014.

[13] OpenContainers. OCI Image Format Specification. https://github.com/opencontainers/image-spec.

[14] M. Barletta, M. Cinque, L. De Simone, R. Della Corte, G. Farina, and D. Ottaviano, "Runphi: Enabling mixed-criticality containers via partitioning hypervisors in industry 4.0," in *Proc. ISSREW*. IEEE, 2022, pp. 134–135.

[15] D. Ottaviano, F. Ciraolo, R. Mancuso, and M. Cinque, "The Omnivisor: A real-time static partitioning hypervisor extension for heterogeneous core virtualization over MPSoCs," in *Proc. ECRTS*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, 2024.

[16] G. Schwäricke, R. Tabish, R. Pellizzoni, R. Mancuso, A. Bastoni, A. Zuepke, and M. Caccamo, "A real-time virtio-based framework for predictable inter-vm communication," in *Proc. RTSS*. IEEE, 2021, pp. 27–40.

[17] G. Avon, A. Buscarino, E. De Marchi, L. Fortuna, A. C. Neto, F. Sartori, and F. Zanon, "Marte2 on arm platforms integration challenges: An asymmetric multiprocessing approach for the iter magnetics diagnostics," *Elsevier Fusion Engineering and Design*, vol. 202, p. 114370, 2024.

[18] Z. Jiang, N. Audsley, P. Dong, N. Guan, X. Dai, and L. Wei, "Mcs-iov: Real-time i/o virtualization for mixed-criticality systems," in *Proc. RTSS*. IEEE, 2019, pp. 326–338.

[19] E. A. Lee, "Cyber physical systems: Design challenges," in *Proc. ISORC*. IEEE, 2008, pp. 363–369.

[20] F. Reghenzani, Z. Guo, and W. Fornaciari, "Software fault tolerance in real-time systems: Identifying the future research questions," *ACM Computing Surveys*, vol. 55, no. 14s, pp. 1–30, 2023.

[21] G. Bernat, A. Colin, and S. M. Petters, "Wcet analysis of probabilistic hard real-time systems," in *Proc. RTSS*. IEEE, 2002, pp. 279–288.

[22] S. Bozhko, F. Marković, G. von der Brüggen, and B. B. Brandenburg, "What really is pwcet? a rigorous axiomatic proposal," in *2023 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2023, pp. 13–26.

[23] M. Barletta, M. Cinque, L. De Simone, and R. Della Corte, "Introducing k4. 0s: a model for mixed-criticality container orchestration in industry 4.0," in *Proc. DASC/PiCom/CBDCom/CyberSciTech*, 2022.

[24] R. Davis and L. Cucu-Grosjean, "A survey of probabilistic schedulability analysis techniques for hard real-time systems." *Leibniz Transactions on Embedded Systems (LITES)*, 2019.

[25] L. Abeni, R. Andreoli, H. Gustafsson, R. Mini, and T. Cucinotta, "Fault tolerance in real-time cloud computing," in *Proc. ISORC*. IEEE, 2023, pp. 170–175.

[26] A. Kalantar, Z. Zimmerman, and P. Brisk, "Fpga-based acceleration of time series similarity prediction: From cloud to edge," *ACM TRETS*, vol. 16, no. 1, pp. 1–27, 2022.

[27] M. Straesser, A. Bauer, R. Leppich, N. Herbst, K. Chard, I. Foster, and S. Kounev, "An empirical study of container image configurations and their impact on start times," in *Proc. CCGrid*. IEEE, 2023, pp. 94–105.

[28] J. Martins and S. Pinto, "Shedding light on static partitioning hypervisors for arm-based mixed-criticality systems," in *Proc. RTAS*. IEEE, 2023, pp. 40–53.

[29] Windriver. Containers at the Intelligent Edge. https://www.windriver.com/resource/containers-at-the-intelligent-edge. As of June 21, 2024.

[30] V. Struhár, M. Behnam, M. Ashjaei, and A. V. Papadopoulos, "Real-time containers: A survey," in *Proc. Fog Computing and the IoT*, 2020.

[31] R. Queiroz, T. Cruz, J. Mendes, P. Sousa, and P. Simões, "Container-based virtualization for real-time industrial systems—a systematic review," *ACM CSUR*, vol. 56, no. 3, pp. 1–38, 2023.

[32] F. Reghenzani, G. Massari, and W. Fornaciari, "The real-time linux kernel: A survey on preempt_rt," *ACM CSUR*, 2019.

[33] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the linux kernel," *ACM SIGBED Review*, 2019.

[34] M. Cinque, R. Della Corte, A. Eliso, and A. Pecchia, "Rt-cases: Container-based virtualization for temporally separated mixed-criticality task sets," in *Proc. ECRTS*, 2019.

[35] M. Barletta, M. Cinque, L. De Simone, and R. Della Corte, "Achieving isolation in mixed-criticality industrial edge systems with real-time containers," in *Proc. ECRTS*. Schloss Dagstuhl - LZI, 2022.

[36] IBM. (2022) Nabla containers: a new approach to container isolation. [Online]. Available: https://nabla-containers.github.io/

[37] F. Manco, C. Lupu, F. Schmidt, J. Mendes, S. Kuenzer, S. Sati, K. Yasukata, C. Raiciu, and F. Huici, "My vm is lighter (and safer) than your container," in *Proc. SOSP*, 2017.

[38] Google LLC. (2022) Google gVisor Homepage. [Online]. Available: https://gvisor.dev/

[39] A. Agache, M. Brooker, A. Iordache, A. Liguori, R. Neugebauer, P. Piwonka, and D.-M. Popa, "Firecracker: Lightweight virtualization for serverless applications," in *Proc. NSDI*, 2020.

[40] Kata Containers. (2022) Kata Containers Homepage. [Online]. Available: https://katacontainers.io/

[41] Xilinx. (2022) RunX. [Online]. Available: https://github.com/Xilinx/runx

[42] K.-H. Chen, M. Günzel, B. Jablkowski, M. Buschhoff, and J.-J. Chen, "Unikernel-based real-time virtualization under deferrable servers: Analysis and realization," in *Proc. ECRTS*. Schloss Dagstuhl - LZI, 2022.

[43] R. Ramsauer, J. Kiszka, D. Lohmann, and W. Mauerer, "Look mum, no vm exits!(almost)," *arXiv preprint arXiv:1705.06932*, 2017.

[44] A. Crespo, I. Ripoll, and M. Masmano, "Partitioned embedded architecture based on hypervisor: The XtratuM approach," in *Proc. EDCC*. IEEE, 2010.

[45] SYSGO GmbH . PikeOS home page. [Online]. Available: https://www.sysgo.com/pikeos

[46] E. Quiñones, S. Royuela, C. Scordino, P. Gai, L. M. Pinho, L. Nogueira, J. Rollo, T. Cucinotta, A. Biondi, A. Hamann *et al.*, "The ampere project:: A model-driven development framework for highly parallel and energy-efficient computation supporting multi-criteria optimization," in *Proc. ISORC*. IEEE, 2020, pp. 201–206.

[47] N. Chen, S. Zhao, I. Gray, A. Burns, S. Ji, and W. Chang, "Msrp-ft: Reliable resource sharing on multiprocessor mixed-criticality systems," in *Proc. RTAS*. IEEE, 2022, pp. 201–213.

[48] D. Hardy, I. Puaut, and Y. Sazeides, "Probabilistic wcet estimation in presence of hardware for mitigating the impact of permanent faults," in *Proc. DATE*. IEEE, 2016, pp. 91–96.

[49] S. Malik and F. Huet, "Adaptive fault tolerance in real time cloud computing," in *Proc. SERVICES*. IEEE, 2011, pp. 280–287.

[50] J. Wang, W. Bao, X. Zhu, L. T. Yang, and Y. Xiang, "Festal: fault-tolerant elastic scheduling algorithm for real-time tasks in virtualized clouds," *IEEE TC*, vol. 64, no. 9, pp. 2545–2558, 2014.

[51] M. Ghose, K. P. Pandey, N. Chaudhari, and A. Sahu, "Soft reliability aware scheduling of real-time applications on cloud with mttf constraints," in *Proc. CCGrid*. IEEE, 2023, pp. 459–468.