REACT: Enabling Real-Time Container Orchestration

Václav Struhár[1], Silviu S. Craciunas[2], Mohammad Ashjaei[1], Moris Behnam[1], Alessandro V. Papadopoulos[1]
[1]Mälardalen University, Västerås, Sweden
[2]TTTech Computertechnik AG, Vienna, Austria

***Abstract*—Fog and edge computing offer the flexibility and decentralized architecture benefits of cloud computing without suffering from the latency issues inherent in the cloud. This makes fog computing very attractive in real-time and safety-critical applications, especially if combined with container-based technologies. Whereas different orchestration systems are available to manage the container placement based on their resource demand, no orchestration system is considering real-time requirements for containerized applications. In this paper, we present the architecture and design of a real-time container orchestrator based on Kubernetes. Moreover, this paper defines metrics for the performance evaluation of real-time containers, and describes an initial model for allocating a mixture of real-time and non-real-time containers. We present an initial implementation of our real-time container extension and evaluate its feasibility on Linux-based systems.**

***Index Terms*—container-to-host mapping, real-time orchestration, real-time container-based virtualization**

I. Introduction

Fog and Edge Computing (FEC) are conceived to overcome some of the main limitations of cloud computing, e.g., tackling the limitations of unbounded communication latency and variance in bandwidth availability [1]. FEC enables the benefits of offloading data collection and decision-making even in application domains that require Real-time (RT) guarantees, e.g., in industrial automation [2], [3]. FEC systems decrease the communication latency for critical data, which can then be processed on the decentralized computational devices on the network's edge. While guaranteeing deterministic communication behavior between nodes can be achieved through the use of, e.g., TTEthernet or Time-Sensitive Networks (TSN), guaranteeing the RT behavior of computation functions can be challenging. The RT execution of functions on nodes cannot always be maintained in the same way due to the complexity and non-determinism of hardware artifacts (e.g., caches), and software layers within the compute nodes (e.g., OS layers, network stack, interrupts). Hence, to maintain the RT behavior of functions, isolation and adaptation/re-allocation of functions at run-time is necessary. Typically, the re-allocation and the isolation of functions have been facilitated through the use of virtualization technologies.

This work was partially supported by the European Commission under the project FORA with grant number 764785, by the Swedish Research Council (VR) under the project PSI, by the Knowledge Foundation (KKS) under the project SACSys, and by the project XPRES.

978-1-7281-2989-1/21/$31.00 ©2021 IEEE

Container-based virtualization is gaining importance in industrial domains as a lightweight alternative to full-blown virtualization [4], [5]. Containers are standalone self-containing software packages, comprising applications and their software dependencies, that simplify the deployment of software [6]. The technology enables sharing of computation resources among several containers while preserving isolation. Recently, there has been an effort to enhance container-based virtualization with time predictability (i.e., RT containers), enabling container applications with RT requirements [7]–[9]. Through these enhancements, containers may be used for time-critical applications required by many industrial systems such as robot control in fog computing [2], [3]. In parallel, due to the continuous change in resource usage by applications running in fog and the availability of the resources, there is a need to dynamically manage and deploy containers in a cluster of compute nodes necessitating container orchestrators that take into account both resource requirements and resource availability. Additionally, container orchestrators offer further benefits, e.g., scalability and availability, fault-tolerance, and efficient resource utilization [10].

RT systems require additional constraints regarding the reaction time to environmental events [11]. So far, container orchestrators do not consider RT requirements of applications running inside containers, neither from the theoretical nor practical points of view. There are no mechanisms of the distribution of containers based on their RT requirements without violating these requirements and how to manage the co-execution of RT and Best Effort (BE) containers on the same node. RT containers are a novel topic [9]. There are no strategies for dealing with dynamic changes in container workloads that may change interference between RT containers. Containers do not provide strict resource isolation as they share not only physical resources but also Operating System (OS) kernel [12], and hence, they are prone to performance degradation in the presence of other competing containers. For example, disk-intensive workloads can induce performance degradation up to 35% [13]. Therefore, if the benefits of containers are to be exploited for RT systems, run-time monitoring and container orchestration that continuously evaluates Quality of Service (QoS) metrics and uses them in container scheduling decisions is necessary.

This paper proposes a container orchestration architecture to support the co-existence of RT and BE containers with temporal isolation. Specifically, the contributions of this paper are as follows:

- The design of a container orchestration architecture enabling

the deployment and online adaptation of both RT and BE containers considering the timing requirements defined for time-critical applications;
- The definition of a mathematical model for the RT components of the designed orchestration architecture, and of the performance metrics needed to implement run-time monitoring and orchestration of the RT containers;
- The implementation of the proposed orchestration over the existing open-source container orchestrator Kubernetes.

II. Background and Prior Work

RT systems are computing systems that require deterministic temporal behavior since the correctness of their functionality depends not only on the value but also on the timing of the computed result [11]. Many software applications utilize periodic tasks that are cyclically executed [14]. Typically, RT systems require that tasks finish their execution in every period instance to be considered correct (assuming an implicit deadline for the tasks).

Virtualization is a technology that allows multiple virtualized applications or systems to be co-located onto the same shared platform [15]. There are two prevalent virtualization technologies: hypervisor- and container-based virtualization. Hypervisor-based virtualization utilizes a hypervisor that is a software layer that creates the different partitions within which each virtualized instance of an OS runs. In contrast, container-based virtualization utilizes OS features to create an isolated environment for processes. Container-based virtualization does not impose any requirements on hardware support for virtualization. Hence, such technology can be utilized on a broad range of devices. Containers co-located on a single computing node run as user-space isolated tasks and share functions of the host's OS (e.g., scheduling, memory allocation). From the user's view, each container appears and executes like a stand-alone OS. In comparison to the hypervisor-based virtualization, container-based virtualization has a negligible overhead, is more resource efficient (e.g., 29.4 times less RAM usage than a Virtual Machine (VM) [16]), has higher flexibility, provides fast booting times [17], and enables a near-native performance [4]–[6]. However, container-based virtualization provides a low level of isolation for memory, disk, and network operations [13]; hence, such operations may lead to degrading QoS of the applications.

The container-based virtualization relies on *namespaces* and *control groups*, referred to as *cgroups*, implemented in the host OS [18]. *Namespaces* partition global resources, e.g., tasks, network, inter-process communication, into different sets, only visible by different task groups [9]. *cgroups* enable the organisation of tasks into hierarchical subgroups with various configurable runtime properties, that allows a dynamic resources redistribution among the subgroups [19].

Looking at the RT support for container-based virtualization, three major directions are aiming to improve RT behavior of containers: hard RT co-kernel that co-exists with a standard Linux Kernel [20]–[22], solutions based on the *preempt_rt* patch for Linux that aims to minimize the latency in the Kernel [23]–[25], and solutions that employ hierarchical scheduling [7], [8], [18]. The RT properties of container-based virtualization depend on the underlying OS. In this paper, we consider the general-purpose OS Linux. While Linux was not designed for RT operation, there are considerable efforts to improve the time determinism on several levels, e.g., introducing RT scheduling as a standard kernel feature and providing time-predictable kernel behavior through full preemption. On the task scheduling level, there are RT scheduling policies in the Linux kernel: First-In-First-Out and Round Robin combined with priority queues to provide different priority levels for tasks, and Earliest Deadline First/Constant-Bandwidth Server that prioritizes tasks dynamically according to their deadlines.

RT scheduling support for containers has been added in [18] through the implementation of hierarchical scheduling combining standard fixed priority scheduler (*SCHED_FIFO* or *SCHED_RR* scheduling policy) and *SCHED_DEADLINE*, consistent with the multiprocessor periodic resource analysis from [26]. The kernel is extended with a reservation-based scheduling policy in which each virtual CPU (vCPU) is assigned a quota and a period, bounding the execution of the vCPU to the respective quota in each time interval of length equal to the period. On the container level, the global *SCHED_DEADLINE* policy selects at each time instant the container with the earliest deadline, while at the task level, the *SCHED_FIFO*/*SCHED_RR* policy is used to schedule tasks within containers. Once there is no task to be scheduled by the *SCHED_FIFO*, other BE tasks, are executed via the default scheduling policy. This enables the co-existence of RT and BE containers. We use this hierarchical patch as the implementation basis for our environment to host containers in compute nodes.

Container orchestrators automate the deployment, management, and scaling of containers in clusters of heterogeneous computing nodes. The main functionality of container orchestrators is the placement of containers in a cluster of compute nodes following placement policies and user-supplied placement constraints. The goal is to choose an optimal compute node to start the requested containers on. The orchestrator matches the resources requirements with the resource capacity of the nodes, e.g., CPU, memory, and disk storage capacity, and applies strategies to maximize the performance (e.g., the highest spread of containers). Additionally, orchestrators address fault-tolerance of the deployed containers, scaling or removing containers based, load balancing, container health monitoring, and efficient resource utilization. There are several container orchestrators available: Kubernetes, Docker Swarm, None of them support the deployment of containers based on their timing requirements [10]. Closest to our work is [27], where Xi et al. utilizes OpenStack to orchestrate RT VMs.

III. Orchestration of real-time containers

This paper introduces a solution to enable the orchestration of RT containers so that the RT requirements are taken into account during the scheduling process. We define a set of

scheduling policies, run-time monitoring mechanisms, performance metrics, and an implementation that supports the deployment and execution of RT containers. The RT container orchestrator provides the following functionalities:

- *Placement of RT and BE containers*: The orchestrator places the RT containers according to their RT requirements (i.e., quota and budget). The orchestrator prevents over-reservation of the resources at the container scheduling level by performing a utilization-bound schedulability test.
- *Run-time monitoring of RT QoS of the containers*: The orchestrator continuously monitors resource usage and the delivered QoS expressed by a set of metrics of the containers deployed in the compute nodes administrated by the orchestrator. As the OS does not enforce strong container isolation, there may be an interference with other noise-perturbing containers that may influence the timing behavior of RT containers. The orchestrator takes the information into account in the next scheduling decisions.

A. System Model

In this section, we define our system model including infrastructure, compute node, container, and task elements.

Infrastructure: We consider a system $\mathcal{S}$ consisting of a container orchestrator F and n connected compute nodes denoted by $f_1, \ldots, f_n$.

Compute node: Each node f_j offers memory and storage resources of a given capacity, denoted with f_j^M and f_j^S, respectively. Each compute node f_j is capable of hosting a mixture of RT containers and BE containers. We define the set of RT and BE containers on node f_j with Π_j^{rt} and Π_j^{be}, respectively. We introduce the Operating System Level Metrics (OSLM) property of a node f_j, denoted by $OSLM_j$, to quantify its performance.

Container: Each container π_k has memory (π_k^M) and storage (π_k^S) demands. An RT container $\pi_k \in \Pi_j^{rt}$ has RT interface expressed as (P_k, Q_k) where Q_k is a CPU quota over period P_k. We introduce the metric of a container π_k, denoted by CLM_k, to capture the performance of RT containers.

Task: Each container π_k accommodates a set of tasks denoted by $\mathcal{T}_k$. Each RT task τ_i is defined by the tuple $\langle a_i, C_i, T_i, D_i \rangle$, where a_i represents the release or arrival time (i.e., the time relative to the period when the task becomes active), C_i is the worst-case execution time bound, T_i is the period, and D_i is the relative deadline of the task. We use two functions $L_i(t_0, t_1)$, and $R_i(t_0, t_1)$ to capture the maximum lateness and maximum response time of the task τ_i in the interval $[t_0, t_1)$. The lateness represents the delay of the task's completion with respect to its deadline, while the response time measures the difference between the finishing and arrival time of a task. The functions return a set of lateness and response time values, respectively, corresponding to the task instances executed within the given time interval. We also use a counter defined as $\mu_i(t_0, t_1)$ to capture the number of deadline misses for task τ_i in the time interval $[t_0, t_1)$. Tasks assigned to BE containers do not have any timing requirements.

There are methods [28] to abstract the timing requirements of a set of RT tasks under certain scheduling algorithms into a periodic resource model in a form similar to the RT interface (Q_k, P_k) of the containers. If the appropriate scheduling mechanisms are in place, the abstracted resource guarantees that when the resource receives an allocation of Q_k over a period P_k, all RT tasks within the resource will meet their RT requirements [28].

B. Performance Metrics

There are several metrics, collectively defined as OSLM [29], to evaluate the performance of a system in terms of task execution. Such metrics are useful to estimate the suitability of the system to run RT tasks. E.g., Interrupts with a non-preemptable section that can influence the RT performance of the system [30], CPU Utilization, number of handled interrupts per second, number of I/O requests per second, and amount of data read/written. There is a number of tools for collecting performance data [31]: *mpstat*, *iostat*. These tools collect the performance data of tasks from *proc* and *cgroups* directories to estimate the OSLM.

On the container level, there is a lack of measurement methodology, tools, and best practices, as well as a lack of metrics on the characterization of the container overhead [31]. Available tools, e.g., *docker stats* and *cAdvisor* allow to estimate the basic set of container-related metrics (e.g., CPU and memory utilization). However, when considering the RT QoS evaluation of containers, the available tools are lacking such capabilities. In this paper, we define Container Level Metrics (CLM) that helps to evaluate the performance of RT containers.

C. Container Level Metrics

Several metrics are used to evaluate the timeliness [11] of tasks running in the system, which we adapt to assess the RT performance of containers. The following properties characterize the RT performance of a task:

- *Number of deadline misses*: represents the number of times that a deadline of a task was exceeded.
- *Lateness*: represents the delay of a task with respect to its deadline [11]. If the task finishes before its deadline, the lateness is negative.
- *Response time*: represents the difference between the finishing time and the start time of a task. [11]

We define CLM to evaluate the RT performance of tasks running in the respective container. For each of the tasks in a container π_k, the CLM are:

- *Number of deadline misses*: characterizes the total number of deadline misses of tasks inside of the container π_k between time t_0 and t_1:

$$\mathcal{M}_k(t_0, t_1) = \sum_{\tau_i \in \mathcal{T}_k} \mu_i(t_0, t_1).$$

- *Maximum lateness*: characterizes the maximum lateness encountered by a task inside the container between time t_0 and t_1:

$$\mathcal{L}_k^{\max}(t_0,t_1) = \max_{\tau_i \in \mathcal{T}_k}\{L_i(t_0,t_1)\}$$

- *Maximum response time*: characterizes the maximum response time encountered by a task inside the container between time t_0 and t:

$$\mathcal{R}_k^{\max}(t_0,t) = \max_{\tau_i \in \mathcal{T}_k}\{R_i(t_0,t_1)\}.$$

We express CLM within the observation interval $[t_0, t_1)$ (usually from system start at $t_0 = 0$ until the current time) as follows:

$$CLM_k(t_0,t_1) = [\mathcal{M}_k(t_0,t_1), \mathcal{L}_k^{\max}(t_0,t_1), \mathcal{R}_k^{\max}(t_0,t_1)]$$

IV. Design of the RT Orchestrator

The orchestration system is based on the master-minion architecture that consists of a master node and a set of minion compute nodes connected in a cluster as described in [10]. The core of the system is the master node that makes global decisions about the cluster; it receives users' requests for container deployments, continuously monitors states of compute nodes in the cluster and schedules containers on computing nodes. The master node's functionality can be distributed across several physical machines to avoid a single point of failure [10].

The compute nodes, depicted in Fig. 1, provide an environment for hosting containers and run a node agent that communicates with the master node through defined APIs. The node agent takes container deployment specifications defining container requirements and deployment parameters.

A. RT Extension of the Master Node

The master node depicted in Fig. 2 is a central point in the architecture; it accepts user-defined container deployment specification enhanced with RT interface and task annotations. It provides mechanisms for admission control and scheduling of containers. Additionally, it continuously collects performance metrics of compute nodes and containers. In the following text, we elaborate on the proposed enhancements:

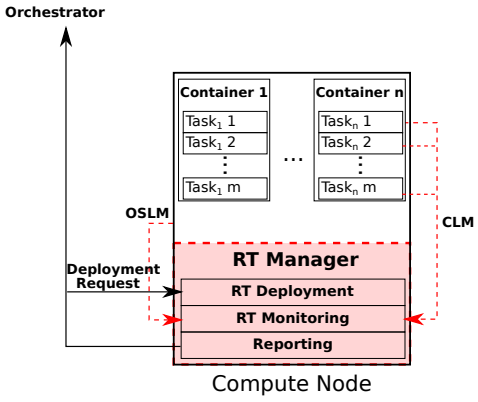

Fig. 1: Compute node.

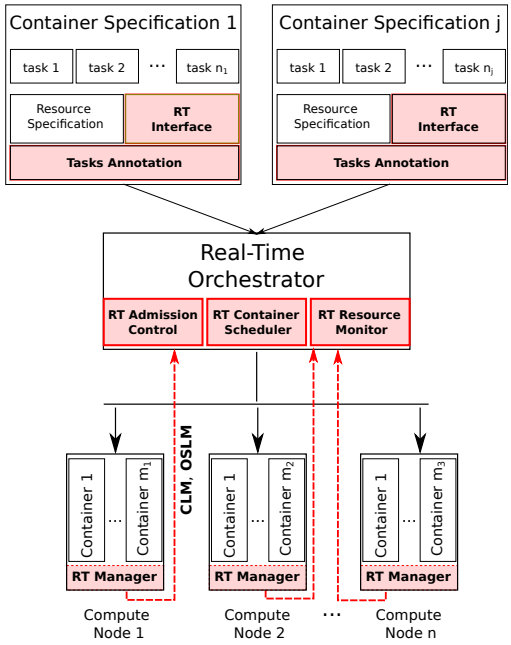

Fig. 2: A high-level container orchestrator architecture enhanced with RT capabilities.

Container Deployment Specification: Each container deployment specification, which is supplied to the master node via a dedicated API, contains the specification of the RT interface and the container annotation. The RT interface specifies the CPU reservation (P_k, Q_k) of the respective container following the periodic resource model of the hierarchical scheduling framework (c.f. [32]). The container specification contains the description of the tasks inside the container to compute the CLM during run-time.

RT Resource Monitor: RT Resource Monitor stores the OSLM and CLM from individual compute nodes. The data are accessible to RT Admission Controller and RT Container Scheduler to support the scheduling decisions.

RT Admission Control: The admission control determines if there are nodes in the cluster with enough available resources to accommodate the resource demands of the new container. Moreover, it performs necessary utilization-bound schedulability tests that reject those nodes on which the RT timing requirements cannot be met.

We perform the checks as defined below to decide if a new container, denoted by π_{new}, can be allocated to a certain node. 1) We check if the available resources (memory, storage), when considering both RT and BE containers, are enough to fit the resource demands of the new container:

$$\forall f_i \in \mathcal{S} : \begin{cases} \pi_{new}^M + \sum\limits_{\pi_k \in \Pi_i^{rt} \cup \Pi_i^{be}} \pi_k^M \leq f_i^M \\ \pi_{new}^S + \sum\limits_{\pi_k \in \Pi_i^{rt} \cup \Pi_i^{be}} \pi_k^S \leq f_i^S \end{cases} \quad (1)$$

2) We check that the new container will not make the existing RT containers unschedulable by performing a necessary

utilization-based test [11]:

$$\forall f_i \in \mathcal{S} : \frac{Q_{new}}{P_{new}} + \sum_{\pi_k \in \Pi_i^{rt}} \frac{Q_k}{P_k} \leq 1 - \delta_i \quad (2)$$

where, δ_i refers to the system overhead of node f_i (discussed in detail below), which reduces the amount of CPU bandwidth available to the containers/tasks. We remind the reader that (P_k, Q_k) is the RT interface of a RT container π_k, which defines that the container will be scheduled for at most Q_k time units over a period of P_k time units. Hence, this utilization-based test (c.f. [14]) defines that if the utilization of the RT containers (including the utilization of the new container) exceeds the bandwidth of the CPU, then the RT requirements of the new container cannot be guaranteed. Please note that the check is not sufficient, i.e., if the check is passed, it does not mean that the RT behavior will always be guaranteed since the actual execution of the RT tasks diverges from the ideal RT schedule due to e.g., timer resolution, scheduling jitter, locking mechanisms (c.f. [30]) or the overhead introduced by the container mechanism.

RT Container Scheduler: The Container Scheduler decides which of the feasible nodes in the cluster, the feasibility being determined through the admission control tests defined above, to assign the new container to. The decision is based on the CLM and OSLM introduced above, which are constantly monitored at run-time. Additionally, the scheduler might decide to change some properties of already running containers (e.g., the RT interface) to be able to fit the new container on a node. Hence, the problem of where to place new containers according to their RT requirements or which containers to modify can be viewed as a multi-objective optimization problem. While the exact mechanism on how to assign (and re-dimension) containers is out of the scope of this paper, we give a brief discussion on the important aspects of this orchestration problem and leave the definition and solution of this optimization problem for future work. Furthermore, selecting the best mix of metrics and optimization objectives to use is also not in the scope of this paper. However, we will briefly discuss several strategies below.

In terms of the observed OSLM properties, we identify several important aspects and strategies which we briefly discuss below.

The *system overhead* (δ) reduces the amount of CPU bandwidth that the containers can use. We show in the experiment section (Sect. VI) that the overhead when co-locating RT and BE containers influences only the BE containers while the RT containers are guaranteed their allotted budget (c.f. Fig. 4). We see that the overhead remains constant and jitters around the constant value both when changing the RT container period and when increasing the number of RT containers. Please note that the overhead analysis needs to be extended in future work to create a more accurate overhead model that produces a safe upper bound for admission control. However, even though the overhead model is deduced empirically and not analytically, we can still use it in the admission control since any overhead impact on RT containers will be corrected by the online reconfiguration algorithm.

Another objective may be to perform load balancing on nodes to not over-utilize some of the nodes. This decreases the probability of deadline misses and lateness for tasks and increases the probability of fitting future container requests.

The optimization function may also select nodes with a low number of context switches as this also influences the system utilization, leaving more CPU bandwidth available for container execution. Furthermore, nodes with a high number of interrupts/sec are more likely to lead to deadline misses due to the irregular nature of interrupt arrivals. Hence, they might not be the best selection for placing RT containers.

In terms of CLM properties, nodes with few (or zero) deadline misses are better candidates for new RT containers. However, adding additional RT containers might increase the number of deadline misses or the lateness/response times of tasks. The container scheduler needs to consider if an increased rate of timing failures is acceptable, depending on the nature of the running RT containers.

When using EDF to schedule containers, the admission test above also becomes necessary, i.e., if the test is passed, the new container can, in theory, be scheduled on the respective node. However, the divergence from the real schedule (described in [30]) can lead to deadline misses and/or negative effects on the lateness and response times of tasks. Hence, the run-time monitoring of the CLM properties can give hints about which containers to re-assign or re-dimension. The observation interval in the CLM properties can be dynamically adjusted to identify which of the new containers has a negative impact on the RT properties of the already running containers.

Feedback-based resource reservation mechanisms can be used to adapt the CLM properties at run-time while keeping hard RT guarantees for all the RT containers [33]. Such mechanisms can be implemented at the orchestration level to compensate for potential unforeseen over-or under-runs, providing a limited impact on the other RT containers. The overhead for feedback-based resource reservation is minimal since it requires implementing an integral control strategy for each container. Such approaches have proven successful also in other domains, such as mixed-critical systems [34].

B. RT Extension of Compute Nodes

We enable RT containers on node-level through the use of the *preempt_rt* patch, hierarchical scheduling of containers [18], hard RT co-kernel, or a combination of these technologies. The overview of the compute node extension is depicted in Fig. 1. A module responsible for the RT related functionality is denoted as *RT manager*. The process of deployment of RT containers is as follows: Upon receiving a deployment request from the orchestrator, the RT manager locate a requested image (either locally or in a remote repository), the container image is fetched, and instantiated with the requested parameters that set-up resources for the container. Concerning the RT functionality, the *RT Manager* offers the following:

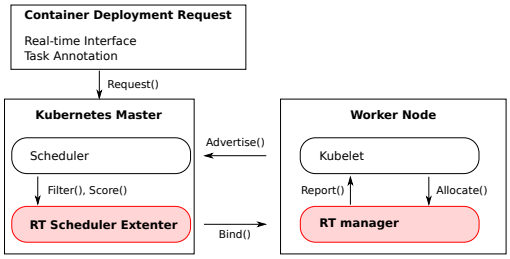

Fig. 3: Scheduling process of Kubernetes.

- Deployment of RT containers: The containers have to be instantiated in RT mode and must be assigned the requested quota and period.
- Monitoring of RT performance: The monitoring functionality assesses the wellness and the performance of the compute node and the containers deployed. There are two monitoring parts: the OSLM monitor and the CLM monitor. The OSLM monitor collects data through the *proc* and *cgroup* filesystem, which contain information related to interrupts, memory usage, CPU utilization. The CLM monitor evaluates the timelines of the containerized tasks in the RT containers.
- Reporting of the performance metrics to the orchestrator: The compute nodes report the OSLM and CLM to the master node, that uses the collected data for the admission control and container scheduling and, additionally, it can take decisions whether to re-allocate and/or re-assign resources of the containers.

V. Implementation

In order to show the feasibility of the proposed system, we have extended the existing open-source orchestrator Kubernetes with the ability to schedule containers onto compute nodes while taking into account their RT requirements. The RT behavior of containers on compute nodes is enabled by using the hierarchical patch[1] presented in [18] which we have extended with the monitoring capabilities. The system allows users to define RT requirements of the containers which need to be deployed (i.e., quota and period) and ensures that at the container scheduling level, the allocation to compute nodes respects the given requirements and does not lead to an overload in the respective node. Additionally, as the containers do not provide strong resource isolation, the system provides run-time monitoring of the container's performance. The implemented extension consists of the following components:

- *The RT Scheduler Extender* on the Master node is an extension of the Kubernetes control plane, which provides admission control and scheduling of RT containers onto compute nodes in the cluster.
- *The RT Manager* on compute nodes provides functionality to deploy RT containers and periodically evaluate and report the RT performance to the Master node.

The architecture of the Kubernetes extension is described in Fig. 3. The Kubernetes Master receives a deployment request to deploy a set of containers (denoted as a Pod in the context of Kubernetes). A new Pod is placed in a queue with other unscheduled pods, the Kubernetes scheduler (*kube-scheduler*) periodically checks the queue. If there is an unassigned pod, the Kubernetes scheduler attempts to place the Pod in a suitable node. First, the scheduler filters out infeasible nodes with insufficient available resources (e.g., insufficient memory or storage) as described in Section IV-A. Subsequently, so-called custom webhooks are triggered during each schedule polling cycle. The webhooks permit to attach custom actions to scheduling events (e.g., filtering and prioritizing events). We have implemented a custom *rt-filter* webhook that takes into account the utilization of the node as well as the RT interface as requested in the container deployment specification. If the utilization, including the new RT container, is above a certain level, the node filtered out as infeasible and not used for hosting the container. In this way, we avoid overloading the host.

The *RT Scheduler Extender* performs a secondary filtering as described in Section IV and ranks the nodes with the custom *rt-scoring* webhook. The *rt-scoring* ranks the nodes according to their suitability to host the RT container. For this work, the scoring is computed as the remaining unreserved CPU capacity. However, the rt-scoring step can incorporate additional properties of the compute node given in the OSLM and CLM to minimize the number of, e.g., deadline misses on compute nodes. As the orchestration is not part of the current paper, we leave this extension to future work.

The Pod is then assigned to the node with the highest score. The node agent on the compute node identifies that the Pod is assigned to its node and deploys it with the given RT parameters. The *RT Manager* continuously monitors the node's state beyond the understanding of the default Kubernetes monitoring metrics and reports them back to the master node. The RT manager monitors the previously described CLM and OSLM: i.e., the number of context switches, interrupts per second, I/O access, as well as task deadline misses, response time, and lateness.

As an input, the Kubernetes master accepts a Pod deployment request that contains the deployment specification of a group of one or more containers. If there are multiple containers in a pod, these are assigned and scheduled on the same node and run in a shared context (i.e., sharing memory and network). As a simplification, we assume that each Pod contains at most one RT container. We amend the deployment configuration (stored in the annotation part of the deployment file) to contain the RT interface (Q_k, P_k) as well as the list of tasks and their $\langle C_i, T_i, D_i \rangle$ parameters.

The *RT manager* runs as a Kubernetes daemon set and provides functionality for run-time monitoring and reporting of the performance of the RT containers and the system. A daemon runs on every compute node in the cluster. The monitoring part continuously monitors OSLM and CLM. The OSLM are computed by utilizing the Linux special *procfs* file system (*/proc* and */cgroups*) that contain information about tasks, similar to [31]. The CLM are derived from the custom tracepoints that we injected into the schedulers implementing

[1] Available at https://github.com/lucabe72/LinuxPatches/tree/HCBS

the *SCHED_FIFO* and *SCHED_DEADLINE* policies in the Linux Kernel. The following events are recorded and periodically evaluated by the daemon:

- Container Started: The container is in a running state, the quota/period has been allocated, and the container's tasks are ready to run.
- Container Throttled: The container used more CPU quota than the allocated one and therefore, the scheduler throttled the container.
- Task Instance started: The start of j^{th} instance of task τ_i.
- Task Instance finished: The end of j^{th} instance of task τ_i.

From these tracepoints, we compute deadline misses, lateness, and response times of tasks within the RT containers.

VI. Evaluation

In this section, we show the system's behavior for co-located RT and BE containers on a single compute node. The set of experiments illustrates the distribution of the CPU time amongst co-located containers. Tables I, II and III show the feasibility of having a mixture of RT and BE containers on a single compute node. We change the reservation period and budget from values 0.1ms to 1000ms and measure the overhead (described below). We show the behavior of low utilization containers (10%) and high utilization (90%). We investigate if RT containers with very short periods introduce significantly more overhead than those with large periods. In the experiments, we instantiate an RT container and a heavy load BE container. The experiments for Table I, II use containers that run CPU intensive operations. We measure the actual time that the containers spend on the CPU. The rest of the CPU that is not used by any container is considered as an overhead. Table III shows a similar experiment where the BE container runs *stress-ng* to generate an excessive workload aiming to affect the assigned CPU time of the RT containers. The stress generating BE containers execute 10 CPU intensive threads, 10 HDD intensive threads, 10 threads generating I/O stress, and 10 threads generating context switches.

We use an Intel i5 machine with 8GB RAM running Debian Linux, Kernel 5.2. patched with Hierarchical Scheduling Patch [18], and Docker v20.10.

We consider the overhead to be the part of the CPU capacity that is not used for any computation of the containerized tasks. It is caused by system-related tasks, context switches or docker-related processes, etc. In the experiments, we execute RT and BE containers simultaneously. Each of the containers executing a loop with a CPU-heavy computation. In theory, the containerized processes should fully utilize the processor; however, the full CPU capacity is not used exclusively for these processes. We can see that the overhead stays the same even when using an RT container with a very short period.

To investigate the overhead, we utilize *systemTap*, which is an instrumentation framework for Linux-based kernels. SystemTap allows instrumenting Linux events with user-defined code in form of loadable kernel modules. We monitor the following events in the Linux kernel:

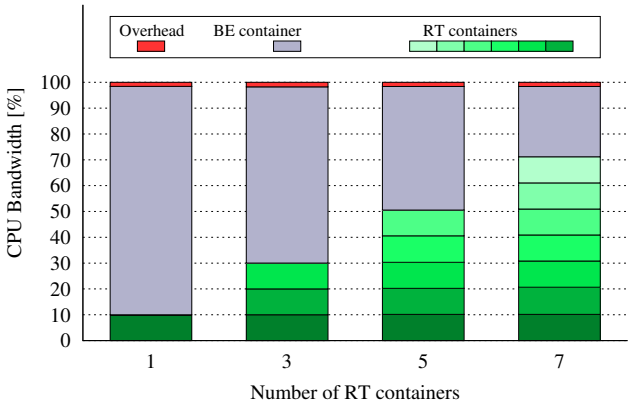

Fig. 4: Distribution of CPU in a multi-container environment.

- *scheduler.cpu_on*: The process is beginning execution on a CPU.
- *scheduler.cpu_off*: The process leaving the CPU.

From the recorded events, we are getting the total measurement time (from the first event to the last one) and each containerized task's total time.

The utilization test in Fig. 4 shows the distribution of CPU time on a single core amongst multiple RT and BE containers when increasing the number of RT containers from 1 up to 7. Each RT container has a RT demand of 10% of the CPU bandwidth (RT period = 100ms, RT budget = 10ms). The experiments indicate that hierarchically scheduled containers using the Hierarchical Scheduling Patch [18] can keep the allocated CPU resource even when competing with BE containers under heavy load. Moreover, the RT containers keep their reserved resource allocation (CPU budget over the periods) with very low run-time jitter on a single core. The system overhead does not influence the RT containers but reduces the remaining CPU utilization used by BE containers.

The experiments indicate that RT containers maintain the target resource reservation even in the presence of heavy RT and BE load. The overhead (indicated in red in Fig. 4) remains relatively constant when increasing the number of containers scheduled on the same node.

VII. Conclusion

In this paper, we have introduced a container orchestration for RT systems designed to prevent over-reservation of the CPU at the container scheduling level. Additionally, we considered the weak isolation inherent in container-based virtualization (e.g., the effects of the use of shared resources or context switches), which can lead to an interference and thereby harm temporal guarantees of RT containers. We have proposed metrics for measuring the RT performance at the container and node levels, which can be used for both the admission control and the online re-configuration of container deployment in order to guarantee timely behavior. We have implemented a scheduler extension and node monitoring system on top of an orchestrating system Kubernetes and have shown the feasibility of co-locating RT and BE containers on the same node in a series of experiments.

TABLE I: RT containers (10% utilization) on a single core with noise generated by 10 CPU intensive containers.

	(1ms, 0.1ms)	**(10ms, 1ms)**	**(100ms, 10ms)**	**(1000ms, 100ms)**
RT Containers	10.0607% ± 0.00193%	10.0021% ± 0.00002%	9.9983% ± 0.00002%	9.9978% ± 0.00002%
NRT Containers	88.7870% ± 0.00406%	88.8963% ± 0.00280%	88.8713% ± 0.00222%	88.8906% ± 0.00264%
overhead	1.1522% ± 0.0035%	1.1016% ± 0.0028%	1.1304% ± 0.0022%	1.1115% ± 0.0025%

TABLE II: RT containers (90% utilization) on a single core with noise generated by 10 CPU intensive containers.

	(1ms, 0.9ms)	**(10ms, 9ms)**	**(100ms, 90ms)**	**(1000ms, 900ms)**
RT Containers	89.9037% ± 0.00031	89.8597% ± 0.00042%	89.8818% ± 0.00066%	89.7780% ± 0.00316%
NRT Containers	9.0345% ± 0.00212	9.0368% ± 0.00193%	9.0349% ± 0.00045%	9.0658% ± 0.00164%
overhead	1.0618% ± 0.0022	1.1035% ± 0.0019%	1.0833% ± 0.0005%	1.1563% ± 0.0018%

TABLE III: RT containers (10% utilization) on one core with noise generated by BE containers executing *stress-ng*.

	(1ms, 0.1ms)	**(10ms, 1ms)**	**(100ms, 10ms)**	**(1000ms, 100ms)**
RT Containers	10.0508% ± 0.00045%	10.0065% ± 0.00005%	9.9956% ± 0.00006%	9.9956% ± 0.00002%
NRT Containers	89.0251% ± 0.00091%	88.9333% ± 0.00034%	88.8779% ± 0.00296%	88.9798% ± 0.00142%
overhead	0.9241% ±0.0037%	1.0602% ±0.0031%	1.1265% ±0.0024%	1.0246% ± 0.0029%

We aim to address the optimization problem arising from the orchestration needs in RT containerized systems in future work. This orchestration includes both the admission/allocation of new container requests as well as the online resource re-dimensioning of already deployed RT containers in case of run-time performance drops.

REFERENCES

[1] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *W. on Mob. cloud comp.*, 2012.

[2] S. M. Salman *et al.*, "Fogification of industrial robotic systems: Research challenges," in *W. on Fog Comp. and IoT (Fog-IoT)*, 2019.

[3] M. S. Shaik *et al.*, "Enabling fog-based industrial robotics systems," in *2020 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA)*, vol. 1, 2020, pp. 61–68.

[4] F. Ramalho and A. Neto, "Virtualization at the network edge: A performance comparison," in *IEEE Int. Symp. A World of Wirel., Mob. and Multim. Net. (WoWMoM)*, 2016.

[5] W. Felter, A. Ferreira, R. Rajamony, and J. Rubio, "An updated performance comparison of virtual machines and Linux containers," in *IEEE Int. Symp. on Perf. Analysis of Syst. and Soft. (ISPASS)*, 2015.

[6] M. G. Xavier, M. V. Neves, and C. A. F. De Rose, "A performance comparison of container-based virtualization systems for mapreduce clusters," in *Euromicro Int. Conf. on Par., Distr., and Netw. Proc. (PDP)*, 2014.

[7] T. Cucinotta *et al.*, "Virtual network functions as real-time containers in private clouds," in *IEEE Int. Conf. on Cloud Comp. (CLOUD)*, 2018.

[8] ——, "Reducing temporal interference in private clouds through real-time containers," in *IEEE Int. Conf. on Edge Comp. (EDGE)*, 2019.

[9] V. Struhár, M. Behnam, M. Ashjaei, and A. V. Papadopoulos, "Real-time containers: A survey," in *W. on Fog Comp. and IoT (Fog-IoT)*, 2020.

[10] M. A. Rodriguez and R. Buyya, "Container-based cluster orchestration systems: A taxonomy and future directions," *Software: Practice and Experience*, 2019.

[11] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Springer, 2011.

[12] C. Zhao *et al.*, "Quantifying the isolation characteristics in container environments," in *Net. and Par. Comp. (NPC)*, 2017.

[13] M. G. Xavier *et al.*, "A performance isolation analysis of disk-intensive workloads on container-based clouds," in *Euromicro Int. Conf. on Par., Distr., and Netw. Proc. (PDP)*, 2015.

[14] C. L. Liu and J. W. Layland, "Scheduling algorithms for multiprogramming in a hard-real-time environment," *J. ACM*, 1973.

[15] S. Xi, J. Wilson, C. Lu, and C. Gill, "RT-Xen: towards real-time hypervisor scheduling in Xen," in *ACM Int. Conf. on Emb. Soft. (EMSOFT)*, 2011.

[16] S. He *et al.*, "Elastic application container: A lightweight approach for cloud resource provisioning," in *IEEE Int. Conf. on Adv. Inform. Netw. and Appl. (AINA)*, 2012.

[17] T. L. Nguyen and A. Lebre, "Conducting thousands of experiments to analyze vms, dockers and nested dockers boot time," INRIA, Research Report RR-9221, 2018.

[18] L. Abeni, A. Balsini, and T. Cucinotta, "Container-based real-time scheduling in the Linux kernel," *ACM SIGBED Review*, 11 2019.

[19] D. Beyer, S. Löwe, and P. Wendler, "Benchmarking and resource measurement," in *Model Checking Software*, 2015.

[20] P. Gerum, "Xenomai - implementing a RTOS emulation framework on GNU/Linux," *White Paper, Xenomai*, 2004.

[21] T. Tasci, J. Melcher, and A. Verl, "A container-based architecture for real-time control applications," in *IEEE Int. Conf. on Eng., Tech. and Innov. (ICE/ITMC)*, 2018.

[22] F. Hofer *et al.*, "Industrial control via application containers: Migrating from bare-metal to IAAS," in *IEEE Int. Conf. on Cloud Computing Technology and Science (CloudCom)*, 2019.

[23] A. Moga, T. Sivanthi, and C. Franke, "OS-level virtualization for industrial automation systems: are we there yet?" in *ACM Symp. on Applied Computing (SAC)*, 2016.

[24] T. Goldschmidt, S. Hauck-Stattelmann, S. Malakuti, and S. Grüner, "Container-based architecture for flexible industrial control applications," *J. of Syst. Architecture*, 2018.

[25] T. Goldschmidt and S. Hauck-Stattelmann, "Software containers for industrial control," in *Euromicro Conf. on Soft. Eng. and Adv. Appl. (SEAA)*, 2016.

[26] A. Easwaran, I. Shin, and I. Lee, "Optimal virtual cluster-based multiprocessor scheduling," *Real-Time Syst.*, 2009.

[27] S. Xi *et al.*, "Rt-open stack: Cpu resource management for real-time cloud computing," in *2015 IEEE 8th International Conference on Cloud Computing*, 2015.

[28] I. Shin and I. Lee, "Periodic resource model for compositional real-time guarantees," in *IEEE Real-Time Syst. Symp. (RTSS)*, 2003.

[29] M. Jägemar, A. Ermedahl, S. Eldh, and M. Behnam, "A scheduling architecture for enforcing quality of service in multi-process systems," in *IEEE Int. Conf. on Emerging Tech. and Factory Aut. (ETFA)*, 2017.

[30] L. Abeni *et al.*, "A measurement-based analysis of the real-time performance of Linux," in *IEEE Real-Time and Emb. Tech. and Appl. Symp. (RTAS)*, 2002.

[31] E. Casalicchio and V. Perciballi, "Measuring docker performance: What a mess!!!" in *ACM/SPEC on Int. Conf. on Perf. Eng. (ICPE)*, 2017.

[32] I. Shin, A. Easwaran, and I. Lee, "Hierarchical scheduling framework for virtual clustering of multiprocessors," in *Euromicro Conf. on Real-Time Syst. (ECRTS)*, 2008.

[33] A. V. Papadopoulos, M. Maggio, A. Leva, and E. Bini, "Hard real-time guarantees in feedback-based resource reservations," *Real-Time Syst.*, 2015.

[34] A. V. Papadopoulos, E. Bini, S. Baruah, and A. Burns, "AdaptMC: A control-theoretic approach for achieving resilience in mixed-criticality systems," in *Euromicro Conf. on Real-Time Syst. (ECRTS)*, 2018.