Hierarchical Resource Orchestration Framework for Real-time Containers

VÁCLAV STRUHÁR, Mälardalen University, Sweden
SILVIU S. CRACIUNAS, TTTech Computertechnik AG, Austria
MOHAMMAD ASHJAEI, MORIS BEHNAM, and ALESSANDRO V. PAPADOPOULOS, Mälardalen University, Sweden

Container-based virtualization is a promising deployment model in fog and edge computing applications, because it allows a seamless co-existence of virtualized applications in a heterogeneous environment without introducing significant overhead. Certain application domains (e.g., industrial automation, automotive, or aerospace) mandate that applications exhibit a certain degree of temporal predictability. Container-based virtualization cannot be easily used for such applications, since the technology is not designed to support real-time properties and handle temporal disturbances. This article proposes a framework consisting of a static offline and a dynamic online phase for resource allocation and adaptive re-dimensioning of real-time containers. In the offline phase, the optimal initial deployment and dimensioning of containers are decided based on ideal system models. Additionally, to adapt to dynamic variations caused by changing workloads or interferences, the online phase adapts the CPU usage and limits of real-time containers at runtime to improve the real-time behavior of the real-time containerized applications while optimizing resource usage. We implement the framework in a real Linux-based system and show through a series of experiments that the proposed framework is able to adjust and re-distribute computing resources between containers to improve the real-time behavior of containerized applications in the presence of temporal disturbances while optimizing resource usage.

CCS Concepts: • **Computer systems organization → Real-time system architecture;**

Additional Key Words and Phrases: Real-time container-based virtualization, real-time, real-time docker

ACM Reference format:
Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V. Papadopoulos. 2024. Hierarchical Resource Orchestration Framework for Real-time Containers. *ACM Trans. Embedd. Comput. Syst.* 23, 1, Article 4 (January 2024), 24 pages.
https://doi.org/10.1145/3592856

This work was supported by the Swedish Research Council (VR) with the project "Pervasive Self-Optimizing Computing Infrastructures (PSI)" (No. 2020-05094), by the Knowledge Foundation (KKS) with the project "Safe and Secure Adaptive Collaborative Systems (SACSys)" (No. 20190021) and by the projects "Federated Choreography of an Integrated Embedded Systems Software Architecture (FIESTA)" and "Excellence in Production Research (XPRES)".

Authors' addresses: V. Struhár, M. Ashjaei, M. Behnam, and A. V. Papadopoulos, Mälardalen University, Västerås, Sweden; emails: {vaclav.struhar, mohammad.ashjaei, moris.behnam, alessandro.papadopoulos}@mdu.se; S. S. Craciunas, TTTech Computertechnik AG, Vienna, Austria; email: silviu.craciunas@tttech.com.

1539-9087/2024/01-ART4 $15.00
https://doi.org/10.1145/3592856

1 INTRODUCTION

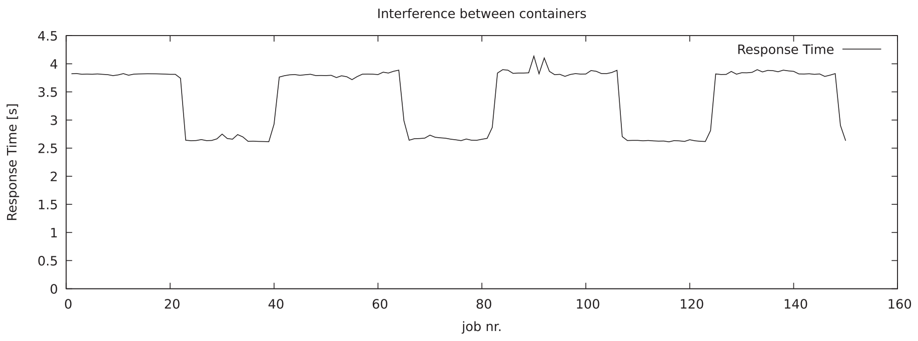
Container-based virtualization and container orchestration became *de facto* standards to package and deploy applications in cloud environments [45]. Virtualization provides functional isolation enabling the co-location of applications on a shared physical machine, while orchestration provides management of containerized applications, automation of deployment, and maintaining scalability in a cluster of physical machines. Such technologies are paving the way for recent trends such as fog and edge computing, where an application is spanned across multiple computing layers. In fog computing, the edge layers host low latency functions, while the cloud layers host computationally demanding functions. Currently, industrial domains, e.g., robot control, automotive, aviation, and mixed-criticality-systems in general [16] seek to adopt container-based virtualization and orchestration in their ecosystems to fulfill Industry 4.0 needs [25, 28, 37, 39]. That is the flexibility of the manufacturing process, decentralization of functions, increasing automation, and resource efficiency.

However, industrial domains have elevated requirements for applications, especially in real-time areas [16, 42]. Industrial applications require both functional correctness and timeliness to produce computation results [3, 39]. In the presence of timing violations, the usefulness of computed results decreases (i.e., soft real-time), or it may even result in catastrophic consequences (i.e., hard real-time). Unfortunately, real-time behavior is challenging to achieve in systems using container-based virtualization and orchestration, since timing predictability is not a major concern of these technologies. As noted in Reference [24], remotely hosted virtual environments suffer from variances in processing times that are changing with the executed workloads [2, 24].

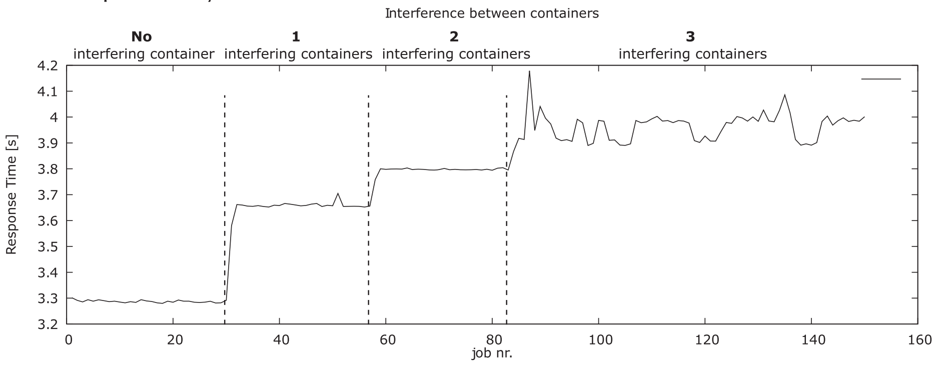
Due to container-based virtualization's limited performance isolation, co-located applications may affect each other's performance in unpredictable ways [32]. For instance, co-located containers may share the **Last Level Cache (LLC)**, i.e., a cache that is accessed by the cores prior to fetching from memory. Intensive simultaneous use of the cache by multiple containers may increase the response time of the containerized applications, and time predictability (i.e., real-time behavior) may be negatively impacted. The effect of performance interference is depicted in the experiment (Figure 1), which shows a performance loss due to the simultaneous use of a shared LLC cache. Figure 1(a) shows how the response time of a container is impacted by a co-located container running on another core. The co-located container periodically changes its workload between a cache-intensive and a non-cache-intensive one. As a result, the response times increase during the cache-intensive periods. Similarly, Figure 1(b) shows the change in response time when an increasing number of containers are co-located with the measured containerized application.

The problem of timing interference between containers is further complicated by the fact that it varies widely across platforms. This fact is crucial in fog and edge computing, where hardware-heterogeneous physical machines are present, and for which the effect of interference may not be known beforehand. An additional factor that hinders the real-time behavior of container-based computing (and computing in general) is the unpredictable changing workload. Containers may be performing updates or serving increased user-generated workloads that may influence the performance of other co-located containers, which may also be unknown in advance.

Taking into account the aforementioned factors influencing the real-time behavior of containers, this article proposes a joint hierarchical container virtualization and orchestration framework that aims to improve the real-time behavior of real-time containers in a multi-container environment. The proposed framework consists of two phases, an offline phase, and an online phase. The offline phase is in charge of solving the deployment problem, i.e., it decides where to place containers and how to dimension their real-time interfaces on the basis of design-time models. This phase is related to the server design problem in hierarchical scheduling [29, 41]. However, the online phase attempts to bridge the gap between the assumptions that are needed for the deployment,



(a) The response time of a co-located container may vary over time due to changes in the workload of the interfering container. In this example the co-located container changes its workload periodically.



(b) An example of changing response times of a container caused by the interfering containers. We successively deploy 1 to 3 interfering containers on different CPU cores. The response time is negatively affected by the amount of co-located containers.

Fig. 1. Two examples of interference between containers utilizing LLC caches. The examples show extended response times of a containerized application caused by the extensive LLC cache usage by co-located containers.

in which idealized models are used, and the actual dynamic behavior of the containers that may negatively impact the runtime performance of real-time containers, as described above. The online phase hence adjusts and distributes the CPU resources among real-time and non-real-time **best-effort (BE)** containers based on a continuous runtime evaluation of the real-time performance of containers. In this work, we emphasize the problem of resource interference and performance loss due to changing workload of co-located containers, a problem that is especially significant in heterogeneous platforms (e.g., fog computing). We focus on the computational part of mitigating the resource interference (i.e., adjusting computational resources) and keep other parts (e.g., location awareness, networking) inherent to fog computing for future work.

In this article, we define and solve both the offline and the online phases of the real-time container orchestration problem. For the offline phase, we use well-known approaches for the server design problem and extend them to the problem at hand, formulating the allocation and dimensioning of containers as an optimization problem that can be solved via, e.g., **integer linear programming (ILP)** and **satisfiability modulo theories (SMT)** solvers or using well-known heuristics if the problem size exceeds the scalability of optimal solutions. In the online phase, the Hierarchical Framework dynamically compensates for the real-time performance variability in a multi-node

container-based environment using a three-level control hierarchy. The first- and second-level controllers compensate for the performance losses in a computing node by continuously measuring and re-adjusting the CPU bandwidth reservation for containers. The third-level controller is in charge of the migration of containers amongst computing nodes in case the required real-time performance cannot be achieved within the scope of the individual node. Finally, we describe the implementation in Linux of our Hierarchical Framework and a series of experiments that shows the viability of our solution.
Contributions of this article are as follows:

- We propose a framework that combines offline placement and online redimensioning of resources for real-time containers. In the offline phase, we formulate the initial deployment of containers as the optimization problem. In the online phase, we enable the self-adaptation of resources for real-time containers based on their timing requirements.
- We present an architectural design and a detailed explanation of the three-level hierarchy of controllers utilized for adapting resources and migrating real-time containers.
- We implement the online adaptation of real-time containers in Linux and provide an evaluation that shows the feasibility of the solution. We utilize and enhance the **Hierarchical Constant Bandwidth Server (HCBS)** patch [1] with the ability to adjust resources at runtime.

The article is organized as follows: Section 2 presents the background and surveys the related work, followed by the system architecture and system model in Section 3. We present the details of our orchestration framework, including the offline and online adaptation phases in Section 4. Section 5 shows experimental results, and finally, Section 7 concludes this work.

2 BACKGROUND AND RELATED WORK

Providing real-time performance for container-based computation is outgoing research, the main research directions are summarized in Reference [43]. The major research directions focus on the mitigation of scheduling latencies and scheduling jitters for containers by using PREEMPT_RT patch [26, 31], applying real-time co-kernels for container-based virtualization [7, 8, 15, 17], and employing the hierarchical scheduling framework for real-time containers to enable temporal isolation of containerized applications [1]. However, none of the studies addresses performance interference between containers nor mitigating performance interference.

Abeni et al. [1] present the Linux Kernel enhanced with hierarchical scheduling that utilizes the Hierarchical Scheduling Framework. The authors hierarchically connect two existing Linux scheduling policies (SCHED_DEADLINE and SCHED_RT). The former policy implements the **Earliest Deadline First (EDF)** algorithm combined with **Constant Bandwidth Server (CBS)** [4], serving as a selector of the container with the earliest deadline. The latter local policy schedules the highest-priority runnable task from the particular container. Thus, a user can reserve a CPU quota over a given period. We base our resource adaptation framework on this approach, which allows us to continuously adapt container resources by changing the CPU quota and the period. Our work utilizes the proposed Kernel patch and adds a mechanism for resource reservation adjustments at runtime.

Container orchestrators are tools that manage containerized applications, automate deployment, and enable scalability in a cluster of physical machines. However, the development of real-time container orchestration is in the early stage. Several studies attempt to enhance existing orchestrators (i.e., Kubernetes,[1] OpenStack[2]) with awareness of real-time requirements. For instance, Struhar et al. [44] and Fiori et al. [24] leverage Kubernetes for the deployment of real-time containers.

[1]Available at https://kubernetes.io/.
[2]Available at https://openstack.org/.

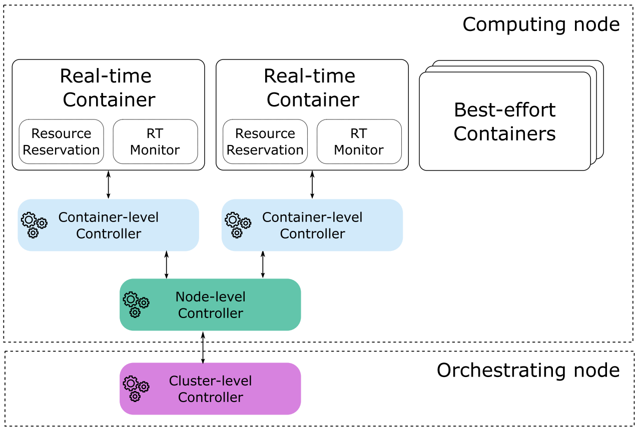


Fig. 2. Architecture of the system.

Similarly, Cucinotta et al. [21] use OpenStack, which is able to deploy real-time containers. The solutions conduct a utilization-based admission test upon a container deployment request. Barletta et al. [9] introduce an orchestration approach for mixed-criticality systems with real-time requirements. The before-mentioned papers extend the orchestrators with the ability to perform schedulability tests and place containers in suitable nodes. However, our work adds additional dimensions for the orchestration: real-time migration (when there are not enough resources) and online adjustment of resources to maintain soft real-time performance.

Containerized applications can be scaled either horizontally or vertically. Horizontal scaling (also known as replication) involves adding computational resources linked with the application [5]. Vertical scaling implies changing functional or non-functional resources such as CPU time, memory, and so on. In the study by Al-Dhuraibi et al. [5], the authors provide a system for elastic scaling of containers that adjusts memory, vCPU cores, and CPU fractions according to the workload. The resource adjustment strategy is based on static thresholds of response times. The study provides a simple strategy to vertically control resources. In our work, we focus on real-time containers containing periodic tasks. We monitor and take control actions continuously, and thus, eliminate long breath-down/break-up periods needed to reach a stable system state. Additionally, we consider the offline phase for computing the parameters of containers and computing the initial placement of the containers.

The study by Rossi et al. [36] proposes reinforcement learning techniques for adapting at runtime the deployment of container-based applications by means of horizontal and vertical elasticity. The system adapts without the need to tune the system manually. Furthermore, both vertical and horizontal scaling (that can introduce performance penalties) is explored. Reference [40] presents a data-driven, machine-learning technique based on Gaussian Processes to build a predictive runtime model of the system's performance, which can adapt itself to the variability in workload changes.

3 SYSTEM ARCHITECTURE AND SYSTEM MODEL

In this section, we present the architecture and system model of the proposed framework. The architecture of the system is depicted in Figure 2. The framework consists of the offline phase and a three-level control architecture for the online phase encompassing container-level, node-level, and cluster-level controllers. The offline phase is (a) responsible for the computation of ideal real-time interfaces (an initial value of the resource reservation) for a given set of containers and their

Table 1. System Model Notations

Model	Notation	Definition
Node	n	total number of nodes
	Π_j^{be}	Set of BE containers on node j
	Π_j^{rt}	Set of RT containers on node j
	f_j^M	Memory capacity
	f_j^S	Storage capacity
Task	$\tau_i \in \mathcal{T}$	$\tau_i = (T_i, \{C_i^1, \ldots, C_i^n\}, D_i, p_i)$
	T_i	Period
	$C_i^1, \ldots, C_i^n$	Node-dependent Worst-Case Execution Times
	D_i	Deadline
	p_i	Task priority
Container	π	$\pi = (\mathcal{T}^k, P_k, Q_k, CLM_k, \pi_k^M, \pi_k^S)$
	m	total number of containers
	Q_k	CPU quota
	P_k	Replenishment period
	CLM_k	Container-level Metrics
	π_k^M	Memory demand
	π_k^S	Storage demand

real-time requirements and (b) responsible for selecting a node for an incoming container. The three-level controller hierarchy mitigates temporal disturbances by a continuous redistribution of system resources and migration of containers in case of a shortage of resources in a computing node.

The system model is based on the React Orchestrator presented in Reference [44] that consists of the following elements: the *Task model*, the *Container model*, the computer *Node model*, the *Cluster model*, and the *Hierarchical scheduler*. The notation of the model is summarized in Table 1 and described as follows.

For the computing node model, we assume that there are n nodes in the system, where each node $f_j, j = 1, \ldots, n$ has a certain amount of memory and storage resources, denoted with f_j^M and f_j^S, respectively. We assume that each node has a single-core CPU with a given CPU frequency, which will affect the **Worst-case Execution Time (WCET)** of a task assigned to it (see below). Multi-core CPUs can be modeled as separate single-core nodes that are part of the cluster. Each computing node f_j is capable of running both **Real-time (*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. The cluster C consists of several nodes, $f_1, \ldots, f_n$, that are connected with the orchestrator.

For the task model, we assume a periodic task model [30] that describes a periodically executed stream of jobs. A task ($\tau_i \in \mathcal{T}$) consists of a stream of jobs. Each job of a task τ_i is periodically activated at the beginning of each period (T_i) and has a prescribed relative deadline D_i. The execution time of each job is specified by WCET, denoted as C_i. Additionally, each task has a relative priority p_i that is used to determine the processing sequence within the container. For heterogeneous systems with a significant difference in hardware capabilities the execution time of tasks will have a high variance depending on where the container is placed. Hence, the execution time C_i of tasks will depend on the CPU frequency of the respective node. For capturing tasks and containers that are allocated on heterogeneous systems, we introduce instead of C_i an array $C_i^1, \ldots, C_i^n$, representing the worst-case execution time of the task τ_i on every node.

For the container model, we assume that there are m containers in the system, each container π_k, $k = 1, \ldots, m$ having a memory (π_k^M) and storage (π_k^S) demand resulting from the individual memory and storage demands of the tasks running in the respective container that has to be satisfied by the node's resources, similar to Reference [44]. A real-time RT container $\pi_k \in \Pi_j^{\mathrm{rt}}$ has an RT interface expressed as (P_k, Q_k) where Q_k is a CPU quota that is required to be provided over a period P_k. We introduce the metric of a container π_k, denoted by CLM_k, to capture the performance of RT containers. Each container is running a pre-defined subset of the tasks $\mathcal{T}^k \in \mathcal{T}$.

In each node, there may be multiple sources of non-determinism that may affect the real-time behavior of containerized tasks, e.g., kernel latencies, interrupts, context switches, caches, and so on. However, we argue that it is difficult to accurately model and capture all these sources of non-determinism in a real system. Hence, we keep our node model simple and do not attempt to include different overhead and interference sources. Our online phase is tailored to address this very problem and not to mandate that all sources of non-determinism in the system are known, since these latencies will be compensated for in the online adaptation mechanism if they negatively affect the real-time behavior of the tasks within containers (see below).

For the scheduler/dispatcher model, we assume that the system uses a two-level hierarchical scheduler. The global scheduler uses the EDF algorithm to select the container with the earliest deadline (the deadline of the container is aligned with its period). The EDF scheduler is combined with the Constant Bandwidth Server with hard allocation. The local scheduler is a fixed priority scheduler that chooses the highest priority runnable task inside of the selected container. This model can be achieved with the modified Linux distribution (e.g., Abeni's HCBS implementation [1]) where a CBS scheduler (`SCHED_DEADLINE`) has a higher priority than the fixed-priority scheduler (`SCHED_RT`) in the kernel scheduling hierarchy. We assume that other tasks (i.e., BE container tasks) are scheduled by the **Completely Fair Scheduler (CFS)**, having the lowest priority.

4 HIERARCHICAL RESOURCE ORCHESTRATION FRAMEWORK

We present the Hierarchical Resource Orchestration Framework, which is a general orchestration framework for static and dynamic allocation and dimensioning of real-time containers. The framework leverages real-time containers, i.e., containers that provide both spatial and temporal isolation of containerized applications. Adding real-time properties to containers has been addressed in Reference [1] by introducing a hierarchical scheduling patch for Linux-based systems. The patch ensures that the container would not violate the CPU resource reservation of other containers (however, it does not solve the possible interference between containers due to sharing of resources other than CPU). Our framework manages the deployment and adaptation of real-time containers in distributed applications featuring heterogeneous computing nodes so that individual real-time task requirements are met. These requirements are not only related to real-time behavior but also resource usages such as memory, I/O, and storage requirements and non-functional requirements such as fault-tolerance, power consumption, or resource efficiency.

The input to the Hierarchical Resource Orchestration Framework is a set of real-time tasks and containers. The containers include either real-time tasks with constrained deadlines assigned to RT containers or non-real-time tasks, which are assigned to BE containers. Real-time tasks are additionally defined using a WCET and a period specifying an upper bound on the computation of the task in each period and the rate at which the task is activated. Both RT and BE containers can coexist on the same core, but they are spatially and temporally isolated. The real-time tasks are pre-allocated to containers, but the containers are not pre-allocated to computing nodes (and cores), although they can have a certain affinity set, constraining the set of nodes to which they can be allocated. As defined in Reference [44], each RT container π_k has additionally an RT interface consisting of (P_k, Q_k) where Q_k is the CPU quota within an interval (period) P_k, defining that the container π_k cannot use more than Q_k time units over an interval of P_k time units.

The system, proposed in this article, consists of two phases: offline and online phase. The former ensures the computation of the initial real-time interfaces, and the latter phase deals with the online adaptation of resources to the containers. Both phases are described in the following sections.

4.1 Offline Phase

For the offline phase, the tasks are pre-assigned to containers, but containers (scheduled using the CBS `SCHED_DEADLINE` scheduler class of Linux) are not assigned to nodes, although they may have a certain affinity to a subset of nodes based on, e.g., ASIL level or node capabilities. The tasks inside the containers are scheduled via a fixed-priority scheduling class (usually the `SCHED_RT` scheduler class in Linux). We assume that the task priorities are given (usually via a deadline-monotonic assignment). In line with the model from Reference [44], a *RT* container π_k has an *RT* interface expressed as (P_k, Q_k) where P_k is a period and Q_k is a CPU quota.

The main objective of the offline phase is to assign to each container an ideal *RT* interface by selecting (P_k, Q_k) such that the fixed-priority tasks are schedulable. This problem is generally referred to as the server design problem in hierarchical scheduling [29, 41]. We use the lower linear approximation for the supply bound function of the periodic resource defined in References [6, 29]. From Reference [29], we have $\forall \pi_k \in \Pi_j^{\text{rt}}$, using $\alpha_k = Q_k/P_k$ and $\Delta_k = 2 \cdot (P_k - Q_k)$, the linear supply lower bound function $lslbf(t)_k$ is defined as

$$lslbf_k(t) = \max\{0, (t - \Delta_k) \cdot \alpha_k\}. \tag{1}$$

With the fixed-priority scheduler within the container, we can use the analysis from Reference [29] to relate the (P_k, Q_k) values to the schedulability of the tasks within the respective container π_k. Theorem 3 of Reference [29] states that the task set $\mathcal{T}^k$ is schedulable under the resource abstraction (P_k, Q_k) using the linear supply lower bound function $lslbf_k(t)$ if

$$\Delta_k \leq \min_{\tau_i \in \mathcal{T}^k} \max_{t \in \mathcal{P}_{i-1}(D_i)} t - \frac{1}{\alpha_k}\left(C_i + \sum_{\substack{\tau_j \in \mathcal{T}^k \\ p_j \geq p_i}} \left\lceil \frac{t}{T_j} \right\rceil \cdot C_j\right), \tag{2}$$

where

$$\begin{cases} \mathcal{P}_0(t) = \{t\}, \\ \mathcal{P}_i(t) = \mathcal{P}_{i-1}\left(\left\lfloor \frac{t}{T_i} \right\rfloor T_i\right) \cup \mathcal{P}_{i-1}(t). \end{cases}$$

Any values $P_k = \frac{\Delta_k}{2(1-\alpha_k)}$ and $Q_k = \alpha_k P_k$, for which α_k and Δ_k conform to Equation (2) are valid. However, as described in Reference [29], there is a trade-off between the wasted bandwidth and the context switch cost, as bigger periods result in fewer context switches, but the allocated container utilization has to be kept low to avoid wasting CPU bandwidth. Therefore, a cost function is proposed in Reference [29] that leverages the two objectives via a weighted sum tuned by two design-time constants. The cost function proposed in Reference [29] contains two design-time constants, c_1 and c_2, which enable the system designer to parameterize the trade-off depending on the design goals of the system. We adapt the cost function J for a container from Reference [29] as follows:

$$J = c_1 \frac{\delta_j}{P_k} + c_2 \alpha_k, \tag{3}$$

where δ_j is the context switch overhead of the node f_j where the container is placed. As can be seen, the cost function assumes that the container has already been assigned and the context switch overhead is known, which is not the case here, since the containers have not been allocated yet.

In homogeneous systems or in heterogeneous systems where there is not a significant variance in the hardware capabilities, we can solve the server design problem immediately by choosing the lowest CPU speed as a reference. Therefore, the second stage allocation of containers to nodes becomes a bin-packing problem (with complexity NP-hard) that can be solved using exact methods or, for larger problem sizes, by heuristic approximations [18].

For heterogeneous systems with a significant difference in hardware capabilities, not only the system overhead but also the execution time of tasks will have a high variance depending on where the container is placed. Hence, both the execution time C_i of tasks, as well as the context switch overhead δ_j, become variables that depend on the allocation. Moreover, both Equation (2) and the cost function from Equation (3) have to be considered as part of the allocation problem. For the purposes of the optimization problem, we remind the reader that instead of C_i representing the single WCET in the standard model (cf. Reference [29]), we use an array $C_i^1, \ldots, C_i^n$, representing the worst-case execution time of the task τ_i on every node.

We define for each container $\pi_k, k = 1, \ldots, m$ the following variables:

- a set of Boolean variables $v_1^k, \ldots, v_n^k$ specifying whether the container is allocated to the respective node or not,
- the container period P_k and budget Q_k.

The optimization problem can therefore be expressed as

$$\underset{\{Q_k, P_k, v_1^k, \ldots, v_n^k | k=1,\ldots,m\}}{\text{minimize}} \quad c_1 \cdot \sum_{k=1}^{m} \frac{\sum_{j=1}^{n} v_j^k \cdot \delta_j}{P_k} + c_2 \cdot \sum_{k=1}^{m} \frac{Q_k}{P_k}, \tag{4}$$

subject to

$$\forall j = 1, \ldots, n : \sum_{k=1}^{m} v_j^k \cdot \pi_k^M \le f_j^M, \tag{5}$$

$$\forall j = 1, \ldots, n : \sum_{k=1}^{m} v_j^k \cdot \pi_k^S \le f_j^S, \tag{6}$$

$$\forall j = 1, \ldots, n : \sum_{k=1}^{m} \frac{v_j^k \cdot Q_k}{P_k} \le 1, \tag{7}$$

$$\forall k = 1, \ldots, m, \forall j = 1, \ldots, n : v_j^k = \{0, 1\}, \tag{8}$$

$$\forall k = 1, \ldots, m : \sum_{j=1}^{n} v_j^k = 1, \tag{9}$$

$$\forall k = 1, \ldots, m : \frac{Q_k}{P_k} \ge \sum_{\tau_i \in \mathcal{T}^k} \frac{\sum_{j=1}^{n} v_k^j \cdot C_i^j}{T_i}, \tag{10}$$

$$\forall k = 1, \ldots, m : \bigwedge_{\tau_i \in \mathcal{T}^k} \bigvee_{t \in \mathcal{P}_{i-1}(D_i)} t - 2 \cdot (P_k - Q_k) - \frac{P_k}{Q_k} \left(\sum_{j=1}^{n} v_k^j \cdot C_i^j + \sum_{\substack{\tau_l \in \mathcal{T}^k \\ p_l \ge p_i}} \sum_{j=1}^{n} \left\lceil \frac{t}{T_l} \right\rceil \cdot v_k^j \cdot C_l^j \right) \ge 0. \tag{11}$$

The first two constraints (Equations (5) and (6)) check the memory availability on each node and the constraint defined in Equation (7) enforces that each node is not overutilized. The next two conditions (Equations (8) and (9)) constrain the Boolean assignment variables to be correct, i.e., one container can only be assigned to one node. The condition in Equation (10) ensures that the bandwidth of the container is sufficient to execute all tasks within the container. Please note that we need the sum over the list of possible WCETs for each task multiplied by the binary assignment variable for the container to node assignment to select the correctly scaled WCET for each task. The last condition (Equation (11)) is the schedulability test from Reference [29], reformulated and extended in the context of the RT container offline allocation problem. We have extended the test from Reference [29] by including the allocation problem via the binary variables $v_1^k, \ldots, v_n^k$, which select the correct WCET for the given node allocation. If a container k is allocated to a specific node j, then all the Boolean allocation variables $v_1^k, \ldots, v_n^k$ will have a value of 0 except for the Boolean variable v_j^k, which will have a value of 1. Hence, equation Equation (11) reduces to the schedulability test from Reference [29] but additionally selects the correct WCET (C_i^j) for the respective node allocation. We also note that the condition in Equation (10) is not necessary for the schedulability test, since it is implied by Equation (11). However, since Equation (11) is quite complex and results in a large number of assertions, we add the necessary condition in Equation (10) to prune our invalid parameter values for the containers and therefore speed up the search when a solver is used (see below).

Constrained satisfiability or optimization problems such as the one defined above can be solved efficiently using SMT solvers or **Optimization Modulo Theory (OMT)** solvers [12]. SMT solvers address more generalized instances of Boolean SAT problems, which benefit from being formulated in more expressive languages such as first-order logic that include non-Boolean variables and constants, quantifiers, functions, and predicate symbols [11]. To be more efficient and make the satisfiability problem decidable, SMT solvers can restrict the interpretations of specific symbols a specific background theory like linear integer arithmetic over natural numbers, arrays, real-numbers, or bit-vectors [11]. OMT solvers add optimization objectives on top of the underlying first-order logic [12]. There are many efficient state-of-the-art SMT and OMT solvers (e.g., CVC4 [10], MathSAT5 [14], openSMT [13], Z3 [22], Yices [19, 23]) that compete each year against each other in the *SMT-COMP* (cf. Reference [46]) and have been used successfully in application areas ranging from model checking [27], static analysis [38], automated test case generation [35], scheduling [20], and optimization [11]. Hence, the optimization problem defined above is a suitable candidate to be solved using, e.g., OMT solvers [12] like Z3 [22] under the quantifier-free fragment of non-linear integer arithmetic (QF_NIA).

Tuning the weights c_1 and c_2 can be done based on the overhead impact when running RT containers. For example, in Reference [44], we see that the system overhead is not highly impacted by the container period or the number of RT containers; hence, we can aim to optimize the bandwidth usage to a higher degree.

We note that it may not always be necessary to obtain the best solution with respect to the optimization objective from Reference [29] (Equation (3)), since the online phase will compensate for the system overhead and dynamic behavior, and also free up CPU bandwidth that is not used by a container (see below). In this case, we can simplify and speed up the offline phase by retrieving any feasible solution for the RT interfaces. We can thus simply encode the offline dimensioning problem into first-order logic, leaving out the optimization objective, and solve it using high-performance SMT solvers like Yices [23] or Z3 [22].

The offline phase only guarantees the schedulability and correct temporal behavior of tasks in an ideal scenario. At runtime, there may be any number of interference sources that affect the temporal behavior. For example, it is impossible to include a complete model of the underlying system (including the runtime interactions between tasks and containers) in the timing analysis. Additionally, certain runtime artifacts (e.g., cache misses, SW/HW interrupts, the overhead of the underlying operating system), as well as unforeseen changes in task workloads (that have not been considered in the WCET calculation), also influence the runtime temporal behavior. Hence, both the dimensioning and the allocation of the containers from the offline stage may not guarantee that task deadlines are always maintained at runtime. Moreover, new containers may be added during runtime, requiring a dynamic adaptation and allocation of the new container entities to nodes. Hence, we next describe the online orchestration phase that is tailored to adapt to these runtime changes and interferences.

4.2 Online Phase

The online phase complements the initial deployment of the containers aiming to mitigate possible performance disturbances by continuously re-adjusting the system resources of the containers. The online phase takes action on three levels: on the container level, on the node level, and on the cluster level. We envision two components (online monitoring component and online resources adjusting component) interacting with each other to be able to respond to unforeseen changes in the temporal behavior of runtime tasks. The online monitoring component is in charge of monitoring the real-time and health aspects of applications. This aspect is described along with the general framework for deploying and implementing an online container orchestrator module in Reference [44]. The authors introduce **Container-level Metrics (CLM)** to capture and continuously evaluate: (i) the number of *deadline misses*, (ii) the *lateness*, and (iii) the response-time of real-time tasks. Additionally, they also monitor **Operating System-level Metrics (OSLM)** that convey a picture of the health of the underlying system and the containers. In Reference [44], special attention is given to the system overhead, which can affect the temporal isolation property and system utilization. These aspects can be used to optimize the response time of tasks as well as to detect overload scenarios or starvation in BE containers.

4.2.1 Runtime Actions. At runtime, there are several actions that we can take based on the CLM and OSLM measurements. The most straightforward parameter to change is the container budget. For example, when detecting a deadline miss of a task, one can increase the budget of the corresponding container, thus giving the tasks more CPU bandwidth to resume their correct behavior. The decision of when and how much to increase the budget is non-trivial as it depends on multiple aspects like the overall system utilization, the effects on other RT and non-RT containers, and the implications on the system overhead. One can also change the period of a container, e.g., to let it run more frequently. This also has complex implications on the overall system behavior and may also affect other tasks running in the same container. Additionally, the implications of changing the period at runtime on preempted but not finished tasks need to be considered.

A third dimension where one can enact changes is container allocation. If we detect at runtime that the system is becoming over-utilized or that it cannot be guaranteed the temporal correctness of all RT tasks and containers, then the system may move one or multiple containers onto another (less congested) node. Here, the complexity comes from identifying which containers to move and to which node(s) to move them to [33]. Additionally, while the container budget and period are more continuous in nature, since we have a whole range of possible values to choose from, the reallocation decision is inherently a binary one. Thus, at runtime, we need to be very careful when switching from slightly adjusting container parameters to deciding that one or multiple containers need to be moved.

The node-level view is depicted in Figure 2 where RT- and BE-containers coexist in a node, and, additionally, there is one container-level controller per RT container. The container-level controller is responsible for adapting local container-level parameters. While the controllers here are local to the container-level containers (and hence apply changes to local container values), they need to synchronize and orchestrate to node-level and cluster-level controllers. By having this controller hierarchy, we can ensure that the holistic view of the distributed system is maintained and the correct overall decisions are made. We envision simple but fast controllers that interact locally with the budget of a container (within some predefined bounds) and can react quickly to runtime violations. Moreover, a node-level controller orchestrates between the container-level controllers, e.g., to modify the allowed bounds for local budget changes and compute correct dimensioning of, e.g., container periods depending on the overall node-level system state. On the next hierarchical level, a centralized controller orchestrates the migration of containers to other nodes.

Another aspect of online redimensioning or migration is resource and task optimization. Even when no real-time requirements are violated, the controllers may decide that there are enough free resources in a node to redimension a particular container (e.g., increasing its budget) to reduce the task response times. Alternatively, a controller may detect that tasks within an RT-Container finish well before their deadlines and decide to reduce the budget to, e.g., optimize non-functional properties such as power consumption or give more bandwidth to BE containers.

4.2.2 Control Hierarchy. The overall control is divided into three levels of hierarchy. Thus, this control structure allows to maintain the separation of concerns and allows for the implementation of different policies for each of the control levels. For instance, cluster-level control can only focus on defining the strategies for migrating containers without a need for reasoning about resource allocation for the containers (that is, the controllers on other levels). Additionally, such an architecture enables the co-existence of various policies, e.g., there can be various per-container container-level controllers. In the following section, we describe the individual controllers.

4.2.3 Container-level Controller. A primary goal of the *container-level controller* is to free up resources whenever possible by lowering the budget. In this way, the utilization of the given container is reduced, releasing computing resources to be used by the node-level controller for other co-located containers. The container-level controller implements a free-up resource policy that decides the time instant of the free-up action and the number of resources to be released. The decision to reduce the budget is taken based on the measured response time of the tasks within the container ($CLM.rt^{\text{Measured}}$). If the container-level controller notices that the tasks finish with enough laxity to meet their deadlines (in a certain span of time), then the tasks must finish with enough laxity. In that case, it can decide that the tasks do not need as many computing resources as they currently have. Consequently, it may decrease their resource usage by reducing their budget. For real-time tasks, the target is to let tasks response times to be as close as possible to their deadlines to decrease the required budget and hence the deadline can be used as the target CML ($CLM.rt^{\text{Target}}$).

The container-level controller that we use in this work frees up resources proportionally to the deviation between the required response time and the measured response (denoted as ε). The more the budget is overshot (resulting in lower measured response time), the more it releases the overshot budget. The controller follows an Integral Controller strategy, similar to the one presented in Reference [34], to adjust the budget $Q(t)$ allocated to the container at a point in time t:

$$\varepsilon(t) = CLM.rt^{\text{Target}}(t) - CLM.rt^{\text{Measured}}(t), \tag{12}$$

$$Q^{\text{new}}(t) = Q(t-1) - K^{\text{clc}} \cdot \max(\varepsilon(t), 0), \tag{13}$$

$$Q(t) = \max(Q^{\text{new}}(t), Q_{\min}), \tag{14}$$

where the variable $\varepsilon(t)$ denotes the disturbances between the target and measured performance of a given container, the variable $Q^{\text{new}}(t)$ is an intermediate value of the budget calculated by the integral controller, and $Q(t)$ denotes the new value of the budget after release from the container Π^{rt}. The parameter $K^{\text{clc}} \in (0, 1]$ is the container-level controller gain, and it is a design parameter for the controller that defines how aggressively the budget should be decreased. Notice that the allocated budget is saturated to a minimum value to guarantee a minimal allocation of time. This strategy is computationally extremely simple (thus, low overhead), and it is implemented for every RT container running in a given node.

4.2.4 Node-level Controller. The node-level container has complete information about the state of the node $j \in \{1, \ldots, n\}$ (e.g., overall RT-container utilization, Operating System-level Metrics), the state of the containers $i \in \Pi^{\text{rt}}_j$ (e.g., their budgets and periods, Container-level Metrics, etc.), and the history of the state changes. While the container-level controller has the main aim of freeing up resources whenever possible, the node-level container attempts to maintain the real-time properties of tasks by using unused computing resources and distributing them to containers that need them. This is done by increasing the budget. Both decisions need to include system-level aspects like overall utilization, effect on other RT and non-RT containers, system overhead, and so on.

The node-level control aims to distribute the system resources to keep the total utilization of real-time containers under a predefined threshold, so even applications outside real-time containers get access to system resources. However, the controller can temporarily exceed the predefined threshold and oversubscribe resources for real-time containers, e.g., in cases of exceptional disturbances or when a migration of a container to another node is initialized. Node-level controllers keep track of the available resources in the system and define resource re-distribution policies that allocate a part of the resources to containers. The controller can also utilize predictive or probabilistic methods to adjust resources for a particular container before the container suffers from interference or a changing workload.

The node-level controller keeps track of the available CPU resources that can be distributed between RT containers as shown in Figure 3. By default, the controller allocates 80% of the CPU bandwidth that can be used by the RT containers. An unused part of it, which is at least 20% of CPU bandwidth, is used by non-real-time BE containers and other (system) tasks. The node-level controller, computes, for every single container $i \in \Pi^{\text{rt}}_j$, an Integral Control strategy:

$$\epsilon_i(t) = CLM.rt_i^{\text{Target}}(t) - CLM.rt_i^{\text{Measured}}(t), \tag{15}$$

$$Q_i^{\text{new}}(t) = Q_i(t-1) + K_i^{\text{nlc}} \cdot \min(\epsilon_i(t), 0), \tag{16}$$

$$Q_i(t) = \min(Q_i^{\text{new}}(t), Q_i^{\text{av}}(t)), \tag{17}$$

where $K_i^{\text{nlc}} \in (0, 1]$ is the node-level controller gain, and it defines how aggressively the budget should be increased while the response time of the container does not meet the target response time. Moreover, $Q_i^{\text{av}}(t)$ is the currently available budget in the node, as seen by the container $i \in \Pi^{\text{rt}}_j$, and it is computed as

$$Q_i^{\text{av}}(t) = Q_j^{\text{tot}} - \sum_{k \in \Pi^{\text{rt}} \setminus i} Q_k(t),$$

where Q_j^{tot} is the 80% total budget on the computing node.

The controller continuously monitors the disturbances between the target and measured performance ($\epsilon(t)$) and attempts to increase the resources. The control mechanism is similar to the container-level controller, however, this controller can only increase resources for the controller

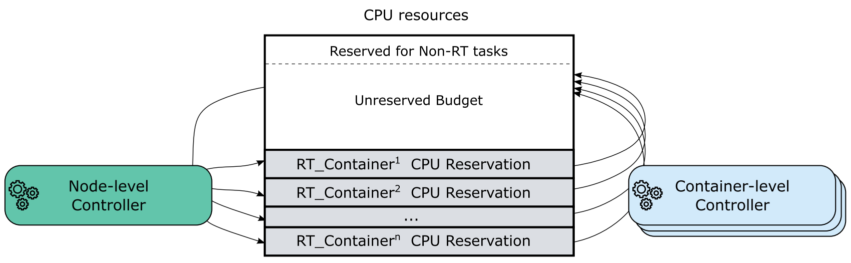

CPU resources
Reserved for Non-RT tasks
Unreserved Budget
RT_Container^1 CPU Reservation
RT_Container^2 CPU Reservation
...
RT_Container^n CPU Reservation
Node-level Controller
Container-level Controller

Fig. 3. CPU resources distribution.

based on the disturbance and available budget. If a container has multiple tasks, then the control action calculation will be based on the worst-behaving tasks with respect to their response times and deadlines, i.e., the task that generates the longest deadline miss.

In case tasks inside a container miss N subsequent deadlines ($\epsilon_i < 0$) and the budget has not increased compared with the previous instances of the controller then the cluster-level controller will be triggered to migrate this container to another node.

4.2.5 Cluster-level Controller. The cluster-level controller monitors individual nodes in the cluster and initializes the migration of containers that are not able to meet demanded real-time performance. The cluster-level controller has a holistic view of the system and can decide to migrate containers to less congested nodes. The node-level controller signals to the cluster-level controller that it cannot meet the real-time requirements of tasks by local changes only. Therefore, the decision to move one or more containers needs to be made. Additionally, the cluster-level container may decide on its own to move certain containers based on, e.g., optimization objectives. However, the controller needs to be very careful not to transform a functioning system into a non-functioning one.

The node-level container selects which container(s) to move, and the cluster-level container decides where to transfer them to. In this case, the cluster-level controller does not need to have some of the local information about the health of the containers, since it does not decide which container to move.

The decision of where to move containers can be made based on more high-level information (e.g., node utilization) and not on more specific data like individual container health, so the container-level controller does not need to have a detailed view of which containers are where and their history (or, e.g., the deadline miss history of a node). As a result, the container-level controller relies on the node-level controller to dimension the container in the right way once it is assigned depending on local information. An alternative is to decide where to move/place a container based on more detailed information and dimension it with a knowledge of the parameters/health/history of other containers on that node. The downside is that the cluster-level controller will need to be updated more frequently and have a more detailed view of each node.

4.3 Implementation

To show the feasibility and the behavior of the system, we implement the proposed Hierarchical Framework in Debian GNU/Linux 10 (buster) patched with the HCBS patch[3] introduced in Reference [1]. This patch allows controlling containers' resources at runtime without any need to modify the container runtime. The resources can be controlled by setting the values in cgroups or by setting the values directly in Linux Kernel (e.g., in the task scheduler or in syscall handlers).

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

We enhance the Linux Kernel to adapt CPU resources for RT containers dynamically. While the HCBS patch allows us to statically set the CPU budget and CPU period for the containers, our additional modifications allow us to dynamically adapt the CPU resources for the RT containers at runtime. We implemented container- and node-controller within the Linux Kernel (specifically, a task scheduler to compute CLM metrics and custom syscalls to trigger control actions). The controllers directly access the container, tasks, and resource reservation entities without causing additional overhead. The cluster-level controller is implemented as a user-space application that periodically observes the states of RT containers and performs migration of a deficient container(s) to another node. The migration process is described in Section 4.3.1.
The workflow and implementation overview is as follows:

- A container is executed with the initial parameters computed in the offline phase (e.g., *docker run –cpu-rt-runtime=40000 –cpu-rt-period=50000 rt_container). The initial values are stored in cgroups as cpu.rt_quota and cpu.rt_period. The values can be changed at runtime from the kernel space.*
- The scheduler selects the highest priority container with available CPU quota, i.e., containers with the earliest deadline following the SCHED_DEADLINE policy. After that, the highest priority runnable tasks are selected from each of the scheduled containers and granted a CPU.
- Each task, implemented in *struct task_struct*, is annotated with a task period and activation start. The values are used to compute the performance metrics.
- On each job activation, the Linux Kernel is notified that the job has started via a custom syscall.
- After each job is completed, the Kernel is notified about the job competition via a custom syscall, and the task is suspended until the time of the next job activation.
- Upon the finish-job syscall, the CLM metrics are updated. The CLM metrics are computed based on the activation time, finishing time, and period of the job. We compute response time, and lateness, and update the deadline miss counter. The CLM metrics are saved in */proc* filesystem and are accessible to all controllers.
- Additionally, upon the finish-job syscall, the custom syscall executes the control hierarchy as follows (as described in Section 4.2):
  – Container-level controller attempts to free up reserved resources if possible.
  – Node-level controller increases resources to the containers with unsatisfactory performance. If there are not enough free resources, then the containers are marked as to-be-migrated (in */proc* filesystem).
- Cluster-level controller periodically check for to-be-migrated containers and performs the migration process.

4.3.1 Migration of Containers. The migration of containers is performed by a custom python-based Cluster-level controller application based on a client-server architecture. The server-side application keeps track of connected computed nodes, deployed containers, and performance metrics of the nodes and individual containers. The client side periodically reports performance metrics to the server and performs the migration of containers as depicted in Figure 4.

Once a container fails to obtain the necessary resources to meet the desired performance, the Node-level controller records it in /proc filesystem. This triggers the Cluster-level controller on the client side to inform the server of the container that needs to be relocated. The server then finds a host that meets the requirements mentioned in Section 4.1 and requests the new host to create the container. When the container is ready, the new host notifies the old host, prompting it to stop the container.

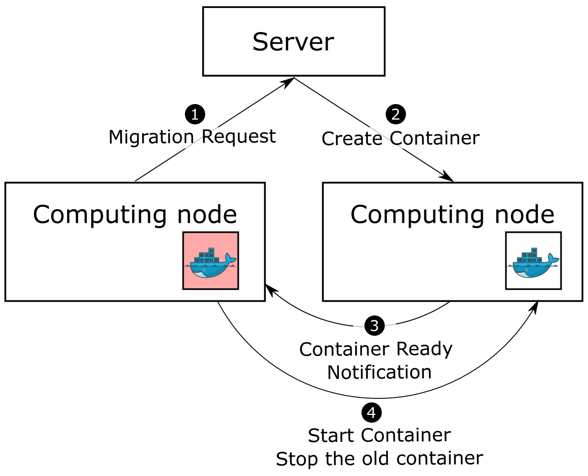


Fig. 4. Migration process of containers.

5 EVALUATION OF THE FRAMEWORK

In this section, we evaluate the proposed Hierarchical Resource Orchestration Framework. We perform the experiments on two COTS platforms: Intel Core i5, 4 GB RAM, Debian Linux 5.2.8 patched with HCBS patch [1] and Intel Core i7, 16 GB RAM, Debian Linux 5.2.8 patched with HCBS patch. Moreover, we run some scalability experiments for the offline phase using the z3 solver in version 4.11.2.

5.1 Demonstration of Performance Interference between Containers

We illustrate the interference caused by the use of shared resources between co-located containers executed on a single computing node as shown in Figure 1. We perform two experiments that run a single RT container collocated with (a) one non-real-time container and (b) multiple non-real-time containers. In the first case (depicted in Figure 1(a)), the co-located non-real-time container is periodically changing its workload between cache-intensive workload (matrix multiplication) and simple workload (simple busy loop). As seen in the subfigure, the response time of an RT container is aligned with the period of cache-intensive (e.g., between 0 s and 20 s, 40 s and 62 s, etc.) workload in the co-located non-real-time container. The response time ranges between 2.7 s and 3.7 s. In the second case (depicted in Figure 1(b)), in addition to the RT container, we gradually execute non-real-time containers performing a cache-intensive workload. We can see that with the increased number of non-real-time containers, the response time of the RT container exhibits increasing response time. The results of the experiment show the performance interference of the container.

In the following sections, we first evaluate the offline phase, and then we show the dynamic adaptation of the CPU resources that can deal with performance disturbances of the executed containers by the proposed hierarchical framework.

5.2 Evaluation of the Offline Phase

In the offline phase, we study the scalability of solving the constrained optimization problem as defined in Section 4.1. Generally, the allocation problem in hierarchical scheduling reduces to the bin-packing problem [47] and is thus contained in the NP-hard complexity class. We implemented the constrained optimization problem using the z3 solver in version 4.11.2.[4]

[4]Available at https://github.com/Z3Prover/z3.

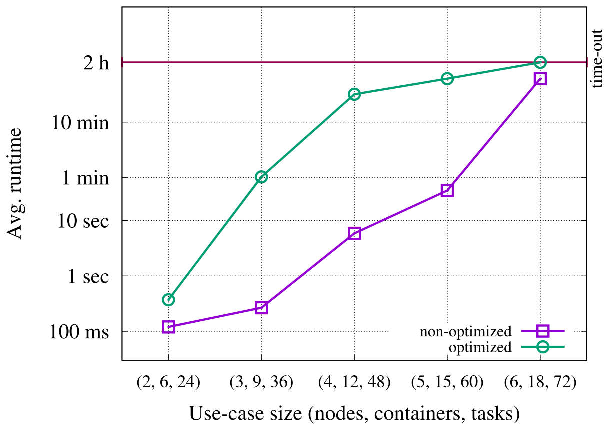


Fig. 5. Scalability results for the offline dimensioning and allocation step with and without optimization.

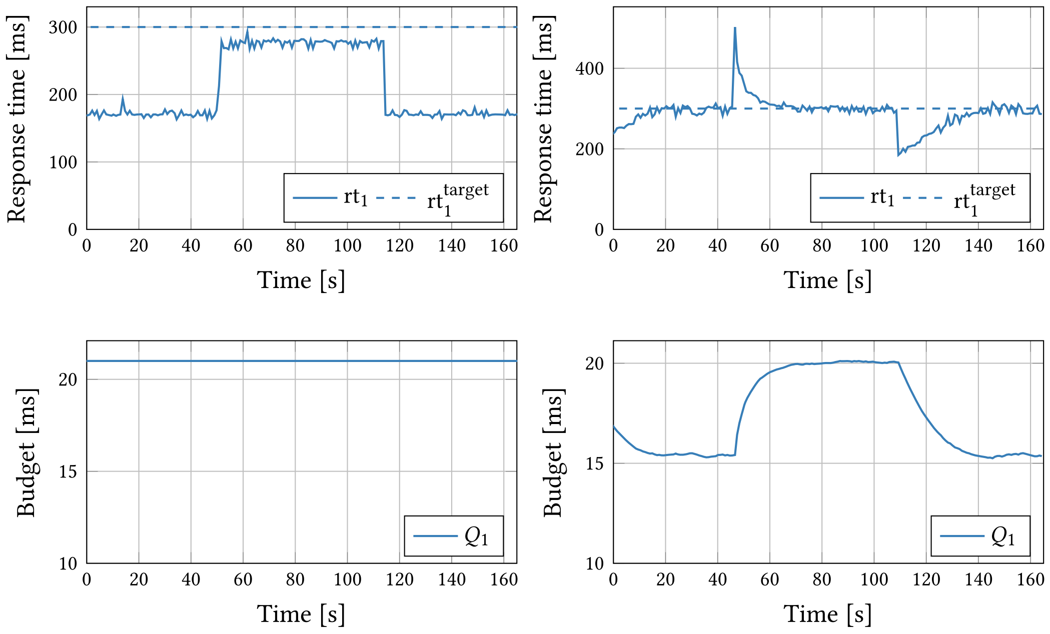
We generate 5 different problem sizes in terms of the number of nodes, the number of containers, and the number of tasks, and for each problem size, we generate 10 test cases. Each node has a memory and storage availability chosen randomly between 10 MB and 20 MB and between $1GB$ and $2GB$, respectively. The tasks are defined by a period chosen randomly from the set $\{200, 300, 400, 500, 600, 700, 800, 900, 1{,}000\}$ ms, a WCET selected randomly between 1 and 10 ms, a storage usage randomly chosen between 2 and 8 MB, and a memory usage chosen randomly between 16 and 64 KB.

In Figure 5, we show the scalability of the offline assignment and dimensioning with the following system configurations (x-axis) in terms of (nodes, containers, tasks): $(2, 6, 24)$, $(3, 9, 36)$, $(4, 12, 48)$, $(5, 15, 60)$, $(6, 18, 72)$. On the logarithmic y-axis, we show the average runtime when solving the offline phase with and without optimization constraints and for each size configuration defined above. We set the timeout to 2 h, after which we deem the test case to be infeasible.

As can be seen, the average runtime increases exponentially, both with and without optimization, as the size of the problem increases. Without optimization criteria, we see that also relatively large use cases can be solved in a reasonable amount of time with a system consisting of 5 nodes, 15 containers, and 60 tasks needing, on average, 34 s. For a system consisting of 6 nodes, 18 containers, and 72 tasks, 5 of the 10 test cases reached the timeout, while the average runtime for the other 5 "solvable" test cases was around 1.6 min. As expected, when the optimization criteria of minimizing the utilization of the containers ($c_1 = 0$ and $c_2 = 1$) were enabled, the average runtime grew more quickly, leading to worse scalability. We can see that small to medium system configurations can be solved in a reasonable time, especially when optimization is not enabled, which is usually the preferred approach, since the online phase will take care of freeing up unused resources and adapting to fluctuations due to system and context switch overheads. For larger systems, using SMT or OMT solvers that deliver optimal solutions will not scale anymore, making it necessary to use heuristics that sacrifice optimality and completeness for faster runtimes.

5.3 Evaluation of Container-level and Node-level Controllers

This section discusses two types of experiments, depicted in Figures 6 and 7, respectively. Both experiments focus on the online CPU budget adaptation using the joint container-level and node-level controllers that decrease/increase the container CPU budget to reach the target performance level. The first experiment (Figure 6) shows the response time and budget of a single container, in the case of a fixed budget (Figure 6(a)), and of a variable budget allocation (Figure 6(b)). The workload is varied within the container.



(a) A baseline scenario without the budget adaptation. (b) A scenario with online budget adaptation.

Fig. 6. An experiment showing an effect of budget adaptation.

In Figure 6(a), we statically set the replenishment quota to 21 ms over the period of 25 ms. This scenario does not employ any adaptation mechanism. The response time changes periodically between 170 and 285 ms, while the target response time is set to 300 ms. We can see the changing response time of the container at times 45s, 116s. The allocated budget remains constant. This leads to the over-allocation of computing resources, part of the unused reserved budget could be reserved for another RT container (that may have insufficient resources). Figure 6(b) shows the same scenario with the enabled online CPU budget adaptation. In this case, the allocated budget is automatically adjusted to match the target response time. As there is no control nor measurement of the time-varying workload, the control mechanism dynamically reacts to its variation without any information on how long such a variation can last. The controller, therefore, serves as an iterative learning mechanism to identify the right allocation of the budget over time. In this scenario, we used two different K parameters in the container-level ($K^{\text{clc}} = 0.3$) and node-level ($K^{\text{nlc}} = 0.6$) controllers to have a more aggressive strategy to increase the budget and a more relaxed strategy to release the budget. When increasing the budget of a container it is better to do it as fast as possible to reduce the quality degradation in terms of deadline misses. In contrast, it is better to release the budget gradually to avoid sudden increases in response times.

The second experiment (Figure 7) shows two RT containers deployed on a single computing node. The figure shows (a) the measured response time and target response time, (b) per-container budget allocation, and (c) the percentage of the entire CPU budget allocation.

The containers are experiencing performance disturbances. For instance, RT container 1 experiences an increased workload between 21 and 63 s, and between 108 and 148 s. The RT container 2 experiences an increased workload between 39 and 95s, and between 152 and 210 s. The per-container budget allocation shows that the budget is increased by the node-level controller and decreased by the container-level controller in line with the change in workload that is experienced.

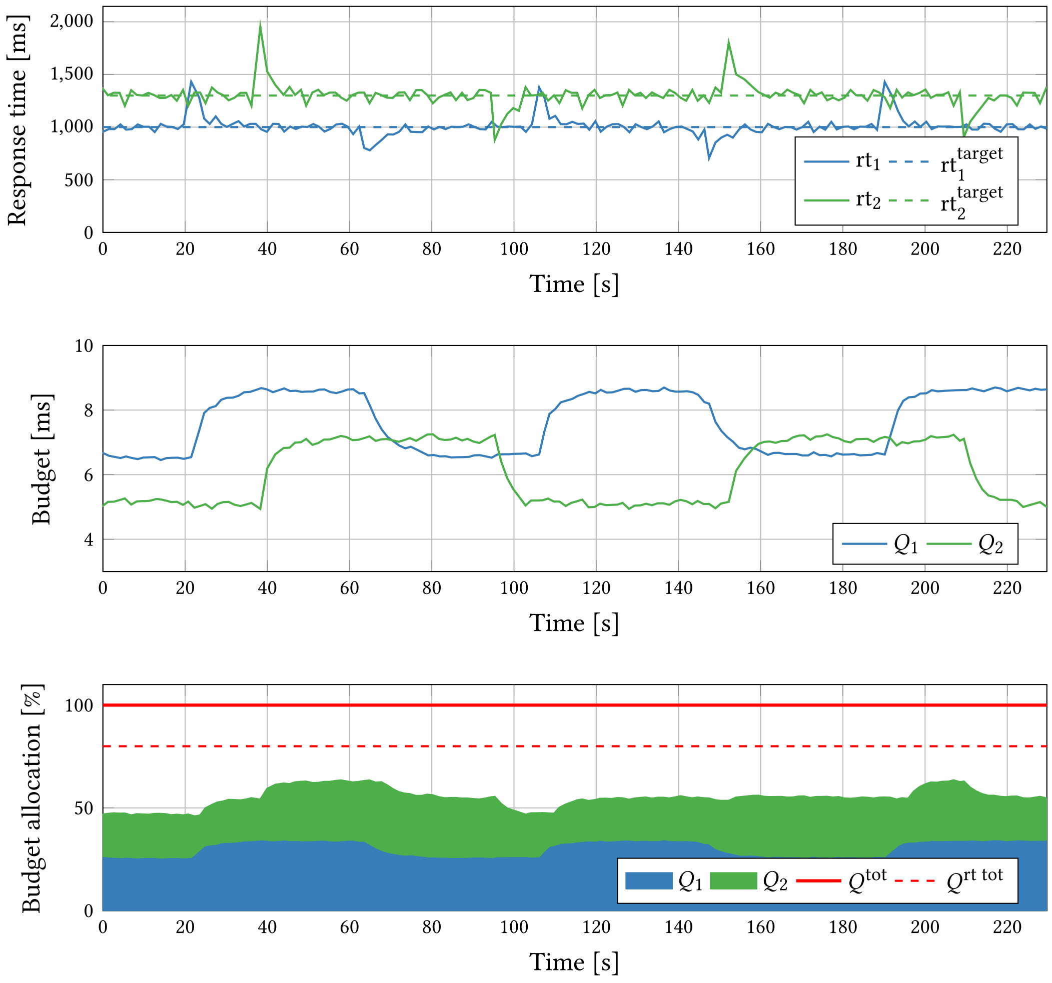


Fig. 7. Distribution of budget amongst containers. Response times and budget allocation per container.

The figure shows how the two controllers are able to dynamically adjust the allocated budgets to match the desired target responses of the respective containers. The bottom graph in Figure 7 shows how the overall budget is partitioned over the two containers. In this specific example, we do not consider the case in which the overall allocated budget exceeds the maximum share allowed for the real-time containers (indicated as a dashed red line in the graph). The case in which such a threshold is exceeded would trigger a container migration, and it is analyzed in the next section.

5.4 Evaluation of Cluster-level Controller

The experiment depicted in Figure 8 shows the collaboration between the three-level control hierarchy proposed in this article. On the x-axis, we depict the timeline of the system and its trace. Container- and node-level controllers are designed to balance CPU resources amongst two RT containers, while the cluster-level controller is designed to monitor the performance of each of the RT containers, and if the performance is not as desired for a period of time, then one of the containers will be stopped and redeployed to another computing (less loaded) node.

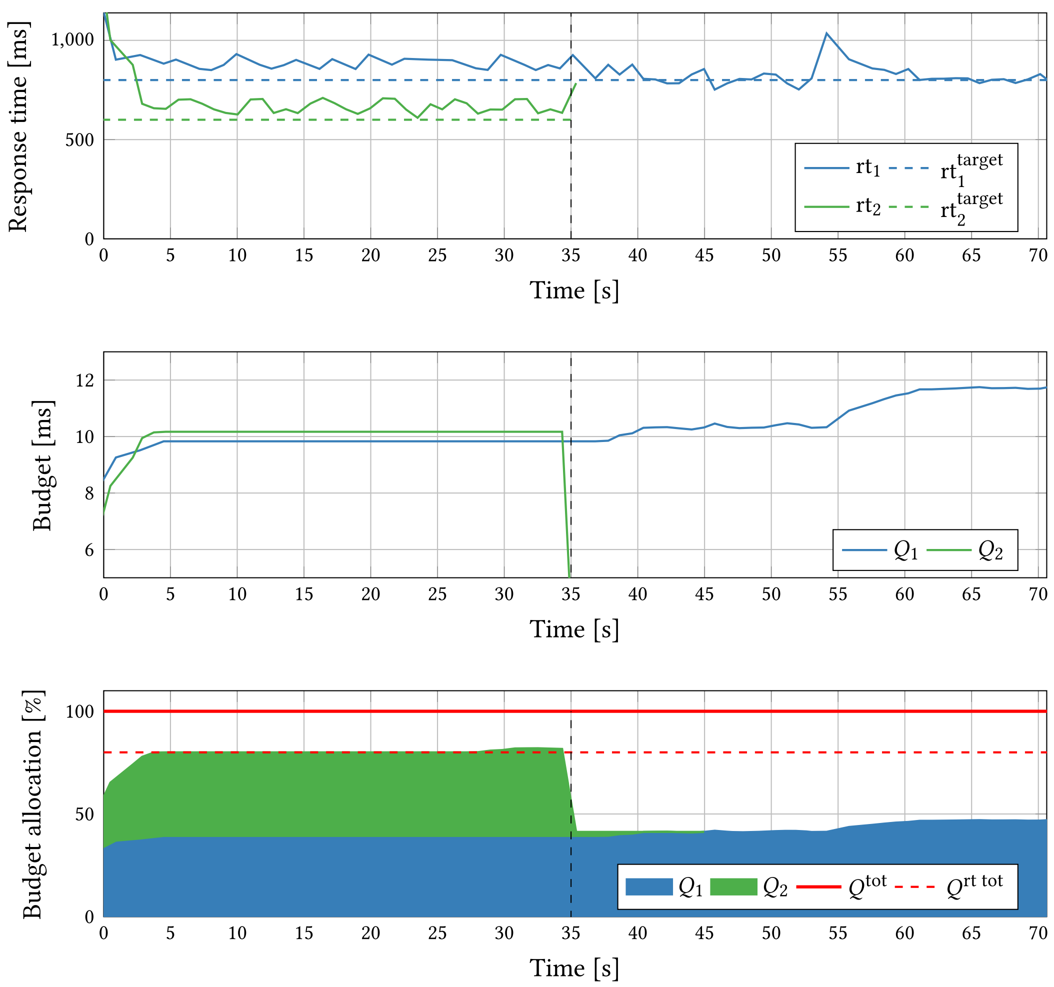


Fig. 8. A demonstration of a cluster-level controller that migrates Container 2 on a less congested computing node.

The experiment shows two RT containers (Initial $Q_1 = 8$ ms and $Q_2 = 7.5$ ms with periods 25 ms) that are unable to reach the target response times (800 and 600 ms) due to insufficient CPU budget (top figure). On the y-axis, we depict the response time in milliseconds, showing both the desired and measured response times of the two containerized tasks with dotted and solid lines, respectively. In the middle part of the figure, we see the budget allocation for both containers (the budget being depicted on the y-axis in milliseconds). Moreover, in the bottom part of the figure, we see that the overall budget allocation is saturating at the maximum set value of 80% of total CPU bandwidth. Hence, since the node- and container-level controllers cannot change the budget allocation in such a way that both containers meet their respective response times, the cluster-level controller performs migration of one of the containers to relieve the computing node.

At the time 35 s, the cluster-level controller detects that the system can not satisfy the response-times requirements of the deployed containers (neither of the containers can reach the target response time). Thus, the controller chooses container 2 to be redeployed on another node. The container is stopped, and the allocated budget (Q_2) is fully released. Then, container 1 can instantly use the newly allocated budget, thus being able to fulfill its response time requirement. We can see in the top part of the figure that the response time of the remaining containerized application reaches the desired level a short time after the migration has been finished. Moreover, at time 54 s, we can see that there is some interference on the remaining container, causing a spike in the response time and the node-level controller being able to allocate even more of the freed-up budget to the remaining container.

6 DISCUSSION

The experiments that we conducted show that the proposed online adaptation mechanism can adapt the CPU bandwidth of real-time containers and match the required performance. However, the control mechanism can also decrease the resource for containers with over-reserved resources and these resources can be used by containers with poor performance.

The implementation in the Linux Kernel enables a more rapid control loop as well as the ability for runtime adaptation of resources without the need for so-called breath-duration periods, which are break intervals that allow a system to reach a steady state after scaling [5].

The system takes control decisions at the end of each job. However, it is possible to adjust resources during the execution of the job, not only at the end of the job. For example, if a system detects that a job may not finish before the deadline, then the system may allocate more resources at the time instant of the detection of a possible delay. However, we keep this topic for future work.

The limitation of the experiment is the use of synthetic tasks; we are seeking a use-case to demonstrate our control mechanism for real-world application (e.g., autonomous driving, robot control). Also, previously we defined CLM and OSLM metrics that contain a multitude of values that describe the real-time performance of containers and the system; however, in our experiments, we use only the basic response time metric.

7 CONCLUSION

In this article, we propose a Hierarchical Resource Orchestration Framework for Real-time Containers that aims to mitigate the timing disturbances present in container-based virtualization due to, e.g., the effects of the use of shared resources, which can affect the temporal behavior of real-time containers. The proposed framework contains two phases: an offline phase that decides the initial deployment and dimensioning of the real-time containers and an online phase that complements the initial deployment to re-adjust the resources when unexpected changes occur.

We implement the framework on a real Linux-based system patched with the HCBS patch to show the feasibility of the proposed framework. In our experiments, we demonstrate (a) the effect of performance interference between containers, (b) the control of resources for the containers, (c) the re-distribution of resources between the containers, and (d) a migration of containers when the overall resources are not sufficient to meet the real-time needs of the containerized applications. Moreover, we also perform a scalability evaluation of an SMT-based implementation of the offline phase. Our experiments show that the proposed framework can mitigate timing disturbances of containerized applications in Linux-based systems and is able to adjust and re-distribute computing resources between containers when needed.

There are several interesting directions for future work. First, the proposed framework utilizes a simple control mechanism to reserve and distribute CPU resources for the RT containers. However, more complex control, decision, and predictive algorithms are needed to capture more complex scenarios. We aim to extend the cluster-level controller with the ability to identify the optimal set of containers to migrate, since it may be more beneficial to migrate the containers that are causing performance interference instead of those with poor real-time performance. Additionally, the offline phase utilizes SMT (or OMT) solvers that deliver optimal solutions but does not scale for large systems. Hence, we aim to investigate efficient heuristics that can scale for large systems, like those found in some industrial automation applications.

REFERENCES

[1] Luca Abeni, Alessio Balsini, and Tommaso Cucinotta. 2019. Container-based real-time scheduling in the Linux kernel. *ACM SIGBED Rev.* 16, 3 (Nov. 2019), 33–38. https://doi.org/10.1145/3373400.3373405

[2] Luca Abeni, Alessandro Biondi, and Enrico Bini. 2022. Partitioning real-time workloads on multi-core virtual machines. *J. Syst. Arch.* 131 (2022), 102733.

[3] Luca Abeni and Giorgio Buttazzo. 2004. Resource reservation in dynamic real-time systems. *Real-Time Syst.* 27, 2 (July 2004), 123–167. https://doi.org/10.1023/b:time.0000027934.77900.22

[4] Luca Abeni, Giuseppe Lipari, and Juri Lelli. 2015. Constant bandwidth server revisited. *ACM SIGBED Rev.* 11, 4 (Jan. 2015), 19–24. https://doi.org/10.1145/2724942.2724945

[5] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. 2017. Autonomic vertical elasticity of docker containers with ElasticDocker. In *Proceedings of the IEEE 10th International Conference on Cloud Computing (CLOUD'17)*. https://doi.org/10.1109/cloud.2017.67

[6] Luis Almeida and Paulo Pedreiras. 2004. Scheduling within temporal partitions: Response-time analysis and server design. In *Proceedings of the 4th ACM international conference on Embedded software (EMSOFT'04)*. https://doi.org/10.1145/1017753.1017772

[7] Marco Barletta, Marcello Cinque, and Raffaele Della Corte. 2021. Hierarchical scheduling for real-time containers in mixed-criticality systems. In *Proceedings of the IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW'21)*.

[8] Marco Barletta, Marcello Cinque, Luigi De Simone, and Raffaele Della Corte. 2022. Achieving isolation in mixed-criticality industrial edge systems with real-time containers. In *Proceedings of the 34th Euromicro Conference on Real-Time Systems (ECRTS'22)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik.

[9] Marco Barletta, Marcello Cinque, Luigi De Simone, and Raffaele Della Corte. 2022. Introducing k4.0s: A Model for Mixed-Criticality Container Orchestration in Industry 4.0. https://doi.org/10.48550/ARXIV.2205.14188

[10] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanović, Tim King, Andrew Reynolds, and Cesare Tinelli. 2011. CVC4. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer, Berlin, 171–177.

[11] Clark Barrett and Cesare Tinelli. 2018. *Satisfiability Modulo Theories*. Springer International Publishing, Cham, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11

[12] Nikolaj Bjørner, Anh-Dung Phan, and Lars Fleckenstein. 2015. νZ—An optimizing SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Springer, Berlin, 194–199. https://doi.org/10.1007/978-3-662-46681-0_14

[13] Roberto Bruttomesso, Edgar Pek, Natasha Sharygina, and Aliaksei Tsitovich. 2010. The OpenSMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Javier Esparza and Rupak Majumdar (Eds.). Springer, Berlin, 150–153.

[14] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. 2013. The MathSAT5 SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, Nir Piterman and Scott A. Smolka (Eds.). Springer, Berlin, 93–107.

[15] Marcello Cinque and Domenico Cotroneo. 2018. Towards lightweight temporal and fault isolation in mixed-criticality systems with real-time containers. In *Proceedings of the 48th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W'18)*.

[16] Marcello Cinque, Domenico Cotroneo, Luigi De Simone, and Stefano Rosiello. 2022. Virtualizing mixed-criticality systems: A survey on industrial trends and issues. *Future Gen. Comput. Syst.* 129 (2022), 315–330. https://doi.org/10.1016/j.future.2021.12.002

[17] Marcello Cinque, Raffaele Della Corte, Antonio Eliso, and Antonio Pecchia. 2019. RT-CASEs: Container-based virtualization for temporally separated mixed-criticality task sets. In *Proceedings of the 31st Euromicro Conference on Real-Time Systems (ECRTS'19)*. https://doi.org/10.4230/LIPIcs.ECRTS.2019.5

[18] E. G. Coffman, M. R. Garey, and D. S. Johnson. 1996. *Approximation Algorithms for Bin Packing: A Survey.*

[19] Computer Science Laboratory–SRI International. [n.d.]. The Yices SMT Solver. Retrieved from http://yices.csl.sri.com/. on 4-Jan-2023.

[20] Silviu S. Craciunas and Ramon Serna Oliver. 2016. Combined task- and network-level scheduling for distributed time-triggered systems. *J. Real-Time Syst.* 52, 2 (2016), 161–200.

[21] Tommaso Cucinotta, Luca Abeni, Mauro Marinoni, Riccardo Mancini, and Carlo Vitucci. 2021. Strong temporal isolation among containers in OpenStack for NFV services. *IEEE Trans. Cloud Comput.* (2021). https://doi.org/10.1109/tcc.2021.3116183

[22] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Proceedings of the International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'08)*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer, Berlin, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[23] Bruno Dutertre. 2014. Yices 2.2. In *Proceedings of the Conference on Computer-Aided Verification (CAV'2014) (Lecture Notes in Computer Science)*, Armin Biere and Roderick Bloem (Eds.). Springer, 737–744. https://doi.org/10.1007/978-3-319-08867-9_49

[24] Stefano Fiori, Luca Abeni, and Tommaso Cucinotta. 2022. RT-kubernetes: Containerized real-time cloud computing. In *Proceedings of the 37th ACM/SIGAPP Symposium on Applied Computing*. ACM. https://doi.org/10.1145/3477314.3507216

[25] Theodosios Gkamas, Vasileios Karaiskos, and Sotirios Kontogiannis. 2022. Performance evaluation of distributed database strategies using Docker as a service for industrial IoT data: Application to industry 4.0. *Information* 13, 4 (2022), 190.

[26] Thomas Goldschmidt, Stefan Hauck-Stattelmann, Somayeh Malakuti, and Sten Grüner. 2018. Container-based architecture for flexible industrial control applications. *J. Syst. Architect.* 84 (Mar. 2018), 28–36. https://doi.org/10.1016/j.sysarc.2018.03.002

[27] Anvesh Komuravelli, Arie Gurfinkel, and Sagar Chaki. 2014. SMT-based model checking for recursive programs. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 17–34.

[28] Heiner Lasi, Peter Fettke, Hans-Georg Kemper, Thomas Feld, and Michael Hoffmann. 2014. Industry 4.0. *Bus. Info. Syst. Eng.* 6, 4 (June 2014), 239–242. https://doi.org/10.1007/s12599-014-0334-4

[29] Giuseppe Lipari and Enrico Bini. 2003. Resource partitioning among real-time applications. In *Proceedings of the 15th Euromicro Conference on Real-Time Systems, 2003. Proceedings.* https://doi.org/10.1109/EMRTS.2003.1212738

[30] Chung Laung Liu and James W. Layland. 1973. Scheduling algorithms for multiprogramming in a hard-real-time environment. *J. ACM* 1 (1973). https://doi.org/10.1016/b978-155860702-6/50016-8

[31] Alexandru Moga, Thanikesavan Sivanthi, and Carsten Franke. 2016. OS-level virtualization for industrial automation systems: Are we there yet? In *Proceedings of the 31st Annual ACM Symposium on Applied Computing*. https://doi.org/10.1145/2851613.2851737

[32] Ripal Nathuji, Aman Kansal, and Alireza Ghaffarkhah. 2010. Q-clouds: Managing performance interference effects for QoS-aware clouds. In *Proceedings of the 5th European conference on Computer systems (EuroSys'10)*. https://doi.org/10.1145/1755913.1755938

[33] Alessandro Vittorio Papadopoulos and Martina Maggio. 2015. Virtual machine migration in cloud infrastructures: Problem formalization and policies proposal. In *Proceedings of the IEEE 54th Annual Conference on Decision and Control (CDC'15)*. IEEE. https://doi.org/10.1109/CDC.2015.7403274

[34] Alessandro Vittorio Papadopoulos, Martina Maggio, Alberto Leva, and Enrico Bini. 2015. Hard real-time guarantees in feedback-based resource reservations. *Real-Time Syst.* 51, 3 (Jun. 2015), 221–246. https://doi.org/10.1007/s11241-015-9224-1

[35] Jan Peleska, Elena Vorobev, and Florian Lapschies. 2011. Automated test case generation with SMT-solving and abstract interpretation. In *NASA Formal Methods*, Mihaela Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi (Eds.). Springer, Berlin, 298–312.

[36] Fabiana Rossi, Matteo Nardelli, and Valeria Cardellini. 2019. Horizontal and vertical scaling of container-based applications using reinforcement learning. In *Proceedings of the IEEE 12th International Conference on Cloud Computing (CLOUD'19)*. https://doi.org/10.1109/cloud.2019.00061

[37] Shaik Mohammed Salman, Václav Struhár, Alessandro Vittorio Papadopoulos, Moris Behnam, and Thomas Nolte. 2019. Fogification of industrial robotic systems: Research challenges. In *Proceedings of the Workshop on Fog Computing and the IoT (Fog-IoT'19)*. https://doi.org/10.1145/3313150.3313225

[38] Sanjit A. Seshia and Jonathan Kotker. 2011. GameTime: A toolkit for timing analysis of software. In *Tools and Algorithms for the Construction and Analysis of Systems*, Parosh Aziz Abdulla and K. Rustan M. Leino (Eds.). Springer, Berlin, 388–392.

[39] Mohammed Salman Shaik, Vaclav Struhar, Zeinab Bakhshi, Van-Lan Dao, Nitin Desai, Alessandro V. Papadopoulos, Thomas Nolte, Vasileios Karagiannis, Stefan Schulte, Alexandre Venito, and Gerhard Fohler. 2020. Enabling fog-based industrial robotics systems. In *Proceedings of the 25th IEEE Conference on Emerging Technologies and Factory Automation (ETFA'20)*. https://doi.org/10.1109/etfa46521.2020.9211887

[40] Shashank Shekhar, Hamzah Abdel-Aziz, Anirban Bhattacharjee, Aniruddha Gokhale, and Xenofon Koutsoukos. 2018. Performance interference-aware vertical elasticity for cloud-hosted latency-sensitive applications. In *Proceedings of the IEEE 11th International Conference on Cloud Computing (CLOUD'18)*. https://doi.org/10.1109/cloud.2018.00018

[41] Insik Shin and Insup Lee. 2003. Periodic resource model for compositional real-time guarantees. In *Proceedings of the International Symposium on System-on-Chip*. IEEE Comput. Soc. https://doi.org/10.1109/REAL.2003.1253249

[42] Michael Sollfrank, Frieder Loch, Steef Denteneer, and Birgit Vogel-Heuser. 2020. Evaluating docker for lightweight virtualization of distributed and time-sensitive applications in industrial automation. *IEEE Trans. Industrial Inform.* 17, 5 (2020), 3566–3576.

[43] Václav Struhár, Moris Behnam, Mohammad Ashjaei, and Alessandro V. Papadopoulos. 2020. Real-time containers: A survey. In *Proceedings of the 2nd Workshop on Fog Computing and the IoT (Fog-IoT'20)*. Schloss Dagstuhl-Leibniz-Zentrum für Informatik, Schloss Dagstuhl-Leibniz-Zentrum für Informatik. https://doi.org/10.4230/OASIcs.Fog-IoT.2020.7

[44] Václav Struhár, Silviu S. Craciunas, Mohammad Ashjaei, Moris Behnam, and Alessandro V. Papadopoulos. 2021. REACT: Enabling real-time container orchestration. In *Proceedings of the 26th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA'21)*. https://doi.org/10.1109/etfa45728.2021.9613685

[45] Sébastien Vaucher, Rafael Pires, Pascal Felber, Marcelo Pasin, Valerio Schiavoni, and Christof Fetzer. 2018. SGX-aware container orchestration for heterogeneous clusters. In *Proceedings of the IEEE 38th International Conference on Distributed Computing Systems (ICDCS'18)*. https://doi.org/10.1109/icdcs.2018.00076

[46] Tjark Weber, Sylvain Conchon, David Déharbe, Matthias Heizmann, Aina Niemetz, and Giles Reger. 2019. The SMT competition 2015-2018. *J. Satisf. Boolean Model. Comput.* 11, 1 (2019), 221–259.

[47] Brandon Woolley, Susan Mengel, and Atila Ertas. 2020. An evolutionary approach for the hierarchical scheduling of safety- and security-critical multicore architectures. *Computers* 9, 3 (2020). https://doi.org/10.3390/computers9030071

Received 10 November 2022; revised 18 February 2023; accepted 27 March 2023