Real-time Container Orchestration Based on Time-utility Functions

Stefan Walser
TTTech Computertechnik AG
Vienna, Austria
stefan.walser@student.tuwien.ac.at

Jan Ruh
TTTech Computertechnik AG
Vienna, Austria
jan.ruh@tttech.com

Silviu S. Craciunas
TTTech Computertechnik AG
Vienna, Austria
silviu.craciunas@tttech.com

***Abstract*—Container-based virtualization is a lightweight deployment solution within heterogeneous Fog and Edge Computing (FEC) systems. When used in real-time FEC applications, orchestration is needed to adapt to variations in system behavior, transient overload scenarios, and changes in resource availability. Container-based orchestration can efficiently and automatically deploy, redimension, and relocate containers according to the measured real-time performance of critical applications. This paper presents a highly configurable orchestration framework for real-time containers that improves the scalability and expressiveness of state-of-the-art approaches. We propose three different heuristics for the offline placement and dimensioning of containers on nodes that scale with the size of modern industrial systems and enhance the decision-making of the online hierarchical controller to include time-utility functions which can model more accurately the usefulness of results in the case of deadline misses. We show our framework's performance and efficiency using synthetic test cases and a real-world test bed.**

***Index Terms*—real-time container, orchestration, virtualization**

I. Introduction

Fog and Edge Computing (FEC) unite the advantages of cloud computing with the ability to transfer low-latency control functions to edge nodes. This helps reduce the communication latency for critical messages, which is crucial for real-time applications, e.g., in industrial automation [1], [2]. Virtualization in FEC improves the deployment and adaptation of services based on the current state and capabilities of the nodes [3]. Container-based virtualization has the benefits of smaller image sizes and faster startup times [4], offering a lightweight deployment within heterogeneous cloud or fog environments with less overhead than full virtualization. As a result, containerization has become the preferred way to package and deploy applications in FEC [5], [6]. Container-based virtualization facilitates FEC because additional instances can be quickly started and migrated between server instances as needed depending on an application's demand, and resources can be optimized by removing unnecessary instances [7]. Hence, in industrial automation, container-based virtualization and orchestration have a significant role in fulfilling Industry 4.0 requirements for flexibility and resource efficiency [5], [8]–[10] but can also be beneficial for other mixed-criticality domains like, e.g., mobile machinery or energy applications.

This research was funded by the European Commission under the Horizon Europe project aerOS, grant number 101069732, and the AIMS5.0 project, which is supported by the Chips Joint Undertaking and its members, including the top-up funding by National Funding Authorities from involved countries under grant agreement no. 101112089.

GNU/Linux operating systems realize containers using namespaces and control groups (cgroups), isolating and providing access limitations to applications (tasks) when accessing resources such as CPU, memory, or IO [11]. However, containerization cannot guarantee meeting critical deadlines in real-time applications. Abeni et al. [11] extended the `rt_group_sched` feature of the Linux kernel to enable the hierarchical scheduling of containers so that containers are scheduled using earliest-deadline first (EDF), and tasks within the containers use a fixed-priority (FP) scheduler. Furthermore, group scheduling allows the user to reserve a period and a quota of CPU time for each container. However, these methods are insufficient to fully utilize containerization in the real-time domains since variations in runtime overhead, runtime changes in workloads, and resource availability changes often lead to a degradation in real-time behavior. An orchestration tool is necessary to automatically deploy, redimension, and relocate containers to maintain their real-time performance at runtime.

Juliàn et al. [12] reviewed automated management in cloud-based architectures aimed at improving network latency, power consumption, and overheads. The authors propose a new classification of state-of-the-art cloud nodes based on their self-* capabilities, e.g., self-configuration, self-orchestration, self-adaptation, or self-learning. The latter act as intelligent resource managers in a virtualized, platform-agnostic meta-operating system and allow the deployment of multiple services in the IoT edge-cloud continuum. We extend this classification with a concept of self-real-timeliness, ensuring the system's real-time performance based on the so-called time-utility function to orchestrate, schedule, and monitor real-time workloads. Struhár et al. [5], [13] present an orchestration framework that uses an offline SMT-based approach to create an initial dimensioning of containers on nodes, and propose a hierarchical online control approach for redimensioning and moving containers based on the runtime performance and system load. However, the approach in [5], [13] suffers from the downside that the offline phase does not scale for real industrial systems and that the online adaptation of containers is based on simple metrics that do not capture the utility of computation results after a deadline miss.

In this paper, we present a highly configurable orchestration framework that uses a similar control hierarchy as in [5], with the following main improvements. We (1) redesign, reimplement, and optimize the controller hierarchy from [5], (2) propose 3 different heuristics for the offline placement and dimensioning phase that scale with the size of modern industrial systems, and (3) enhance the decision-making of the online hierarchical controller to include time-utility functions which can model more accurately the usefulness of results in the case of deadline misses. We show the scalability of our offline heuristics and the performance of our online orchestration based on time-utility functions using synthetic test cases and a real-world test bed.

II. Related Work

The Linux kernel has been enhanced with real-time capabilities through the PREEMPT_RT patch [14], a Co-Kernel approach with a custom API [15], and hierarchical scheduling [11] which schedules task containers via EDF scheduler and the tasks within the container via FP.

Several works study real-time container orchestration. A survey of real-time containerization and orchestration approaches is presented in [16]. Fiori et al. [17] adapt the orchestration framework Kubernetes such that timing guarantees are fulfilled using the HCBS patch presented in [11], similar to our work. Wang et al. [18] evaluate different RT container orchestration setups using FogBus2 [19] and K3s: Lightweight Kubernetes [20]. Singh et al. [21] use Docker Swarm in a game-theoretic approach, which includes deadlines and security aspects in the orchestration. Yin et al. [22] introduce a mechanism for deciding whether to offload tasks to the cloud or run them on local edge nodes for industrial fog-computing scenarios. Govindaraj et al. introduced in [23] a container migration scheme that reduces the downtime of the containers. Krüger et al. [24] used RT container orchestration to mitigate failures in power-grid systems.

Finally, our work builds upon Struhár et al. [5], who present a hierarchical resource orchestration framework that calculates initial container placements in an *offline-phase* via an SMT-based approach and relocates containers in the *online-phase* based on simple metrics. We have not only introduced a more scalable heuristic-based offline step and extended the online control decisions using time-utility functions (see below) but also redesigned and rebuilt the online phase while keeping a similar hierarchical control structure. The approach in Struhár et al. is entirely implemented in the Kernel, and thus, it cannot be extended easily and is less portable. We kept the changes to the Kernel to a minimum, adding only three hook functions, and implemented the logic in a loadable kernel module. Furthermore, instead of having a separate Container-Level-Controller that adapts the budget of a container, we implemented this functionality in the Node-Level-Controller (NLC), which is running in userspace as opposed to the kernel-level implementation in [5].

III. System Model

We use the notation, correctness conditions, and cost function from [5], [25]. We assume a system with n homogeneous nodes and m containers. Every container $\pi_k, k = 1, \dots, m$ has a set of Boolean variables $v_1^k, \dots, v_n^k$ specifying whether the container is assigned to node n. P_k is the containers period and Q_k its budget. $f_j, j = 1, \dots, n$ represents a node in the system. C_i^j is the worst-case execution time (WCET) of task i on node j, which means the tasks can have different WCETs on different nodes. Each container π_k has a predefined set of tasks denoted with $\mathcal{T}^k$. The cost function initially defined in [25] and adapted in [5] enables a design-time parametrization via c_1 and c_2, trading-off the importance of the context switch overhead σ_j in node f_j against the overhead of the container dimensioning in terms of budget and period:

$$\min_{\{Q_k, P_k, v_1^k, \dots, v_n^k | k=1,\dots,m\}} c_1 \sum_{k=1}^{m} \frac{\sum_{j=1}^{n} v_j^k \sigma_j}{P_k} + c_2 \sum_{k=1}^{m} \frac{Q_k}{P_k} \quad (1)$$

For the offline correctness conditions and the complete formal model, we refer the reader to [5]. The overhead σ_j can usually be measured using instrumentation and is dependent on e.g. number of tasks, macrotick, hardware architecture, etc. [26].

IV. Offline Container Placement and Dimensioning

The offline phase aims to find an initial placement of containers to nodes and a correct dimensioning in terms of budget and period for each container. This problem is similar to the bin-packing problem and is, hence, NP-complete. In [5], the authors found containers' initial placement and dimensioning using an optimal method via SMT solver that does not scale for larger systems. We propose three heuristic approaches to solve the initial placement and dimensioning problem: a greedy algorithm, a metaheuristic based on simulated annealing, and a compositional SMT approach.

A. Simulated Annealing (SA)

Simulated Annealing [27] is a well-known metaheuristic used in search problems that tend to get stuck in local minima. In each step, a new neighbor solution (`neighbor()`) is computed that changes a subset of parameters of the current candidate solution, and the cost function is evaluated. If the result is better, the new solution is saved and otherwise discarded. However, worse solutions are also accepted to overcome local minima with a certain probability. This probability depends on a temperature value that decreases every iteration; hence, the chance of accepting worse solutions decreases. The algorithm can stop after a fixed amount of iterations or after the cost function evaluates to a value below a specified threshold.

The Simulated Annealing-based solver uses the Dual Annealing function from the `scipy` package [28]. Every container has a variable for its node, quota, and period. The `neighbor()` function uses the Cauchy-Lorentz visiting distribution from [28], [29]. Furthermore, we implement two variants for the simulated annealing solver:

Minimum: The search space limit for quota and period of each container is the minimal period of the container's tasks.
Hyperperiod: The search space limit for quota and period of each container is the hyperperiod of the container's tasks.

Hypothetically, we expect that the hyperperiod variant yields better results since all possible periods are considered. However, the resulting search space is larger so that the runtime becomes too long for the variant to deliver feasible results for large systems (c.f. Section VI).

B. Greedy Algorithm (Greedy)

The greedy algorithm reduces the search space, considering only a fraction of the possible container periods and quotas. We acknowledge that this approach does not yield an optimal solution, yet it is designed to be scalable for large systems.

A random container-to-node assignment is generated at the start of each search run. If it is infeasible in terms of the memory requirements, a new assignment is generated until the constraints are satisfied. The algorithm selects a list of periods for the search for each container, iterates over the list, and tries to assign the same period to all containers. An initial quota is guessed by setting it to the utilization of the container. Next, the algorithm attempts to decrease the container utilization as much as possible by subtracting the maximum quota that is known to be insufficient, halving the result, and checking if the tasks inside the container are schedulable with the given quota. If the quota does not allow a feasible scheduling, the quota is doubled. This procedure continues until the candidate quota has already been considered. In the end, the quota is set to the lowest quota for which the tasks in the container are schedulable. If the whole system is schedulable, the algorithm terminates; otherwise, a new search run is started up to a given maximum runs, after which the input is considered infeasible. The upper limit of the quota and period of a container is the hyperperiod of the tasks. The step distance is set to $50\mu s$.

The greedy solver provides two variants as well:

Standard: The algorithm begins its search for solutions at the lower end of the search space.
Inverted: The algorithm starts to search for solutions at the upper end of the search space. This way, the cost function can be minimized further if the context-switching overhead is considered because the resulting periods are larger.

C. Compositional SMT (C-SMT)

We enhanced the SMT-based method presented in [5] with an outer metaheuristic loop that divides the system-level problem into subproblems for each container. First we find the quota and period for every container independently and then we solve the allocation problem with the found container values. This approach reduces the runtime compared to the other presented solvers since fewer constraints are given to the SMT solver, reducing the complexity of each run, as shown in Section VI. Here, we focus on computing allocations for homogeneous systems; however, we note that this approach can be extended for heterogeneous systems by computing the (quota, period) pair for each type of node to account for the individual computation times and scheduling overheads. The resulting list of pairs can then be used in the second step for the allocation. This solver provides a parameter that allows tuning the optimality of the allocation. It specifies for each container the difference (maximum error) between the current cost function value compared to the previously found optimal solution. This can be helpful for high utilization systems where it may be useful to reduce the bandwidth of every container as much as possible. However, finding optimal allocations results in a longer runtime since the SMT solver takes more iterations to arrive at a lower-cost solution.

V. Online Container Orchestration

In [30], task response times are used as a metric to detect a real-time performance degradation that triggers the re-dimensioning or relocation of a container. This so-called temporal error only captures if a deadline has been met or missed, hence assuming hard task deadlines. However, there are processes in real-time systems with firm or soft deadlines that still maintain utility even if individual job deadlines are missed. Using temporal errors for container adaptations renders it impossible to reflect this maintained utility upon a missed deadline or to capture the overall system utility. Therefore, we improve upon [5] by introducing a time-utility (TU) function that assigns each containerized task a semantic value capturing the degree of temporal error.

A. Time-Utility Functions

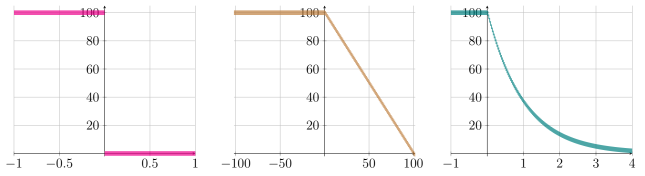
A time-utility function (TUF) assigns a utility to the result of a task depending on its completion time, even after the deadline has expired. The usefulness of the result, therefore, depends on the response time of the task [31]. In the initial phase of TUF-based scheduling, only maximal utility accrual (UA), i.e. the sum of the total utility, concerning the timeliness of the tasks has been considered. However, further parameters, such as energy consumption or predictability, have been added, and various TUF/UA models have been studied in the academic literature, e.g., [32]–[34].

We use TUFs commonly used in real-world applications [32] that yield 100.0 if the task finishes before its deadline; otherwise, they monotonically decrease until reaching a utility of zero. We define the tardiness or temporal error δ_i^j of a task τ_i's job j as the difference between its completion time and its deadline, i.e., a task's temporal error is negative if the task completed before the deadline and zero or positive if it missed its deadline. Therefore, a time utility function returns a task's utility given its temporal error. We consider the following families of functions:

Step We use a step function to model hard deadlines, i.e., $\delta > 0$ as illustrated in Figure 1a:

$$\text{step}(\delta) = \begin{cases} 100, & \text{if } \delta \leq 0 \\ 0, & \text{otherwise} \end{cases}$$

Linear We use a linear function with slope $-m$ to model tasks for which the decline of the time utility is proportional



(a) Step TUF (b) Linear TUF (c) Exponential TUF

Fig. 1: Visualizing the TUFs considered in this work.

to the temporal error δ of a task's job, e.g., tasks with soft deadlines, as illustrated in Figure 1b:

$$\text{linear}(\delta) = \begin{cases} 100, & \text{if } \delta \leq 0 \\ \max\{-m \cdot \delta + 100, 0\}, & \text{otherwise} \end{cases}$$

Exponential We use an exponential function with exponential decay factor $-m$ to model tasks for which the time utility degrades more rapidly with the temporal error δ of a task's job, as illustrated in Figure 1c:

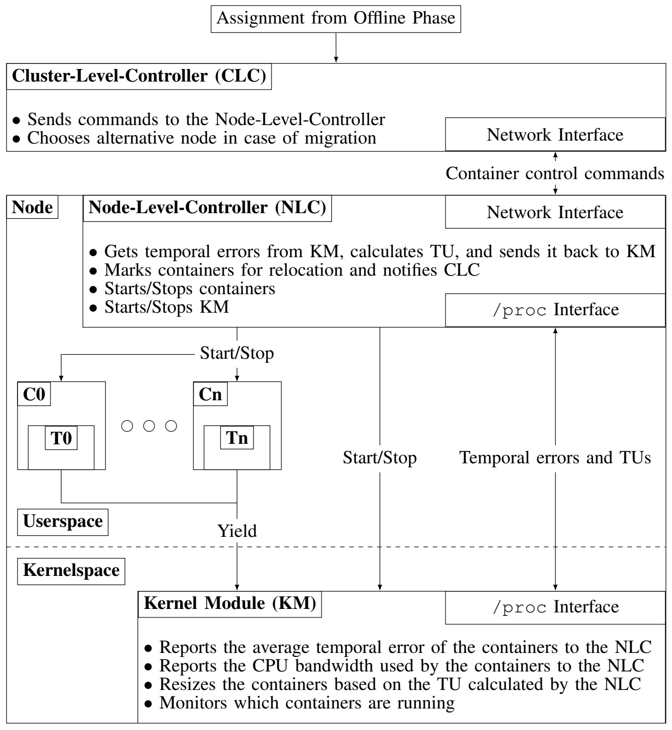
$$\exp(\delta) = \begin{cases} 100, & \text{if } \delta \leq 0 \\ \exp\left(-m \cdot \delta\right) * 100, & \text{otherwise} \end{cases}$$

The choice of the parameter m for each task determines the function's shape by spanning the interval between the deadline and the instant when the TUF degrades to zero, also referred to as the expiration time [31]. Note that both the function type and the parameter m depend significantly on the specific application. For instance, in Maynard et al. [35], the authors illustrate how they derived application-specific time-utility functions for an air-defense application while developing a command, control, and battle management system.

B. Implementation

We have redesigned and implemented the hierarchical orchestrator framework utilizing TUFs from scratch in GNU/Linux. However we retain a similar controller structure as proposed in [5]. Figure 2 shows the three main components, the Cluster-Level-Controller (CLC), the Node-Level-Controller (NLC), and the Linux kernel module (KM), containing the logic of the online orchestration phase.

1) Cluster-Level-Controller (CLC): We have implemented the communication between the CLC and the NLC via bidirectional HTTP requests. The CLC configures registered NLCs and can issue to start, to stop, and to relocate a container on registered nodes by communicating with the respective NLCs. Initially, the CLC starts containers by sending start commands to the NLCs according to an allocation found in the offline phase. In the online phase, or during runtime, we have implemented a simple relocation policy based on the current CPU utilization in the cluster. When a NLC demands relocation of a container due to a diminished TU, the CLC requests the current CPU utilization from NLCs running on registered nodes. The CLC decides to relocate the container with degraded performance to the node with the lowest CPU utilization and stops the container on the node it was previously running on. If there is no node with lower CPU utilization than the node the container is currently assigned to, the CLC aborts the relocation.



Assignment from Offline Phase
Cluster-Level-Controller (CLC)
• Sends commands to the Node-Level-Controller
• Chooses alternative node in case of migration
Network Interface
Container control commands
Node
Node-Level-Controller (NLC)
Network Interface
• Gets temporal errors from KM, calculates TU, and sends it back to KM
• Marks containers for relocation and notifies CLC
• Starts/Stops containers
• Starts/Stops KM
/proc Interface
Start/Stop
C0
T0
Cn
Tn
Start/Stop
Temporal errors and TUs
Userspace
Yield
Kernelspace
Kernel Module (KM)
/proc Interface
• Reports the average temporal error of the containers to the NLC
• Reports the CPU bandwidth used by the containers to the NLC
• Resizes the containers based on the TU calculated by the NLC
• Monitors which containers are running

Fig. 2: Architecture Diagram Online Phase

2) Node-Level-Controller (NLC): After start up the NLC awaits to receive its configuration from the CLC containing, e.g., KM parameters, the relocation interval, or the update interval between the NLC and the KM. The relocation interval determines a NLC's periodic checks if a container needs relocation, i.e., if a container's time-utility (TU) dropped below a threshold. We can pick the behavior of the relocation interval from three available options:

Fixed: A fixed value specified in seconds. Experimental evaluation indicated that a value of five seconds yields good results that prevents oscillating relocation.
Minimum: Equal to the smallest task period on the node.
Maximum: Equal to the greatest task period on the node.

If the NLC detects multiple containers for relocation in the same period, we introduce a *lowest utility first* policy where the container with the lowest utility is chosen for relocation.

Just as the relocation interval, the update interval determines the frequency with which the NLC communicates with the kernel module. The update interval depends on the task periods. There are three options to derive the update interval:

Minimum: Equal to the smallest task period on the node.
Maximum: Equal the greatest task period on the node.
Medium: Equal to the difference between the greatest and smallest period divided by 2.

If not active yet, the NLC loads the KM and starts the update thread that periodically communicates with the KM using the configured update interval. The communication via the `/proc` filesystem consists of two steps. First, the NLC reads each task's average temporal error, the container bandwidth and quota used during the last period, and if the KM detected any container that should be relocated. Then, the NLC uses the average temporal errors to calculate the time-utilities (TUs) of all tasks, sorts the TUs in ascending order, and writes them to the KM. As a result, in the KM the tasks with the lowest TU get a quota increase first, and tasks with a higher TU might not receive an increased quota to improve their TU if the CPU has no more capacity (the minimal quota `min_quota` is $100\mu s$).

3) Kernel Module (KM): The function of the KM is twofold. Firstly, the KM keeps track of hosted containers and monitors each container's tasks' temporal error. Secondly, the KM adapts each container π_k's CPU quota Q_k when receiving tasks' updated TUs from the NLC via the `/proc` filesystem. When calculating a new quota Q_k^* for containers $\pi_k, k = 1, \ldots, m$, we differentiate two cases: (1) If a task's TU received from the NLC is 100, i.e., the task's jobs completed earlier than their deadlines, the KM reduces the hosting container's quota, or (2) if the TU falls below 100 we increase a container's quota.

For case (1), a TU of 100 indicates that we can decrease its hosting container's quota so that the new quota is:

$$Q_k^* = \max\left(Q_k - \frac{g_r \cdot \Delta_k}{100}, \text{min\_quota}\right)$$

Here, g_r is a configurable parameter that specifies how aggressively the KM reduces a container's quota (set to 0.05 in our experiments). Furthermore, a container's average temporal error in an arbitrary interval $[t_{l-1}, t_l]$ since the $(l-1)$-th quota update is determined by the sum of its tasks $\tau_i \in \mathcal{T}^k$ individual average temporal errors Δ_i^k. Each task τ_i spawns $|\mathcal{J}| = \lfloor \frac{t_l - t_{l-1}}{T_i} \rfloor$ jobs during that interval, each associated with a temporal error $\delta_i^j, j \in \mathcal{J}$. Therefore, a task's average temporal error is given by $\Delta_i^k = \frac{1}{|\mathcal{J}|} \sum_{j \in \mathcal{J}} \delta_i^j$, i.e., the sum of the temporal errors δ_i^j of the task's jobs divided by the number of jobs $|\mathcal{J}|$ in the interval $[t_{l-1}, t_l]$. Lastly, we scale each container's tasks's average temporal errors to their container's replenishment period P_k:

$$\Delta_k = \frac{1}{|\mathcal{T}^k|} \sum_{\tau_i \in \mathcal{T}^k} \frac{\Delta_i^k}{\max(\frac{T_i}{P_k}, 1)}$$

In the (2) case, if a task's TU lies within $[0, 100)$, its container's quota is increased to reduce its temporal error to:

$$Q_k^* = Q_c + \frac{g_i \cdot \Delta_k \cdot (100 - \frac{1}{|\mathcal{T}^k|} \sum_{\tau_i \in \mathcal{T}^k} TU_i)}{100^2}$$

As for the quota decrease, the parameter g_i is configurable and specifies the slope of the quota increase. Note that the formula for quota increase uses the container's average temporal error to scale the quota increase given by the inverse of the container's tasks average TU. As a result, if several tasks have the same absolute TU, the quota of those hosting containers with a high average temporal error is increased more.

C. Modifications to the Linux Kernel

We have kept the modifications to the Linux kernel to a minimum yet it was necessary to add three hooks into the Linux kernel patched with the Hierarchical Constant Bandwidth Server (HCBS) [11]. We did not use the PREEMPT_RT patch [14] since it is incompatible with HCBS [11]. When our KM is loaded, we inject the three hooks calling the KM at the following locations in the kernel via function pointers:

Yield `sched_yield` System Call Hooks A containerized real-time task invokes the yield system call to indicate job completion. We add a hook to the `yield` system call of the Linux scheduler subsystem taking a timestamp that indicates the instant of job completion. By placing the hook in the system call definition we ensure accurate temporal errors. We insert an additional hook in the fixed-priority (FP) scheduler specific `sched_yield` call in which we use the previously created timestamp to calculate a jobs temporal error.

Free Real-Time Group Hook When a container stops execution there is a call to the `free_rt_sched_group` of the FP scheduler. We add a hook to the function to ensure freeing of KM resources used internally for tracing timing metadata.

VI. Experiments

We performed separate experiments to evaluate the offline phase and the online phase.

A. Offline Container Dimensioning

We evaluate our proposed methods from Section IV that have been implemented in `python3` in terms of runtime and solution quality, comparing them to the SMT-based approach from [5]. All offline dimensioning experiments were run on a *Dell Latitude 5420* with an Intel(R) 4-Core(TM) i7-1185G7 @ 3.00GHz CPU, 32 GB RAM, and Windows 10. For the following results, we set the target system utilization to 50%. For each system size, we aggregated the results of 100 runs to retrieve the average runtime. In the case of simulated annealing and the greedy solvers, we compute ten different allocations for each of the 100 systems to account for randomness in the algorithms. For each test case, we used four tasks per container. Figure 3 shows the average runtime with error bars (y-axis) of all solvers for different system sizes (x-axis). For the basic SMT solution [5], no cost function is taken into account since adding the cost function to find an optimized solution would increase its runtime additionally. Furthermore, we set a timeout of 300 seconds for the basic SMT approach depicted as a red line. For the Simulated Annealing (SA) heuristic, we selected the variant that uses the minimum task period occurring in a container to limit the search space since we could not solve even small systems in under ten minutes when considering the hyperperiod of a container's tasks as the search space limit for quota and container period. In the *benchmark_50* set for the smallest systems with size *1-3-12*, the search space is approximately $1.75714 * 10^{11}$ times bigger when using the hyperperiod as a limit instead of the minimum task period. For the greedy solver, the non-inverted version was chosen, and for C-SMT, the maximum error was set to 0.1 to get a feasible solution quickly without spending too much time on optimization.

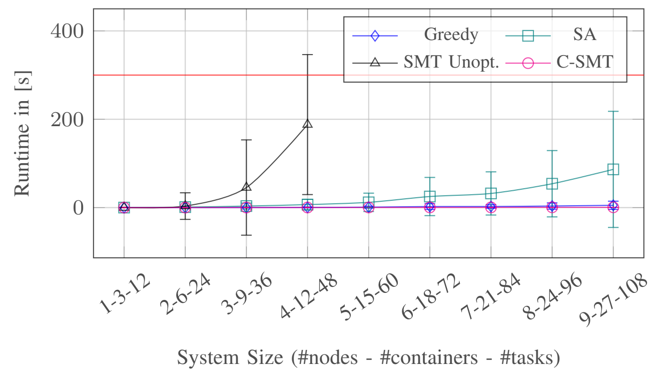
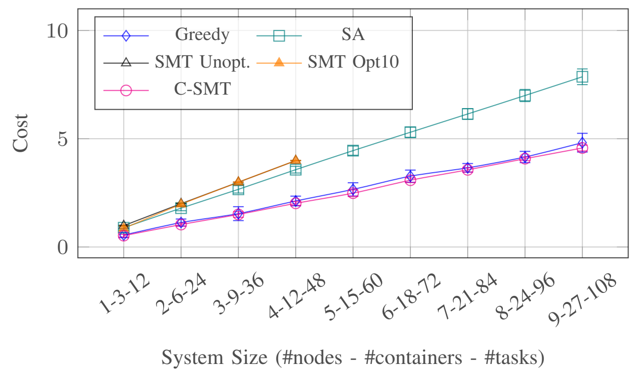


Fig. 3: Solver Runtimes Comparison



(a) Without Overhead

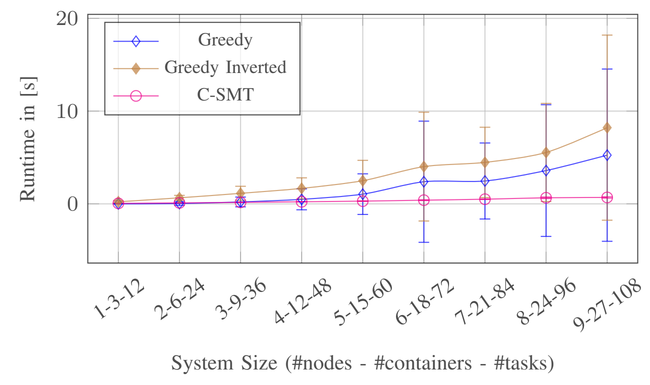
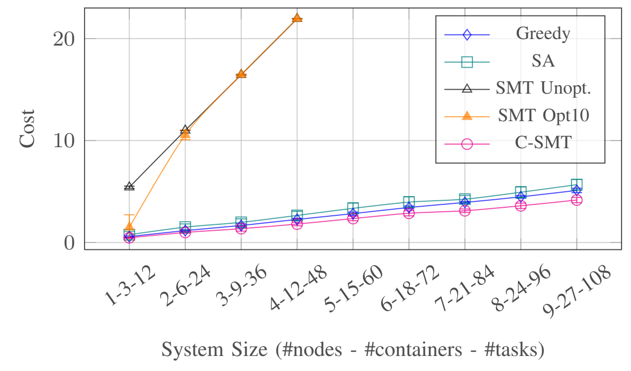


Fig. 4: Greedy and Compositional SMT Runtimes Comparison



(b) With Overhead

Fig. 5: Solver Qualities with and without overhead

The basic SMT method [5] exhibited the worst runtime. For all system sizes larger than *4-12-48*, the basic SMT approach [5] times out at 300 seconds, and the data points are not shown in the plot. We can see that the runtime of the SA solver performs better, yet an exponential trend indicating bad scalability with growing system size is visible. We trace this back to the neighbor function that is part of the `scipy` package [28] and does not fit the problem well. Finding better suited neighbor functions is out of the scope of this paper. Lastly, the greedy solver and the C-SMT approach exhibit a stable and low runtime for increasing system sizes. In Figure 4, we zoom in onto both approaches and compare the normal and the inverted greedy solvers to C-SMT. The C-SMT approach scales better showing only a small increase in runtime for growing system sizes compared to the greedy approaches that do show an increased runtime for bigger systems on this scale.

Only the runtime is not sufficient to evaluate the performance of a solver. In addition, we evaluate the quality of the found solutions utilizing the cost function introduced in Equation (1) as a metric. Figure 5 compares the cost in an optimal scenario without scheduling overhead σ_j $(c_1 = 0, c_2 = 1.0)$ with a scenario that weighs scheduling overhead and the minimal container budget and period equally $(c_1 = c_2 = 0.5)$.

For the basic SMT solver we compare two implementations. The unoptimized SMT solver (SMT Unopt.) does not consider the cost function in its constraints. In the second implementation (SMT Opt10), after finding and evaluating the cost of a solution, we set the found cost as a new minimum constraint for the solution quality and start the next iteration. We repeat the process for ten iterations.

In the first scenario illustrated in Figure 5a, not considering scheduling overhead, we observe similar cost of found solutions across solvers that linearly increase with system size. However, a trend is visible indicating an increasing gap between the cost of solutions found by the Greedy and C-SMT solvers compared to the basic SMT and SA solvers.

Figure 5b visualizes the second scenario, we can see that both basic SMT approaches perform badly finding solutions with high cost. We explain this with the basic SMT solvers selecting solutions with small period and quota first resulting in high CPU utilization and cost. An incremental approach does not improve this behavior if the number of iterations remains low and if we allow more iterations the long runtime renders this approach unfeasible as indicated by both basic SMT approaches not finding solutions within five minutes for system sizes beyond (*4-12-48*). The Greedy, SA, and C-SMT solvers all find comparable low-cost solutions for all tested system sizes when weighing scheduling overhead and the minimal container budget and period equally. We find that SA and Greedy find solutions with longer periods resulting in lower cost compared to the short period solutions found by the basic SMT approaches. To analyze the quality of the Compositional SMT solver (C-SMT), we set the maximum error of the cost function to 0.000001. This led to the best

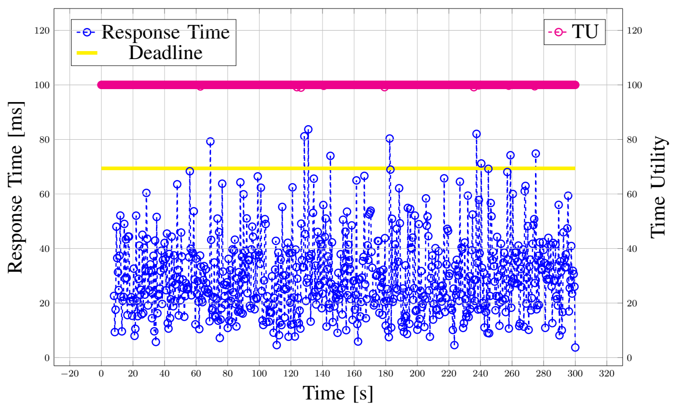


Fig. 6: Scenario 1 - Task 9: Maximum Response Time per half-second and Average TU per half-second over Time

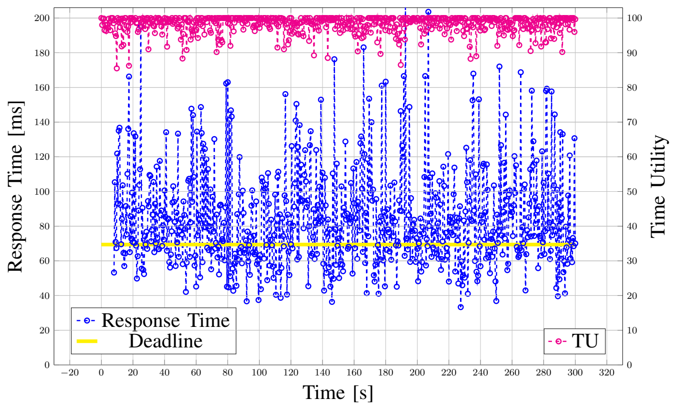


Fig. 7: Scenario 2 - Task 9: Maximum Response Time per half-second and Average TU per half-second over Time

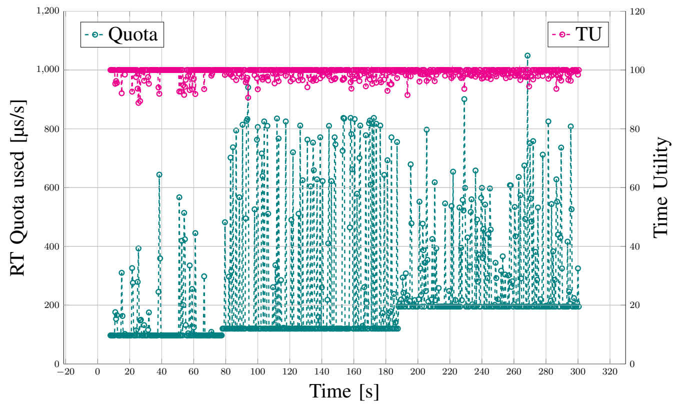


Fig. 8: Scenario 3 - Node 2: Maximum Quota per half-second and Average TU per half-second over Time

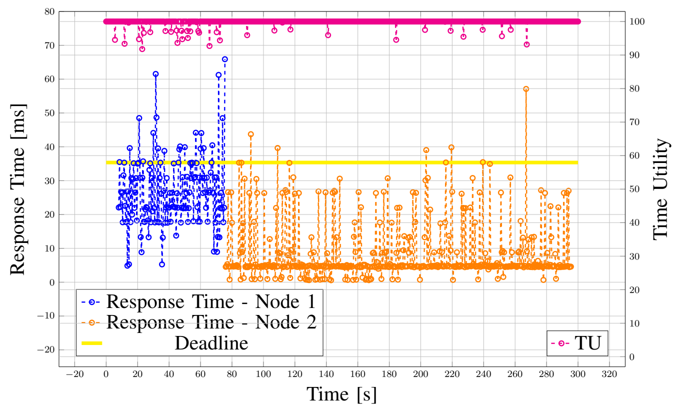


Fig. 9: Scenario 3 - Task 5: Maximum Response Time per half-second and Average TU per half-second over Time

results compared to all other solvers but at the cost of additional runtime. When the scheduling overhead was not considered, the runtime increased on average by a factor of seven compared to our runtime experiments.

We can see from the experiments that our three proposed heuristics scale well with increasing system size compared to the SMT-based method presented in [5]. Furthermore, we show that the C-SMT solver exhibits a trade-off between the quality of the solution and the runtime scaling well with increased system size.

B. Online Container Orchestration

We executed the CLC in an Ubuntu 22.04.3 LTS VMWare Workstation 16.2.5 build-20904516 virtual machine on the same *Dell Latitude* laptop that we used for the previous experiments. For the cluster we used two Nerve MFN100 edge computing devices [36] also running Ubuntu 22.04.3 LTS. Node 1 comes with an *Intel(R) Atom(TM) Processor E3940 @ 1.60GHz* and 4 GiB of RAM and node 2 with an *Intel(R) Atom(TM) Processor E3950 @ 1.60GHz* and 8 GiB of RAM. We connected node 1, node 2, and the VM executing the CLC via the built-in Ethernet switch of node 1. For the online experiments, we used one task per container, where the task runs within a Docker container simulating a periodic workload by iterating through a loop. The number of iterations correlates to the desired WCET. We derive the actual execution time from the upper bound used in the offline phase (`offline_bound`) using an exponential tail Gumbel distribution [37], [38]. In practice, it is often too complex or costly to calculate the actual worst-case execution time (WCET), and a measurement-based approach is used where the application is profiled with many different inputs [39]. The resulting maximal observed execution time (MOET) is only correct for the tested inputs and may underestimate the WCET [39]. We have added a parameter called `WCET_scaler`, which specifies the relation between the given WCET and the MOET. If the parameter is, e.g., 0.8, the task may need up to 20% longer than the given upper bound to execute. Our parameter, hence, specifies how well each task was profiled: $\text{true\_WCET} = \frac{\text{offline\_bound}}{\text{WCET\_scaler}}$.

We evaluated the online phase with the following scenarios:

1) 10 containers, `WCET_scaler` $= 0.6$, System utilization is at 70%, linear TU
2) 10 containers, `WCET_scaler` $= 0.261856$, System utilization is at 70%, linear TU
3) 7 containers, `WCET_scaler` $= 0.261856$, Utilization is at 50%, step TU, node 1 at 100%, node 2 at 0%.

The value 0.261856 for the `WCET_scaler` is the expected value of the Gumbel distribution. These scenarios represent those cases in which tasks were poorly profiled.

Figure 6 shows the graph of the response time of a selected task and its TU over time. For this scenario, the interval of the TU evaluation in the NLC was set to *medium*. For node 1, on which the task was running, this was about 60 ms. We aggregated the data points for each half-second interval to keep the plots meaningful. For the response times, we plot the maximum value of the interval; for the TU, we plot the average value. It is also possible that the response time graph indicates missed deadlines, but the TU is still 100% because the TU is not calculated after each task period. If response times around a missed deadline are short enough, they can compensate for the miss, and the TU is still maximal.

A further reduction of the `WCET-scaler` led to constant deadline misses and a significantly reduced TU. Figure 7 shows the response time and TU graph of the same task. However, the CLC could not relocate any container.

To demonstrate the relocation feature of the online phase, we started containers only on node 1. Because of the low `WCET_scaler` value, a lot of tasks missed their deadlines. Container C6 was relocated immediately after startup. Container C5 was relocated after about 70 seconds (Figure 9) and C3 after about 180 seconds. Figure 8 contains a graph of the quota used on node 2. The quota jumps indicate that a container was relocated to the node.

VII. Conclusion

We have extended and improved existing work on hierarchical-based container orchestration for real-time heterogeneous FEC systems, introducing a highly configurable orchestration framework for real-time containers. We have proposed three different heuristics for the offline placement and dimensioning of containers that overcome the scalability issues of SMT- or ILP-based approaches. Furthermore, we have added time-utility functions to the decision-making of the online hierarchical controller, enabling more complex system models for the utility of results when deadline misses occur. We have demonstrated the scalability and performance of our framework using synthetic test cases and a real-world test bed.

References

[1] S. M. Salman, V. Struhár *et al.*, "Fogification of industrial robotic systems: Research challenges," in *Proc. Fog-IoT*, 2019.

[2] M. S. Shaik, V. Struhár *et al.*, "Enabling fog-based industrial robotics systems," in *Proc. ETFA*, 2020.

[3] R. Morabito, I. Farris *et al.*, "Evaluating performance of containerized iot services for clustered devices at the network edge," *IEEE Internet Things J.*, vol. 4, no. 4, 2017.

[4] B. Xavier, T. Ferreto *et al.*, "Time provisioning evaluation of kvm, docker and unikernels in a cloud platform," in *Proc. CCGrid*, 2016.

[5] V. Struhár, S. S. Craciunas *et al.*, "Hierarchical resource orchestration framework for real-time containers," *ACM TECS*, 2023.

[6] S. Vaucher, R. Pires *et al.*, "SGX-aware container orchestration for heterogeneous clusters," in *Proc. ICDCS*, 2018.

[7] A. Tosatto, P. Ruiu *et al.*, "Container-based orchestration in cloud: state of the art and challenges," in *Proc. CISIS*, 2015.

[8] T. Gkamas, V. Karaiskos *et al.*, "Performance evaluation of distributed database strategies using docker as a service for industrial iot data: Application to industry 4.0," *Information*, vol. 13, no. 4, 2022.

[9] H. Lasi, P. Fettke *et al.*, "Industry 4.0," *Business & information systems engineering*, vol. 6, no. 4, 2014.

[10] S. M. Salman, V. Struhár *et al.*, "Fogification of industrial robotic systems: Research challenges," in *Proc. Fog-IoT*, 2019.

[11] L. Abeni, A. Balsini *et al.*, "Container-based real-time scheduling in the linux kernel," *ACM SIGBED Review*, vol. 16, no. 3, 2019.

[12] R. S-Julián, I. Lacalle *et al.*, "Self-* capabilities of cloud-edge nodes: A research review," *Sensors*, vol. 23, no. 6, 2023.

[13] V. Struhár, S. S. Craciunas *et al.*, "React: Enabling real-time container orchestration," in *Proc. ETFA*, 2021.

[14] "A realtime preemption overview," https://lwn.net/Articles/146861/, accessed: 24.03.2023.

[15] T. Tasci, J. Melcher *et al.*, "A container-based architecture for real-time control applications," in *Proc. ICE/ITMC*. IEEE, 2018.

[16] R. Queiroz, T. Cruz *et al.*, "Container-based virtualization for real-time industrial systems—a systematic review," *ACM Comput. Surv.*, vol. 56, no. 3, 2023.

[17] S. Fiori, L. Abeni *et al.*, "RT-kubernetes: containerized real-time cloud computing," in *Proc. SAC*, 2022.

[18] Z. Wang, M. Goudarzi *et al.*, "Container orchestration in edge and fog computing environments for real-time IoT applications," in *Proc. ICCIDA*, 2022.

[19] Q. Deng, M. Goudarzi *et al.*, "Fogbus2: a lightweight and distributed container-based framework for integration of iot-enabled systems with edge and cloud computing," in *Proc. BiDEDE*, 2021.

[20] R. Labs, "K3s - lightweight kubernetes: K3s," https://rancher.com/docs/k3s/latest/en/, Dec 2023, accessed: 07.12.2023.

[21] C. Singh, P. Kumari *et al.*, "Secure industrial iot task containerization with deadline constraint: A stackelberg game approach," *IEEE Transactions on Industrial Informatics*, vol. 18, no. 12, pp. 8674–8681, 2022.

[22] L. Yin, J. Luo *et al.*, "Tasks scheduling and resource allocation in fog computing based on containers for smart manufacturing," *IEEE Transactions on Industrial Informatics*, vol. 14, no. 10, 2018.

[23] K. Govindaraj and A. Artemenko, "Container live migration for latency critical industrial applications on edge computing," in *Proc. ETFA*, 2018.

[24] C. Krüger, A. Narayan *et al.*, "Real-time test platform for enabling grid service virtualisation in cyber physical energy system," in *Proc. ETFA*, 2020.

[25] G. Lipari and E. Bini, "Resource partitioning among real-time applications," in *Proc. ECRTS*. IEEE, 2003.

[26] S. Craciunas, R. Serna Oliver *et al.*, "Optimal static scheduling of real-time tasks on distributed time-triggered networked systems," in *Proc. ETFA*, 2014.

[27] S. Kirkpatrick, C. D. Gelatt Jr *et al.*, "Optimization by simulated annealing," *Science*, vol. 220, no. 4598, 1983.

[28] "Dual annealing documentation," https://docs.scipy.org/doc/scipy/reference/generated/scipy.optimize.dual_annealing.html, accessed: 20.06.2023.

[29] Y. Xiang, D. Sun *et al.*, "Generalized simulated annealing algorithm and its application to the thomson model," *Phys. Lett. A*, vol. 233, 1997.

[30] V. Struhár, M. Behnam *et al.*, "Real-time containers: A survey," in *Proc. Fog-IoT*, 2020.

[31] E. D. Jensen, "A timeliness model for asychronous decentralized computer systems," in *Proc. ISAD*. IEEE, 1993.

[32] B. Ravindran, E. D. Jensen *et al.*, "On recent advances in time/utility function real-time scheduling and resource management," in *Proc. ISORC*, 2005.

[33] S. A. Aldarmi and A. Burns, "Dynamic value-density for scheduling real-time systems," in *Proc. ECRTS*. IEEE, 1999.

[34] A. Burns, D. Prasad *et al.*, "The meaning and role of value in scheduling flexible real-time systems," *J. Syst. Archit.*, vol. 46, no. 4, 2000.

[35] D. P. Maynard, S. E. Shipman *et al.*, "An example real-time command, control, and battle management application for alpha," *Archons Project TR-88121, CMU*, 1988.

[36] TTTech Computertechnik AG, "Nerve," https://www.tttech-industrial.com/nerve, accessed: 07.11.2023.

[37] K. P. Silva, L. F. Arcaro *et al.*, "On using gev or gumbel models when applying evt for probabilistic wcet estimation," in *Proc. RTSS*, 2017.

[38] A. Burns and S. Edgar, "Predicting computation time for advanced processor architectures," in *Proc. ECRTS*. IEEE, 2000.

[39] R. Wilhelm, J. Engblom *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM Trans. Embed. Comput. Syst.*, vol. 7, no. 3, 2008.