2024 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)
October 14-18, 2024. Abu Dhabi, UAE

Optimizing Kubernetes Deployment of Robotic Applications with HEFT-based Container Orchestration

Francesco Lumpp*, Franco Fummi and Nicola Bombieri
Department of Innovation Medicine - University of Verona - Italy
name.surname@univr.it

Abstract— **This study addresses the challenge of deploying robotic software with Quality of Service (QoS) constraints in Edge-Cloud computing clusters. The paper introduces HEFT4K, an event-driven scheduling method tailored for Kubernetes-managed systems based on the Heterogeneous Early Finish Time (HEFT) algorithm. This algorithm reduces software execution time (makespan) and facilitates re-mapping in case of node failures, involving only essential containers to maintain uninterrupted robot functionality. Experimental results, conducted on a real-world robot and synthetic benchmarks, show a 75% speedup in makespan compared to the standard Kubernetes scheduler, enhancing the efficiency of QoS-focused scheduling for robotic applications in distributed systems.**

I. Introduction

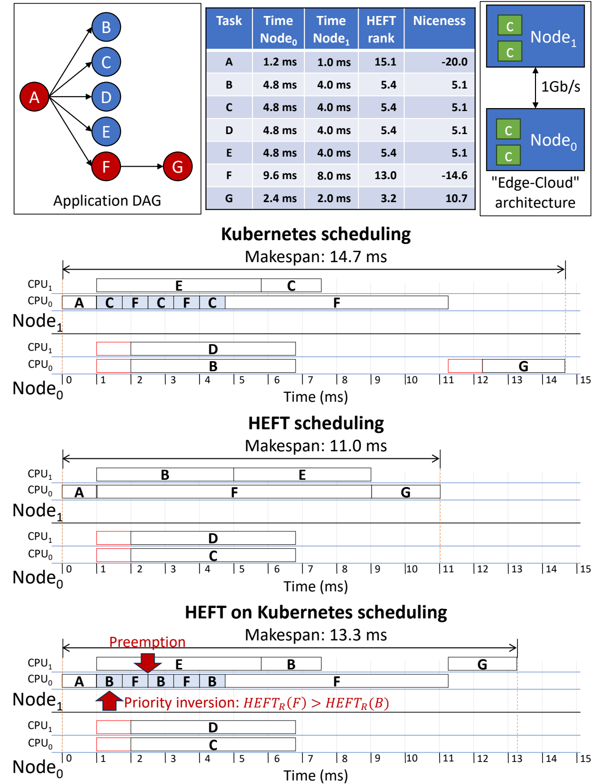
Rigid accuracy standards for robotic software are mandatory due to the frequent use of robots in safety-critical tasks. Apart from functional constraints, these standards encompass extra-functional constraints like quality of service (QoS), reliability, scalability, and energy efficiency [1]. Fulfilling these constraints necessitates the set up of robotic software to function across *heterogeneous* computing architectures, which entails the allocation of software components across Edge-Cloud computing clusters [2]. The Edge-Cloud computing paradigm combines the responsivness of edge-computing to the computing power of the cloud (or local cloud). This is achieved through a complex software and hardware architecture that allows the distribution of software across these two different layers, taking advantage of both worlds [3].

In this scenario, *containerization* has emerged as an essential necessity. It enhances resource utilization, facilitates platform-independent development, and ensures secure software deployment [4]. However, standard software containerization falls short due to the ever-increasing complexity of software designed for autonomous and intelligent robots. The partitioning of services and tasks into distinct containers becomes crucial to address the growing size of container images, enhancing the adaptability of container mapping to cluster nodes and bolstering the system's resilience against node failures [5].

Within this *multi-container* context, a significant challenge involves maintaining uninterrupted robot functionality despite disruptions. Consequently, there is a growing interest within robotics companies in embracing *orchestration* platforms for software deployment [6], [7]. This raises a new big challenge, as state-of-the-art orchestrators such as Kubernetes rely on "scheduling" policies designed to enhance system-level efficiency (e.g., load balancing, optimizing data transfer, and overall system energy efficiency) to allocate containers across the Edge-Cloud computing nodes (referred to as "nodes" hereafter). Due to their inherent focus on automating software deployment, these orchestrators do not support global synchronization of containers, which would require significant source code modifications to implement. As a result,

*corresponding author.

This study was carried out within the MICS (Made in Italy – Circular and Sustainable) Extended Partnership and received funding from Next-Generation EU (Italian PNRR – M4 C2, Invest 1.3 – D.D. 1551.11-10-2022, PE00000004). CUP MICS D43C22003120001 - Cascade funding project CollaborICE.



Task	Time $Node_0$	Time $Node_1$	HEFT rank	Niceness
A	1.2 ms	1.0 ms	15.1	-20.0
B	4.8 ms	4.0 ms	5.4	5.1
C	4.8 ms	4.0 ms	5.4	5.1
D	4.8 ms	4.0 ms	5.4	5.1
E	4.8 ms	4.0 ms	5.4	5.1
F	9.6 ms	8.0 ms	13.0	-14.6
G	2.4 ms	2.0 ms	3.2	10.7

Fig. 1: Application example represented by a DAG and the corresponding makespan achieved with native Kubernetes, (ideal) HEFT, and HEFT on Kubernetes schedulers. Red boxes represent data transfer, blue boxes represent preemption.

this limitation, rooted in the emphasis on system-level efficiency, hinders the implementation of scheduling algorithms aimed at QoS constraints, such as minimizing the application makespan.

Fig. 1 shows, for example, the outcome of the Kubernetes scheduling applied to an application represented as a Directed Acyclic Graph (DAG). This application runs on a heterogeneous (distributed) cluster of two nodes, each equipped with two CPUs. Each graph node represents a containerized task, while the graph edges represent the temporal dependencies between tasks. In the optimal scenario, task communication relies on a standard API such as the Robotic Operating System (ROS) or any other *publish-subscribe* paradigm. This approach ensures that temporal task-to-task dependencies are implicitly met. However, the lack of global synchronization leads to priority inversions between containers in the critical path (A, F, G) and other tasks. In the example, task B is executed before task F, even though task F has a higher HEFT rank. This is caused by the OS's lack of knowledge of the HEFT ranking of tasks (i.e., the tasks' priorities), leading to a priority inversion. Furthermore, in such distributed architectures, factors like preemption mechanisms (shown in blue time slots in Fig. 1), which

often cannot be disabled, and data transfers (red slots), impact the application makespan.

We propose a scheduling approach tailored for distributed systems under Kubernetes control to overcome this limitation. Our objective is to reduce the makespan without centralized or distributed synchronization of containers. Our starting point is the Heterogeneous Early Finish Time (HEFT) algorithm [8], which is one of the most efficient mapping and synchronization algorithms in the current state of the art for makespan optimization. HEFT typically demands centralized synchronization of tasks (containers), making it incompatible with Kubernetes. We take advantage only of the definition of the priority ranking of tasks and the mapping topology, while we discard synchronization information like the starting time of tasks. Fig. 1 (*HEFT on Kubernetes scheduling*) shows the outcome of implementing this approach in Kubernetes in the example and provides a comparison of the resulting makespan (13.3 ms) with the theoretically ideal scheduling according to HEFT (*HEFT scheduling*, 11 ms). While the ranking-based mapping method improves upon the initial Kubernetes scheduling (14.7 ms), it struggles with the issues of priority inversion and preemption during container execution as it has no concept of HEFT ranking, priority or niceness.

To tackle this limitation, we propose a solution to adjust container priorities through their *niceness* during the deployment phase. The niceness parameter in the Linux scheduler influences how often a process is executed. Additionally, our approach relies on event-driven re-scheduling to align with the orchestrator's automated deployment, scaling, and recovery capabilities. It starts with an initial offline mapping, and subsequent re-mappings are triggered as needed, such as in the event of a node failure. The re-mappings target the minimal subset of containers to ensure uninterrupted robot functionality to the greatest extent possible.

To summarize, the main novel contributions of this work are the following:

- The adaptation and the corresponding analysis of the most efficient QoS-oriented scheduling approach (HEFT) at the state of the art in Kubernetes.
- A new scheduling algorithm for Kubernetes called HEFT4K that, starting from the HEFT task ranking, takes advantage of the OS niceness concept to reduce priority inversions and preemptions of tasks.
- An event-driven remapping strategy that supports Kubernetes' scaling and recovery capabilities while minimizing the interruption of robot functionality.

We present experimental results obtained on both synthetic benchmarks and a real-world case study of a Robotnik Kairos mobile robot's mission. Our observations indicate that on average, HEFT4K achieves a 75% speedup in makespan compared to the standard Kubernetes scheduler and a loss of ≈2% w.r.t. an ideal HEFT scheduling as ground truth.

II. Related Work

Different works have been proposed to improve Kubernetes scheduling, aiming at more efficient scaling [9], [10] and topological distribution of tasks [11]. Other works proposed approaches to better distribute tasks into a cluster with the aim of reducing the service downtime after node failure [12]. Others proposed a Kubernetes-based monitoring tool for edge devices [13]. The scheduler maps containers based on the collected data, which includes the device physical status such as CPU temperature, to avoid node throttling or crashing.

In [14], the authors proposed a framework to trade between system performance and energy consumption by using a stateless migration policy for Kubernetes based on pre-made energy and performance models.

Work	No code modif.	On demand remap.	Edge-Cloud	Makespan oriented	Task sched.	System resource optim.
[9]	✗	✓	✗	✗	✗	✗
[11]	✓	✓	✓	✗	✗	✗
[10]	✗	✓	✗	✗	✗	✓
[12], [13]	✗	✓	✗	✗	✗	✗
[14]	✓	✓	✗	✗	✗	✗
[15]	✓	✗	✗	✓	✗	✓
[16]	✗	✗	✗	✗	✗	✓
[17]	✓	✗	✓	✓	✗	✓
[18]	✗	✗	✓	✓	✓*	✗
[19]	✗	✓	✓	✗	✗	✓
[20]	✗	✗	✓	✗	✗	✓
HEFT4K	✓	✓	✓	✓	✓**	✓

TABLE I: Summary of the scheduling features proposed in the state of the art. *Optimizes task scheduling, but requires global synchronization. **Optimizes task scheduling, but does not control it directly.

In all of these studies, the primary objective centers around system-level optimization. For the application-level optimization (i.e., makespan reduction), the authors in [15] proposed a framework to analyze data from prior application executions to facilitate efficient scheduling. They employ a secondary scheduler for Kubernetes in conjunction with a database to implement the Earliest Finish Time (EFT) mapping algorithm.

In [16], the authors proposed the use of a particle swarm optimization based algorithm to improve the Kubernetes scheduler for container-to-node allocation in big data applications. In [17], the authors proposed a framework that schedules and monitors repetitive, short-lived, tasks, such as deep learning training or batch operations. They use progress as a metric to reduce the makespan. The application progress is obtained by modifying the application source code and sending its progress to an external monitoring framework.

For containerized Edge-Cloud environments oriented at microservices and Function-as-a-Service, the authors of [18] proposed an algorithm that computes the data flow between edge nodes to obtain the lowest possible makespan. However, the work has several limitations, including complex modifications to the source code to implement global synchronization (like [8]), high overhead, and low scalability. Additionally, it does not support on-demand remapping, which is essential for Kubernetes-based scheduling.

Effort from industry-leading companies has also been made with the Telemetry Aware Scheduler (TAS) from Intel [19] and Trimaran from IBM [20]. TAS can use many node characteristics to decide where to deploy a container. Trimaran monitors standard Kubernetes metrics providers (i.e., Kubernetes Metrics Server, Prometheus Server, and SignalFx), and extends the scoring phase of the Kubernetes scheduler, without interfering with the other native plugins, accounting for both average load and load spikes.

Table I summarizes the scheduling features of the related work. They include necessity of source code modification, implementation of event-driven (on-demand) task remapping, support for Edge-Cloud distributed architectures, orientation to makespan reduction, global synchronization of tasks and implementation of system resource optimization.

In general, most of the work in the literature addresses the problem of scheduling DAG-based applications for virtual machine-

based cloud systems. Other works rely on the use of expensive exact algorithms that do not allow for online scheduling or orchestration capabilities due to their high execution time. Others, such as [8] and [18], are based on explicit global synchronization, which requires invasive kernel and user application modifications and can cause sensible overhead. In contrast, our work tackles the problem of task scheduling by optimizing Kubernetes scheduling decisions through HEFT, as well as by including a priority-based system to avoid priority inversions and mitigate preemption. It also implements a smart on-demand re-scheduling of containers to improve the system resilience to node failures.

III. The HEFT4K scheduling

Let $G^c = (V^c, E^c)$ be a connected but not necessarily complete graph representing the distributed Edge-Cloud computing architecture. Each vertex $v_i^c \in V^c$ is a computing node and each edge $e_{j,k}^c \in E^c$ is the communication link between v_j^c and v_k^c. Each edge has a bandwidth value associated with $l(e_{j,k}^c)$.

We define $G^a = (V^a, E^a)$ as the DAG representing the software application a, where each vertex $v_i^a \in V^a$ is a containerized task to be scheduled on a distributed computing architecture. Each edge $e_{j,k}^a \in E^a$ is a temporal dependency between tasks (v_j^a, v_k^a). Each task has a corresponding *computational cost* $t(v_{i,j}^a)$ for each node $v_j^c \in V^c$ (ms). Each edge has a corresponding *communication cost* $d(e_{j,k}^a)$. The communication value is represented in Mbits. Combined with the network bandwidth, it allows to calculate the actual communication costs after the task-to-node mapping. The goal is to find a mapping of V^a in V^c that achieves the best makespan without the need for global synchronization among tasks.

Let $\text{HEFT}_{sim}(v_i^a, G^c)$ be the scheduling simulation function that, starting from the execution time of each task v_i^a in each node v_j^c and the communication latency of each link $e_{j,k}^c$ (which we obtain with the open shortest path first routing algorithm) represented by G^c, applies the HEFT algorithm (look ahead version [21]). The algorithm generates a ranking of the tasks. By following the rank list, it calculates, for each task, the task-to-node mapping, the start and the finish times of the task in the selected node. The mapping and the scheduling times guarantee that the execution of tasks follows the priority order of the generated rank (*condition 1*), and the temporal constraints between tasks (i.e., the execution order of the DAG nodes in each path) are satisfied (*condition 2*).

Nevertheless, due to the lack of global synchronization, the Kubernetes scheduling cannot apply the start and finish times of tasks on the distributed nodes. Although adopting the HEFT mapping in Kubernetes without task synchronization achieves makespan improvements over the standard Kubernetes scheduling (i.e., 17.7% on average), it suffers from priority inversion and preemption of tasks, which often lead to makespan increase. In addition, HEFT is a static scheduler, which requires re-execution at each deployment event of Kubernetes. It requires the shut down and re-start of the whole container set, which involves unsustainable interruptions of the robot functionality.

We propose *HEFT4K*, which relies on three points. First, we assume that the robotic software application implements task communication through a standard API, such as ROS or any other publish-subscribe paradigm. This guarantees that condition 2 is satisfied. At the deployment time, we associate a *niceness* value to each task for the preemption rules implemented by the target operating system. Even though condition 1 cannot be guaranteed, the niceness values force the priority of the tasks by considering the HEFT ranking (see Section III-A). It finally implements an event-driven remapping approach to deploy tasks after node failure without requiring the application deployment from scratch (Section III-B).

Fig. 2: Scheduling the example of Fig. 1 with HEFT4K.

A. HEFT ranking-based priority through task niceness

In the Linux kernel, the standard scheduler relies on a dynamic priority-based scheduling algorithm. Initially, each process is assigned a base priority. This base priority increases each time the process remains idle or waits in the ready queue. The operating system allows adjustment of the base priority through the niceness value, which can elevate a process's base priority, resulting in more frequent scheduling:

$$base_priority = (100) + 20 + niceness,\\ niceness \in [nice_H, nice_L] \subseteq [-20, 19] \quad (1)$$

A base priority of 100 is the highest possible for a non-real-time process. It is important to note that lower values indicate higher priorities. This scheduler ensures that every process receives a fair share of CPU time and prevents any process from becoming starved. Eventually, even a process with the lowest base priority will see its dynamic priority rise, making it the first in queue. Once a process is scheduled to run, its dynamic priority is reset to its base value.

We leverage this dynamic priority system to assign a niceness value which depends on the HEFT rank of each task. The HEFT rank represents the task scheduling order computed by HEFT within each device's scheduler. This approach effectively mitigates priority inversions and drastically reduces preemptions, addressing the challenge of limited control over a process's start and finish times.

We define $\text{HEFT}_R(v_i^a)$ as the function that returns the HEFT rank for a given task. Then, HEFT4K calculates the task niceness as follows:

$$niceness(v_i^a) = \frac{(nice_H - nice_L) * \text{HEFT}_R(v_i^a)}{max(\text{HEFT}_R(G^a))} + nice_L \quad (2)$$

where $nice_H$ and $nice_L$ denote the upper and lower bounds for the niceness values to be assigned to each task, as defined in (1). Let $optimize_niceness(v_i^a)$ be a function that computes the niceness of a node v_i^a according to equation (2).

Fig. 2 shows the resulting scheduling for the running example. Compared to HEFT on Kubernetes in Fig. 1, we observe the elimination of priority inversions and preemptions along the critical path, particularly with task *F*. This leads to a notable enhancement of approximately 20.9% in the makespan. In comparison to the scheduling implemented by Kubernetes, the speedup is 33.6%.

B. Event-based remapping

To complement the orchestrator functionality, we also account for scenarios where a computing node becomes unavailable (e.g., due to loss of connectivity or depleted battery) and the tasks running on that node necessitate an online re-deployment (i.e., restart of the tasks on a different node without restarting the whole system). This operation can lead to suboptimal remapping and extended makespan as standard orchestrators like Kubernetes implement such a re-deployment of containers targeting workload balancing instead of makespan optimization. To address this issue, HEFT4K incorporates the *partial* remapping of tasks. This approach specifically remaps

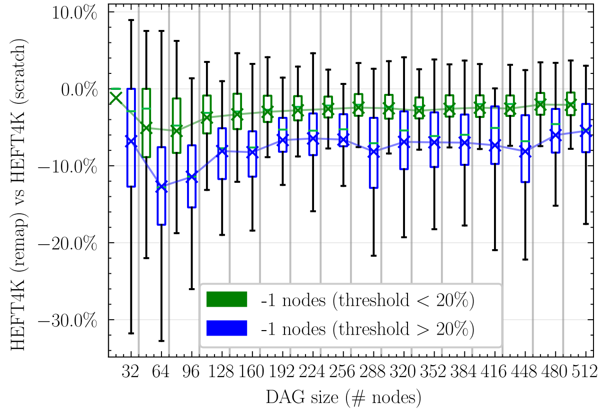


Fig. 3: Performance loss when the sum of the HEFT rank lost due to a node failure is higher than 20%.

only the tasks that were lost onto operational nodes, with a focus on optimizing makespan to minimize the impact of node failures.

Conversely, there are scenarios where partial re-mapping might result in notable makespan increase, and a complete task re-mapping from the beginning, along with a corresponding temporary service interruption, would be more acceptable. HEFT4K employs a *threshold* mechanism to determine whether to choose partial or total remapping. If the cumulative HEFT ranks of the rescheduled tasks exceeds a certain percentage of the total sum of HEFT ranks for the entire DAG, the decision is made to reschedule the entire application. The idea is that, given that the HEFT rank encompasses task computation time, communication time, and priority, exceeding the HEFT rank threshold implies that either numerous less critical tasks or a few highly crucial ones were assigned to the failed node. In either scenario, this indicates a significant deterioration in the current mapping.

The threshold value is user-defined and system-dependent. We developed a benchmark that, starting from a given δ-value representing the maximum increase of makespan and the architecture characteristics (node compute capability, communication, etc.), it extrapolates the threshold for any DAG size. The benchmark iterates over the threshold values until the δ is reached, giving the best compromise between rescheduling frequency and performance.

Fig. 3, for example, depicts the results for $\delta = 10\%$ in our experimental setup, i.e., the performance difference between a re-mapping created from scratch and a partial re-mapping for different DAG sizes, when a node in the cluster is lost. We found the corresponding near-optimal value of the threshold at ≈20%. Green denotes the DAGs that lost less than 20% of the total HEFT rank of the application, while blue denotes applications that lost more than 20%.

Algorithm 1 outlines the complete re-mapping procedure. The algorithm first identifies all tasks affected by an offline node (line 1). It checks if a critical task has been impacted or if the threshold has been surpassed (lines 3-6). If either condition is met, it performs a complete re-mapping of the DAG. Otherwise, a partial re-mapping is performed, and the list of tasks is sorted according to their HEFT ranks (lines 8-9). For each task in the graph, if it is not one of the affected tasks, and for each node in the Edge-Cloud cluster that differs from the node on which the current task was originally scheduled, it modifies the CPU time to the maximum. This adjustment compels the HEFT algorithm to remap tasks that were not affected on the same node as before, while dropped tasks are mapped around locked tasks in the most optimal manner possible (line 18). The algorithm computes the niceness value (line 19). Finally, it restores all modified runtimes in G^a (line 20).

Algorithm 1: Rescheduling algorithm in case of nodes that go offline.

input : The graphs G^a and G^c
output: The container-to-node-mapping.

1 **Function** `reschedule` (G^a, G^c)**:**
2 $\quad OT \leftarrow get_offline_tasks()$
3 $\quad$ **if** $\exists v_i^a \in OT \mid v_i^a$ *is crit. or* $\sum_{i \in OT} HEFT_R(v_i^a) > threshold$ **then**
4 $\quad\quad$ **for** $v_i^a \in G^a$ **do**
5 $\quad\quad\quad$ sched $\leftarrow$ HEFT$_{sim}$ (v_i^a, G^c)
6 $\quad\quad$ **end**
7 $\quad$ **else**
8 $\quad\quad$ $G^a \leftarrow$ sort(G^a, HEFT$_R$)
9 $\quad\quad$ $G^a_{copy} \leftarrow G^a$
10 $\quad\quad$ **for** $v_i^a \in G^a$ **do**
11 $\quad\quad\quad$ **if** $v_i^a \notin OT$ **then**
12 $\quad\quad\quad\quad$ **for** $v_j^c \in G^c$ **do**
13 $\quad\quad\quad\quad\quad$ **if** $v_j^c \neq n(v_i^a)$ **then**
14 $\quad\quad\quad\quad\quad\quad$ $t(v_{i,j}^a) = 9999999$
15 $\quad\quad\quad\quad\quad$ **end**
16 $\quad\quad\quad\quad$ **end**
17 $\quad\quad\quad$ **end**
18 $\quad\quad\quad$ sched $\leftarrow$ HEFT$_{sim}$ (v_i^a, G^c)
19 $\quad\quad\quad$ $optimize_niceness(v_i^a)$
20 $\quad\quad\quad$ $G^a \leftarrow G^a_{copy}$
21 $\quad\quad$ **end**
22 $\quad$ **end**
23 **return** sched

This algorithm allows for improved rescheduling efficiency and is also efficient itself. The overall complexity is the same as HEFT because the HEFT rank sorting takes $O(|V^a| \times \log |V^a|)$, the CPU time modification phase added to HEFT takes $O(|V^a| \times |V^c|)$ and, the niceness computation phase takes $O(|V^a|)$. Thus, the overall complexity remains $O(|V^a|^2 \times |V^c|)$ (from [8]).

Finally, HEFT4K implements the complete re-deployment also when a node failure involves a *critical task*, i.e., a task whose offline affects the entire application and causes the system downtime.

IV. Experimental results

We first tested HEFT4K with 100k synthetic DAGs on a computing platform simulating an Edge-Cloud architecture composed of 6 nodes, with 2 CPUs per node. We implemented a parametric DAG generator to build DAGs with different characteristics, including size, degree of nodes, and execution times. This translates into different levels of topological constraints among nodes, including tree-like shapes (high average degree and medium/small diameter) and linear shapes (low average degree and high diameter). Table II summarizes the range of the considered values. Without loss of generality, we only consider one application DAG at a time since multiple DAGs can be unified into a single composite DAG by merging them at a designated virtual root node, which does not alter the intrinsic scheduling behavior or dependencies of the individual DAGs.

We then tested HEFT4K on the software that implements the mission of a Robotnik RB-Kairos, which is a skid-steering mobile platform equipped with a Universal Robots UR5 and a Schunk WSG50 gripper. The computing cluster consists of three nodes: two on-board programmable devices, i.e., an NVIDIA Jetson Xavier and a Jetson Nano. The onboard nodes communicate through a Gigabit Ethernet switch (802.3ab). The third node consists of an off-board desktop with an octa-core CPU and 16GB of RAM. It communicates with the other cluster nodes through WiFi (802.11ac).

Graph characteristics	min.	max.	avg.	median
Nodes (#)	4	511	181.8	130
Edges (#)	3	1576	305.6	158
Outdegree	0.9	3.1	1.4	1.2
Critical path (ms)	4	446	64.3	41
Task comp. cost (ms)	1	20	10	10
Task comm. cost (Mb)	1	5	2.5	2.5
Node comm. (Mb/s)	500	1000	750	750

TABLE II: Range of values considered for the random generation of DAGs and cluster configuration.

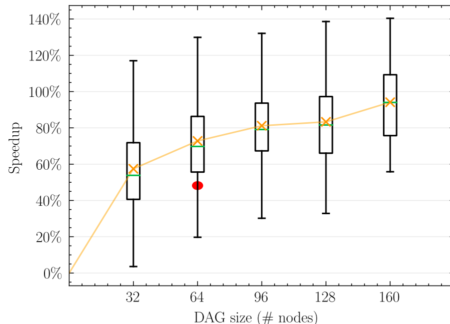


Fig. 4: Speedup of HEFT4K vs. the standard Kubernetes on different DAG sizes.

The software is a ROS2-compliant application that executes a robot mission through a 42-task DAG, with each task placed in its own container to have the most scheduling flexibility. The application allows interaction between the mobile robot and an agile industrial production chain. The robot starts at a designated position, aligned with the production chain, and remains idle until a pallet arrives on the conveyor belt. Using a sequence of arm and gripper operations, the robot grasps production pieces from the conveyor belt and holds them in its cargo bay. Then, it moves towards the warehouse area, unloads the pieces from the cargo bay, and returns to the production line, aligning itself with the conveyor belt for the next cycle.

For both the synthetic benchmarks and the real case study, we compared the results obtained with the standard Kubernetes scheduler, HEFT on Kubernetes, the most efficient scheduler for Edge-Cloud architectures targeting QoS in literature (DPE) [18], and the proposed HEFT4K. We also tested the event-driven rescheduling of HEFT4K, analyzing its ability to maintain a positive makespan when sequentially removing nodes from the cluster.

A. HEFT4K vs. state of the art Kubernetes scheduling

Fig. 4 summarizes the comparison results between HEFT4K and the standard Kubernetes scheduler. Even with graphs of size 32 or less (first box plot from the left), HEFT4K achieves a significant speedup of 40%+, and in the upper 25% of cases, the speedup grows to over 60%. With larger graphs, the speedup exceeds 100%. The red dot in Fig. 4 represents the speedup (48.2%) measured with the real case of study. With DAG sizes larger than 160 tasks, the Kubernetes scheduler fails. This is due to the fact that, differently from HEFT4K, Kubernetes stops the deployment of tasks when there are no resources available (i.e., free computing nodes).

Fig. 5 shows the comparison results between HEFT4K and DPE [18][1]. On average, HEFT4K achieves a speedup of 16% and reaches

[1]In [18], DPE achieves a performance improvement of ≈40% over a standard HEFT [8] with one CPU per node. We measured a loss of performance ≈15% of DPE when compared to HEFT with the look-ahead variant [21] with multiple CPUs per node.

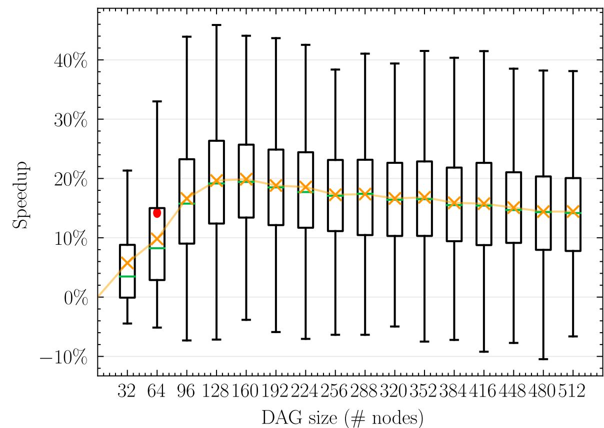


Fig. 5: Speedup of HEFT4K vs DPE scheduler on different DAG sizes.

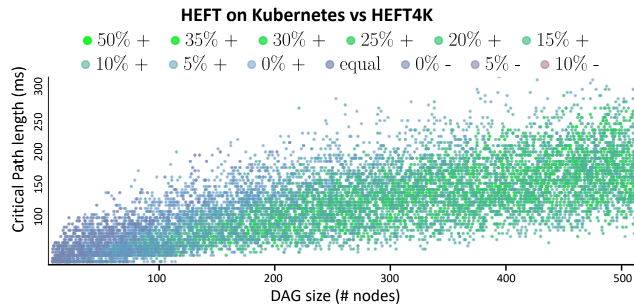


Fig. 6: Heatmap comparing the performance of HEFT4K and HEFT on Kubernetes, plotting the number of tasks in the DAG and the length of the critical path.

up to 40% speedup with DAG sizes larger that 64 tasks. In a limited number of cases (< 25%), the DPE is slightly faster, on average, with 4% speedup w.r.t. HEFT4K. This is due to the fact that DPE does not use preemption and takes advantage of global synchronization. In the smaller graphs, small scheduling inefficiencies caused by the Linux scheduler, can translate in large percentage slowdowns.

We also compared HEFT4K and DPE on the Robotnik RB-Kairos case study. We obtained a 14.2% speedup on our 42-task DAG workload, which is in line with the average speedup obtained for the same DAG size in the synthetic dataset.

B. HEFT4K vs. HEFT on Kubernetes

The speedup achieved by HEFT4K on HEFT on Kubernetes is on average ≈ 13% across all DAGs and cluster configurations. Figure 6 shows a three dimensional heat map to represent the speedup of HEFT4K over HEFT on Kubernetes, by considering the DAG size and the critical path length. The speedup is limited with small graphs with short critical paths (lower left-most part of the plot). On the other hand, the speedup increases linearly with the DAG size and with the critical path length, albeit not as quickly with the critical path length (i.e., the bottom half of the Figure has lower speedups compared to the top half). This is due to the fact that larger graphs with a longer critical path are generally more sequential and thus have less scheduling freedom. Therefore, the niceness optimization cannot really improve the scheduling, resulting in lower speedups.

C. Performance of event-based remapping

We reused the same six-node architecture and 100k DAGs as the previous tests and repeated the scheduling five times. We removed

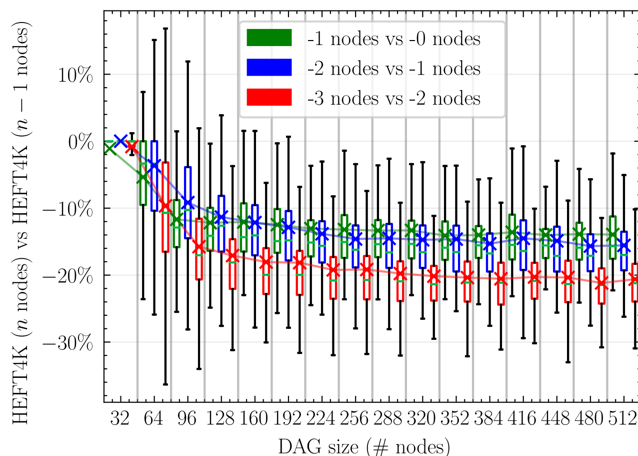


Fig. 7: Speedup of HEFT4K event-based remapping when rescheduling DAGs in case of node shutdown, compared to the previous makespan obtained with HEFT4K.

one random node from the cluster in each iteration until only one node was left. Fig. 7 shows the result in terms of makespan increase (i.e., negative speedup in %).

The loss of one to three cluster nodes translates into a performance loss of $\approx 6\%$, for each lost node, when looking at DAG sizes smaller than 96 nodes. When considering larger DAG sizes, the performance degradation increases, although remaining below 15% on average when one or two nodes are lost. When three nodes are lost, the average trends towards -20%. However, half the cluster is offline and the mapping strategy is limited.

The results for losing 4 and 5 nodes are similar, trending toward an average performance loss of $\approx 25\%$ (not reported for readability).

Overall, the algorithm provides strong resilience against node faults, achieving positive performance, especially when losing one to two nodes of the computing cluster, which represents the most common scenario in real application use cases.

The overall time required to return the application to a working state after a node failure, and thus its downtime, depends strongly on its size and architecture. To recover from a node failure, the system needs to first execute the rescheduling algorithm and assess whether a partial rescheduling would significantly degrade performance. The time required by the rescheduling algorithm is negligible (few ms in our setup) since it is as efficient as HEFT. In addition, it needs to restart some (or all) containers on a new cluster node. However, restarting a container is a rather fast process that takes on average less than a second, and is parallelized, restarting multiple containers at once.

V. Conclusions

This work addressed the challenge of deploying robotic software with orchestrator platforms like Kubernetes while adhering to QoS constraints. We proposed a scheduling approach called HEFT4K based on the HEFT task ranking that takes advantage of the OS niceness concept to reduce priority inversions and preemptions of tasks. The scheduler implements event-driven remapping to support Kubernetes' scaling and recovery capabilities. Compared to the state of the art Kubernetes scheduler, HEFT4K improved the makespan by an average of 75% on a synthetic dataset and a real case study.

References

[1] T. Iqbal, S. Rack, and L. D. Riek, "Movement coordination in human-robot teams: A dynamical systems approach," *IEEE Transactions on Robotics*, vol. 32, no. 4, pp. 909–919, 2016, cited By :45. [Online]. Available: www.scopus.com

[2] P. Thakur and V. Kumar Sehgal, "Emerging architecture for heterogeneous smart cyber-physical systems for industry 5.0," *Computers and Industrial Engineering*, vol. 162, 2021.

[3] F. Lumpp, M. Panato, N. Bombieri, and F. Fummi, "A design flow based on docker and kubernetes for ros-based robotic software applications," *ACM Trans. Embed. Comput. Syst.*, vol. 23, no. 5, aug 2024.

[4] F. Lumpp, F. Fummi, H. D. Patel, and N. Bombieri, "Enabling kubernetes orchestration of mixed-criticality software for autonomous mobile robots," *IEEE Transactions on Robotics*, vol. 40, pp. 540–553, 2024.

[5] P. Melo, R. Arrais, and G. Veiga, "Development and deployment of complex robotic applications using containerized infrastructures," in *2021 IEEE 19th International Conference on Industrial Informatics (INDIN)*, 2021, pp. 1–8.

[6] M. Shaik and et al., "Enabling fog-based industrial robotics systems," in *IEEE Symposium on Emerging Technologies and Factory Automation, ETFA*, vol. 2020-September, 2020, pp. 61–68.

[7] N. Nikolakis, R. Senington, K. Sipsas, A. Syberfeldt, and S. Makris, "On a containerized approach for the dynamic planning and control of a cyber - physical production system," *Robotics and Computer-Integrated Manufacturing*, vol. 64, p. 101919, 2020.

[8] H. Topcuoglu, S. Hariri, and M.-Y. Wu, "Performance-effective and low-complexity task scheduling for heterogeneous computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 13, no. 3, pp. 260–274, 2002.

[9] C.-C. Chang, S.-R. Yang, E.-H. Yeh, P. Lin, and J.-Y. Jeng, "A kubernetes-based monitoring platform for dynamic cloud resource provisioning," in *GLOBECOM 2017 - 2017 IEEE Global Communications Conference*, 2017, pp. 1–6.

[10] G. Rattihalli, M. Govindaraju, H. Lu, and D. Tiwari, "Exploring potential for non-disruptive vertical auto scaling and resource estimation in kubernetes," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, 2019, pp. 33–40.

[11] F. Katenbrink, A. Seitz, L. Mittermeier, H. Müller, and B. Bruegge, "Dynamic scheduling for seamless computing," in *2018 IEEE 8th International Symposium on Cloud and Service Computing (SC2)*, 2018, pp. 41–48.

[12] M. Chima Ogbuachi, C. Gore, A. Reale, P. Suskovics, and B. Kovács, "Context-aware k8s scheduler for real time distributed 5g edge computing applications," in *2019 International Conference on Software, Telecommunications and Computer Networks (SoftCOM)*, 2019, pp. 1–6.

[13] M. C. Ogbuachi, A. Reale, P. Suskovics, and B. Kovács, "Context-aware kubernetes scheduler for edge-native applications on 5g," *Journal of communications software and systems*, 2020.

[14] I. Rocha, C. Göttel, P. Felber, M. Pasin, R. Rouvoy, and V. Schiavoni, "Heats: Heterogeneity-and energy-aware task-based scheduling," in *2019 27th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*, 2019, pp. 400–405.

[15] G. El Haj Ahmed, F. Gil-Castiñeira, and E. Costa-Montenegro, "Kubcg: A dynamic kubernetes scheduler for heterogeneous clusters," *Software: Practice and Experience*, vol. 51, no. 2, pp. 213–234, 2021. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/spe.2898

[16] J. Li, B. Liu, W. Lin, P. Li, and Q. Gao, "An improved container scheduling algorithm based on pso for big data applications," in *Cyberspace Safety and Security*, J. Vaidya, X. Zhang, and J. Li, Eds. Cham: Springer International Publishing, 2019, pp. 516–530.

[17] Y. Fu, S. Zhang, J. Terrero, Y. Mao, G. Liu, S. Li, and D. Tao, "Progress-based container scheduling for short-lived applications in a kubernetes cluster," in *2019 IEEE International Conference on Big Data (Big Data)*, 2019, pp. 278–287.

[18] S. Deng, H. Zhao, Z. Xiang, C. Zhang, R. Jiang, Y. Li, J. Yin, S. Dustdar, and A. Y. Zomaya, "Dependent function embedding for distributed serverless edge computing," *IEEE Transactions on Parallel and Distributed Systems*, vol. 33, no. 10, pp. 2346–2357, 2022.

[19] Intel. (2023) Telemetry Aware Scheduling. [Online]. Available: www.intel.com/content/www/us/en/developer/articles/technical/telemetry-aware-scheduling.html

[20] IBM. (2023) Trimaran: Real Load Aware Scheduling. [Online]. Available: github.com/kubernetes-sigs/scheduler-plugins/tree/master/kep/61-Trimaran-real-load-aware-scheduling

[21] L. F. Bittencourt, R. Sakellariou, and E. R. M. Madeira, "Dag scheduling using a lookahead variant of the heterogeneous earliest finish time algorithm," in *2010 18th Euromicro Conference on Parallel, Distributed and Network-based Processing*, 2010, pp. 27–34.