GNN-MiCS: Graph Neural-Network-Based Bounding Time in Embedded Mixed-Criticality Systems

Behnaz Ranjbar, Paul Justen, and Akash Kumar, *Senior Member, IEEE*

***Abstract*—In Mixed-Criticality (MC) systems, each task has multiple WCETs for different operation modes. Determining WCETs for low-criticality modes (LO modes) is challenging. A lower WCET improves processor utilization, but a longer one reduces mode switches, maintaining smooth task execution even with low utilization. Most research focuses on WCETs for the highest criticality mode, with fewer solutions for LO modes in graph-based applications. This paper proposes GNN-MiCS, a machine learning and graph neural networks scheme to determine WCETs for directed acyclic graph applications in LO modes. GNN-MiCS generates test sets and computes proper WCETs based on the application graph to enhance system timing behavior. Experiments show our approach improves MC system utilization by up to 45.85% and 22.45% on average while maintaining a reasonable number of mode switches in the worst-case scenario.**

***Index Terms*—Mixed-Criticality, Mode Switching Probability, Neural Network, Utilization, Worst-Case Execution Time (WCET).**

I. Introduction

NOWADAYS, Mixed-Criticality (MC) systems are widely used in embedded real-time applications such as medical devices, automotive, and avionics to meet requirements like cost, space, timing, and power consumption [1]–[3]. These systems handle tasks with different criticality levels, ensuring flawless execution of High-Criticality (HC) tasks to prevent catastrophic consequences while optimizing processor utilization and Quality-of-Service (QoS) by executing more Low-Criticality (LC) tasks [2]–[4].

MC systems typically define multiple Worst-Case Execution Time (WCET) for different criticality levels and operational modes [2]–[4]. A common type is the dual-criticality system with LC and HC tasks, each having two WCETs: a lower ($WCET^{LO}$) and a higher ($WCET^{HI}$) bound. $WCET^{HI}$ is the maximum execution time under all conditions, but relying on it reduces processor utilization and QoS as fewer LC tasks can be scheduled [2], [5]. Thus, $WCET^{LO}$ is used to maximize processor utilization and QoS in low-criticality mode (LO mode), while ensuring task guarantees in high-criticality mode (HI mode).

For dependent MC tasks, the system initially follows a schedule based on $WCET^{LO}$. If an HC task exceeds its $WCET^{LO}$, the system switches to HI mode, using $WCET^{HI}$ for unexecuted HC tasks to ensure their correct execution, possibly dropping unexecuted LC tasks [6]. A significant gap between $WCET^{LO}$ and $WCET^{HI}$ means more tasks are scheduled at design-time, leading to frequent mode switches and more LC tasks dropped at run-time. Conversely, a smaller gap reduces processor utilization due to fewer tasks being scheduled. Thus, $WCET^{LO}$s are crucial for enhancing timing behavior, QoS, and mode-switching probability.

While numerous approaches, such as those in [7] and tools like OTAWA [8], exist for determining $WCET^{HI}$, there are fewer methods for determining $WCET^{LO}$ in MC systems. Existing approaches [9], [10] often set $WCET^{LO}$s as a percentage of $WCET^{HI}$s, which may result in poor processor utilization or frequent mode switches. [2] proposed determining $WCET^{LO}$ based on the application's Average-Case Execution Time (ACET) using Chebyshev's theorem; however, this is limited to independent tasks and pessimistically determines mode switching probability, leading to poor utilization. [11] presented a Neural-Network (NN)-based approach to determine $WCET^{LO}$ based on task time distribution, but it is limited to independent tasks and requires precise knowledge of task functions and distributions, which is impractical at design-time for some applications. In [12], Machine-Learning (ML) techniques determine the $WCET^{LO}$ at run-time if there is sufficient dynamic slack, but this approach has significant timing overhead and relies on the generation of dynamic slack.

Our study introduces a novel learning-based approach called *GNN-MiCS*, a Graph Neural-Network-based bounding time approach to determine the $WCET^{LO}$s in Mixed-Criticality Systems. Our scheme aims to achieve the following objectives: 1) effectively minimize the frequency of system mode switches, 2) achieve high processor utilization, thereby enhancing the QoS, and 3) ensure the schedulability of the system at each criticality level, i.e., all tasks can schedule and execute correctly before their deadlines. To the best of our knowledge, there is no method yet to determine $WCET^{LO}$ of dependent MC tasks with no run-time timing overhead while making a trade-off between the QoS and mode switches. Our implementation and experimental data are publicly available[1].

Contributions: The main contributions of this paper are:

- Presenting a novel scheme to analyze and obtain the low WCETs of dependent MC tasks in order to improve the system timing behavior.
- Proposing a learning-based approach, called GNN-MiCS, to design an MC system, which makes a reasonable trade-off between mode-switching probability and utilization that can be assigned to LC tasks.
- Scheduling the tasks before deadlines with insignificant and no timing overheads at design- and run-time, respectively.

II. System Model

We deal with periodic dependent MC tasks, each denoted as τ_i={$\zeta_i, WCET_i^{LO}, WCET_i^{HI}, d_i, Su_i, Pr_i, P_i$}, analo-

This work was supported in part by Deutsche Forschungsgemeinschaft (DFG) through the Project *LeanMICS* under Grant 534919862.

B. Ranjbar and A. Kumar are with the chair of Embedded Systems, Faculty of Electrical Engineering and Information Technology, Ruhr-Universität Bochum, 44801 Bochum, Germany (e-mail: {behnaz.ranjbar, akash.kumar}@rub.de).

P. Justen is with Technische Universität Dresden, 01062 Dresden, Germany e-mail: paul.justen@mailbox.tu-dresden.de).

[1]The source code is available at https://etit.ruhr-uni-bochum.de/esys/downloads/

gous to [6], [9], [10]. We adopt a dual-criticality system where each dependent MC task can be classified as either high-critical (ζ_i = HC) or low-critical ζ_i = LC). Moreover, each task τ_i has a specific local deadline d_i. The relationships between tasks, including their successors and predecessors, are determined by Su_i and Pr_i, respectively. A task is executable only after all its predecessors have completed their execution. Each MC task has distinct WCETs, $WCET^{LO}$ and $WCET^{HI}$. For every LC task, $WCET^{LO}$=$WCET^{HI}$, while for every HC task, $WCET^{LO} \leq WCET^{HI}$. If a task is a predecessor to an HC task, it is also considered an HC task [6]. Additionally, all tasks share a common period P_i which represents the period of the task graph.

An MC system initially starts the operation in the LO mode. If the execution time of any HC task exceeds its $WCET^{LO}$, the system switches to the HI mode, and in this mode, some/all LC tasks might be dropped to guarantee the correct execution time of the tasks. During LO mode, mapping and scheduling algorithms consider the $WCET^{LO}$ of tasks, while in HI mode, tasks are scheduled according to $WCET^{HI}$. In the context of dependent MC task models, the system safely switches back to LO mode at the end of each period [3], [5], [6].

Algorithm 1 Neural Network Model

Require: $graph \leftarrow graph[features, values, steps, deadline]$
▷ $features \leftarrow [crits, wcets_{low}, wcets_{high}, acets, stds]$ ◁
▷ $crits, wcets_{low}, wcets_{high}, acets, stds \leftarrow array[\#tasks]$ ◁
▷ $values \leftarrow array[\#tasks]$ ◁
▷ $steps \leftarrow array[[sender, receiver], \dots]$ ◁
function COLLECT(graph, step)
  $value_{sender} = graph.values[sender]$
  $value_{receiver} = graph.values[receiver]$
  $value_{new} = NN\_collect(value_{sender}, value_{receiver})$
  $graph.values[receiver] = value_{new}$
  output graph
function APPLY(graph, step)
  $value_{receiver} = graph.values[receiver]$
  $features_{receiver} = graph.features[receiver]$
  $value_{new} = NN\_apply(value_{receiver}, feature_{receiver})$
  $graph.values[receiver] = value_{new}$
  output graph
function OUTPUT(graph)
  $values = graph.values$
  $results = NN\_output(values)$
  output results
for all $step \in graph.steps$ **do**
  $g = collect(g, step, params)$
  $g = apply(g, step, params)$
$results = output(g, params)$
output results

III. Proposed Method: GNN-MiCS

Identifying the appropriate $WCET^{LO}$s for HC tasks is challenging. Our approach designs MC systems and analyzes task graph applications using ML models to select the optimal $WCET^{LO}$, reducing mode switches and improving LC task execution. For embedded applications, designers typically know system tasks and can compute parameters, like $WCET^{HI}$s, at design-time. We utilize task graph applications from the tool in [13]. MC tasks are scheduled for LO and HI modes using static scheduling tables from the algorithm of [13]. Our approach uses Graph Neural Network (GNN) to determine $WCET^{LO}$ for HC tasks, improving QoS by scheduling more LC tasks.

Using optimization methods like Integer Linear Programming (ILP) to obtain $WCET^{LO}$ is inefficient compared to ML-based approaches in terms of scalability, adaptability, and speed. ML models handle large, complex task sets more efficiently than ILP, since using ILP is computationally expensive as task size and complexity increase. ML techniques learn from historical data, generalizing to provide accurate estimates for new task sets without solving new optimization problems each time. Once trained, ML models offer quick $WCET^{LO}$ estimates, beneficial for iterative design and real-time adjustments, whereas ILP requires considerable computation time for each new task set. Among ML techniques, GNN is preferred for dependent MC task graph models due to their ability to handle graph structures, learn complex dependencies, adapt to varying input sizes, and generalize across different task sets. Leveraging a program's dependency graph for a GNN integrates the inherent information within the structure into the learning process by traversing the graph and computing the information in each node.

Algorithm 1 outlines the pseudo-code of the NN model using GNN layers and shows fundamental operations, using a data structure of steps (edges) and nodes. Each node contains values and features, with values computed and features remaining constant. Features include task criticality (ζ), $WCET^{LO}$, $WCET^{HI}$, ACET, and Standard Deviation (STD). ACET and STD are needed to compute task overrun probability based on [2], which are computed by running tasks several times on a CPU processor core with various inputs. These two parameters helps determine the probability of mode switching.

The steps follow a dependency-sensitive traversal of the graph and are iterated over during the model's application process to generate output. Each step involves a sender and receiver, referencing corresponding nodes by index. Operations in the model are facilitated by dedicated NNs, with the four fundamental operations including *init*, *collect*, *apply*, and *output*. The NNs share identical layouts with parameters like the number of layers, neurons per layer (using a tanh activation function), and output vector size. The NN for the output operation reduces the number of neurons per layer until only one remains to yield singular values. Various task graph sets, generated by the tool [13] are utilized as training data set.

During training, the process iterates repeatedly as follows:

- *Initialisation*: For each node, the initial values are computed based on the features using the init-function. The outcome is an array, ideally matching the size of the number of neurons per layer.
- *Traversal*: The model sequentially processes each step (or edge), executing the collect- and apply-functions. In the collect-function, values from both connected nodes are combined to compute a new value, which replaces the receiving node's value, merging information from the predecessor nodes. Subsequently, in the apply-function, the new value (newly acquired information) is combined with the receiver node's features, updating the node's value. This ensures effective information propagation through the graph. Incorporating both sender and receiver values during the collect-function prevents loss of information, especially for nodes with multiple predecessors.
- *Model Results*: To prepare the model results for using in subsequent loss calculations, the output-function condenses each node's value array into a singular value for the optimized $WCET^{LO}$ calculation. The loss function then calculates a performance score based on utilization and task overrun probability using the graph with the optimized $WCET^{LO}$.

– **Resource Utilization**: The enhancement is characterized by a notable increase in the utilization of LC tasks in LO mode (U_{LC}^{LO}). This is restricted by the upper limits imposed by the schedulability constraints in LO mode, according to the scheduling algorithm in [13]. This open-source algorithm creates two scheduling tables for LO and HI modes. U_{LO}^{LO} is calculated based on $WCET^{LO}$ [2] using the tables form [13] and the available free slack for scheduling LC tasks.
– **Mode Switching Probability**: its reduction positively impacts the performance/functionality of MC systems, notably by reducing the frequent drops of LC tasks in the HI mode. $P_i^{overrun}$ represents the probability of task τ_i exceeding $WCET^{LO}$, while P_{Sys}^{MS} is the mode switching probability of the system. Since each HC task can overrun independently of other tasks and causes the system to switch to the HI mode, P_{Sys}^{MS} is computed as follows [2]. Note that, $P_i^{overrun}$ is calculated based on the formula mentioned in [2]:

$$P_{Sys}^{MS} = 1 - \prod_{\zeta_i \in HC} (1 - P_i^{overrun}) \quad (1)$$

To enhance the system's timing behavior, maximizing the equation $score = U_{LC}^{LO} \times (1 - P_{Sys}^{MS})$ is crucial [2]. Therefore, an approach capable of determining appropriate values for $WCET^{LO}$ should result in a higher value for this equation [2], [11].

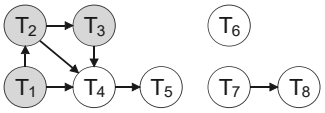
For training on multiple graphs, they are grouped into batches. The size of these groups can be specified for each run, but typically set to 20% of the total number tends to yield optimal results. During batching, graphs are combined into a single graph object with concatenated values and features, and steps are merged element-wise with adjusted values based on an offset. The batched graph resembles multiple sub-graphs, with steps containing sender and receiver values referencing nodes from each sub-graph. This method, along with accelerated array computing, speeds up training. Metric calculations remain graph-wise, producing results for each graph, which are then aggregated by summing and dividing by the batch size to obtain the average result.

IV. Evaluation Results

A. Experimental Setup

In our experiments, we evaluate our scheme by synthetic task sets, known as task graphs, generated by the tool mentioned in [13]. These applications involve four key parameters: *c* (number of cores), *U* (system utilization), *d* (outgoing edge percentage), and *n* (number of tasks). The parameter *d* represents the probability of having outgoing edges from one task to other tasks. *U/c* is a normalized utilization, which refers to both LC and HC tasks, each with their predefined $WCET^{HI}$. Since we use the same configuration and system inputs as [5], by taking inspiration from actual execution times as detailed in [5], we consider the same scenario in [5] to provide ACET and STD within the ranges of $[\frac{1}{5} * WCET^{HI}, \frac{1}{3} * WCET^{HI}]$ and $[0.05 * WCET^{HI}, 0.1 * WCET^{HI}]$, respectively.

We compare our proposed method, GNN-MiCS, against the re-implemented approaches in [2], [9], [10], [13]. $WCET^{LO}$s are set in [13] based on its generated tool. Besides, most of the state-of-the-art works, like [9], [10] have defined a fraction of $WCET^{HI}$ as $WCET^{LO}$. If $\lambda = \frac{WCET^{LO}}{WCET^{HI}}$, researchers in [9] have considered $\lambda \in [0.001, 1]$ in their experiment. Researchers in [10] have also considered $\lambda \in [\frac{2}{3}, 1]$. Since these papers have the same policy in determining $WCET^{LO}$, we select [10] as a representative methodology for all these research works in the experiments. In addition, [2] computed $WCET^{LO}$ based on the task ACET by exploiting Chebyshev's theorem.



	T1	T2	T3	T4	T5	T6	T7	T8
WCET^{LO}	30	60	20	20	30	60	20	20
WCET^{HI}	30	80	50	20	30	60	20	20

Fig. 1. Real-life application task graph (UAV).

From the machine learning perspective, the NNs comprise 4 layers, each containing 32 neurons. The vector used for calculation also consists of 32 values and the learning rate is set to $1 \times e^{-5}$.

B. Investigating Timing Overhead and Different Configuration of ML Technique

We first analyze the timing overhead and different configurations of the NN process. To this end, 1000 task sets are running on 8 cores, and for the learning part, the batch size is 200. The number of layers, neurons, and size are varied to show the results ({layer, neuron, size}). The training time ($Time^{train}$, computed on a machine with Intel(R) Xeon (R) CPU E5-2630 v2@2.6GHz) and the score value ($max(U_{LC}^{LO} \times (1 - P_{Sys}^{MS}))$ for each configuration are as follows.

- {4,32,32}: ($Time^{train}$,$score$)=(350.74s,0.278)
- {8,64,64}: ($Time^{train}$,$score$)= (621.16s,0.2488)
- {16,128,128}: ($Time^{train}$,$score$)= (1523.63s,0.2529)
- {32,256,256}: ($Time^{train}$,$score$)= (5848.11s,0.2644)

Instead of larger NNs improving results significantly, a smaller configuration with 4 layers, 32 neurons, and a vector size of 32 performs better than larger setups. This might be because the limited data provided is insufficient for more complex NNs. Although larger configurations show a trend of improved results, they also increase training time considerably. Finding an optimal NN configuration requires an extensive search for correlations between configuration parameters and training attributes, or brute-forcing different configurations. Additionally, handling larger NNs may necessitate more processing power to explore the model's performance limits. However, in the case of complex and huge task graph applications, utilizing larger NN would be reasonable as it can improve the results significantly.

Besides, since the GNN-MiCS is a design-time approach, $WCET^{LO}$s are computed at design-time and computed training time is reasonable. The determined $WCET^{LO}$s are then used at run-time with no timing overheads.

C. Evaluation With Real UAV Application Model

To evaluate GNN-MiCS in terms of maximum utilization that can be assigned to LC tasks, mode switching probability, and the score ($U_{LC}^{LO} \times (1 - P_{Sys}^{MS})$), we first consider a real application. We selected the Unmanned Aerial Vehicles (UAV) application [13] (Fig. 1), which includes 8 tasks: three HC tasks (T_1 to T_3) for system avoidance, navigation, and stability, and five LC tasks (T_4 to T_8) for sensor data recording, GPS coordination, and video transmission [13]. This UAV

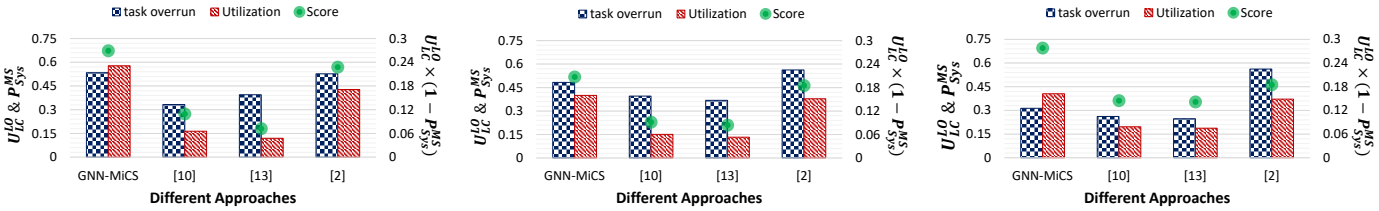


Fig. 2. Objectives and system goal (score) for different approaches, GNN-MiCS, [2], [10], [13]

TABLE I
OBJECTIVES AND SYSTEM GOAL (SCORE) OF DIFFERENT APPROACHES FOR THE REAL-LIFE APPLICATION

	$max(U_{LC}^{LO})^*$	P_{Sys}^{MS**}	$max(U_{LC}^{LO}) \times (1 - P_{Sys}^{MS})^*$
GNN-MiCS	0.269	0.262	0.198
[10]	0.076	0.067	0.071
[13]	0.188	0.237	0.143

* Higher is better ** Lower is better

application runs on two cores, with an overall task graph deadline (equal to period) of 200 units. Training and predicting on a single graph can lead to model overfitting but help assess the approach's effectiveness. We will later expand the training set using the tool from [13] to generate task sets.

Table I shows significant improvement in the score with GNN-MiCS over the unaltered graph. While [10] has a lower mode switching probability, their maximum utilization that can be assigned to LC tasks is inferior to GNN-MiCS due to choosing $WCET^{LO}$ too pessimistic. Additionally, although GNN-MiCS has a mode switching probability nearly equal to [13], it achieves significantly higher maximum utilization. Thus, GNN-MiCS outperforms other methods by effectively balancing mode switching and utilization, as evidenced by having a higher score value than other approaches.

D. Evaluation With Synthetic Task Sets

Now, we evaluate GNN-MiCS in terms of maximum utilization that can be assigned to LC tasks, mode switching probability, and the score ($U_{LC}^{LO} \times (1 - P_{Sys}^{MS})$), using synthetic task sets. Fig. 2 shows the parameters if we set $WCET^{LO}$ according to different approaches by varying numbers of cores.

Fig. 2 shows the average results from 1000 task graph sets across different core numbers ($c = 2, 4, 8$) for training and predicting phase. For all core counts, the mode-switching probability in [13] and [10] is lower (better) than in GNN-MiCS and [2], but their utilization is worse due to overly pessimistic $WCET^{LO}$ estimates. Although [2] achieves a higher score ($U^{LO}LC \times (1 - PSys^{MS})$) compared to [10], [13], it has worse utilization and mode switching probability and consequently the score, compared to GNN-MiCS. GNN-MiCS provides higher utilization and fewer mode switches by obtaining an appropriate $WCET^{LO}$ for HC tasks compared to other approaches. It improves utilization by up to 45.85% and 22.45% on average while maintaining a reasonable number of mode switches in the worst-case scenario compared to the other methods. Additionally, the ML model effectively balances utilization and mode switching across different core configurations (utilization is fixed for each number of cores (U/c =0.75)), optimizing the score for each configuration.

V. CONCLUSION AND FUTURE WORK

The article introduces a novel machine-learning-based approach for determining tasks' low WCETs, aiming to balance system utilization and mode-switching probability while ensuring task schedulability and timeliness. This approach enhances MC system utilization by up to 45.85% and 22.45% on average, while keeping mode switches reasonable in the worst-case scenario, compared to other methods.

In future research, we aim to create more practice-oriented graphs to assess the model's applicability in realistic scenarios. We will also explore the impact of neural network parameters on the training process and results, focusing on fine-tuning model calculations. Additionally, we plan to investigate the effects of varying system and application parameters, such as utilization and edge percentage.

REFERENCES

[1] S. Baruah, V. Bonifaci, G. DAngelo, H. Li, A. Marchetti-Spaccamela, S. van der Ster, and L. Stougie, "The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems," in *Proc. of ECRTS*, 2012.
[2] B. Ranjbar, A. Hoseinghorban, S. S. Sahoo, A. Ejlali, and A. Kumar, "Improving the timing behaviour of mixed-criticality systems using chebyshev's theorem," in *Proc. of DATE*, 2021.
[3] A. Burns and R. I. Davis, "A survey of research into mixed criticality systems," *ACM Comput. Surv.*, vol. 50, no. 6, nov 2017.
[4] D. Liu, J. Spasic, N. Guan, G. Chen, S. Liu, T. Stefanov, and W. Yi, "Edf-vd scheduling of mixed-criticality systems with degraded quality guarantees," in *Proc. of RTSS*. IEEE, 2016.
[5] B. Ranjbar, A. Hosseinghorban, S. S. Sahoo, A. Ejlali, and A. Kumar, "Bot-mics: Bounding time using analytics in mixed-criticality systems," *IEEE TCAD*, vol. 41, no. 10, 2022.
[6] B. Ranjbar, A. Hosseinghorban, M. Salehi, A. Ejlali, and A. Kumar, "Toward the design of fault-tolerance-aware and peak-power-aware multicore mixed-criticality systems," *IEEE TCAD*, vol. 41, no. 5, 2022.
[7] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra *et al.*, "The worst-case execution-time problem—overview of methods and survey of tools," *ACM TECS*, vol. 7, no. 3, 2008.
[8] C. Ballabriga, H. Cassé, C. Rochange, and P. Sainrat, "Otawa: An open toolbox for adaptive wcet analysis," in *Software Technologies for Embedded and Ubiquitous Systems*, S. L. Min, R. Pettit, P. Puschner, and T. Ungerer, Eds., Berlin, Heidelberg, 2010.
[9] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga, "Priority-based scheduling of mixed-critical jobs," *Real-Time Systems*, vol. 55, no. 4, 2019.
[10] J. Choi, H. Yang, and S. Ha, "Optimization of fault-tolerant mixed-criticality multi-core systems with enhanced wcrt analysis," *ACM TODAES*, vol. 24, no. 1, 2018.
[11] V. Kumar, B. Ranjbar, and A. Kumar, "Esomics: Ml-based timing behavior analysis for efficient mixed-criticality system design," *IEEE Access*, vol. 12, pp. 67 013–67 024, 2024.
[12] B. Ranjbar, A. Hosseinghorban, and A. Kumar, "Adaptive: Agent-based learning for bounding time in mixed-criticality systems," in *Proc. of DAC*, 2023.
[13] R. Medina, E. Borde, and L. Pautet, "Availability enhancement and analysis for mixed-criticality systems on multi-core," in *Proc. of DATE*, 2018.