Latency and reliability improvements of high critical tasks in mixed criticality systems

Hakduran Koc and Vamsi Krishna Karanam

Received: July 18, 2020. Revised: August 29, 2020. Accepted: September 4, 2020. Published: September 15, 2020.

***Abstract*—**Reliability and execution latency of high critical tasks are crucial for a successful execution in a mixed criticality system with tight design constraints. In this paper, we focus on two main design problems, namely latency-constrained maximum reliability problem and reliability-constrained minimum latency problem, for the applications running tasks of different criticality. The target architecture can run in two operating modes: low criticality mode (normal operating mode) and high criticality mode. For the former problem, we first find the minimum execution latency assuming the system runs in low criticality mode. Then, using this latency as lower bound, we present a heuristic algorithm to improve the reliability of high critical tasks in the application. The proposed algorithm assigns high critical tasks to the highest reliable processing elements in the technology library, and then, schedules low critical tasks without exceeding the given latency constraint. Similarly, for the latter problem, we first determine the highest reliability assuming the system runs in low criticality mode. Then, considering the overall system reliability, the proposed approach reduces the latest completion time of high critical tasks by giving them priority over low critical ones when selecting processing elements. The experimental evaluation conducted using task graphs shows up to 14.69% reliability improvement and 20.05%, on the average, latency improvement for the high critical tasks in the system.

Keywords*—Mixed criticality system, system reliability, execution latency, hardware/software co-design, scheduling, task mapping, critical task*.

I. INTRODUCTION

MIXED criticality systems (MCS) have become widely used in recent years as they can execute various hardware and software tasks of different criticality (e.g., safety critical and mission critical). They became an integral part of today's daily life and can be found almost everywhere such as aircrafts, cars, automated machines, and remotely piloted vehicles. In such a system, while a mission critical task typically prioritizes the system goal to obtain the results targeted, a safety critical task works to keep the system safe and prioritizes system survival. Significant amount of research can be found in the literature to improve various metrics of MCSs such as reliability, performance, and power consumption [1].

Hakduran Koc is with Computer Engineering Program at University of Houston-Clear Lake, Houston, TX 77058 USA (corresponding author, phone: +1 281-283-3877; fax: 281-226-7608; e-mail: KocHakduran@uhcl.edu).

Vamsi Krishna Karanam was with Computer Engineering Program at University of Houston-Clear Lake, Houston, TX 77058 USA (e-mail: KranamV1321@uhcl.edu).

The application running on a mixed criticality system consists of tasks of different criticality. The criticality of a task denotes the impact it has on the overall system output and may change over time during the execution. In this work, we classify tasks as High Critical (HC) or Low Critical (LC) ones. An effective task scheduling and binding involve classification in order to ensure the optimal use of available resources in the most efficient way. The system arranges tasks based on its level of criticality within the given constraints following dual criticality scheduling. Even though the reliability of HC tasks is generally given more importance over their execution times, reducing the latest completion time for those tasks becomes crucial in some scenarios given that the execution latency of the system is kept at a certain level (e.g., some time-sensitive military applications). In addition, tasks changing the criticality levels in the course of execution brings additional challenges in the task scheduling and binding process in terms of execution latency and reliability.

The system utilized in this work runs in two different operating modes: low criticality mode (normal mode of operation) and high criticality mode. In low criticality mode, all tasks are treated to be of equal criticality, and they are scheduled accordingly in order to meet the design constraints (execution latency and system reliability). In high criticality mode, the HC tasks are held in priority when scheduling the tasks. Some LC tasks may be excluded from scheduling to meet the target optimization metric. The target architecture is a hardware/software co-design environment with two CPUs and two ASICs. Each task can be executed by any processing element with varying reliability and latency requirements.

In this work, we investigate two major design problems, namely latency-constrained maximum reliability problem and reliability-constrained minimum latency problem, and present a new heuristic algorithm for each. Given the task graph representing the application, the proposed algorithms partition the tasks in high or low critical ones while prioritizing high critical tasks based on reliability or latency during the scheduling process. The system is allowed to transition between high criticality and low criticality operation modes if dictated by the requirements. The data is provided to the algorithms which start the scheduling process by choosing the tasks based on the predefined rules of priority and assigning them to the appropriate processing elements within the system. These rules take into consideration the number of tasks dependent on the current task, criticality, and the distance from sink node to compare the priority of the task in

question. In the event of multiple tasks meeting all conditions equally, the task with the lowest latency on the available processing element is scheduled.

More specifically, the proposed approach for the former problem first schedules all tasks in the system assuming all tasks are of equal reliability (low criticality execution mode) and returns the final reliability and the latency of the system. In high criticality mode, the algorithm first considers HC tasks during the scheduling process. After scheduling all HC tasks, LC tasks are scheduled in the gaps available before the execution deadline. The algorithm runs until all LC tasks are scheduled or it may exclude some LC tasks depending on overall system requirements. The algorithm returns the overall reliability of the system and the reliability of the high critical tasks in both the modes of operation. By prioritizing HC tasks, we increase system viability by ensuring the completion of high critical tasks early while incorporating low critical tasks where possible.

Keeping in mind that the main goal for the latter problem is to reduce the execution latency (the latest completion time) of HC tasks, the proposed method first schedules all tasks in the system and calculates the final latency and reliability of the system in low criticality execution mode. Then, in high criticality mode, the algorithm prioritizes HC tasks for scheduling. After ensuring all HC tasks are scheduled, the LC tasks are scheduled in the idle time internals. The algorithm runs untill all LC tasks are scheduled or it can exclude some LC tasks depending on overall system requirements. The algorithm finally returns the overall latency of the system and the latency of the high critical tasks in both modes of operation. The proposed algorithm also considers the changes on the criticality levels of the tasks in the course of execution due to the system goal, operating environment or an approaching emergency situation.

The rest of the paper is organized as follows: Section II gives the literature review, Section III describes the elements that form the system, Section IV shows an illustrative example for reliability and execution latency calculations using the technology library, Section V explains the details of the proposed algorithms, Section VI presents the experimental evaluation, and finally, Section VII concludes the paper.

II. Related Work

Early studies into mixed criticality systems noted that the application specific integrated circuits enhance the performance of the system at a really high cost. In systems that utilize HW/SW co-design, the repeatedly executed tasks typically dictate the entire execution time [7]. Distinguishing a particular node within the hot path that decreases overall latency once it is allotted to a hardware element is presented in [8]. The planned fault tolerance strategies were check pointing, rollback recovery, and active replication time redundancy and area redundancy [11]. A quasi-static scheduling strategy was studied based on fault occurrence and execution time in [14].

As fault tolerance became more common, a brand new metric of criticality was introduced to task scheduling while permitting tasks to alter criticality over time [1]. If a task exceeded its execution time limit, it moves the system into high criticality mode untill all high critical tasks were completed [5]. The zero slack approach makes an attempt to reduce the occurrences of high criticality tasks preempting low criticality tasks [17]. When utilized in conjunction with a priority-based preventive scheduling algorithm, this proves to be an efficient method.

Priority assignment in multiprocessor real time systems is employed in [20]. Research into multicore systems found an underutilization because of high WCET's and therefore the requirement for temporal isolation of critical tasks [15]. The underutilization was further studied and a scheduling algorithm was designed in [4]. The proposed algorithm scheduled critical tasks while employing a constant bandwidth server to schedule low critical tasks in [10]. An alternate metric of processor acceleration factor was presented and the effectiveness of a reservation-based scheduling and priority-based scheduling were tested in [12]. A scheduling algorithm from a single-core processor to a multicore process system is given in [16].

While multicore processors become widely used, task scheduling should also consider potential core necessities with completely different scheduling implementations [2]. For a selected mission or safety critical application situation dependability attributes are the first concern. Transient or soft errors do not affect the hardware permanently; however, may still have an effect on the result [13]. Multicore processors conjointly led to the introduction of federated scheduling, wherever each individual task is either restricted to execute on a selected processor or has exclusive access to all processors [6]. Many programming algorithms were given in [9], and a UML model was created while considering concurrency, completely different resource allocations, and multiple platform configurations [18].

Sha et al. [21] discussed how mode changes can be accommodated within a given framework of priority driven real-time scheduling. Schneider et al. [22] presented a multi-layered schedule synthesis scheme for MCCPS that aims to jointly schedule deadline-critical and QoC-critical tasks at different scheduling layers. Zhou et al. [23] proposed a design framework comprising a hyper-period optimization algorithm, which reduces the size of the schedule table and preserves schedulability, and a re-scheduling algorithm to reduce the number of preemptions. Zeng et al. [24] presented design methodologies to guarantee both safety and schedulability for real-time mixed-criticality systems on identical multicores. Assuming hardware/software transient errors and model safety requirements on different criticality levels explicitly according to safety standards, they further propose fault-tolerant mixed-criticality scheduling techniques with task replication and re-execution to enhance system safety. Pathan et al. [25] came up with an effective scheduling policy that can guarantee certification of the system at each criticality level while

maximizing the utilization of the processors. Müller et al. [26] reviewed EDF-VD's schedulability criteria and determined its schedulability region to better understand and design mixed-criticality systems. Maurer et al. [27] presented a generic component and communication model for CPS that not only allows the coexistence of computing paradigms of different criticality but also supports the data exchange between them.

Research into criticality led to the idea of dual criticality involving high critical and low critical tasks. Once a HC task exceeds its assigned execution time, the algorithmic rule switches HC mode and all LC tasks are abandoned [3]. The trend shifted to utilizing the network on chip architectures to overcome these problems as the number of cores increased. The designed reliability-aware task scheduling on NOC primarily based platform uses changed clustered replication with the bulk voting to attain reliability [19].

Koc et al. [28] studied latency-constrained task mapping to improve reliability of HC tasks. Previous research into MCSs has primarily centered on task scheduling which supported hardware/software co-design restrictions, task criticality, processor accessibility, or fault tolerance. This paper extends the discussions in [28] and proposes two new heuristic algorithms to reduce the execution latency of HC tasks based on task scheduling.

III. Preliminaries

A. Target Architecture Model

The target architecture is a multicore HW/SW co-design environment with four processing elements along with a shared memory and a synchronization/communication unit as shown in Figure 1. Processing elements include two software components (CPU 1 and CPU 2) and two hardware components (ASIC 1 and ASIC 2). The synchronization and communication unit is the circuitry that manages the communication among the components through the memory with a uniform access latency. The inputs are read from memory when a task is about to start executing and the result is stored in memory just after a task finishes its execution. For the sake of simplicity (and in order to not deviate from the main purpose of this paper), we assume a uniform access latency for each memory read and write, and include the memory access latencies into the execution latencies of tasks.

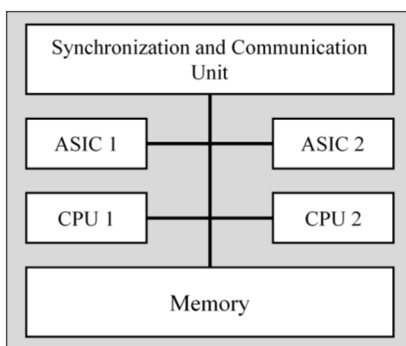

Fig. 1 The target HW/SW co-design environment with two CPUs, two ASICs, memory, and synchronization /communication unit.

Each task can be executed by any processing element in the system with varying latency and reliability values. ASICs typically execute tasks much faster than CPUs. The reliability and execution latency values of each task when running on different processing element is given in a technology library.

B. Application Modelling

The functionality of an application running on the system is graphically represented using a task graph. A task graph is a directed acyclic graph, $G_s(V, E)$. Each vertex node, $V = \{v_i;\ i = 0, 1, 2, 3, \dots, n\}$ represents an independent task performing various operations. The intercommunication between tasks is denoted by edges. The edges represent dependencies between tasks, $E = \{(v_i, v_j);\ i, j = 0, 1, \dots, n\}$. A task can start executing only after all its predecessors complete their executions. The result of a predecessor task is passed to its successors upon completion. Tasks are executed on various processing elements.

An example task graph is shown in Figure 2. Each node in the task graph denotes a task in the system while an edge represents the data dependency between predecessors and successors within the graph. This dependency implies that a task cannot start executing before all its predecessors complete their executions. For example, T13 cannot be scheduled to execute before T4, T11, T5 and T12 complete their executions. Each task is mapped to a processing elements in the target architecture during task scheduling. Source and sink vertices are used for synchronization and have zero execution time. While LC tasks are illustrated by white nodes, the nodes representing HC tasks are filled with color. Note that if a task is marked critical, all its predecessors would be deemed critical as well to avoid discrepancies in data dependency.

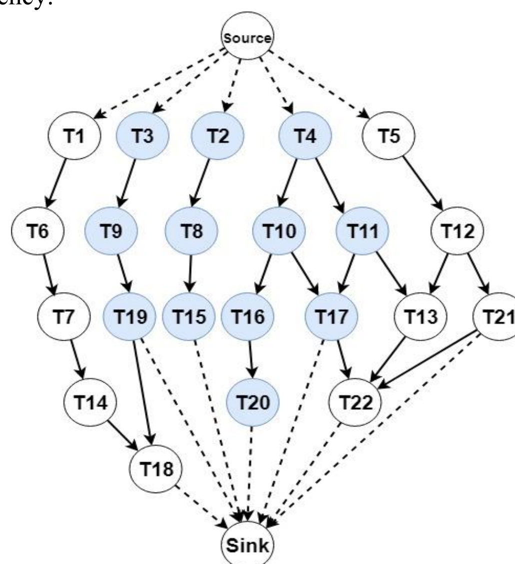

Fig. 2. Example Task Graph

C. Mixed Criticality System

A mixed criticality system can be defined as a group of tasks (with different criticality) interconnected with one another and working towards generating an output. Each MCS

includes a combination of critical and non-critical tasks (e.g., HC and LC tasks) that contribute to success. Execution of critical tasks ensures that the system works safely and produces output whereas execution of non-critical tasks improves performance of the system result. It can also be expressed that critical tasks are essential to the survival of the system whereas non-critical tasks improve the effectiveness of the system.

Criticality Modes of a Task: In this work, we classify a task as high critical task or low critical task. This helps us in the scheduling process as our system allows a task to change its criticality during the execution. Criticality additionally helps confirm the importance of the task within the system. We specify the condition that a HC task cannot have a LC task as a predecessor. In Figure 3, the tasks in Figure 2 are grouped by their criticality. HC tasks are colored in.

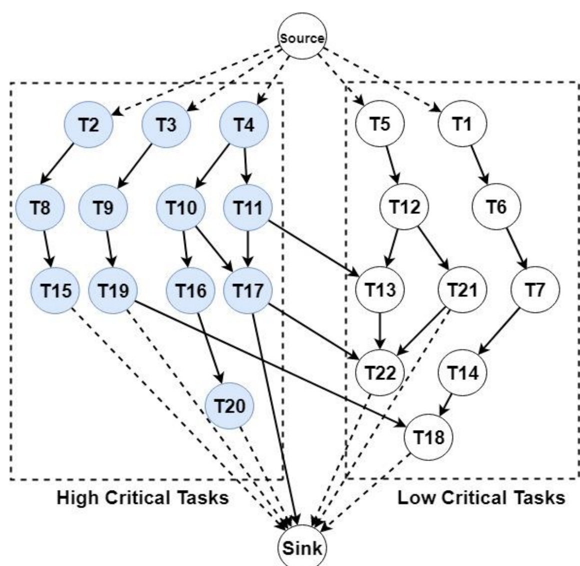

Fig. 3. Re-arranged example task graph with HC tasks (blue) and LC tasks (white). Some LC tasks depend on HC tasks.

Criticality Modes of the System: The MCS utilized in this work operates in two different operating modes: high criticality mode and low criticality mode. Low criticality mode considers all tasks to be of equal priority and schedules them accordingly. Low criticality mode can be considered as normal operational mode for the system. The system transitions into high criticality mode based on predefined conditions such as a) when a crisis or emergency approaches, b) when a high critical task error occurs, c) when the system is not able to schedule all tasks within given constrains, or d) whenever a manual override is deemed necessary. Once the system is in high criticality mode, it ignores all low critical tasks for the present and schedules the high critical tasks in line with their priority assignment. Finally, after all HC tasks are scheduled, LC tasks are placed within the slack generated between high critical tasks.

D. Reliability Modeling

Reliability is defined by how efficiently a task can run on a specific processor, which varies for each task, when running on each of the processing elements in the target architecture. Each and every task must be executed correctly for a successful execution of the entire system. So, the overall reliability of a system with multiple units is determined by the product of the reliabilities of individual tasks in the system as shown in (1).

$$R_{overall} = \prod_{i=1}^{n} R_i \tag{1}$$

Equation (1) suggests that the individual reliability of the tasks should be improved in order to enhance the overall reliability of the system. This may be achieved by adding a redundant element to each task (i.e., parallel configuration). The elements in this configuration are connected in parallel manner within which the input divides into all the components and the outputs of the components combine into one. Reliability of such a parallel system is calculated by (2).

$$R_{parallel} = 1 - \prod_{i=1}^{n} (1 - R_i) \tag{2}$$

E. Design Constraints

This paper addresses two main design problems: latency-constrained maximum reliability problem and reliability-constrained minimum latency problem. For the first problem, we find the minimum execution latency (using Algorithm 1) assuming the system runs in low criticality mode. Then, this latency is used (by Algorithm 2) as an upper bound when finding the most reliable design solution. Similarly, for the second problem, we first find the most reliable design assuming the system runs in low criticality mode, and then, this reliability value is used as lower bound for the system reliability when determining the design that provides the minimum completion time for the high critical tasks in the system. Please note that the algorithms will be explained in detail later.

F. Technology Library

Each task can run on any available processing element (CPU 1, CPU 2, ASIC 1, or ASIC 2). The reliability and execution latency values of each task on each processor are tabulated into a technology library. The technology library for the tasks T1 to T22 in the sample task graph in Figure 2 is

TABLE I. TECHNOLOGY LIBRARY FOR TASKS IN FIGURE 2. BLUE IS FOR HC TASKS AND WHITE IS FOR LC TASKS. R: RELIABILITY, L: LATENCY.

		T1	T2	T3	T4	T5	T6	T7	T8	T9	T10	T11	T12	T13	T14	T15	T16	T17	T18	T19	T20	T21	T22
ASIC2	**R**	0.997	0.998	0.994	0.995	0.985	0.979	0.993	0.996	0.988	0.992	0.998	0.969	0.989	0.979	0.978	0.998	0.974	0.976	0.990	0.997	0.997	0.995
	L	10	12	9	8	10	11	12	12	7	13	16	12	12	15	6	12	9	6	12	15	14	10
ASIC1	**R**	0.995	0.996	0.998	0.992	0.991	0.994	0.979	0.97	0.986	0.955	0.991	0.987	0.993	0.971	0.990	0.991	0.971	0.998	0.993	0.995	0.986	0.993
	L	8	10	12	6	7	9	10	15	5	17	12	7	10	10	12	8	9	8	8	12	8	11
CPU2	**R**	0.993	0.980	0.980	0.977	0.975	0.976	0.999	0.995	0.98	0.989	0.985	0.990	0.987	0.973	0.985	0.987	0.973	0.989	0.980	0.996	0.993	0.984
	L	55	45	30	65	55	35	35	50	65	30	50	30	35	20	55	55	30	25	65	40	25	30
CPU1	**R**	0.994	0.994	0.986	0.987	0.989	0.969	0.995	0.993	0.972	0.985	0.990	0.978	0.996	0.991	0.987	0.983	0.970	0.985	0.990	0.995	0.987	0.995
	L	50	40	35	45	35	50	25	25	35	35	30	45	25	40	50	60	35	30	50	35	45	50

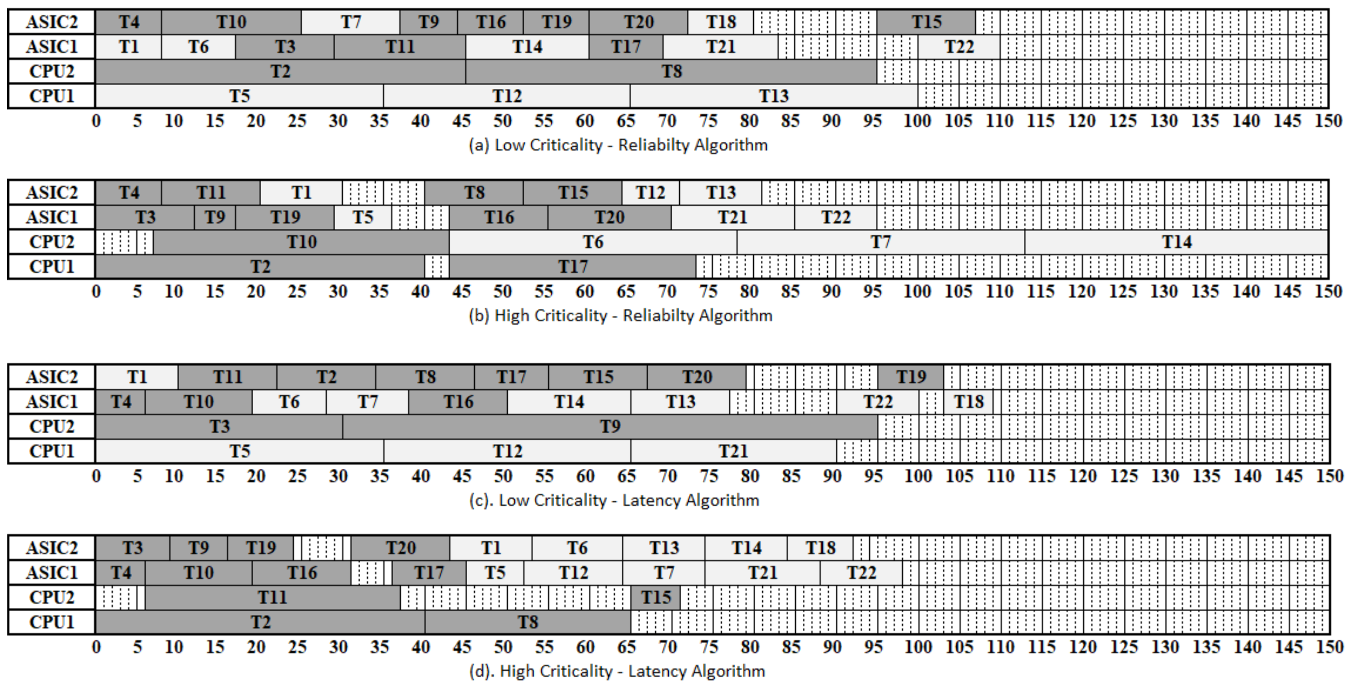

Fig. 4. Scheduling of the task graph in Fig. 3 generated by the proposed approaches: a) Initial schedule to find the minimum execution latency; b) The schedule that provides the maximum reliability for HC tasks; c) Initial schedule to find the maximum reliability; d) The schedule for minimum completion time of HC tasks in the system.

given in Table 1. The library contains the data that is utilized by the algorithms to decide which processor is the best to achieve the target reliability or execution latency constraint. The latency values are given in clock cycles (e.g., executing T1 using ASIC 2 and CPU 2 takes 10 clock cycles and 55 clock cycles, respectively). In the table, R is for reliability and L is for execution latency. Blue columns represent the high critical tasks and white columns are for the low critical tasks.

IV. Motivational Example

Let us consider the task graph given in Figure 2 and its associated technology library in Table 1. The graph contains 22 tasks (12 of them are HC) interconnected with each other from source to sink. HC and LC tasks are later grouped separately in Figure 3. The technology library provides us the reliability and execution time of each task on each processor.

The approach proposed for the latency-constrained maximum reliability problem first starts with scheduling the source vertex to clock cycle 0. Please note that source and sink nodes have execution time of zero cycles. Then, T1, T2, T4 and T5 are selected by the algorithm among all successors of the source vertex based on the distance of each successor to the sink vertex (i.e., the nodes with higher distance is selected first). The selected nodes are mapped to the available processing elements ASIC 1 (T1 - 8 cycles), ASIC 2 (T4 - 8 cycles), CPU 1 (T5 - 35 cycles), and CPU 2 (T2 - 45 cycles). It waits for a processor to free up to schedule the tasks in queue and waits for each task's predecessors to be executed before the task is scheduled. The algorithm runs until the sink vertex is scheduled. The schedule generated by the algorithms are represented in Figure 4. The darker rectangles represent HC tasks and the white ones represent LC tasks. In Fig. 4(a), the tasks are scheduled assuming the system is in low criticality mode. It is observed that LC tasks are scheduled alongside HC tasks that make some HC tasks to be scheduled later in the order. This method still allocates all the tasks to their highest reliable processor available. In Fig. 4(b), scheduling is done following the high criticality mode of the system. This scheduling order achieves lower latency and higher reliability for HC tasks fulfilling the target intended. It is also observed that some LC tasks go beyond the intended deadline. These tasks are dropped to reduce latency as they do not affect the success of the system. The latency and reliability results for this illustrative example are given in Table 2 under the reliability priority scheduling. Note that LC mode represents the normal operating mode where all tasks have the same level of criticality while HC tasks are given priority when the system runs in HC mode. The column labeled "Overall" give the reliability and latency values for all tasks; the column labelled "HC tasks" presents the values only for high criticality tasks. The reliability values are calculated using the series configuration in (1). The results show 5% reliability improvement (0.854 to 0.897) as well as 46% latency improvement for HC tasks. It can also be seen that, when we consider all tasks, the algorithm shows an improvement in the reliability of the high criticality mode at the cost of latency. If a few low critical tasks can be dropped, this latency overshoot can be rectified (i.e., T7 and T14).

Similarly, the approach proposed for the reliability-constrained minimum latency problem first schedules the source vertex to clock cycle 0. Then, T1, T4, T3 and T5 are selected by the algorithm among all successors of the source

TABLE II. THE RELIABILITY AND EXECUTION LATENCY VALUES FOR THE ILLUSTRATIVE EXAMPLE FOR THE SAMPLE TASK GRAPH.

Properties	Reliability Priority Scheduling				Latency Priority Scheduling			
	LC mode		HC mode		LC mode		HC mode	
	Overall	HC tasks	Overall	HC tasks	Overall	HC tasks	Overall	HC tasks
Reliability	0.786	0.854	0.831	0.897	0.743	0.837	0.776	0.887
Latency	110	107	162	73	107	79	97	71

vertex based on the distance of each successor to the sink vertex. The selected nodes are mapped to the available processing elements ASIC 1 (T4 - 6 cycles), ASIC 2 (T1 - 10 cycles), CPU 1 (T5 - 35 cycles), and CPU 2 (T3-30 cycles). Then, once a processing elements becomes available, the task with longest delay to the sink is scheduled if all its predecessors have completed their executions. This process continues until all tasks are scheduled. In Figure 4(c), the tasks are scheduled assuming the system is in low criticality execution mode. It is observed that the LC tasks are scheduled alongside HC tasks that make some high critical tasks to be scheduled later in the order. This method still allocates all the tasks to the fastest processor available. Figure 4(d) shows the schedule generated assuming the system runs in the high criticality mode. This scheduling order achieves faster latency and higher reliability for HC tasks. The latency and reliability results for this illustrative example are given in Table 2 under the latency priority scheduling. As seen in the table, the execution latency of HC tasks is reduced by 11% (79 to 71 cycles) while the latency of all tasks goes down by 10 clock cycles. The table also shows that better reliability values are achieved by the proposed algorithm for both HC tasks and all tasks when the systems is running in high criticality mode.

V. DETAILS OF PROPOSED APPROACH

In this section, we will explain the details of the algorithms proposed as solutions to the two main design problems investigated in this paper: the latency-constrained maximum reliability problem and the reliability-constrained minimum latency problem. Algorithm 1 is used to find the minimum execution latency assuming the system runs in low criticality mode (i.e., all tasks are of the same level of criticality). This latency is used as an input for Algorithm 2 that is proposed as solution to the latency-constrained maximum reliability problem in this paper. After changing the selection criteria and the target optimization metric in the proposed two algorithms, the modified version of Algorithm 1 is used to find the maximum reliability value assuming the system runs in low criticality mode. This reliability value is used as input to the modified version of Algorithm 2 that is proposed as a solution to the reliability-constrained minimum latency problem.

A. Latency-Constrained Maximum Reliability Problem

Algorithm 1 is developed to schedule the tasks when the system is in low criticality mode while Algorithm 2 is used to schedule the tasks when the system is in high criticality mode. Algorithm 2 is referred as Reliability Priority Algorithm. Each algorithm first lists out all the tasks available for scheduling into set T. Then, it sorts the tasks based on the number of successors. Considering the inputs from the technology library, the algorithm chooses the task with the greatest number of successors and schedules it. This decision is made in this way because the distance from each task to the sink cannot be determined until all tasks are scheduled. By doing so, our goal is that the next tasks in line would be available earlier and the longer path getting stuck in slower processors is avoided.

Once the first tasks are scheduled, the algorithm updates its set of available tasks as new tasks arrive and compares them with their priority. The priority of tasks in case of equal number of successors is determined by checking the following conditions in the given order: the task with higher criticality, the task with greater number of high critical successors, and the task nearest to the source. If there is still a tie, the task that can run with the highest reliability in the available processor is chosen. This order is devised to obtain higher reliability. If there is a task available to schedule, the algorithm waits until a processor becomes available. After all the tasks are scheduled to the most reliable processor available, the overall execution latency is calculated. When the system switches into high criticality mode, the LC tasks are not considered in the first

Algorithm 1: Scheduling with Low Criticality System Mode

INPUT: Task graph with 'n' tasks, Tech Library with (R_{ji}, L_{ji}) where $j \in J$ (1,4) refers to processors and $i \in I$ (1, n) denotes the tasks in task graph,
OUTPUT: Schedule of LC reliability priority mode, L_o, R_o, L_{hc}, R_{hc}
1. Update $I_{current}$ = {i | i are tasks that are available to be scheduled, $i \in I$}
2. Update $J_{current}$ = {j | j are processors that are idle, $j \in J$}
3. Form the set of 'I′' where
4. I = {i | i are the tasks with the greatest number of successors, $i \in I_{current}$}
5. **if** |I| = 1
6. **then** $T_{current} = T_i$
7. **else** I′ = I
8. I = {i | i is High Critical task, $i \in I'$}
9. **if** |I| = 1
10. **then** $T_{current} = T_i$
11. **else** I′ = I
12. I = {i | i is nearest to the source, $i \in I'$}
13. **if** |I| = 1
14. **then** $T_{current} = T_i$
15. **else** I′ = I
16. I = {i | i has greatest $R_j(i)$, $i \in I'$ & $j \in J_{current}$}
17. **if** |I| = 1
18. **then** $T_{current} = T_i$
19. Assign $T_{current}$ to $j \ni R_j(i)$ is the highest, $j \in J_{current}$
20. **if** | $J_{current}$ | = 0
21. **then** wait for time T
22. Update $I_{current}$ & $J_{current}$
23. **if** | $I_{current}$ | = 0
24. **then** wait for time T
25. Update $I_{current}$ & $J_{current}$
26. Repeat step 4 until | I | = 0 i.e. all the tasks are scheduled

Algorithm 2: Scheduling with High Criticality System Mode

INPUT: Task graph with 'n' number of tasks, Tech Library with (R_{ji}, L_{ji}) where j ∈ J (1,4) refers to processors and I ∈ I (1, n) denotes the tasks in task graph,
OUTPUT: Schedule of Low Criticality reliability priority mode, L_o, R_o, L_{hc}, R_{hc}

1. Update $H_{current}$ = {h | h are High Critical tasks that are available to be scheduled, h ∈ I}
2. Update $J_{current}$ = {j | j are processors that are idle, j ∈ J}
3. Form a set of 'H' where
4. H = {h | h are the tasks with the greatest number of successors, h ∈ $H_{current}$}
5. **if** |H| = 1
6. **then** $T_{current}$ = T_h
7. **else** H′ = H
8. H = {h | h is nearest to the source, h ∈ H′}
9. **if** |H| = 1
10. **then** $T_{current}$ = T_h
11. **else** H′ = H
12. H = {h | h has greatest R_j(h), h ∈ H′ & j ∈ $J_{current}$}
13. **if** |H| = 1
14. **then** $T_{current}$ = T_h
15. Assign $T_{current}$ to j ∋ R_j(h) is the highest, j ∈ $J_{current}$
16. **if** | $J_{current}$ | = 0
17. **then** wait for time T
18. Update $H_{current}$ & $J_{current}$
19. **if** | $H_{current}$ | = 0
20. **then** wait for time T
21. Update $H_{current}$ & $J_{current}$
22. Repeat step 3 until | H | = 0 i.e. all the High Critical tasks are scheduled
23. Form a set of tasks $L_{current}$ where
24. $L_{current}$ = {l | l are the remaining tasks that are available to be scheduled with greatest number of successors, l ∈ I}
25. **if** |L| = 1
26. **then** $T_{current}$ = T_l
27. **else** L′ = L
28. L = {l | l is nearest to the source, l ∈ L′}
29. **if** |L| = 1
30. **then** $T_{current}$ = T_l
31. **else** L′ = L
32. L = {l | l has greatest R_j(l), l ∈ L′ & j ∈ $J_{current}$}
33. **if** |L| = 1
34. **then** $T_{current}$ = T_l
35. Assign $T_{current}$ to j ∋ R_j(l) is the greatest, j ∈ $J_{current}$
36. **if** | $J_{current}$ | = 0
37. **then** wait for time T
38. Update $L_{current}$ & $J_{current}$
39. **if** | $L_{current}$ | = 0
40. **then** wait for time T
41. Update $L_{current}$ & $J_{current}$
42. Repeat step 23 until | L | = 0 i.e. all the tasks are scheduled

cycle for the scheduling process. All HC tasks are scheduled first following the order with the task having the greatest number of successors scheduled first. After all HC tasks are scheduled, the algorithm processes LC tasks and completes the scheduling.

More specifically, Algorithm 1 takes the task graph representing the application and the technology library as inputs. In steps 1 to 4, it compiles the data and arranges tasks in the order of predecessors. In steps 5 to 18, the algorithm selects the task that has the highest priority and assigns it to the suitable processor. Then, the algorithm loops back to select the suitable processor. Then, the algorithm loops back to select the next task to be scheduled. In Algorithm 2, the first four steps categories the tasks as high critical and low critical (e.g., construction of the graph in Figure 3). Then, the steps 5 to 15 select the highest critical task and assigns it to the most reliable processor. Steps 16 to 22 are repeated until all high critical tasks are scheduled. The steps from 23 until the end schedule the rest of the tasks accordingly until all tasks are scheduled.

B. Reliability-Constrained Minimum Latency Problem

After changing the selection criteria of the tasks to be scheduled and the target optimization metric for Algorithm 1 and Algorithm 2, they can be used to solve the reliability-constrained minimum latency problem. More specifically, a) the tasks that can run with the smallest latency in the available processor should be selected (instead of the tasks that can run with the highest reliability); b) the tasks should be scheduled to the fastest processors (instead of the most reliable processor); and, c) the highest critical task should be assigned to the fastest processor (instead of the most reliable processor). The modified version of Algorithm 2 is referred as Latency Priority Algorithm.

Now let us list the specific changes in Algorithm 1 and Algorithm 2 in order to modify them for the reliability-constrained minimum latency problem. The following changes are needed in Algorithm 1:

Step 16: I = {i | i has lowest L_j(i), i ∈ I′ & j ∈ $J_{current}$}
Step 19: Assign $T_{current}$ to j ∋ L_j(i) is the lowest, j ∈ $J_{current}$

In addition, the following modifications are needed in Algorithm 2:

Step 12: H = {h | h has lowest L_j(h), h ∈ H′ & j ∈ $J_{current}$}
Step 15: Assign $T_{current}$ to j ∋ L_j(h) is the lowest, j ∈ $J_{current}$
Step 32: L = {l | l has lowest L_j(l), l ∈ L′ & j ∈ $J_{current}$}
Step 35: Assign $T_{current}$ to j ∋ L_j(l) is the lowest, j ∈ $J_{current}$

VI. Experimental Evaluation

In this section, we present the experiment evaluation for the proposed algorithms: Reliability Priority Algorithm for the latency-constrained maximum reliability problem and Latency Priority Algorithm for the reliability-constrained minimum latency problem. Assuming the target architecture described in Section 3 (i.e., HW/SW co-design architecture with two CPUs and two ASIC components along with a synchronization and communication unit), the algorithms are tested using task graphs generated by TGFF tool. Each graph has a unique technology library. The task graphs are named as TG1 (22 tasks), TG2 (16 tasks), TG3(25 tasks), TG4 (31 tasks), TG5 (43 tasks), TG6 (28 tasks), TG7(36 tasks), and TG8 (40 tasks). A detailed experimental evaluation using these eight graphs is presented in Table 3 and Table 4. The first two columns in these tables give the labels of the task graphs and the number of tasks in each graph. The next four columns report the reliability values and the execution latencies for each graph in low criticality execution mode for all tasks and only HC tasks. The next four columns report the same when the system runs in high criticality execution mode. The last two columns in Table 3 present the reliability improvements in percentage considering all tasks and only HC tasks for Reliability Priority Algorithm while the last two columns in

TABLE III. THE RELIABILITY IMPROVEMENTS FOR THE LATENCY-CONSTRAINED MAXIMUM RELIABILITY PROBLEM

Task Graph	Number of tasks	Reliability Priority Algorithm								% Reliability Improvement	
		Low Criticality Mode				High Criticality Mode					
		Reliability		Latency		Reliability		Latency		Overall system	High Critical tasks
		Overall	HC task	Overall	HC task	Overall	HC task	Overall	HC task		
TG1	22	0.7863	0.8548	110	107	0.8317	0.8978	162	73	6.41	4.71
TG2	16	0.7839	0.9263	155	155	0.7879	0.9281	157	157	0.51	0.19
TG3	25	0.5680	0.7301	164	162	0.6440	0.8067	255	125	13.38	10.49
TG4	31	0.6459	0.7789	243	173	0.6933	0.8933	233	171	7.34	14.69
TG5	43	0.6097	0.7549	205	205	0.6518	0.7773	245	160	6.91	2.97
TG6	28	0.6526	0.7961	201	184	0.6648	0.8682	219	173	1.87	11.57
TG7	36	0.6192	0.7813	195	187	0.6215	0.8598	239	162	0.37	10.05
TG8	40	0.6745	0.7382	225	219	0.6939	0.8141	252	177	2.88	10.28

TABLE IV. THE EXECUTION LATENCY IMPROVEMENTS FOR THE RELIABILITY-CONSTRAINED MINIMUM LATENCY PROBLEM

Task Graph	Number of tasks	Latency Priority Algorithm								% Latency Improvement	
		Low Criticality Mode				High Criticality Mode				Overall system	High Critical Tasks
		Reliability		Latency		Reliability		Latency			
		Overall	HC task	Overall	HC task	Overall	HC task	Overall	HC task		
TG1	22	0.7431	0.8378	107	79	0.7769	0.8870	97	71	9.35	10.13
TG2	16	0.6963	0.8503	103	72	0.7114	0.8546	103	60	0	16.67
TG3	25	0.5314	0.7183	110	110	0.4742	0.6644	160	68	-45.45	38.18
TG4	31	0.5366	0.7122	205	200	0.5536	0.7355	178	116	13.17	42.0
TG5	43	0.5742	0.7045	157	135	0.5595	0.7216	200	118	-27.38	12.0
TG6	28	0.5812	0.7513	147	127	0.5634	0.7643	152	113	-3.40	11.02
TG7	36	0.5915	0.7441	165	145	0.5841	0.7145	161	126	2.42	13.10
TG8	40	0.5435	0.7112	185	185	0.5356	0.6571	192	153	-3.78	17.29

Table 4 present the latency improvements for Latency Priority Algorithm.

The gray columns in Table 3 report the reliability values and the reliability improvement for each graph. As seen in the last column, the proposed reliability priority algorithm is successful in increasing the reliability of the HC tasks by 8.12% on the average (in the range of 0.19% and 14.69%). Please note that the results of the proposed algorithm are compared with those of Algorithm 1. Algorithm 1 is a reliability-centric algorithm which, in a sense, represents state-of-the-art. Prioritizing HC tasks over others resulted in lesser holdup times for HC tasks and making more reliable processors to be available when they are scheduled. We compare the HC task reliabilities in both system modes for each task graph to obtain the reliability improvement as shown in the gray columns in the table. The algorithm aims at increasing the HC tasks' reliability to ensure the system success in critical times and emergencies. Scheduling LC tasks after HC tasks gives HC tasks better access to reliable processors, thereby inducing an improvement in HC task reliability. The improvement is consistent in tasks graphs with higher number of task and in graphs with fewer tasks.

TG1 is our sample task graph. TG2 has a HC task reliability improvement of just 0.19% due to the low number of tasks available (only 16 tasks) to schedule at any given point in the execution. Most of the tasks have more reliable processors readily available in both operating modes. Although the algorithm returns high reliability, the improvement of HC tasks' reliability is low. We also observe in some cases that when the system reliability is increased, it comes at the cost of an increased latency (e.g., TG3 and TG5). Assigning HC tasks to highly reliable but slower processors cause this. This overall latency overhead can be avoided as the LC tasks can be dropped to run within the time boundary. This does not affect the primary goal of the algorithm which is improving the reliability of HC tasks in the system. The algorithm generates higher reliability improvements for TG4 (14.69%) and TG6 (11.57%) as the lower dependency requirements of these graphs allows higher reliable components to be assigned to HC tasks. The algorithms also keep track of high critical task latency as achieving reliability at an enormous cost to latency is not a viable option. Another observation is that the overall system reliability typically decreases as the number of tasks increase as discussed in reliability modelling.

Table 4 presents the execution latency and the latency improvement for each graph in the gray columns. The

proposed Latency Priority Algorithm (the modified version of Algorithm 2) improves the execution latency (latest completion time) of the HC tasks by 20.05% on the average (in the range of 10.13% and 42%). Please note that the results of Latency Priority Algorithm are compared with the modified version of Algorithm 1, which is, in a sense, represents state-of-the-art. Prioritizing HC tasks over others let the algorithm schedule HC tasks to faster processors (mostly ASICs). We compare the HC task execution latencies in both system modes for each task graph to obtain the latency improvement as shown in the gray columns. The improvement is consistent in all task graphs.

The experimental evaluation reports the lowest improvement (10.13%) for TG2, which has smallest number of tasks making small number of tasks available to be scheduled to the fastest processing elements. The graphs TG3 and TG4 gives the highest reliability improvements (38.18% and 42%, respectively) as they have relatively large number of HC tasks available to be assigned to faster processing elements. One interesting observation is that even though TG3 gives one of the highest latency improvements for HC tasks, the table reports the highest degradation for the overall system latency (HC tasks + LC tasks). This is mainly due to the dependency requirements of the task graph. Please note that the main goal for the proposed method is to reduce the execution latency (the latest completion time) of HC tasks. In real applications, the algorithm would drop LC tasks that exceeds the latency constraint which is set by the first algorithm. The algorithm also keeps track of high critical task reliabilities because a significant increase in reliability cost is not a viable option when achieving better execution times.

VII. Conclusion

In this paper, we proposed two algorithms, namely Reliability Priority Algorithm and Latency Priority Algorithm, as solutions to two main design problems, the latency-constrained maximum reliability problem and the reliability-constrained minimum latency problem, respectively, in the context of mixed criticality systems. The application composed of high critical and low critical tasks run on a hardware/ software co-design environment. For each problem, an initial algorithm determines the highest system reliability or the lowest execution latency assuming all tasks are of equal criticality. Then, this value is used as a constraint by the corresponding proposed algorithm. The experimental evaluation conducted clearly shows the viability of the proposed algorithms.

The future work includes focusing on behavior of high criticality tasks and investigating the scenarios where the criticality of tasks changes in during the execution when the system is in high criticality mode.

References

[1] A. Burns and R. I. Davis, "A Survey of Research into Mixed Criticality Systems," ACM Computing Surveys, vol. 50, no. 6, pp. 1–37, Nov. 2017.

[2] D. Tamas-Selicean and P. Pop, "Task Mapping and Partition Allocation for Mixed-Criticality Real-Time Systems," IEEE 17th Pacific Rim International Symposium on Dependable Computing, pp. 282–283, 2011.

[3] R. Medina, E. Borde, and L. Pautet, "Scheduling Multi-periodic Mixed-Criticality DAGs on Multi-core Architectures," IEEE Real-Time Systems Symposium (RTSS), pp. 254–264, 2018.

[4] M. Bagheri and G. Jervan, "Fault-Tolerant Scheduling of Mixed-Critical Applications on Multi-processor Platforms," 12th IEEE International Conference on Embedded and Ubiquitous Computing, pp. 25–32, 2014.

[5] S. Vestal, "Preemptive Scheduling of Multi-criticality Systems with Varying Degrees of Execution Time Assurance," 28th IEEE International Real-Time Systems Symposium (RTSS 2007), pp. 239–243, 2007.

[6] S. Baruah, "The Federated Scheduling of Systems of Mixed-Criticality Sporadic DAG Tasks," IEEE Real-Time Systems Symposium (RTSS), pp. 227–236, 2016.

[7] E. Azari and H. Koc, "Improving performance through path-based hardware/software partitioning," 5th International Conference on Digital Information Processing and Communications (ICDIPC), pp. 54–59, 2015.

[8] S. Tosun, N. Mansouri, E. Arvas, M. Kandemir, Y. Xie, and W.-L. Hung, "Reliability-Centric Hardware/Software Co-Design," 6th International Symposium on Quality of Electronic Design (ISQED), pp. 375–380, 2005.

[9] B. Nimer and H. Koc, "Improving reliability through task recomputation in heterogeneous multi-core embedded systems," The International Conference on Technological Advances in Electrical, Electronics and Computer Engineering (TAEECE), pp. 72–77, 2013.

[10] P. K. Saraswat, P. Pop, and J. Madsen, "Task Mapping and Bandwidth Reservation for Mixed Hard/Soft Fault-Tolerant Embedded Systems," 16th IEEE Real-Time and Embedded Technology and Applications Symposium, pp. 89–98, 2010.

[11] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Synthesis of fault-tolerant embedded systems with checkpointing and replication," 3rd IEEE International Workshop on Electronic Design, Test and Applications (DELTA06), pp. 5, pp. 447, 2006.

[12] S. K. Baruah et al., ''Scheduling real-time mixed-criticality jobs,'' IEEE Trans. Comput., vol. 61, no. 8, pp. 1140–1152, Aug. 2012

[13] C. Bolchini and A. Miele, "Reliability-Driven System-Level Synthesis for Mixed-Critical Embedded Systems," IEEE Transactions on Computers, vol. 62, no. 12, pp. 2489–2502, Dec. 2013.

[14] V. Izosimov, P. Pop, P. Eles, and Z. Peng, "Scheduling of Fault-Tolerant Embedded Systems with Soft and Hard Timing Constraints," Design, Automation and Test in Europe, pp. 915–920, 2008.

[15] M. S. Mollison, J. P. Erickson, J. H. Anderson, S. K. Baruah, and J. A. Scoredos, "Mixed-Criticality Real-Time Scheduling for Multicore Systems," 10th IEEE International Conference on Computer and Information Technology, pp. 1864–1871, 2010.

[16] H. Li and S. Baruah, "Global Mixed-Criticality Scheduling on Multiprocessors," 24th Euromicro Conference on Real-Time Systems, pp. 166–175, 2012.

[17] D. D. Niz, K. Lakshmanan, and R. Rajkumar, "On the Scheduling of Mixed-Criticality Real-Time Task Sets," 30th IEEE Real-Time Systems Symposium, pp. 291–300, 2009.

[18] P. Penil, H. Posadas, J. Medina, and E. Villar, "UML-based single-source approach for evaluation and optimization of mixed-critical embedded systems," Conference on Design of Circuits and Integrated Systems (DCIS), pp. 1–6, 2015.

[19] A. Namazi, S. Safari, and S. Mohammadi, "CMV: Clustered Majority Voting Reliability-Aware Task Scheduling for Multicore Real-Time Systems," IEEE Transactions on Reliability, vol. 68, no. 1, pp. 187–200, 2019.

[20] R. I. Davis and A. Burns, "Priority Assignment for Global Fixed Priority Pre-Emptive Scheduling in Multiprocessor Real-Time Systems," 30th IEEE Real-Time Systems Symposium, pp. 398–409, 2009.

[21] L. Sha, R. Rajkumar, J. Lehoczky, and K. Ramamritham. Mode change protocols for priority driven preemptive scheduling. Journal of Real-Time Systems, 1(3):244–264, 1989.

[22] R. Schneider, D. Goswami, A. Masrur, M. Becker, and S. Chakraborty. Multi-layered scheduling of mixed-criticality cyber-physical systems. Journal of Systems Architecture, 59(10, Part D):1215 – 1230, 2013.

[23] Y. Zhou, S. Samii, P. Eles, and Z. Peng. Partitioned and overhead-aware scheduling of mixed criticality real-time systems. In Proc. of 24th Asia and South Pacific Design Automation Conference, ASPDAC, pages 39–44. ACM, 2019.

[24] L. Zeng, P. Huang and L. Thiele, "Towards the design of fault-tolerant mixed-criticality systems on multicores," 2016 International Conference on Compilers, Architectures, and Synthesis of Embedded Systems (CASES), Pittsburgh, PA, 2016, pp. 1-10.

[25] R. M. Pathan, "Schedulability Analysis of Mixed-Criticality Systems on Multiprocessors," 2012 24th Euromicro Conference on Real-Time Systems, Pisa, 2012, pp. 309-320.

[26] D. Müller and A. Masrur, "The schedulability region of two-level mixed-criticality systems based on EDF-VD," 2014 Design, Automation & Test in Europe Conference & Exhibition (DATE), Dresden, 2014, pp. 1-6.

[27] S. Maurer and R. Kirner. Cross-criticality interfaces for cyber-physical systems. In Proc. 1st IEEE Int'l Conference on Event-based Control, Communication, and Signal Processing, 2015.

[28] H. Koc, V. K. Karanam and M. Sonnier, "Latency Constrained Task Mapping to Improve Reliability of High Critical Tasks in Mixed Criticality Systems," IEEE 10th Annual Information Technology, Electronics and Mobile Communication Conference (IEMCON), Vancouver, BC, Canada, 2019, pp. 320-324.

Hakduran Koc received his B.S. degree in Electronics Engineering from Ankara University, Turkey in 1997. After working in the industry for two years, he joined Syracuse University, NY where he received his M.S. and Ph.D. degrees in Computer Engineering in 2001 and 2008, respectively. During his graduate study, he was at the Pennsylvania State University as visiting scholar. He is currently chair and an associate professor of Computer Engineering at University of Houston-Clear Lake. He is a senior member of IEEE. His research is in the areas of digital design, embedded systems, and computer architecture.

Vamsi Krishna Karanam received his B.S. degree in Computer Engineering from Vellore Institute of Technology, India in 2016 and his M.S. degree in Computer Engineering from University of Houston-Clear Lake in 2019. He is a student member of IEEE. His research is in the areas of linear circuits, embedded systems, and cyber physical systems.

Creative Commons Attribution License 4.0 (Attribution 4.0 International, CC BY 4.0)

This article is published under the terms of the Creative Commons Attribution License 4.0
https://creativecommons.org/licenses/by/4.0/deed.en_US