ASDYS: Dynamic Scheduling Using Active Strategies for Multi-Functional Mixed-Criticality Cyber-Physical Systems

Yang Bai, Yizhi Huang, Guoqi Xie, Renfa Li, Wanli Chang

***Abstract*—Emerging cyber-physical systems (CPS), such as in the domains of automotive, robotics, and industrial automation, often run complex functions with different criticality levels on a heterogeneous and distributed architecture. The ever stronger interactions between the cyber components and the physical environment lead to dynamic and irregular release of these functions. This paper investigates dynamic scheduling of such mixed-criticality functions, where each function is modelled by a Directed Acyclic Graph (DAG) with no assumption on its period or minimum inter-arrival time. Unlike the existing methods that passively address the mixed criticality with a remedy when deadline misses are observed — this results in high deadline miss ratio (DMR) and it is particularly undesirable for the high-criticality functions — we propose a novel dynamic scheduling approach using active strategies (ASDYS in short), where the mixed criticality is actively treated throughout the scheduling process. Automotive CPS are used as an example for illustration. Experimental results show that our approach is significantly better than the existing methods in both the DMR of high-criticality functions and the overall system DMR.**

***Index Terms*—Cyber-physical systems, dynamic scheduling, mixed-criticality, multi-DAG, heterogeneous distributed architectures**

I. Introduction

Increasingly complex and diverse functions are running on distributed cyber-physical systems (CPS), with a demand of heterogeneous computing architectures. Many of the functions closely interact with the physical environment and often have different criticality levels. Taking the automotive domain as an example, in a modern vehicle, there can be tens of millions of lines of code and hundreds of functions executing on more than one hundred heterogeneous electronic control units (ECUs). These ECUs communicate with various sensors and actuators via shared bus. For these functions, ISO 26262 defines four Automotive Safety Integrity Levels (ASILs) to reflect their different criticality [1], [2].

There is a trend that functions on CPS may get added or removed online depending on varying scenarios. This makes static scheduling difficult and motivates dynamic scheduling. For instance, the automotive domain is moving from the AUTOSAR (AUTomotive Open System ARchitecture) Classic standard to the AUTOSAR Adaptive standard [3], driven by the demands from autonomous and highly automated driving. In AUTOSAR Classic, which has dominated the automotive industry for two decades, the functions to run are fixed, and static scheduling is suitable. All software modules are completely specified during the design process, and the whole stack is compiled and linked in one piece. In AUTOSAR Adaptive, which is service-oriented, dynamic scheduling is supported. Functions dynamically arrive and leave without being known beforehand. They can be created or destroyed with memory allocated accordingly. The emphasis is on adaptive resource sharing. A simple example scenario would be that, depending on the driving condition (such as urban area or highway, speed, distance from neighbour vehicles, risk of illegal road usage, accuracy of mapping, and weather), different sensing and planning functions may be used. Their release time is largely dynamic and unpredictable.

This paper studies CPS running complex and dynamically released functions with different criticality levels on a heterogeneous and distributed architecture. Dynamic scheduling of such functions is investigated, where each end-to-end function runs on multiple processing units (such as ECUs) and can be modelled as a Directed Acyclic Graph (DAG) with no assumption on its period or minimum inter-arrival time. For example, in a vehicle, functions (such as adaptive cruise control) are often composed of sensor operations (such as LIDAR data processing), decision making and planning, control algorithms, and actuation operations. Therefore, many such functions need to be modelled by DAG and executed on multiple distributed heterogeneous ECUs.

Existing methods developed for dynamic scheduling of multiple DAGs may be applied in this context [4], [5]. However, they either do not consider mixed criticality, or passively address it — i.e., the criticality features are only considered and utilised when impending deadline misses are observed, trying to save the high-criticality functions as a remedy — leading to high deadline miss ratio (DMR).

Main contributions: We propose a novel dynamic scheduling approach using active strategies (ASDYS in short) for multi-functional mixed-criticality CPS on a heterogeneous distributed architecture, aiming to minimise DMR[1]. This approach models the functions under the realistic assumption that a task may be supported by only a limited set of processing

Yang Bai, Yizhi Huang, Guoqi Xie, and Renfa Li are with the Key Laboratory for Embedded and Cyber-Physical Systems, College of Computer Science and Electronic Engineering, Hunan University, Changsha 410082, China. (e-mail: baiyang@hnu.edu.cn; huangyizhi@hnu.edu.cn; xgqman@hnu.edu.cn; lirenfa@hnu.edu.cn).

Wanli Chang is with the Department of Computer Science, University of York, York YO10 5GH, UK. (e-mail: wanli.chang@york.ac.uk).

The corresponding authors are Guoqi Xie, Renfa Li, and Wanli Chang.

[1]In modern CPS, deadline misses can often be tolerated to a certain extent on the application level. There have been extensive works in this direction, such as on weakly-hard systems [6].

units. We actively account for the criticality features and prioritise high-criticality functions throughout the scheduling process, by incorporating traffic shaping in the scheduling framework and adaptively handling newly arrived functions. A new dynamic scheduling algorithm for multiple DAGs is reported. Experimental results show that our ASDYS is significantly better than the existing methods [4], [5] in both the DMR of high-criticality functions and the overall system DMR. In addition, the time complexity is polynomial and there is a large reduction of the execution time.

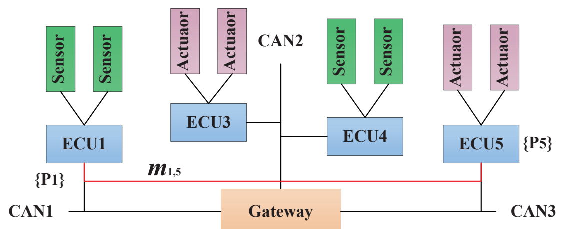

Fig. 1. Architecture of a CAN cluster with buses connected by a central gateway.

II. Related Work

Dynamic scheduling based on simple task models has been studied in systems with mixed criticality [7], [8]. Vestal [9] first formalises the mixed-criticality scheduling problem, which is then extended to the dynamic setting in [7], [8]. A dynamic strategy is proposed in [7] to switch the criticality mode of systems in runtime. Adaptive dynamic scheduling methods proposed in [8] regulate the incoming workload of low-criticality tasks at runtime according to the demand of high-criticality tasks. The main difference of these works above from our paper is that they do not consider the inter-dependencies between tasks, which are modelled by DAG.

Static scheduling based on DAG models has been investigated in several recent works [10]–[15]. The federated scheduling approach for mixed-criticality systems with sporadic DAG tasks running on identical processors is proposed in [10], [11], where high-utilisation tasks are assigned to dedicated processors and tasks are scheduled in a work-conserving manner. Different from the above works [10], [11] that address single-DAG scheduling, periodic multi-DAG scheduling on time-triggered systems is discussed in [12]. A meta-heuristic for scheduling multi-periodic mixed-criticality DAG tasks on multiprocessor systems is reported in [13]. Low DMR of the high-criticality applications and short makespan are targeted in [14], which proposes a static scheduling algorithm for multiple DAGs that are simultaneously released. It has no strategy to deal with newly arrived functions.

The most relevant research to our work lies in dynamic scheduling of multiple DAGs on heterogeneous distributed architectures [4], [5]. Such scheduling processes have two main components: (i) allocating and scheduling tasks of the existing DAGs; (ii) dynamically handling newly arrived DAGs. Dynamic workflow scheduling is studied in [4], which uses a wait queue for each processor and places the selected task in the wait queue if the selected processor is not idle. It does not consider mixed criticality in the entire process. The work [5] aims to address multiple criticality levels, which are unfortunately not accounted for properly. The higher-criticality functions have priorities over the lower-criticality ones in neither of the above two components, i.e., task allocation and scheduling, as well as handling of new DAGs. Only when impending deadline misses are observed, the criticality features get considered and measures are taken to save the high-criticality functions, in the sacrifice of low-criticality functions. This is a passive strategy with very limited effectiveness, as reflected by the high DMR of the high-criticality functions and the high overall system DMR.

III. Models

This section explains the models used in this work, for the platform architecture, functions, criticality, and problem formulation. We would like to make a note that unlike the usual assumption that a task can be supported by all processing units [4], [5], our models take a more realistic assumption that some tasks may only be supported by a limited set of processing units. Many CPS, such as the automotive E/E (Electrical/Electronic) architecture, are heterogeneous. That is, the ECUs are not identical and they support different sets of tasks. This poses a larger challenge in finding an appropriate schedule, as the feasible choices are limited.

A. Platform Architecture

We consider multiple heterogeneous distributed processing units that are connected by a network. This is commonly found, e.g., in the automotive domain, where multiple heterogeneous distributed ECUs are connected with CAN (controller area network) buses and a central gateway, as illustrated by Figure 1. Functions may be triggered by sensors receiving signals from the physical components of the automobile (including driver actions) or the environment (including infrastructure and other vehicles). We denote a set of heterogeneous distributed processing units as $P = \{p_1, p_2, \ldots, p_{|P|}\}$, whose size is $|P|$. There is a trend in the automotive domain to employ some powerful central ECUs, which can be used to run the scheduling algorithm, and instructs the distributed less powerful ECUs via the central gateway and buses. This trend is aligned with the moving from AUTOSAR Classic to AUTOSAR Adaptive, driven by the demands from autonomous and highly automated driving. In this work, we consider non-preemptive scheduling.

B. Function Model

A single function, which is to be executed by heterogeneous distributed processing units in P, can be modelled as a DAG [5], [16], where the nodes stand for tasks and the edges indicate the dependencies with communication cost between the nodes. An example will be illustrated at the end of this section. We denote the mth function in the system function set MS as F_m=(N, W, M, C):

- $N = \{F_m.n_1, \ldots, F_m.n_N\}$ represents the task set containing all N task nodes in F_m.
- W is an $|N| \times |P|$ matrix, where $w_{i,k}$ is the worst-case execution time (WCET) of the task $F_m.n_i$ running on p_k. Due to the heterogeneity, the $w_{i,k}$ values on different processing units are different. In addition, $w_{i,k}$ takes the value ∞ if p_k does not support $F_m.n_i$.
- M is a set of edges, and $m_{i,j} \in M$ represents the task dependency and communication from $F_m.n_i$ to $F_m.n_j$.
- C denotes the set of end-to-end worst-case response time (WCRT) for communication, where $c_{i,j} \in C$ is for $m_{i,j}$. If $F_m.n_i$ and $F_m.n_j$ are on the same processing unit, $c_{i,j}$ is taken to be 0. Otherwise, the communication response time can be derived from the task allocation. In this paper, we assume a simple and conservative computation that $c_{i,j}$ takes the maximum of all possible task allocations. Note that $c_{i,j}$ sometimes may not be straightforward to compute, especially when contention cannot be resolved with pessimistic resource sharing approaches. This is however not the focus of this work.
- Other attributes: $F_m.arrivaltime$ is the release time instant, $F_m.criticality$ denotes the criticality level, $F_m.lowerbound$ indicates the minimum makespan of a function when all the processing units are monopolised by the function, $F_m.deadline$ is the relative deadline, and $F_m.deadline \geq F_m.lowerbound$. There is no dependency between different functions.

C. Criticality Model

The concrete specifications and identifications of criticality vary in different industries, often involving more than two levels. For example, as mentioned earlier in Section I, in the automotive domain, the criticality is formalised by the ASIL in ISO 26262 with four levels A, B, C, and D, where the specific level of a function can be evaluated from three orthogonal dimensions, i.e., severity, exposure, and controllability. Taking the perspective of severity (severity of the damage to relevant people caused by a hazard) for instance, low DMR is desired by all functions and particularly important for the high-criticality (severity) ones.

In the rest of this paper, we will continue to use this example for illustration, where the set of criticality levels is $S = \{S_0, S_1, S_2, S_3\}$ and $F_m.criticality \in S$. The highest level is S_3. Besides, we assume that all tasks of the same function inherit its criticality level, i.e., $F_m.n_i.criticality = F_m.criticality$. Functions do not move between criticality levels.

In a mixed-criticality CPS, there can be multiple dynamically released functions to be executed on distributed processing units (such as ECUs), belonging to P. The set of these functions to be allocated and scheduled is denoted as $MS = \{F_1, F_2, \ldots, F_{|MS|}\}$, where $|MS|$ is the size. A system has its criticality level as $MS.criticality \in S$, which can be changed during runtime. Its default value is the lowest level S_0. Essentially, the functions with lower criticality levels than $MS.criticality$ will not be handled. $MS.criticality$ gets elevated when impending deadline misses of functions with higher criticality levels than it are observed. Details will be explained in Section IV-D.

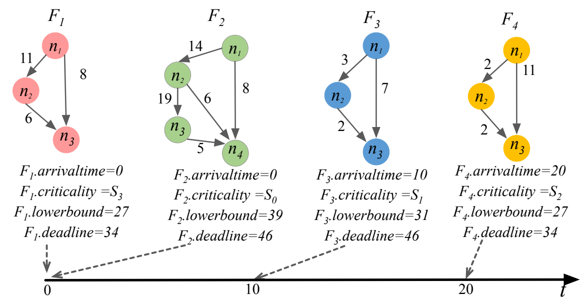

Fig. 2. The motivating example with four functions.

D. Problem Formulation

Given a set of dynamically released functions MS to be executed on a set of heterogeneous processing units P and a set of criticality levels S, we aim to propose a dynamic scheduling approach to reduce the DMR of these functions:

$$DMR(S_x) = \frac{|MS^{\text{miss}}(S_x)|}{|MS(S_x)|}, \tag{1}$$

where $|MS^{\text{miss}}(S_x)|$ represents the number of functions with the criticality level of x missing their deadlines, and $|MS(S_x)|$ represents the number of all functions with the criticality level S_x. In this work, the high-criticality functions are assigned with the criticality level of S_3, and $DMR(S_3)$ is the most important metric, followed by the DMR of functions at other criticality levels S_2, S_1 and S_0.

E. Motivating Example

An example of four dynamically arriving functions with different criticality levels is illustrated in Figure 2, and it will again be used when explaining the proposed scheduling approach. Table I shows the WCET matrices W, where ∞ indicates that a task is not allowed to run on this processing unit. The task's upward rank value $rank_{\text{u}}$ will be explained and used later in Section IV-D and IV-E. For each DAG in Figure 2, a directed edge from the task $F_m.n_i$ to $F_m.n_j$ represents the dependency and communication between them. The value of WCRT $c_{i,j}$ is the number beside this edge. The parameters $lowerbound$ and $deadline$ of a function will be discussed in the next section with Table II.

IV. ASDYS: The Dynamic Scheduling Approach

A dynamic scheduling approach for multiple DAGs mainly consists of two parts: (i) how to allocate and schedule the existing tasks on the processing units; (ii) how to handle newly arriving functions. Our proposed approach ASDYS, designed for multi-functional CPS on heterogeneous distributed architectures, uses active strategies to treat the mixed criticality in both parts of the scheduling process and reduce the DMR. After the schedule is computed, certain deadline misses may be saved with a remedy.

TABLE I
WCET MATRICES OF THE MOTIVATING EXAMPLE

(a) WCET matrix of F_1

Task	$F_1.n_1$	$F_1.n_2$	$F_1.n_3$
p_1	∞	17	13
p_2	6	12	9
p_3	11	6	10
$rank_u$	47.8	28.3	10.7

(b) WCET matrix of F_2

Task	$F_2.n_1$	$F_2.n_2$	$F_2.n_3$	$F_2.n_4$
p_1	14	16	7	6
p_2	15	7	7	11
p_3	8	15	5	13
$rank_u$	79.3	60	21	10

(c) WCET matrix of F_3

Task	$F_3.n_1$	$F_3.n_2$	$F_3.n_3$
p_1	12	13	∞
p_2	18	10	18
p_3	9	∞	7
$rank_u$	42	26	12.5

(d) WCET matrix of F_4

Task	$F_4.n_1$	$F_4.n_2$	$F_4.n_3$
p_1	8	20	20
p_2	13	7	15
p_3	18	11	8
$rank_u$	44	29	14

TABLE II
LOWER BOUNDS AND DEADLINES OF THE MOTIVATING EXAMPLE

(a) Deadlines of functions

Function	F_1	F_2	F_3	F_4
$F_m.abs\_deadline$	34	46	56	54
$F_m.deadlineslack$	7	7	15	7

(b) Lowerbounds and deadlines of tasks in F_1 and F_2

Task	$F_1.n_1$	$F_1.n_2$	$F_1.n_3$	$F_2.n_1$	$F_2.n_2$	$F_2.n_3$	$F_2.n_4$
$lowerbound$	6	18	27	8	23	28	39
$abs\_deadline$	13	25	34	15	30	35	46

(c) Lowerbounds and deadlines of tasks in F_3 and F_4

Task	$F_3.n_1$	$F_3.n_2$	$F_3.n_3$	$F_4.n_1$	$F_4.n_2$	$F_4.n_3$
$lowerbound$	9	22	31	8	17	27
$abs\_dealine$	34	43	56	35	44	54

A. Preliminaries

Lower bound: A function's lower bound refers to its minimum makespan when all processing units are monopolised by it. In this work, we use the Heterogeneous Earliest-Finish-Time (HEFT) algorithm [16], which allocates the tasks of a DAG to multiple heterogeneous processing units with the objective of minimising the makespan. The lower bound of a function F_m is equal to the exit task's actual finish time computed by HEFT. As defined in [5], each task $F_m.n_i$ has an individual lower bound $lowerbound(F_m.n_i)$ equal to its actual finish time in the HEFT computation. The obtained lowerbounds of functions and tasks in the motivating example are reported in Figure 2 and Table II.

Deadline and deadline-slack: For each function, a known relative deadline $F_m.deadline$ as introduced earlier, is provided according to the physical requirements. It limits the length of time between the function arrival and its execution completion. The absolute deadline of a function F_m is calculated as:

$$F_m.abs\_deadline = F_m.deadline + F_m.arrivaltime. \quad (2)$$

The time slack between a function's relative deadline and lower bound is denoted as $F_m.deadlineslack$ (defined in [5]):

$$F_m.deadlineslack = F_m.deadline - F_m.lowerbound. \quad (3)$$

Similarly, the absolute deadline of a task $F_m.n_i$ can be calculated as:

$$\begin{aligned} abs\_deadline(F_m.n_i) &= F_m.arrivaltime \\ &+ lowerbound(F_m.n_i) + F_m.deadlineslack. \end{aligned} \quad (4)$$

The values for the motivating example are shown in Table II.

B. Scheduling Framework

The scheduling framework is shown in Figure 3. The basic structure is similar to [4], [5], and we propose a shaper (in grey) to prioritise the high-criticality functions. The underlying algorithm (Algorithm 1 in Section IV-D) is also new, aiming to actively address the mixed criticality and reduce DMR.

Before explaining the shaper, we will first define a related term.

Definition 1: **Criticality-slack.** A function's criticality-slack refers to the difference between its criticality and the system's current criticality. For F_m, given $F_m.criticality = S_{x_1}$ and $MS.criticality = S_{x_2}$, where x_1, x_2 are integers in $[0, 3]$, its criticality-slack denoted by $F_m.criticalityslack$ is computed as

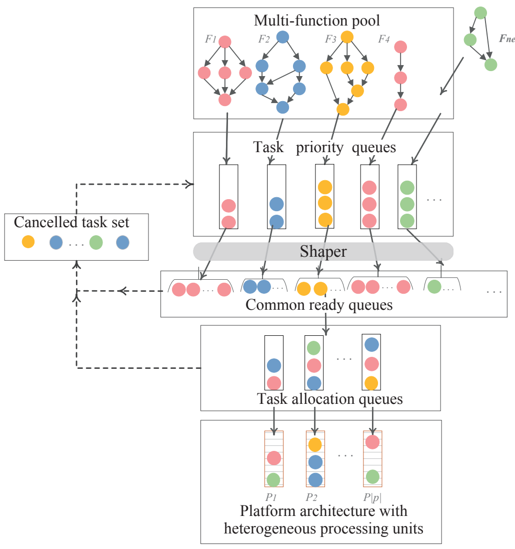

Fig. 3. The scheduling framework.

$$F_m.criticalityslack = x_1 - x_2. \quad (5)$$

Shaper: The *shaper* is a component to limit the maximum number of tasks in a given function F_m that can join the current scheduling round. This maximum number $N_{\max}(F_m)$ is computed as

$$N_{\max}(F_m) = F_m.criticalityslack + 1. \quad (6)$$

Essentially, a higher-criticality function will have more tasks being handled. A function with lower criticality than the system has no task joining this scheduling round. More details will be explained later in this paper.

We now describe the scheduling framework, emphasising on the *shaper*:

- The multi-function pool holds the arriving functions.
- Each function F_m in the multi-function pool has a corresponding task priority queue $F_m.task\_priority\_queue$.
- Our proposed algorithm, which will be explained in Section IV-D, schedules tasks round by round. In

each round, the algorithm checks every task priority queue in the order from $F_1.task\_priority\_queue$ to $F_{|MS|}.task\_priority\_queue$, and selects certain tasks from each task priority queue. The selected tasks are put into the common ready queue $MS.common\_ready\_queue$ and then they wait for being allocated and scheduled. The unselected tasks remain in their task priority queues and they wait to be selected in the next round.

- As is known, traffic shapers are used in computer networks to control and regulate the speeds, at which data packets of different traffic types are injected to the network so as to achieve a certain quality of service (QoS). Inspired by the above, we use the shaper in this work to adaptively control and regulate the speeds, at which the tasks of functions with different criticality levels are sent to $MS.common\_ready\_queue$, in order to reduce the DMR of high-criticality functions. In other words, for a function F_m, the *shaper* decides how many tasks in $F_m.task\_priority\_queue$ can be sent to $MS.common\_ready\_queue$ in a certain round of scheduling.
- A task allocation queue $p_k.task\_allocation\_queue$ is maintained for each processing unit p_k. If a task in $MS.common\_ready\_queue$ has been selected, and the processing unit for it to be allocated to (decided by the algorithm that will be explained later) is p_k, it will be inserted to $p_k.task\_allocation\_queue$.
- The cancelled task set temporarily stores tasks that have been cancelled according to rules in our algorithm.

C. Triggering Events

In order to respond to changes in the dynamic system, it is sometimes necessary to cancel the computed schedules (and allocations) of certain tasks and come up with new ones. We propose the concept *triggering events* to be used in our algorithm.

Definition 2: **Triggering events.** Triggering events refer to the events that may trigger rescheduling and reallocation of tasks. We consider two types of triggering events: *new arrival triggering events* and *deadline alert triggering events*.

Definition 3: **New arrival triggering events.** With the arrival of one or more functions, assuming HI to be the highest criticality level of the new functions, if there is at least one unexecuted task, whose criticality level is lower than HI, then this new arrival event is called a *new arrival triggering event*.

Definition 4: **Deadline alert triggering events.** An allocation for $F_m.n_i$ is called a *deadline alert triggering event* if both the conditions in (7) are satisfied, where $AFT(F_m.n_i)$ is the actual finish time of $F_m.n_i$ computed by the scheduling algorithm,

$$\begin{cases} F_m.criticality > MS.criticality; \\ AFT(F_m.n_i) > abs\_deadline(F_m.n_i). \end{cases} \tag{7}$$

Intuitively it means that, if the deadline of a function is going to get missed (referring to the lower condition) and the function is of high criticality (compared to the system criticality level, referring to the upper condition), then this task allocation should trigger rescheduling and reallocation.

D. Scheduling Algorithm

We propose a dynamic scheduling algorithm (including task allocation) for multiple DAGs, which actively addresses the mixed criticality, to be run with the scheduling framework discussed above, as shown in Algorithm 1. Our entire approach is designed to effectively reduce DMR of all functions. The shaper and the adaptive handling of newly arrived functions, which will be elaborated later in this subsection, actively prioritise high-criticality functions and prevent them from being interfered with. In general, this algorithm does the followings: (i) tasks from different functions get handled round by round; (ii) in each round, the scheduling orders of the ready tasks are dependent on the criticality levels; (iii) every task is allocated to the processor that provides its earliest finish time; and (iv) triggering events may cause re-allocation.

Step 0: Initialising system criticality (Line 1). The system criticality $MS.criticality$ is initialised to S_0, i.e., the lowest criticality level. The initial non-empty function set MS_{init} is composed of the functions released during system initialisation.

Step 1: Prioritising tasks in functions (Lines 2-4). For each F_m that has already arrived in the multi-function pool, we put all the tasks of F_m in its task priority queue $F_m.task\_priority\_queue$, with the descending order of the tasks' upward rank values (denoted by $rank_{\text{u}}$ as shown in Table I). For a task $F_m.n_i$,

$$rank_{\text{u}}(F_m.n_i) = \\ F_m.\overline{w_i} + \max_{F_m.n_j \in succ(F_m.n_i)} \{F_m.c_{i,j} + rank_{\text{u}}(F_m.n_j)\}, \tag{8}$$

where $F_m.\overline{w_i}$ is the average WCET of $F_m.n_i$ over all supporting processing units and $succ(F_m.n_i)$ is the set of $F_m.n_i$'s immediate successor tasks. The $rank_{\text{u}}$ value of a task is first proposed in [16] and it has been widely used to prioritise a DAG's tasks. Intuitively, a task is assigned a higher priority, if (i) it has long execution time itself; (ii) its successors (depending on it) have long execution time; (iii) the communication latency between it and its successors is long. As long as there are tasks in the task priority queues, we perform Step 2 to 6 (Lines 5-44).

Step 2: Preparing tasks for allocation (Lines 6-16). By employing the *shaper* component, our algorithm adaptively selects the tasks for allocation round by round. In each round, for a function F_m satisfying $F_m \geq MS.criticality$, we try to select the top $N_{max}(F_m)$, as computed by (6), tasks from the head of $F_m.task\_priority\_queue$, and put them in $MS.common\_ready\_queue$ (ordered from high to low criticality and then rank for the same criticality). If the number of tasks in $F_m.task\_priority\_queue$ is smaller than $N_{max}(F_m)$, all of them get selected.

Intuitively, the system criticality level $MS.criticality$ acts as a threshold to prevent functions with lower criticality levels from being handled in this and following steps. Among the functions above this threshold, $N_{max}(F_m)$ is used for further prioritisation according to their criticality levels, allowing more tasks in higher-criticality functions to participate in a round. $MS.criticality$ takes the lowest value S_0 by default (i.e., every function can be handled) and only gets elevated if impending deadline misses are observed (i.e., the system is not able to sustain all functions and has to abandon the low-criticality ones).

For example, after initialisation, the $N_{\max}$ of the functions with the lowest criticality level S_0 is 1, according to (5) and (6). That is, all functions get scheduled to some extent. By comparison, the $N_{\max}$ of the functions with the highest criticality level S_3 is 4, reflecting prioritisation. Afterwards, when impending deadline misses are observed, the system criticality level is raised from S_0 to S_1. In this case, the $N_{\max}$ of the functions with the lowest criticality level S_0 becomes 0. That is, these S_0 functions are abandoned. Details about how the impending deadline misses are treated will be explained in Step 4. As long as there are tasks in $MS.common\_ready\_queue$, we perform Step 3 to 5.

Step 3: Task allocation (Lines 17-19). We take out the task at the head of $MS.common\_ready\_queue$ and compute its allocation. Assuming that the selected task $F_m.n_i$ is allocated on p_k, the earliest finish time, i.e., $EFT(F_m.n_i, p_k)$, is the earliest time when $F_m.n_i$ can finish its execution on p_k. It depends on $F_m.w_{i,k}$ (the WCET of the task $F_m.n_i$ on p_k) and the earliest time when $F_m.n_i$ can start its execution on p_k, which is denoted by $EST(F_m.n_i, p_k)$. The relation is shown below

$$EFT(F_m.n_i, p_k) = EST(F_m.n_i, p_k) + F_m.w_{i,k}. \tag{9}$$

On the other hand, $EST(F_m.n_i, p_k)$ depends on both the earliest idle time of p_k and $F_m.n_i$'s immediate predecessors,

$$EST(F_m.n_i, p_k) = \max \begin{cases} avail[k]; \\ \max\limits_{F_m.n_j \in pre(F_m.n_i)} \{AFT(F_m.n_j) + F_m.c_{j,i}\}, \end{cases} \tag{10}$$

where $avail[k]$ is the earliest idle time of p_k and $pre(F_m.n_i)$ denotes the set of immediate predecessors of $F_m.n_i$.

The task $F_m.n_i$ is then allocated to the processor $\widehat{p_k}$ (in fact, $\widehat{p_k}.task\_allcation\_queue$ instead of the processor itself), which provides the minimum earliest finish time. The actual finish time of $F_m.n_i$, i.e., $AFT(F_m.n_i)$ is equal to $EFT(F_m.n_i, \hat{p}_k)$. The actual start time $AST(F_m.n_i)$ is equal to $AFT(F_m.n_i) - F_m.w_{i,k}$. Only at the time of $AST(F_m.n_i)$, is the task $F_m.n_i$ assigned to the processor $\widehat{p_k}$.

Once the allocation is decided, the WCRT of the whole function F_m (note the difference from the WCRT of communication messages explained in Section III-B) is equal to $AFT(F_m, n_{exit}) - F_m.arrivaltime$, i.e., the actual finish time of the exit task $F_m.n_{exit}$ minus the release time of F_m.

Step 4: Checking *deadline alert triggering event* and cancelling tasks (Lines 20-25). We check if this allocation of $F_m.n_i$ in Step 3 is a *deadline alert triggering event* based on Definition 4. If it is, we elevate $MS.criticality$ to be the same as $F_m.criticality$. The unexecuted tasks getting out of the task priority queues in this round and the previous round, i.e., including the tasks entering the task allocation queues in this and previous round, and all tasks in the common ready queues (all inserted in this round), are cancelled. Note that we do not cancel tasks that have already started executing on the processing units. These cancelled tasks will be moved to $cancelled\_task\_set$, and then back to the corresponding task priority queues. Once a cancelled task gets a second chance, it may be allocated to a different processing unit.

Step 5: Resetting system criticality (Lines 26-30). Once the scheduling of F_m causing the *deadline alert triggering event* in Step 4 is completed, the system criticality $MS.criticality$ is reset to S_0. The remaining tasks in $MS.common\_ready\_queue$ are then moved back to the task priority queues via $cancelled\_task\_set$.

Step 6: Handling newly arrived functions (Lines 34-42). If one or more new functions arrive (implemented by the interrupt service routine), with the highest criticality level of HI, we check whether this is a *new arrival triggering event* with Definition 3. If it is, we cancel the unexecuted tasks in the

Algorithm 1 The scheduling algorithm

```
Input: $P = \{p_1, p_2, \ldots, p_{|P|}\}$, $S = \{S_0, S_1, S_2, S_3\}$, $MS_{init} = \{F_1, F_2, \ldots, F_{|MS|}\}$
Output: Scheduling results
 1: $MS.criticality \leftarrow S_0$ and $MS \leftarrow MS_{init}$;
 2: for ($m \leftarrow 1; m \leqslant |MS|; m++$) do
 3:     sort $F_m$'s tasks to $F_m.task\_priority\_queue$ in descending order of $rank_u$ ;
 4: end for
 5: while (task priority queues are not all empty) do
 6:     for ($m \leftarrow 1; m \leqslant |MS|; m++$) do
 7:         if ($F_m.criticality < MS.criticality$) then
 8:             continue;
 9:         end if
10:         $N_{max}(F_m) \leftarrow F_m.criticalityslack + 1$,
11:         $cnt \leftarrow N_{max}(F_m)$;
12:         while (($cnt--$)&&(!$F_m.task\_priority\_queue.empty()$)) do
13:             $n_i \leftarrow F_m.task\_priority\_queue.out()$;
14:             $common\_ready\_queue.insert(n_i)$;
15:         end while
16:     end for
17:     while (!$MS.common\_ready\_queue.empty()$) do
18:         $F_m.n_i \leftarrow common\_ready\_queue.out()$;
19:         Assign $F_m.n_i$ to $task\_allocation\_queue(p_k)$ with the minimum EFT;
20:         if ($AFT(F_m.n_i) > abs\_deadline(F_m.n_i)$&& $F_m.criticality > MS.criticality$) then
21:             $MS.criticality \leftarrow F_m.criticality$;
22:             $cancelled\_task\_set \xleftarrow{\text{remove}}$ tasks in $MS.common\_ready\_queue$;
23:             $cancelled\_task\_set \xleftarrow{\text{remove}}$ tasks in $task\_allocation\_queues$ allocated in the current and previous round
24:             $cancelled\_task\_set \xrightarrow{\text{remove back}} task\_priority\_queues$
25:         end if
26:         if (Scheduling of $F_m$, which causes the system criticality to rise, is completed) then
27:             $cancelled\_task\_set \xleftarrow{\text{remove}}$ tasks in $MS.common\_ready\_queue$;
28:             $cancelled\_task\_set \xrightarrow{\text{remove back}} task\_priority\_queues$
29:             $MS.criticality \leftarrow S_0$;
30:         end if
31:     end while
32:     if (new functions $MS_{new}$ arrive) then
33:         $HI \leftarrow$ the highest criticality of functions in $MS_{new}$;
34:         if (at least one task in $task\_allocation\_queues$ has lower criticality level than $HI$) then
35:             $cancelled\_task\_set \xleftarrow{\text{remove}}$ tasks in $task\_allocation\_queues$ with lower criticality levels than $HI$;
36:             $cancelled\_task\_set \xleftarrow{\text{remove}}$ tasks in $MS.common\_ready\_queue$;
37:             $cancelled\_task\_set \xrightarrow{\text{remove back}} task\_priority\_queues$
38:         end if
39:         for ($m \leftarrow 1; m \leqslant |MS_{\text{new}}|; m++$) do
40:             $MS$.add($F_m^{\text{new}}$);
41:             sort $F_m^{\text{new}}$'s tasks to $F_m^{\text{new}}.task\_priority\_queue$ in descending order of $rank_u$
42:         end for
43:     end if
44: end while
```

task allocation queues with lower criticality levels than HI, and all tasks in the common ready queues. These cancelled tasks will be put back in the task priority queues. The newly arrived functions will be added to the multi-function pool and sorted for the task priority queues as in Step 1. Such cancellation has negligible cost.

Whilst the system is running, it is highly unlikely, especially for those complex CPS we are investigating like autonomous vehicles, that task priority queues get all emptied. There is always some function waiting to be scheduled and run. Therefore, Algorithm 1 runs continuously. If no functions are queuing, it means that the system is shut down or suspended, following which there will be a new initialisation.

A summary of the active strategies: The proposed algorithm uses active strategies to prioritise high-criticality functions in the scheduling process: (i) In each scheduling round, the *shaper* adaptively sets the number of tasks that can participate according to the criticality levels of both the functions and systems, and the tasks in the common ready queues are ordered from high to low criticality (followed by $rank_u$ values for the same criticality), giving more opportunities to functions with higher-criticality levels; (ii) When responding to the newly arrived functions, task cancelling operations are adaptively performed based on the criticality levels of both the new functions and the current unexecuted tasks. This strategy avoids the undesirable situation that the newly arrived functions with lower-criticality levels interfere with the previously computed schedules of the higher-criticality functions. The reduction of rescheduling efforts also contributes to the timing behaviour of the algorithm. Assuming a function to have at most α tasks, there are a maximum of $|MS| \cdot \alpha$ tasks in the system function set MS. The most frequent and time-consuming operation in the proposed algorithm is to find the proper processing unit for a task through computing its EFT value, which requires traversing the task's immediate predecessors (at most α) and all the $|P|$ processing units. Therefore, the asymptotic time complexity of this algorithm is $\mathcal{O}(|MS| \cdot \alpha^2 \cdot |P|)$.

E. Scheduling Process of the Motivating Example

The motivating example in Figure 2 is again used to illustrate the scheduling process. $MS.criticality$ is initialised to S_0. At time instant 0, F_1 and F_2 arrive concurrently. Since $F_1.criticality = S_3$ and $F_2.criticality = S_0$, according to (5) and (6), $N_{\max}(F_1) = 4$ and $N_{\max}(F_2) = 1$. So in this scheduling round, at most four tasks in F_1 and one task in F_2 are allowed to participate. In the first round, all the three tasks of F_1 (in the descending order of the $rank_u$ values) and one task of F_2 are selected and moved to $MS.common\_ready\_queue$. The allocation order is $F_1.n_1, F_1.n_2, F_1.n_3, F_2.n_1$, from high to low criticality. Since all tasks in F_1 get allocated, next, one task in F_2 is selected and allocated in each round. The results are shown in Figure 4.

When the time is 10, a new function F_3 with the criticality level of S_1 arrives. Therefore, HI is S_1. The unexecuted tasks $F_2.n_4$ and $F_2.n_3$ are cancelled since $F_2.criticality < HI$, as shown in Figure 5. After the above task cancelling, the

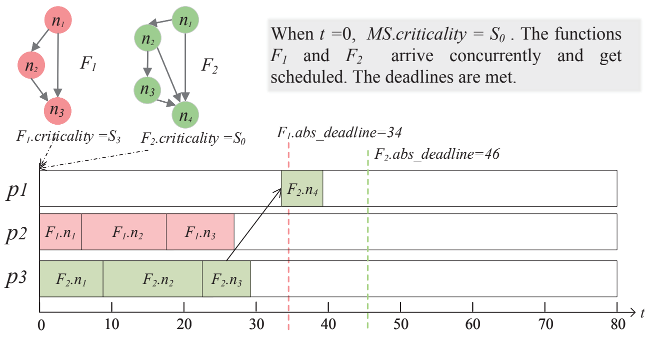

Fig. 4. Scheduling results for F_1 and F_2 that arrive at t=0.

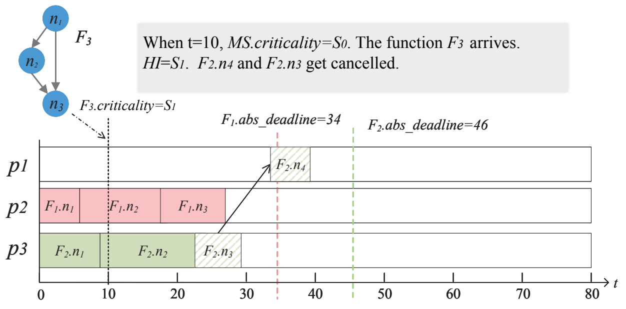

Fig. 5. Task cancelling caused by the arrival of F_3 at t=10.

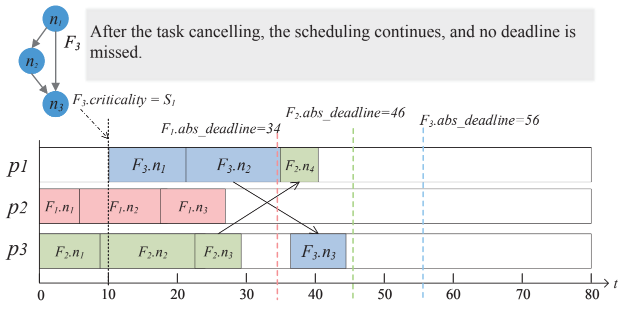

Fig. 6. Scheduling results after the task cancelling in Figure 5.

scheduling continues as follows. Since $F_2.criticality = S_0$, $F_3.criticality = S_1$, and $MS.criticality$ is still S_0. According to (5) and (6), $N_{\max}(F_2) = 1$, and $N_{\max}(F_3) = 2$. The tasks $F_3.n_1$, $F_3.n_2$, and $F_2.n_3$ are allocated successively in the first round, ordered from high to low criticality followed by rank. The tasks $F_3.n_3$ and $F_2.n_4$ are allocated in the next round. Figure 6 shows the related scheduling results, and no deadline is missed.

When the time is 20, F_4 with the criticality level of S_2 arrives, making HI S_2. The unexecuted tasks belong to F_3 and F_2, which both have lower criticality than HI, and thus get cancelled, as shown in Figure 7. Afterwards, since $F_4.criticality = S_2$, $F_2.criticality = S_0$, $F_3.criticality = S_1$, and $MS.criticality = S_0$, according to (5) and (6), $N_{\max}(F_4) = 3$, $N_{\max}(F_2) = 1$, and $N_{\max}(F_3) = 2$. The tasks $F_4.n_1$, $F_4.n_2$, $F_4.n_3$, $F_3.n_2$, $F_3.n_3$, and $F_2.n_3$ are allocated successively in the first round, and $F_2.n_4$ is allocated in the next round, as shown in Figure 8.

It is noted that the *deadline alert triggering events* do not appear in this example. The active strategies reduce their

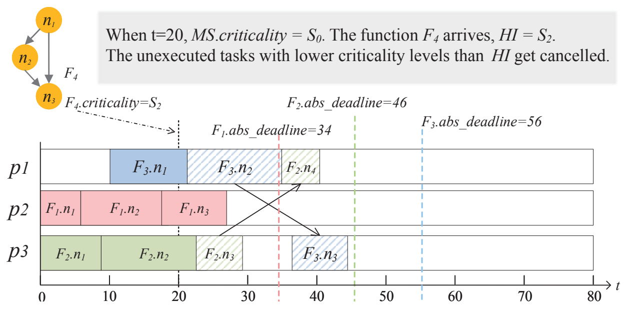
When t=20, MS.criticality = S0. The function F4 arrives, HI = S2.
The unexecuted tasks with lower criticality levels than HI get cancelled.
F4.criticality=S2
F2.abs_deadline=46
F1.abs_deadline=34
F3.abs_deadline=56

Fig. 7. Tasks cancelling caused by the arrival of F_4.

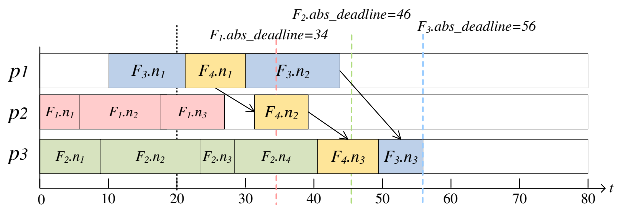

Fig. 8. Scheduling results after the task cancelling in Figure 7.

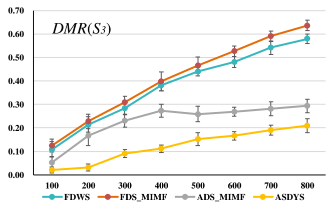

Fig. 9. $DMR(S_3)$ on $|MS|$.

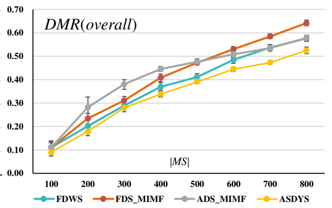

Fig. 10. $DMR(overall)$ on $|MS|$.

occurrence, making it difficult to present both two types of triggering events with a simple example and a short scheduling process.

We would like to stress that, whilst the automotive systems are used as an example, the proposed approach is fairly general and can be deployed for dynamic scheduling of any systems with mixed-criticality functions, such as in the domains of robotics and industrial automation. The extension from four to any number of criticality levels is trivial.

V. Experimental Results

We compare our proposed approach with three existing works for evaluation, i.e., FDWS [4], FDS_MIMF [5], and ADS_MIMF [5], all of which address the dynamic scheduling of multiple DAGs. As discussed before, FDWS does not consider mixed criticality, and MIMF passively utilises the criticality features, trying to fix the observed deadline misses with a remedy. Note that these three methods are the closest to our approach and they can be directly applied in the context we study.

A. Experimental Setting and Metrics

We mainly evaluate these approaches from two aspects. The first is on the scheduling results, focusing on the DMR of functions $DMR(S_x)$ referring to (1), and especially the high-criticality ones, i.e., $DMR(S_3)$ in this work. The second aspect is on the timing efficiency, which is reflected primarily by the time cost in simulation, with the average number of times a task is rescheduled as an auxiliary indicator.

The functions are randomly generated according to the model in Section III-B within the following realistic parameter ranges [17] under uniform distribution. The WCET of a task and the WCRT of communication are between 100 and 400 time units, i.e., $100 \leqslant w_{i,k} \leqslant 400$, and $100 \leqslant c_{i,j} \leqslant 400$. For a function F_m, the number of tasks it contains (denoted by $|N|$ as mentioned before) is $8 \leqslant |N| \leqslant 23$. There is no restriction imposed on the communication network topology or protocol for the sake of generality. The deadline-slack of F_m is set as $F_m.deadlineslack = F_m.lowerbound/40$ [5]. To reflect the increasing complexity of CPS in both functions and platforms, we consider up to 800 functions dynamically arriving and running on 100 heterogeneous distributed processing units. For each task, certain processing units, in the range of $[0, 9]$, are randomly chosen to be non-supportive.

The simulator is implemented with Java. The global scheduler executing Algorithm 1 is aware of all the parameters of functions except for the arrival time, which is randomly initialised. Both the task priority queues and task allocation queues are maintained with the scheduler. All the algorithms under comparison run on the same PC with Intel i7 CPU (4.00GHz) and 16GB RAM.

B. Results

Experiment 1: The approaches are compared under different workloads, represented by the function set size $|MS|$ varying from 100 to 800. We limit the time range between the first and the last arrived function to be 40000 time units, and the functions arrive randomly under uniform distribution. The number of functions with each of the four critical levels is set to be the same. The obtained values of the evaluation metrics are statistic averages.

Table III shows the DMR results, among which $DMR(S_3)$ and $DMR(overall)$ are plotted in Figure 9 and 10. The utilisation of the processing unit can be computed as

$$U = \sum_{k=1}^{|P|} busytime(p_k) / \sum_{k=1}^{|P|} makespan(p_k) \qquad (11)$$

and also reported in Table III, where $|P|$ is the size of processor set, $makespan(p_k)$ is the time span between 0 and the finish time instant of the last task executed on p_k, and $busytime(p_k)$ is the total time when there are tasks executing on p_k. The utilisation in DAG scheduling is usually not high, due to the dependency constraints between tasks. Considering mixed criticality and trying to reduce DMR make it even worse.

With increasing workloads (reflected by $|MS|$), the $DMR(overall)$ and $DMR(S_x)$, where $x \in [0, 3]$, of all approaches under comparison increase. This is expected as the resources are limited. As shown in Table III and Figure 9, our proposed ASDYS always has the lowest $DMR(S_3)$, i.e., the DMR of functions with the highest criticality level, which is the most important metric in this work. As shown in Table III and Figure 10, our ASDYS has the lowest $DMR(overall)$.

TABLE III
DMR COMPARISON WITH DIFFERENT FUNCTION SET SIZES (REPRESENTED AS SAMPLE MEAN±1.96SE (STANDARD ERROR), 95% CONFIDENCE INTERVAL)

|MS|	Approach	DMR (sample mean±1.96SE)					Reschedule /task	U
		Overall	S3	S2	S1	S0		
100	FDWS	0.11±0.02	0.11±0.03	0.10±0.04	0.10±0.03	0.14±0.04	0.00	0.08
	FDS_MIMF	0.11±0.02	0.12±0.03	0.13±0.05	0.07±0.05	0.12±0.05	2.03	0.08
	ADS_MIMF	0.11±0.03	0.05±0.03	0.11±0.03	0.11±0.06	0.17±0.05	2.34	0.08
	ASDYS	**0.09±0.02**	**0.02±0.01**	**0.06±0.02**	**0.10±0.05**	**0.18±0.04**	**1.63**	**0.08**
200	FDWS	0.20±0.02	0.21±0.03	0.21±0.03	0.19±0.06	0.18±0.04	0.00	0.14
	FDS_MIMF	0.23±0.02	0.23±0.03	0.23±0.04	0.22±0.06	0.24±0.03	2.91	0.14
	ADS_MIMF	0.28±0.04	0.17±0.04	0.28±0.06	0.34±0.06	0.32±0.04	3.83	0.14
	ASDYS	**0.18±0.02**	**0.03±0.01**	**0.13±0.03**	**0.26±0.05**	**0.27±0.03**	**2.51**	**0.14**
300	FDWS	0.29±0.02	0.28±0.03	0.26±0.02	0.30±0.03	0.30±0.03	0.00	0.20
	FDS_MIMF	0.31±0.02	0.31±0.03	0.30±0.04	0.31±0.05	0.32±0.02	3.15	0.20
	ADS_MIMF	0.38±0.02	0.23±0.03	0.34±0.03	0.42±0.03	0.52±0.04	4.21	0.20
	ASDYS	**0.28±0.02**	**0.09±0.02**	**0.23±0.03**	**0.37±0.04**	**0.42±0.03**	**2.81**	**0.20**
400	FDWS	0.37±0.02	0.38±0.02	0.37±0.02	0.37±0.04	0.36±0.04	0.00	0.26
	FDS_MIMF	0.41±0.02	0.40±0.04	0.39±0.03	0.42±0.03	0.42±0.03	3.39	0.26
	ADS_MIMF	0.45±0.01	0.27±0.03	0.39±0.03	0.52±0.02	0.60±0.02	4.49	0.27
	ASDYS	**0.34±0.01**	**0.11±0.02**	**0.32±0.03**	**0.42±0.04**	**0.51±0.03**	**2.98**	**0.26**
500	FDWS	0.41±0.01	0.44±0.02	0.39±0.03	0.40±0.04	0.41±0.02	0.00	0.32
	FDS_MIMF	0.47±0.01	0.47±0.04	0.48±0.02	0.47±0.03	0.46±0.03	3.41	0.31
	ADS_MIMF	0.48±0.01	0.26±0.03	0.43±0.02	0.58±0.03	0.64±0.03	4.39	0.33
	ASDYS	**0.39±0.01**	**0.15±0.03**	**0.39±0.03**	**0.47±0.03**	**0.55±0.02**	**2.94**	**0.32**
600	FDWS	0.48±0.02	0.48±0.02	0.48±0.03	0.49±0.02	0.48±0.03	0.00	0.37
	FDS_MIMF	0.53±0.01	0.53±0.02	0.51±0.02	0.55±0.02	0.54±0.03	3.44	0.37
	ADS_MIMF	0.51±0.01	0.27±0.02	0.45±0.03	0.60±0.02	0.71±0.02	4.37	0.38
	ASDYS	**0.44±0.01**	**0.17±0.02**	**0.41±0.03**	**0.55±0.03**	**0.64±0.02**	**2.95**	**0.37**
700	FDWS	0.54±0.01	0.54±0.03	0.52±0.02	0.55±0.02	0.54±0.03	0.00	0.42
	FDS_MIMF	0.58±0.01	0.59±0.02	0.59±0.02	0.57±0.02	0.59±0.03	3.50	0.42
	ADS_MIMF	0.53±0.01	0.28±0.03	0.48±0.03	0.63±0.02	0.75±0.02	4.36	0.44
	ASDYS	**0.47±0.01**	**0.19±0.02**	**0.45±0.02**	**0.57±0.03**	**0.69±0.03**	**2.98**	**0.42**
800	FDWS	0.58±0.01	0.58±0.02	0.58±0.02	0.55±0.02	0.59±0.03	0.00	0.48
	FDS_MIMF	0.64±0.01	0.64±0.02	0.64±0.02	0.64±0.02	0.65±0.02	3.58	0.47
	ADS_MIMF	0.58±0.01	0.29±0.03	0.53±0.03	0.69±0.02	0.80±0.01	4.39	0.50
	ASDYS	**0.52±0.01**	**0.21±0.03**	**0.51±0.02**	**0.63±0.02**	**0.74±0.03**	**3.02**	**0.48**

TABLE IV
FOUR CASES WITH FEWER $S3$ FUNCTIONS

Case	$\|MS(S_3)\|$	$\|MS(S_2)\|$	$\|MS(S_1)\|$	$\|MS(S_0)\|$
Case 1	20	100	100	180
Case 2	40	100	100	160
Case 3	60	100	100	140
Case 4	80	100	100	120

This indicates that the proposed approach does not compromise the DMR of lower-criticality functions when improving the DMR of high-criticality functions. As reported in Table III, the $DMR(S_x)$ values at the four criticality levels for FDWS and FDS_MIMF are much more balanced than those for ADS_MIMF and the proposed ASDYS. This observation is consistent with the characteristics of the approaches where FDWS does not consider mixed criticality and FDS_MIMF not effectively.

Figure 11 shows the average simulation time cost comparison. The average number of times a task gets rescheduled is shown in Table III as an ancillary metric. FDWS always has the least time cost due to its simplicity. Our ASDYS has clear improvement compared with ADS_MIMF, mainly since ADS_MIMF frequently and unconditionally cancels tasks when responding to new arrival, which leads to much more rescheduling, whilst ASDYS adaptively and only partially cancels and reschedules tasks.

Experiment 2: Missing the deadlines of functions with the highest criticality level may have severe consequences. In the design process of safety-critical CPS, some of the functions at the highest criticality level may be decomposed into multiple lower-criticality functions, such as by SIL (Safety Integrity Level) decomposition [1] [18], thereby reducing the amount of these functions. This experiment evaluates the performance when reducing the proportion of S_3-level functions. The total size $|MS|$ is fixed to be 400, with $MS(S_2)$ and $MS(S_1)$ both fixed as 100. We then reduce $MS(S_3)$ from 100, and corre-

TABLE V
DMR COMPARISON WITH FEWER S_3 FUNCTIONS (REPRESENTED AS SAMPLE MEAN±1.96SE (STANDARD ERROR), 95% CONFIDENCE INTERVAL)

Case	Approach	DMR(sample mean±1.96SE)					Reschedule /task	U
		Overall	S3	S2	S1	S0		
1	FDWS	0.36±0.01	0.37±0.09	0.34±0.02	0.36±0.03	0.37±0.02	0.00	0.26
	FDS_MIMF	0.39±0.02	0.39±0.08	0.38±0.03	0.41±0.03	0.40±0.02	3.34	0.26
	ADS_MIMF	0.44±0.01	0.18±0.04	0.27±0.02	0.43±0.03	0.56±0.02	4.30	0.26
	ASDYS	**0.34±0.02**	**0.03±0.03**	**0.16±0.02**	**0.36±0.03**	**0.47±0.03**	**3.48**	**0.26**
2	FDWS	0.37±0.01	0.35±0.06	0.37±0.03	0.37±0.03	0.36±0.02	0.00	0.26
	FDS_MIMF	0.39±0.01	0.40±0.03	0.38±0.03	0.42±0.03	0.39±0.02	3.31	0.26
	ADS_MIMF	0.44±0.02	0.22±0.03	0.32±0.03	0.44±0.04	0.57±0.02	4.36	0.27
	ASDYS	**0.33±0.02**	**0.04±0.02**	**0.19±0.02**	**0.34±0.02**	**0.49±0.03**	**3.36**	**0.26**
3	FDWS	0.35±0.02	0.35±0.04	0.36±0.03	0.36±0.02	0.34±0.03	0.00	0.25
	FDS_MIMF	0.40±0.01	0.41±0.04	0.40±0.03	0.41±0.02	0.40±0.03	3.31	0.25
	ADS_MIMF	0.44±0.01	0.25±0.04	0.34±0.02	0.48±0.03	0.57±0.02	4.34	0.27
	ASDYS	**0.33±0.01**	**0.07±0.02**	**0.23±0.02**	**0.40±0.03**	**0.47±0.01**	**3.22**	**0.25**
4	FDWS	0.35±0.02	0.35±0.03	0.34±0.05	0.36±0.03	0.35±0.04	0.00	0.25
	FDS_MIMF	0.39±0.02	0.41±0.04	0.38±0.03	0.40±0.04	0.39±0.02	3.30	0.25
	ADS_MIMF	0.45±0.02	0.24±0.03	0.41±0.03	0.50±0.03	0.57±0.03	4.35	0.27
	ASDYS	**0.33±0.02**	**0.08±0.02**	**0.29±0.04**	**0.42±0.03**	**0.46±0.03**	**3.06**	**0.26**

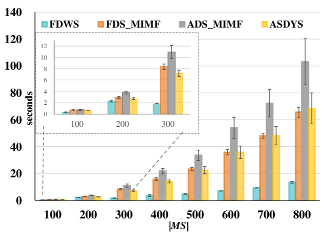

Fig. 11. Time cost on $|MS|$.

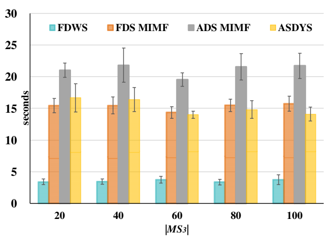

Fig. 12. Time cost in 4 cases.

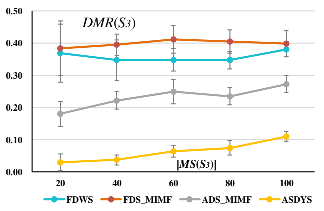

Fig. 13. $DMR(S_3)$ in 4 cases.

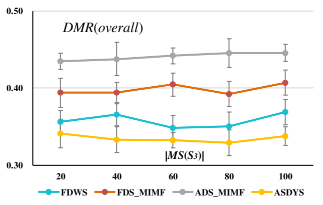

Fig. 14. $DMR(overall)$ in 4 cases.

spondingly increase $MS(S_0)$ from 100. Four cases are shown in Table IV. The results (statistical averages) are reported in Table V and Figure 12. The $DMR(S_3)$ and $DMR(overall)$ are shown in Figure 13 and 14.

Clearly, for all the four cases, our ASDYS is better than the existing approaches in both $DMR(S_3)$ and $DMR(overall)$, especially $DMR(S_3)$. The observation on the timing efficiency (shown in Figure 12) is similar to Experiment 1, where FDWS is simple and fast, and ASDYS outperforms ADS_MIMF.

VI. CONCLUSION AND FUTURE WORK

There is a demand from the industry, such as for highly automated driving, to schedule complex dynamically arriving mixed-criticality functions on distributed heterogeneous architectures. This paper proposes ASDYS as the first approach precisely treating this scenario. Aiming at minimising the DMR of functions, ASDYS actively prioritises higher-criticality functions throughout the scheduling process, which is reflected in the scheduling framework design and the algorithm development. Experimental results show that ASDYS achieves significantly lower DMR for the functions on the highest criticality level, and also performs better in the overall DMR accounting for functions on all criticality levels. Considering the polynomial time complexity, ASDYS can be directly applied to industrial systems, such as on the central vehicular computer, which is expected in future highly automated automobiles.

This work may be extended in several directions. First, it is possible to further reduce the DMR with new scheduling algorithms. Second, the proposed approach has no guarantee on deadlines even for high-criticality functions. One simple yet conservative solution is isolation, i.e., the resources are partitioned and the hard real-time functions get their dedicated portions. Otherwise, complex response time analysis needs to be developed. Third, the workload models can be refined, with, e.g., minimum inter-arrival time of functions, which potentially leads to better performance and which is helpful for the certification. Fourth, the time cost of the dynamic scheduling algorithm can be reduced, where one angle is to adapt it for hardware acceleration.

Acknowledgement

This work was supported by the National Natural Science Foundation of China (Grants 61932010, 61672217, 61702172 and 61972139), the CCF-Tencent Open Fund (Grant CCF-TecentRAGR20190119), the Industry-University-Research Project of United Automotive Electronic Systems Co.,Ltd. (Grant 20TMS0047CN), the Natural Science Foundation of Hunan Province (Grant 2018JJ3076), the Open Research Project of the State Key Laboratory of Synthetical Automation for Process Industries (SAPI), Northeastern University, China (Grant PAL-N201803), and the Fundamental Research Funds for the Central Universities, Hunan University, China.

References

[1] "Road vehicles-functional safety, iso 26262," 2011.

[2] "Road vehicles-functional safety,2nd edition, iso 26262," 2018.

[3] AUTOSAR, "Adaptive platform 19.03," 2019. [Online]. Available: https://www.autosar.org/standards/adaptive-platform/adaptive-platform-1903/

[4] H. Arabnejad and J. Barbosa, "Fairness resource sharing for dynamic workflow scheduling on heterogeneous systems," in *Parallel and Distributed Processing with Applications (ISPA), 2012 IEEE 10th International Symposium on*. IEEE, 2012, pp. 633–639.

[5] G. Xie, G. Zeng, Z. Li, R. Li, and K. Li, "Adaptive dynamic scheduling on multi-functional mixed-criticality automotive cyber-physical systems," *IEEE Trans. Veh. Technol*, vol. 66, no. 8, pp. 6676–6692, 2017.

[6] G. Bernat, A. Burns, and A. Liamosi, "Weakly hard real-time systems," *IEEE Transactions on Computers*, vol. 50, no. 4, pp. 308–321, 2001.

[7] X. Gu and A. Easwaran, "Dynamic budget management with service guarantees for mixed-criticality systems," in *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2016, pp. 47–56.

[8] B. Hu, K. Huang, G. Chen, L. Cheng, and A. Knoll, "Adaptive runtime shaping for mixed-criticality systems," in *2015 International Conference on Embedded Software (EMSOFT)*. IEEE, 2015, pp. 11–20.

[9] S. Vestal, "Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance," in *28th IEEE International Real-Time Systems Symposium (RTSS 2007)*. IEEE, 2007, pp. 239–243.

[10] J. Li, D. Ferry, S. Ahuja, K. Agrawal, C. Gill, and C. Lu, "Mixed-criticality federated scheduling for parallel real-time tasks," *Real-time systems*, vol. 53, no. 5, pp. 760–811, 2017.

[11] S. Baruah, "The federated scheduling of systems of mixed-criticality sporadic dag tasks," in *2016 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2016, pp. 227–236.

[12] M. Hu, J. Luo, Y. Wang, and B. Veeravalli, "Scheduling periodic task graphs for safety-critical time-triggered avionic systems," *IEEE Transactions on Aerospace and Electronic Systems*, vol. 51, no. 3, pp. 2294–2304, 2015.

[13] R. Medina, E. Borde, and L. Pautet, "Scheduling multi-periodic mixed-criticality dags on multi-core architectures," in *2018 IEEE Real-Time Systems Symposium (RTSS)*. IEEE, 2018, pp. 254–264.

[14] G. Xie, G. Zeng, R. Li, and K. Li, "High-performance real-time scheduling," in *Scheduling Parallel Applications on Heterogeneous Distributed Systems*. Springer, 2019, pp. 147–179.

[15] Y. Liu, G. Xie, X. Chen, L. Jin, Y. Tang, and R. Li, "An active scheduling policy for automotive cyber-physical systems," *Journal of Systems Architecture*, vol. 97, pp. 208–218, 2019.

[16] 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.

[17] J. Gan, "Tradeoff analysis for dependable real-time embedded systems during the early design phases," Ph.D. dissertation, Technical University of Denmark (DTU), 2014.

[18] D. Tămaş-Selicean and P. Pop, "Design optimization of mixed-criticality real-time embedded systems," *ACM Transactions on Embedded Computing Systems (TECS)*, vol. 14, no. 3, p. 50, 2015.