Extending resources for avoiding overloads of mixed-criticality tasks in cyber-physical systems

Md. Al Maruf, Akramul Azim

1Department of Electrical and Computer Engineering, Ontario Tech University, Oshawa, Canada
2E-mail: md.maruf@uoit.ca

Abstract: With the increasing number of services and industries including nuclear, chemical, aerospace, and automotive sectors in cyber-physical systems (CPSs), systems are being severely overloaded. CPSs comprises mixed-critical tasks which are of either safety-critical (high) or non-safety critical (low). In traditional task scheduling, most of the existing scheduling algorithms provide poor performance for high-criticality tasks when the system experiences overload and do not show explicit separation among different criticality tasks to take advantage of using cloud resources. Here, we propose a framework to schedule the mixed-criticality tasks by analyzing their deadlines and execution times which leverage the performance of parallel processing through OpenMP. The proposed framework introduces a machine learning-based prediction for a task offloading in the cloud. Moreover, it illustrates to execute a selected number of low-criticality tasks in the cloud while the high-criticality tasks are run on the local processors during the system overload. As a result, the high-criticality tasks meet all their deadlines and the system achieves a significant improvement in the overall execution time and better throughput. In addition, the experimental results employing OpenMP show the effectiveness of using the partitioned scheduling over the global scheduling method upon multiprocessor systems to achieve the tasks isolation.

1 Introduction

Nowadays, cyber-physical systems (CPSs) are becoming complex as they demand advanced support with faster computation [1]. In the past, most of the systems were isolated from each other, but currently the connectivity of the embedded systems is increasing with the expansion of the Internet of Things (IoT) [2] that consist of systems with multiple criticalities. Moreover, many mixed-critical systems are now becoming connected to the cloud for fast and parallel computation. A mixed-criticality (MC) system is defined as the composition of several tasks having various criticality levels where the criticality level is represented by the task’s tolerance to a deadline miss [3]. The criticality level of any task can be described as high or low. A high-criticality task has stringent timing requirements related to safety where the low-criticality or soft real-time tasks have the quality of service requirements. In most cases, the critical level of a task is determined based on prior analysis of the system design. Therefore, in this paper we assume two types of tasks: (a) high-criticality task: that does not allow or tolerate any deadline miss and (b) low-criticality task: that can tolerate occasional deadline misses.

In MC applications, which are often found in CPSs, high-criticality tasks may experience deadline misses because of migration caused during the parallel execution of tasks. The design of a scheduling technique can significantly impact the execution of tasks to finish within the deadline. Different scheduling techniques and methods exist to handle the timing constraints for mixed-critical systems. Moreover, the MC systems have tasks with varying levels of criticality. Some scheduling algorithms also exist to ensure MC timing requirements related to the different safety levels of the system.

Scheduling MC applications in CPSs is an important and critical element in the system design upon uni and multiprocessor platforms. Although various scheduling algorithms exist, each of them has some advantages and limitations compared with other scheduling algorithms in different scenarios. For example, earliest deadline first (EDF) is considered as the most effective and optimal algorithm for dynamic scheduling on both periodic and aperiodic tasks [4]. The EDF scheduling, each task is selected based on the priority that has the earliest deadline compared to the other tasks.

According to the EDF scheduling policy, it ensures the maximum processor utilisation for which it is easy to schedule a set of tasks that maintain system utilisation of ≤1. The system utilisation is calculated by the summation of the ratio of worst execution time divided by period for each task. Although we consider EDF as one of the optimal scheduling approaches, it does not perform well if the MC systems experience any overload (a situation that occurs when the system exceeds the demand of its maximum possible utilisation). Another popular scheduling algorithm is the rate monotonic algorithm that also does not perform well in an overload situation. When using this algorithm, high-criticality tasks can be adequately scheduled, but the low-criticality tasks may be preempted and miss the deadlines. Although no safety violations occur due to the deadline miss of low-criticality tasks, this will affect the overall performance of a MC application.

While examining different MC scheduling algorithms (e.g. partitioned-EDF, global EDF), we discover that they perform better when the system is not overutilised [5]. To handle such a situation, we see increasing use of elastic MC (E-MC) task model that aims at ensuring the minimum service requirements for low-criticality tasks. In another way, the E-MC task model also allows modifiable periods for low-criticality tasks. Thus, this model can enhance the required service levels at runtime without exceeding the worst-case execution time for high-criticality tasks [6]. It is challenging to provide better performance to schedule the MC tasks when the system utilisation exceeds the optimal value for a given scheduling algorithm upon uniprocessor or multiprocessor platforms. Moreover, most research works particularly consider the uniprocessor scheduling to characterise system overload, and implementation associated with multiprocessor schedulings do not examine the effectiveness of parallel processing using OpenMP while overloading. OpenMP is a parallel programming interface that implements a library for scheduling tasks. It provides the flexibility to divide the total number of tasks into multiple chunk sizes and assigns them into multiple processors which reduce the overall execution time [7].

The proposed framework presents a way to schedule the MC tasks for both uniprocessor and multiprocessor systems when the system exceeds the limit of maximum utilisation. The proposed architecture has several states that are responsible for scheduling.
the MC tasks in the system efficiently. We maintain a task queue where tasks arrive randomly and wait for entering into a ready queue state to be scheduled for execution. In this task queue state, we employ a machine learning (ML)-based classification algorithm [8] on the arrival tasks and predicts which tasks are schedulable and which tasks require to offload. By applying early prediction of task offloading, we can minimise the overhead of task schedulability test and CPU utilisation check in the ready queue state as less number of tasks are filtered out from the task queue state. Moreover, due to the characteristics of deterministic execution and less complexity, we choose to use partitioned scheduling algorithm over the global scheduling method for high-criticality tasks in multiprocessor systems implemented through OpenMP [9] that supports parallel processing.

The rest of the paper is arranged as follows: Section 2 describes the motivation of this work and Section 3 discusses the system architecture. Section 4 discusses the proposed approach for MC task scheduling after explaining the advantages and limitations of the existing approaches. Section 5 demonstrates the experimental analysis of the whole work. Section 6 discusses some related work and provides better scheduling result for aperiodic tasks. In this task queue state, the scheduling algorithm to leverage parallel execution.

### 2 Motivation

In real-time embedded systems, a noticeable current trend is to choose parallel processor structures where the systems are connected with cloud services. The main objective of using such architecture is to implement the system in such a way so that the applications are interconnected and share information with each other (e.g. connected autonomous car driving, smart grid). Here, the high-critical or safety-critical tasks are scheduled in such a way so that they always meet their deadlines.

Scheduling algorithms can guarantee meeting deadlines, for example EDF is a well-known technique that guarantees full processor utilisation, facilitates efficient use of available resources, and provides better scheduling result for aperiodic tasks. In contrast to EDF, rate-monotonic (RM) is more straightforward to implement but more suitable for high-critical task [10]. As a result, they render poor real-time performance for low-criticality tasks [11].

Besides that, the adaptation of parallel framework OpenMP in embedded systems is increasing because of a number of reasons such as (a) the acceleration of processing using parallel programming and therefore allowing processors to execute more computational activities and (b) parallel computation offloading to cloud for improved system performance. Overall, mixed-critical real-time embedded systems can benefit from the flexibility delivered by OpenMP and the cloud service (Fig. 1).

### 3 System model and architecture

In conventional systems, real-time computing is studied separately because of its applicability to some specific areas. Currently, the trend has been changed as most of the systems contain MC tasks which make the scheduling difficult. The increasing use of CPSs and the IoT has opened the possibility to connect the MC systems with other networks where safety and security are also considered as important challenges [12]. In a MC CPS [13], we assume, each task is defined as,

\[ t_i = (\delta_i, p_i, T_i, C_i, D_i) \]

where

- \( \delta_i \) represents the degree of criticality,
- \( p_i \) is the priority of any task,
- \( T_i \) stands for the minimum inter-arrival time between two tasks,
- \( C_i \) refers to the worst-case execution time, and
- \( D_i \) refers to the deadline of a task.

The degree of criticality \( \delta_i \) is evaluated by observing the laxity of the worst-case execution time \( C_i \). From the perspective of a dual criticality task model, the degree of criticality can be identified as high criticality or low criticality. We assume that a high-criticality task has two types of execution modes which are overrun mode \( C(\text{HI}) \) and normal mode \( C(\text{LO}) \). Usually, the high-criticality tasks will try to complete its execution being in normal mode. On the other hand, the low-criticality tasks run only in a single mode. In such a task model, \( D_i \) refers to the deadline that demands to complete the functionality of tasks within this period.

To visualise the overall system architecture, we consider that the MC systems are connected with the local server where each MC system contains a set of different criticality tasks. We also assume that all the tasks are executed in the local processor by default. The number of local processors is defined by \( m \). However, if the local processor is unable to schedule the MC tasks and the system experiences an overload, the low-criticality tasks can be run in parallel in the cloud so that the number of task deadline misses becomes minimised.

### 4 Analysis of scheduling algorithms and proposed framework for MC tasks

Depending on a scheduling algorithm, the tasks access the system resources at a different point in time, but all of the algorithms ensure meeting deadlines for high-criticality tasks when the system becomes overloaded. In the following, we will see the working mechanism of two popular scheduling mechanisms, namely EDF and RM upon uniprocessor system. We also discuss scheduling for MC tasks in multiprocessor systems. In the case of a multiprocessor system, the partitioned scheduling and global scheduling techniques are applied on top of any uniprocessor scheduling algorithm to leverage parallel execution.

#### 4.1 EDF scheduling in uniprocessor system

To better understand the EDF scheduling during the overload, we show an example that illustrates the advantages and disadvantages of using EDF for executing MC tasks [14].

We assume a task set \( T \) that includes two tasks having different criticality levels (low or high) need to be scheduled in any uniprocessor system. Let us assume that the low-criticality task is \( t_1 \), where \( t_2 \) is referred to as the high-criticality task. We also assume that the high-criticality task \( t_2 \) has the normal and overrun mode or configuration where the low-criticality task \( t_1 \) always runs on the normal mode. Different activities are performed in different modes of operation. The interarrival times, criticality levels and the worst-case executions times are described in Table 1. Moreover, the tasks have implicit deadlines means the deadlines equivalent to the respective period \( T \). In a typical case, the tasks scheduling using the EDF algorithm is shown in Fig. 2.

| \( r \) | \( \delta \) | \( C(\text{LO}), \text{ms} \) | \( C(\text{HI}), \text{ms} \) | \( T, \text{ms} \) |
|---|---|---|---|---|
| \( t_1 \) | low (LO) | 2 | 2 | 4 |
| \( t_2 \) | high (HI) | 1 | 5 | 6 |

---

Fig. 1 Mixed-criticality CPSs connected with a cloud server

Table 1 MC task set for EDF [14]
In Fig. 2, we see that the periodic low-criticality task has the deadline at every 4 ms where the periodic high-criticality task has the deadline at every 6 ms. At the very beginning, if both of the tasks start to be scheduled, the low-criticality task \( \tau_1 \) will have a higher priority as it has the earliest deadline of 4 ms. In the normal mode, \( \tau_1 \) will be executed for 2 ms first, and after that, the \( \tau_2 \) will get the opportunity to be completed which takes 1 ms. This process continues as long as both the tasks run on the normal execution mode. The problem occurs when the high-criticality task starts running in the overrun mode. In this scenario, \( \tau_2 \) is executed for an extra 4 ms which increases the execution time. Fig. 2 shows that at the time of 14 ms, \( \tau_2 \) enters into the overrun mode with a deadline of 19 ms and it misses the deadline because the low-criticality task has the higher priority to run due to its earlier deadline. Therefore, the system fails to schedule the high-criticality task.

As we know, in any scenario, if the tasks run in the usual or normal mode, the EDF guarantees safety by scheduling both the tasks appropriately. The EDF algorithm only fails when the high-criticality tasks run in the overrun mode, and low-criticality tasks get higher priority based on the earliest deadline. To overcome the deadline misses of high-criticality task, we assume that it will be executed first. Moreover, the system moves into a safe state by running the high-criticality tasks before the low-criticality tasks. Fig. 3 shows that task \( \tau_2 \) gets the opportunity to be executed first in each period. The prioritisation of high-criticality task helps to meet the deadline even if \( \tau_1 \) runs in the overrun mode. Although the system becomes safe after executing high-criticality tasks, we now observe the deadline misses of low-criticality tasks. During the run-time from 12 to 16 ms, the low-criticality task \( \tau_1 \) fails to meet the deadline and the task \( \tau_1 \) needs to be dropped immediately to assure safety. The overall discussion clearly illustrates that the EDF scheduling policy struggles to schedule the MC tasks when the system switches to overrun mode due to experiencing overloads.

4.2 RM scheduling in uniprocessor system

In many real-time systems, the scheduling algorithms are categorised based on the priority assignment where most of the real-time operating systems support preemptive task scheduling for which the highest priority tasks always get preference to be executed first. The standard priority assignment policies are based on dynamic priority, static priority, or mixed priority. A dynamic priority algorithm assigns the priority of each task during the runtime where the static priority algorithm specifies the task priority before the execution of all the tasks. In addition, the mixed priority algorithm offers both options for selecting the priority of any task at the very beginning of the execution or during the runtime. In the case of the static scheduling algorithm, the priority of the tasks remains identical throughout the execution time. On the other hand, the dynamic algorithm selects the priority by analysing different essential parameters that influence the execution schedule. However, the static priority algorithm is easy to implement compared to the other algorithms. RM considers the priority of a task is based on the period of that task [15] where

- the task with the smaller period is referred to as the high-priority task,
- the priority algorithm is static that holds the fixed priority for each task as the algorithm considers all the tasks are of equal importance.

Let us assume that \( \tau_1 \) and \( \tau_2 \) are two periodic tasks in a MC system. Moreover, \( \tau_1 \) requires an execution time of 1 ms out of 2 ms time slot and \( \tau_2 \) needs 2.5 ms out of 5 ms time slot. We consider the scheduling of both tasks upon a single processor where preemption of any task is always acceptable by a higher priority task. However, the RM algorithm fails to meet deadlines for the task set that is described in Table 2. Fig. 4 shows that the high-criticality task misses the deadline for the low-criticality task.

4.3 Scheduling in multiprocessor system

With the increasing availability of multiprocessors, many real-time systems prefer the multiprocessor architecture for high computational tasks. Although the multiprocessor systems support parallel tasks execution, still research is ongoing to find better strategies to schedule tasks efficiently. To schedule an \( n \) number of tasks on an \( m \) number of processors, a few numbers of scheduling approaches are commonly used. Among these algorithms EDF, RM, proportionate-fair (Pfair) and EDF first fit (EDF-FF) algorithms [16] show less advantages and disadvantages related to the deadline miss and overall performance. For example, the RM scheduling approach provides predictable worst-case execution for critical tasks but the algorithm runtime complexity is higher than the EDF for its recursive characteristics. One important problem of this algorithm is that the CPU utilisation can be guaranteed on multiprocessors for independent tasks is only 41% [16].

Moreover, the characteristics of multiprocessors are also important in tasks scheduling. Multiprocessors can be categorised as follows [17]:

- **Type 1**: each processor possesses similar computational rate and abilities.
- **Type 2**: each processor has a similar configuration of capabilities including varying speeds.
- **Type 3**: each processor is separated from each other and heterogeneous.

One of the significant contributors to the multiprocessor scheduling problem is resource allocation. This problem states that the scheduling algorithm requires to map the appropriate task in the proper processor. Besides, it demands the timing guarantee for each task to ensure the stability of the system. However, the partitioned and global [17] scheduling algorithms are frequently practiced in multiprocessor systems.

4.3.1 Partitioned scheduling: A partitioned scheduling algorithm allocates resources for an individual task to a particular processor [17]. Fig. 5 shows an example of the partitioned scheduling that each task \( \tau_i \) is assigned on the individual processor. For example, \( \tau_1 \), \( \tau_2 \), and \( \tau_3 \) are running separately in processor one two and three, respectively, where processor three will remain idle later as no task

---

**Table 2** MC task set for RM

| \( \tau \) | Criticality | Execution time, ms | Period, ms |
|---|---|---|---|
| \( \tau_1 \) | LO | 1 | 2 |
| \( \tau_2 \) | HI | 2.5 | 5 |

Fig. 2 MC tasks scheduling using EDF [14]

Fig. 3 MC tasks scheduling (higher priority for high-criticality task) [14]

Fig. 4 MC task scheduling using RM

---

This is an open access article published by the IET under the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/)
is assigned in it. The partitioned scheduling algorithm has both pros and cons. The algorithm has the following advantages:

- It allows static task assignments.
- Partitioned scheduling supports most of the single processor scheduling approaches.
- If any high-criticality task runs on overrun mode, it affects only the task-associated processor where other processors remain unaffected.
- As tasks are not allowed to switch the processors, there is no migration cost.
- To design partitioned scheduling algorithms, each processor maintains a separate queue for handling tasks.

The principal limitations of this partitioned scheduling algorithm are listed below:

- This algorithm cannot help to share all the available resources equally.
- Since tasks avoid migration, the system may experience a low processor utilisation.
- Although any processor can become idle at runtime, it cannot be assigned for mapped tasks to other processors.

4.3.2 Global scheduling: Using a global scheduling algorithm, all the tasks can also be scheduled upon multiprocessors. The global queue cannot be applied to overcome the multiprocessor scheduling problem in safety-critical systems because it allows task migration among the available processors. The migration of any task usually adds extra overhead, and the system may show unpredictable behaviour due to the dynamic changes in the task environment. Since the algorithm maintains a single global queue, the implementation and design become more complicated for different MC applications [17]. Fig. 6 presents an example of global scheduling. In comparison with the partitioned scheduling approach, the advantages of global scheduling are given below:

- All the immediate tasks are kept in a single global queue.
- Since task migration is allowed, the global scheduling ensures the maximum resource utilisation.
- Most of the multiprocessor operating system now supports the global scheduling.

On the other hand, some of the limitations of this approach are addressed below:

- Most of the uniprocessor scheduling algorithms cannot utilise the processors completely in multiprocessors.
- Task migration may lead to an unpredictable state for the safety-critical system.
- Due to task migration, various processors may cause varying computational rates which decrease the optimal performance.
- Tasks migration causes an extra overhead in the system because all the processors use shared memory for communicating among them.
- Poor resource utilisation for high-criticality tasks.

4.4 Schedulability test for MC tasks

4.4.1 Uniprocessor schedulability: In EDF scheduling, the task with the shortest deadline is considered for earliest execution because of the highest priority [18]. A standard way to know the schedulability condition for a task set using EDF is that the total processor utilisation bound for the defined task set should be ≤1. We assume that we have a set of periodic tasks \( r_1, r_2, ..., r_m \) where the condition [18] for EDF schedulability test is the following:

\[
\sum_{j=1}^{n} \frac{C_i}{T_j} = \sum_{j=1}^{n} u_i \leq 1
\]  

Let us assume that all the tasks of Table 1 have deadlines that are equal to their respective periods. In this scenario, the total task utilisation is equal to 1.3 which is higher than 1. We use the least common multiple (LCM) that defines the size of the process periods. We envision a timeline equal in length to the least common multiple of the time periods for scheduling a set of tasks within it. In the following calculation, we observe that the tasks are not schedulable:

\[
\text{LCM}(4, 6) = 12
\]

\[
U(t) = \frac{2}{4} + \frac{5}{6} = 1.3 \geq 1
\]

4.4.2 Multiprocessor schedulability: Similarly, to schedule successfully, any periodic task set on \( m \) processors can have utilisation \( U(t) \) no greater than \( m^2/(2m - 1) \) [19]:

\[
U(t) \leq \frac{m^2}{2m - 1}
\]  

In a safety-critical system, the partitioned algorithm is used more than the global scheduling upon a multiprocessor platform. The reason behind the choice of partitioned scheduling is to use the processors better where the global scheduling shows more unpredictability due to processor migration and unavailable cache information.
For example, Table 3 shows a set of tasks and these tasks are MC tasks. When scheduling these tasks in a two-core system, the schedulability test using (3) indicates that the tasks are not schedulable in static scheduling because process utilisation is higher than the defined value. Therefore, not all tasks meet their deadlines.

Calculation:
\[ \Rightarrow LCM(2, 5, 3) = 10 \]
\[ \Rightarrow U(t) = \frac{1}{2} + \frac{2.5}{3} + \frac{3}{5} = 1.6 \]
\[ \Rightarrow U(t) \leq \left( \frac{m^2}{2m - 1} \right) = \frac{4}{3} = 1.3 \]

In uniprocessor and multiprocessor systems, once system utilisation becomes higher than the optimal value, then the low-criticality or high-criticality tasks are not always guaranteed to be scheduled not to meet the deadlines.

4.5 Proposed framework for MC tasks scheduling

The proposed framework presents a scheduling guideline for MC tasks when the existing scheduling approaches fail to schedule them if an overload occurs (system utilisation exceeds its limit). To understand the overall proposed approach, we show an abstract architecture employing ML-based task offloading prediction which is shown in Fig. 7. The architecture has several states that are responsible for decides which task will be executed where and when it be ready to execute. Therefore, we define six states which are discussed below:

- **Task queue**: All the dynamically arrival tasks in the system are put into the task queue. In this state, we propose to run a ML classifier that predicts upon available tasks in the task queue and tags which tasks are not schedulable in the system. After identifying non-schedulable low-criticality tasks, the selected tasks are transferred to the offloading state for execution in the cloud server.
- **Ready queue**: The ready queue accepts those tasks that are coming from the task queue. In this queue, tasks usually wait to be scheduled on different processors. However, it does not necessarily need to be placed beside the main memory. The tasks of this queue are generally stored as a linked list and different scheduling approaches can be used to manage the ready queue. In this stage of our work, the task set provides a fixed number of tasks where we perform the CPU utilisation and schedulability test for a particular time frame. If the ready task set fails to schedule any low-criticality task, it transfers that task to the offloading state again so that no high-criticality task misses the deadline while executing locally.
- **Running**: When a task enters into this state, it starts to execute and can be preempted or blocked at any time according to the scheduling policies.
- **Waiting**: When tasks require input–output (I/O) requests for further execution, they wait in the waiting state and select their next move based on instructions.
- **Cloud offloading**: In this state, the tasks are set to offload in the cloud.
- **Terminated**: The tasks that are completed or dropped enter into this terminated state.

A MC CPS may combine two or more distinct tasks with different criticality level. In this scenario, it is assumed that the MC CPS contains an n number of tasks where the number of processors is m. The workflow of our proposed approach to schedule all these MC tasks is discussed below.

4.5.1 ML classifier: ML classification is a way to classify a given data into a desired and distinct number of classes assigning a label to each class. Alternatively, it is a predictive modelling approach that maps a function (f) from input variables (X) to discrete output variables (y = f(X)). In a formal way, a learning algorithm processes a training set \( \tau = (\tau_i, y_i) \), \( \tau_i \in \tau \) and \( y_i \in [1, -1] \); i = 1, ..., n comprising n examples \( (\tau_i, y_i) \), where \( \tau_i \) represents a task input and \( y_i \) is the associated output. The output of this prediction can be a numerical value for the regression model or a class label for the classification model [20].

The proposed approach aims to use Naive Bayes classifier [21] to estimate the syntactical and semantic similarity of the execution pattern of low- and high-criticality tasks. Although other classifiers can be employed, we select Naive Bayes as the classifier model, is fast to build and it can be easily modified with new training data without having rebuild the model. Here, \( y_i \) (dependent variable) is predicted using a k number of input features of task \( \tau_i \) (independent variable). Naive Bayes classifier is based on the Bayes rule of conditional probability and assumes each attribute of the features is independent and equally important. Therefore, the demand prediction model using the Naive Bayes can be given as

\[
P(y | \tau_1, \tau_2, ..., \tau_k) = \frac{P(y)P(\tau_1 | y)P(\tau_2 | y) \cdot ... \cdot P(\tau_k | y)}{P(y)}
\]

In this approach, we consider a few features for training the model such as \( \tau_i \) is the execution time of task \( \tau_i \), \( \tau_i \) is the last arrival time of task \( \tau_i \), \( \tau_i \) is the previous schedulability status, \( \tau_i \) is the required time to test the schedulability of task \( \tau_i \), \( \tau_i \) is the criticality of task \( \tau_i \), and \( \tau_i \) is the total execution cost for offloading in cloud. Our goal is to predict those tasks which are certainly not possible to schedule locally and those tasks must be offloaded in the cloud. As a result a selected number of tasks are offloaded to the cloud and remaining tasks are transferred to ready queue state for scheduling.

In the ready queue state, we check the utilisation of the tasks that determines whether all the MC tasks will be adequately schedulable or not. In any situation, if the utilisation of the tasks exceeds the optimal value (for uniprocessor or partitioned scheduling it is 1), this proposed framework separates the MC tasks based on their criticality levels and offload the low-criticality tasks in the cloud and the high-criticality tasks in local processors. This execution ensures that the safety-critical tasks can achieve their timing requirements and the low-criticality tasks are scheduled as many as possible with improved performance. To schedule the MC tasks that are waiting in the ready queue state, we extend our proposed architecture which is shown in Fig. 8. We summarise the necessary modules as follows:

4.5.2 Execution controller: For scheduling the tasks, initially, a schedulability test is performed by the execution controller to check the system overload or the tasks resource utilisation based on their deadlines and execution times. If the utilisation of tasks is >1 (uniprocessor or partitioned scheduling) or higher than \( m^2/(2m - 1) \) (multiprocessor), our framework separates the tasks into two sets \( (\tau_{\text{high/critical}}) \) based on their criticality and overall execution cost calculation. The execution controller continuously scans the task resource utilisation through overload monitor function and estimates cost \( (\Delta_{\text{cloud offloading}}) \) considering response time, the worst execution time of the cloud services. Moreover, the proposed framework computes the communication delay \( (d_{\text{high/critical}}) \) and low-criticality tasks processing time \( (C_{\text{low/critical}}) \) using a suitable cloud analysis tool to ensure the schedulability of the system. In our case, we use the Cloud Analyst [22] which is a CloudSim-based tool for modelling and analysis of large-scale cloud computing infrastructure. The reason behind the selection of this tool is that it provides the flexibility to the application designers in identifying the optimal configuration for their application. The designer can design the cloud data centres and select the load balancing algorithms and available value-added services. Moreover, it is easy to use due to its graphical simulation interface, a high degree of configurable options, flexibility, repeatability, and ease of extension. While the execution controller verifies the
The processing of high-criticality tasks is that the cloud environment and the task execution time, respectively:

The low-criticality tasks will be examined first to run in the local scheduled only if the local resources are available and no other internet. The high-criticality or safety-criticality tasks are criticality tasks in the cloud server. The reason behind the local $\chi$ does not guarantee to meet the deadline or latency bound over the cloud.

Fig. 8 Framework for MC tasks scheduling

estimation of low-criticality tasks execution cost and time, the task assignment method will decide which tasks should be executed in the cloud.

4.5.3 Task assignment: A task assignment or partitioner manages the tasks so that they are correctly scheduled. OpenMP is used to address the issue where the partitioned scheduling algorithm (P-EDF) will be used to increase the predictability of high-criticality tasks. The system executes all the high-criticality tasks on the local processors and the selected numbers of low-criticality tasks in the cloud server. The reason behind the local processing of high-criticality tasks is that the cloud environment does not guarantee to meet the deadline or latency bound over the internet. The high-criticality or safety-criticality tasks are scheduled in such a way so that they do not miss their timing requirements. To schedule the low-criticality tasks we have two options. The first option is that the low-criticality tasks will be scheduled only if the local resources are available and no other high-criticality tasks requests exist. In this case, the schedulability of the low-criticality tasks will be examined first to run in the local processors along with high-criticality tasks. Equations (2) and (3) will be used to test the task schedulability.

The other option is that the tasks will be offloaded to the cloud. The low-criticality tasks that reach the deadlines with the estimated new worst execution time are selected for offloading in the cloud. The new worst execution time $\chi$ of a task is determined using (5) where $a_i$ is the clock time when a low-criticality task is released to offload in the cloud, $d_{\text{cloud}}$ and $c_{\text{cloud}}$ are the communication delay and the task execution time, respectively:

$$c_i = a_i + d_{\text{cloud}}(\tau_{\text{low}}^i) + C_{\text{cloud}}(\tau_{\text{low}}^i)$$

Thus, the task assignment method starts releasing the selected tasks periodically until the system attains stability. This simultaneous task execution will increase the schedulability guarantee of the system. Algorithm 1 (see Fig. 9) illustrates the whole framework for executing the tasks in cloud and multiprocessors parallelly.

4.5.4 Task isolation: This framework allows isolation among the MC tasks, which is essential for security and safety purposes. Once the low-criticality tasks are executed in the cloud, it does not make any interference with the high-criticality tasks. The deadline misses of low-criticality tasks will not affect the high-criticality tasks. In this way, the system remains in a stable state because there is less chance of an attack in the high-criticality tasks.

5 Experimental analysis

The goals of this experiment are: (a) monitor real-time system overload, (b) classify tasks to offload in the cloud using ML, (c) analyse the MC tasks execution time for different scheduling approaches, and (d) visualise the performance of tasks execution in the cloud and local processors. The applicability of experimental goals is illustrated using several scenarios.

Scenario 1: When an overload occurs in a CPS? Nowadays, many real-time video and image processing applications are integrated into CPSs for object detection where systems may often experience overload. To visualise the system overload, we consider a real-time image processing application that reads input images using Dask [23] parallel computing library.

To design our experiment, we consider a quadcopter mixed-criticality system that consists of a flight controller (high-criticality tasks) and a camera controller (low-criticality tasks). The camera controller manages the image processing tasks for object detection.

In this experiment, we read 200 images with each image size of $(6623 \times 6426)$ using python Dask array. The chunk size to read an input image is $(200 \times 200)$. The whole experiment is run on a multiprocessor system which has four logical x64-based processors, 16.0 GB RAM, 64-bit Windows 10 operating system. During the image processing, the application performs the different distinct tasks that have multiple jobs to compute the image data. In this example, we have five individual tasks that combine a total of 5135 jobs to read the images parallelly.

Observation: We observe the task stream of the image processing to identify the system overload. Fig. 10 shows a bar graph which shows the computing progress of five tasks. Applying the separate calculation, we estimate that the final merging task named ‘rechunk–split–rechunk–merge’ should complete ten jobs within 2 min after the processing starts. However, at time 2 min 59.3 s, we observe that the task named ‘rechunk–split–rechunk–merge’ does not have any progress due to system overload. This overload depicts the number of inputs to a system exceeds its processing capacity for which the task misses its deadline eventually. Moreover, we visualise the CPU utilisation and the memory usages during that period to verify the system overload has occurred actually. Fig. 11 shows that the overall CPU utilisation is always 100% after a certain time and the memory usages are much higher than the usual usages. Hence, offloading a few tasks in the cloud can drastically reduce the system CPU utilisation for less number of tasks’ execution which eventually reduces the system overload, and the system can retain its stable state.

We also observe the Dask’s default global task scheduling policy that adds overhead (extra execution time) for tasks migration. Fig. 12 shows a small task stream portion of the overall computation where the task migration overhead is evident clearly. Furthermore, the effect of the migration cost on the missed deadline is compared among two popular scheduling approaches focusing OpenMP in Scenario 4.

Scenario 2: Task classification using ML algorithm: In this scenario, we use the Naive Bayes classifier to predict which low-criticality tasks need to be offloaded in the cloud if there is an overload situation.

| Task set | Criticality | Deadline | Execution time |
|---------|------------|----------|----------------|
| $\tau_1$ | LO         | 2        | 1              |
| $\tau_2$ | LO         | 5        | 2.5            |
| $\tau_2$ | HI         | 5        | 3              |

Fig. 7 State machine diagram for executing MC tasks

Fig. 8 Framework for MC tasks scheduling

This is an open access article published by the IET under the Creative Commons Attribution License (http://creativecommons.org/licenses/by/3.0/)
We train the Naive Bayes model using the synthetic data of 200 tasks and the features are considered according to the assumptions which are listed in the proposed ML classifier section. To identify the value of each feature, we run multiple times all the tasks at a time by changing their timing proprieties and store the output. We manually validate the output class for training the model. We consider two output class local execution (‘0’ if probability ≤ 0.5) and cloud execution (‘1’ if probability > 0.5). After that we level each of the tasks in the training data either as ‘0’ or ‘1’. To test the training model, we fit ten tasks as test data and try to predict which tasks require to be offloaded in the cloud. Among these ten tasks first three tasks are marked as high-criticality tasks and others are labelled as low-criticality tasks.

Observation: Using this Naive Bayes algorithm, we see the test accuracy in predicting a task execution platform is almost 61%. The test accuracy can be further improved by analysing the impact of other features which we consider for our future work. Fig. 13 shows that tasks τ₆ to τ₁₀ are identified for offloading in the cloud and the remaining tasks (τ₁ to τ₅) are classified for local execution. However, the early prediction of task offloading after analysing the task queue state can avoid the further expensive schedulability test (e.g. RM test) in the ready queue state. Thus, it can help to improve the performance of the system reducing the overall execution time.

Scenario 3: Execution-time analysis: In this scenario, we visualise the insights of the total execution time of MC tasks in uniprocessor by examining several key factors and their influence in meeting the tasks deadlines.

To design the scenario, it is considered that we have a set of ten tasks, where each task has an individual criticality level and deadline. The defined tasks are τ₁ (generating prime numbers), τ₂ (matrix multiplication), τ₃ (integer factorisation), τ₄ (integer sort), τ₅ (Fibonacci sequence generation), τ₆ (Pi digit calculation), τ₇ (bubble sort), τ₈ (insertion sort), τ₉ (heap sort), and τ₁₀ (merge sort). To set the deadline, we execute each task 100 times on different processors and store the execution times to find the worst-case execution time which is assumed as deadline. Therefore, τ₁[2, 6], τ₂[1.6, 5], τ₃[1.5, 4], τ₄[1.2, 1.2], τ₅[0.7, 0.7], τ₆[0.8, 0.8], τ₇[0.7, 0.7], τ₈[1.0, 1.0], τ₉[0.5, 0.5], and τ₁₀[0.6, 0.6] seconds show the execution time and the deadline of each task. After calculating the tasks utilisation, we observe that the tasks utilisation is much higher than the optimal value for both the quad-core processor and uniprocessor. The proposed framework schedules all the high-

![Algorithm 1: MC tasks scheduling approach](image1)

![Tasks computing progress in image processing](image2)

![CPU utilisation and memory usage monitoring during system overload](image3)

(a) CPU utilisation, (b) Memory usage
criticality tasks applying OpenMP including the static approach in local processor, so that the deadlines of high-criticality tasks are met. To execute the tasks in parallel, OpenMP provides an advantage to select the number of execution threads based on the available tasks and processors in a system. Moreover, a programmer can define the OpenMP thread function to set the number of threads automatically against the available tasks that will be executed. On the other hand, the low-criticality tasks are executed simultaneously in the cloud.

Initially, $\tau_1$, $\tau_2$, and $\tau_3$ are defined as high-criticality tasks and others as low-criticality tasks. According to the proposed approach, all three high-criticality tasks are scheduled first using the static approach. The Amazon Web Services (AWS) is used to run the low-criticality tasks in the cloud environment for this experiment.

Fig. 14 shows the total execution time for a quad-core processor system. It is also demonstrated that the execution of low-criticality tasks on the cloud environment decreases the overall response time significantly. The total execution time for the different number of tasks increases when the utilisation of the tasks becomes higher than 1. In this situation, the proposed architecture executes the low-criticality tasks in the cloud in parallel. Such execution requires a less overall execution time for all the MC tasks. Fig. 14b shows the total execution time for a quad-core processor system. It shows that the execution of the low-criticality tasks in the cloud environment decreases the overall execution time significantly.

**Scenario 4: Deadline-based analysis:** According to the proposed architecture, it is shown that the local processors should run all the high-criticality tasks using the partitioned scheduling algorithm for multiprocessor systems to meet the deadlines of the tasks. In this scenario, an experimental analysis demonstrates the
reason for selecting the partitioned scheduling over the global scheduling algorithm.

In this experiment, $r_1$, $r_2$, and $r_3$ are defined as high-criticality tasks like earlier and others as low-criticality tasks. A multiprocessor system is used in the experiment which has four logical processors, 64-bit Ubuntu 16.04 operating system, and 5.4.0 GCC compiler which supports OpenMP 3.0 and 4.0 versions.

Observation: Table 4 shows the number of missed deadlines for partitioned scheduling approach, where the number of missed deadlines for high-criticality tasks is zero, but the number of deadline misses for low-criticality tasks is high. The experiment is run for more than once, and it shows that each time the low-criticality tasks miss their deadlines.

On the other hand, Table 5 shows that the number of missed deadlines is much lower than the partitioned scheduling approach. Although the number of deadline misses is low, it shows that the number of missed deadlines of high-criticality tasks is noticeable.

Scenario 5: Cloud communication time and cost analysis: In this scenario, we analyse the cloud response and processing time including virtual machine and data transfer cost for efficient offloading decision. The Cloud Analyst [24] tool is used as an example to assist in the experimental analysis.

To estimate the tasks execution time affected by varying key factors, the execution controller analyses the load distribution on various cloud data centres according to processing demand. However, we consider the Cloud Analyst [24] tool as an example, but other tools can be associated to analyse the tasks communication delay and cost. The analysis tool uses the round-robin algorithm to achieve improved response time and processing time. Moreover, we perform a simulation considering the user data request and available data centres which are listed in Table 6. The ‘User base’ section of Table 6 shows the number of task requests from different users and the ‘Data centres’ section shows the available cloud servers information for computation.

The simulation is conducted for 60 min where the results of overall response time, data processing time, VMware cost and data transfer cost are listed in Table 7. Throughout the similar experiment, the execution controller obtains the idea about the worst execution time and the execution cost for each task offloading decision. Once the execution controller identifies all the results of relevant parameters that have an impact on the task execution, the task assignment module decides which tasks will be executed where.

Overall, we realise that most of the MC tasks scheduling approaches focus on high-criticality tasks for timing guarantee which provides poor performance at the same time for low-criticality tasks. According to the proposed architecture, it is efficient to run the low-criticality tasks on the cloud when the

---

### Table 4 Result partitioned scheduling approach

| $r_1$ | $r_2$ | $r_3$ | $r_4$ | $r_5$ | $r_6$ | $r_7$ | $r_8$ | $r_9$ | Deadline misses |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-----------------|
| ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |
| ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |
| ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |
| ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |
| ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |
| ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |
| ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |
| ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |

### Table 5 Results using the global scheduling approach

| $r_1$ | $r_2$ | $r_3$ | $r_4$ | $r_5$ | $r_6$ | $r_7$ | $r_8$ | $r_9$ | Deadline misses |
|-------|-------|-------|-------|-------|-------|-------|-------|-------|-----------------|
| x     | x     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |
| x     | x     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |
| x     | x     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |
| x     | x     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |
| x     | x     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |
| x     | x     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |
| x     | x     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |
| x     | x     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | ✓     | 5               |

### Table 6 User base and data centres configuration

| Name | Region | Requested tasks per user, h | Data size, bytes | Peak hours start (GMT) | Peak hours end (GMT) | Avg. peak users | Avg. off peak users |
|------|--------|-----------------------------|------------------|------------------------|----------------------|-----------------|---------------------|
| user base: | | | | | | | |
| UB1  | 0      | 10                          | 1000             | 14                     | 15                   | 1200            | 100                 |
| UB2  | 1      | 8                           | 1000             | 14                     | 15                   | 1000            | 100                 |

### Table 7 Data centre #VMware (Xen) Image size OS (linux) Memory Bandwidth Cost per VM/H Data transfer cost, $/Gb Memory cost, $/s

| Data centres | | | | | |
| DC1  | 5 | 10,000 | 1000 | 1000 | 0.1 | 0.1 | 0.05 |
| DC2  | 5 | 10,000 | 1000 | 1000 | 0.1 | 0.1 | 0.05 |
utilisation of the tasks exceeds the optimal value defined for schedulability.

6 Related works

There are many MC scheduling algorithms which show different techniques to schedule high-criticality tasks efficiently, but in most of the cases, they ignore improving the execution of the low-criticality tasks. Examples are given as follows.

Multicore EDF with virtual deadlines: In this work, a single core MC scheduling algorithm is proposed by introducing virtual deadlines in EDF algorithm (EDF-VD) [23]. They present a scheduling architecture that can be implemented on virtualised multiprocessor systems. In addition, they show a comparison between the partitioned scheduling and global scheduling algorithm through the analysis of schedulability guarantees.

Adaptive mixed criticality (ACM-WH) framework: Oliver Gettings et al. [26] present a novel scheduling approach that assures the least quality of service for low-criticality tasks when the high-criticality tasks run in the overrun mode. The scheduling technique is called ACM-WH for weak and hard tasks set. Pathan [27] analyses the response time and presents a sufficient schedulability condition for global fixed-priority scheduling algorithm on preemptive multiprocessors.

Erickson et al. [28] present the first multicore MC multicore framework that demonstrates the difficulty of task scheduling when a transient overload occurs. They address a few conditions that could cause an overload and increase the response time in the long run. The proposed MC framework deals with these conditions using a virtual-time mechanism to adjust the period of low-criticality tasks. However, a few works also available that only focus on the tasks offloading to the cloud rather than considering the tasks criticality or the system overload. For example, Nan et al. [29] show the tradeoff between average response time and average cost by developing an online algorithm, unit-slot optimisation, based on the technique of Lyapunov optimisation. The intended algorithm schedules the tasks among available resources in the fog computing systems. Krishna et al. [30] propose another model for task offloading using a learning automata-based decision-making algorithm. The model analyses the execution time and energy consumption during the allocation of the tasks to the suitable VMs in the cloud.

Ekberg and Yi [31] propose the demand bound approach which analyses the constrained deadline tasks. The scheduling approach used in EDF-VD which sets the virtual deadlines of various criticality tasks. This allows the low-criticality tasks to be abandoned when the high-criticality tasks run in overrun mode. Giorgio Buttazzo et al. [32] propose the elastic scheduling approach by allowing the low-criticality tasks to change their execution rate for the different quality of services. This algorithm shows how to control the execution rates of running tasks when the system is overloaded. Similarly, another elastic task model in the MC system is used to decrease the overload from the high-criticality tasks by extending the low-criticality tasks period [33]. It allows the low-criticality tasks to minimise the utilisation by introducing a stretching factor in making a scheduling decision.

All the methods introduced so far share a straightforward concept – to have different system models for various criticality levels but when the high-criticality tasks run on the overload mode, they either drop the low-criticality tasks or change the execution rates for achieving the minimum quality of service. These affect the overall performance in terms of user experience and quality of service. However, if we use the advantage of cloud service, we have an opportunity to meet all the requirements of the MC tasks.

7 Conclusion

In MC CPSs, various tasks that are having different levels of priorities and criticality need to be executed in such a way so that the high- and low-criticality tasks achieve timing guarantees for the reliable and correct system operations. However, most of the existing MC algorithms only focus on timing guarantees of high-criticality tasks where the system introduces a poor performance for the low-criticality tasks. The existing scheduling algorithm shows different ways the schedulability can be ensured where the total calculated tasks utilisation follow an optimal value both for uniprocessor and multiprocessor systems. The system fails the schedulability test when the utilisation of tasks exceeds the optimal value, and the tasks miss their deadlines. In this situation, we propose a new framework to execute the low-criticality tasks in the cloud and the high-criticality tasks in the local machine so that the timing guarantees are ensured for high-criticality tasks. To minimise the overhead of schedulability and CPU utilisation test for a large number of tasks, we incorporate the Naive Bayes ML classifier that predicts which tasks will be offloaded in the cloud. In addition, we also run the high-criticality tasks in parallel using OpenMP and the partitioned scheduling algorithm upon multiprocessor systems. Separating the low-criticality tasks from high-criticality tasks also creates strong tasks isolation which is essential for the CPSSs.

8 References

[1] Yue, X., Cai, H., Yan, H., et al.: ‘Cloud-assisted industrial cyber-physical systems: an insight’. Microprocess. Microsyst., 2015, 39, (8), pp. 1262–1270
[2] Jazdi, N.: ‘Cyber physical systems in the context of industry 4.0’. 2014 IEEE Int. Conf. on Automation, Quality and Testing, Robotics, Chj-Napoca, Romania, 2014
[3] Burns, A., Davis, R.: ‘Mixed criticality systems-a review’. Technical Report, Department of Computer Science, University of York, December 2013, pp. 1–69
[4] Dertouzos, M.: ‘Control robotics: the procedural control of physical processes’. Proc. IFIP Congress, Stockholm, Sweden, 1974, pp. 807–813
[5] Su, H., Zhu, D., Mossé, D.: ‘Scheduling algorithms for elastic mixed-criticality tasks in multicore systems’. 2013 IEEE 19th Int. Conf. on Embedded and Real-Time Computing Systems and Applications, Taipei, Taiwan, 19 August 2013, pp. 352–357
[6] Su, H., Zhu, D.: ‘An elastic mixed-criticality task model and its scheduling algorithm’. 2013 Design, Automation & Test in Europe Conf. & Exhibition (DATE), Grenoble, France, 2013
[7] Mattson, T.G., Sanders, B., Masiungil, B.: ‘Patterns for parallel programming’ (Pearson Education, UK, 2004)
[8] Mahdavinejad, M.S., Rezvan, M., Barekatain, M., et al.: ‘Machine learning for internet of things data analysis: a survey’, Digit. Commun. Netw., 2018, 4, (3), pp. 161–175
[9] Dagum, L., Menon, R.: ‘OpenMP: an industry-standard API for shared-memory programming’, Comput. Sci. Eng., 1998, 5, (1), pp. 46–55

[10] Buttazzo, G.C.: ‘Rate monotonic vs. EDF: judgment day’, Real-Time Syst., 2005, 29, (1), pp. 5–26

[11] Ren, J., Pfan, L.T.X.: ‘Mixed-criticality scheduling on multiprocessors using task grouping’. 2015 27th Euromicro Conf. on Real-Time Systems, Lund, Sweden, 2015

[12] Wolf, M., Serpanos, D.: ‘Safety and security of cyber-physical and internet of things systems [point of view]’. Proc. IEEE, 2017, 108, (6), pp. 983–984

[13] Lakshmanan, K., De Niz, D., Rajkumar, R., et al.: ‘Resource allocation in distributed mixed-criticality cyber-physical systems’. 2010 IEEE 30th Int. Conf. on Distributed Computing Systems, Genova, Italy, 21 June 2010, pp. 169–178.

[14] Narayana, S.: ‘Orchestrating mixed-criticality melody: reconciling energy with safety for mixed-criticality embedded real-time systems’, 2015

[15] Lehoczky, J., Sha, L., Ding, Y.: ‘The rate monotonic scheduling algorithm: exact characterization and average case behavior’. Proc. Real-Time Systems Symp., Santa Monica, CA, USA, 5 December 1989, pp. 166–171.

[16] Srinivasan, A., Holman, P., Anderson, J.H., et al.: ‘The case for fair multiprocessor scheduling’. Proc. Int. Parallel and Distributed Processing Symp., Nice, France, 2003

[17] Osmolovskiy, S., Fedorov, I., Vinogradov, V., et al.: ‘Mixed-criticality scheduling in real-time multiprocessor systems’. Proc. of the 18th Conf. Open Innovations Association FRUCT, FRUCT Oy, Saint-Petersburg, Russia, 2016

[18] Baker, T.P.: ‘An analysis of EDF schedulability on a multiprocessor’, IEEE Trans. Parallel Distrib. Syst., 2005, 16, (8), pp. 760–768

[19] Richter, J., Kothis, H., Bischl, B., et al.: ‘Faster model-based optimization through resource-aware scheduling strategies’. Int. Conf. on Learning and Intelligent Optimization, Springer, Chan, 2016

[20] Weg,J., Sarkar, A.M., Lee, Y.-K., Lee, S.: ‘A smoothed naive Bayes-based classifier for activity recognition’, IETE Tech. Rev., 2010, 27, (2), pp. 107–119

[21] Wickremasinghe, B., Buyya, R.: ‘Cloudlast: a CloudSim-based tool for modelling and analysis of large scale cloud computing environments’, MEDC Proj. Rep., 2009, 22, (6), pp. 433–459

[22] Rocklin, M.: ‘Dask: parallel computation with blocked algorithms and task scheduling’. Proc. of the 14th Python in Science Conf. No. 130–136, Austin, TX, USA, 2015

[23] Samal, P., Mishra, P.: ‘Analysis of variants in round robin algorithms for load balancing in cloud computing’. Int. J. Comput. Sci. Inf. Technol., 2013, 4, (3), pp. 416–419

[24] Xi, S., Xu, M., Lu, C., et al.: ‘Real-time multi-core virtual machine scheduling in xen’. 2014 Int. Conf. on Embedded Software (EMSOFT). IEEE, New York, NY, USA, 2014

[25] Gettings, O., Quinton, S., Davis, R.I.: ‘Mixed criticality systems with weakly-hard constraints’. Proc. of the 23rd Int. Conf. on Real Time and Networks Systems, Lille, France, 2015

[26] Pathan, R.M.: ‘Schedulability analysis of mixed-criticality systems on multiprocessors’. 2012 24th Euromicro Conf. on Real-Time Systems, Pisa, Italy, 2012

[27] Erickson, J.P., Kim, N., Anderson, J.H.: ‘Recovering from overload in multicore mixed-criticality systems’. 2015 IEEE Int. Parallel and Distributed Processing Symp., Hyderabad, India, 2015

[28] Nan, Y., Li, W., Bao, W., et al.: ‘Cost-effective processing for delay-sensitive applications in cloud of things systems’. 2016 IEEE 15th Int. Symp. on Network Computing And Applications (NCA), Cambridge, MA, USA, 2016

[29] Krishna, P.V., Misra, S., Nagaraju, D., et al.: ‘Learning automata based decision making algorithm for task offloading in mobile cloud’. 2016 Int. Conf. on Computer, Information and Telecommunication Systems (CITS), Kunming, China, 2016

[30] Ekberg, P., Yi, W.: ‘Outstanding paper award: bounding and shaping the demand of mixed-criticality sporadic tasks’. 2012 24th Euromicro Conf. on Real-Time Systems, Pisa, Italy, 2012

[31] Buttazzo, G.C., Lipari, G., Caccamo, M., et al.: ‘Elastic scheduling for flexible workload management’. IEEE Trans. Comput., 2002, 51, (3), pp. 289–302

[32] Jan, M., Zaourar, L., Petel, M.: ‘Maximizing the execution rate of low criticality tasks in mixed criticality system’. Proc. WMC, RTSS, Vancouver, Canada, 2013, pp. 43–48