sustainability

MDPI

Article

An Adaptive Emergency First Intelligent Scheduling Algorithm for Efficient Task Management and Scheduling in Hybrid of Hard Real-Time and Soft Real-Time Embedded IoT Systems

Sehrish Malik [1], Shabir Ahmad [1], Israr Ullah [1], Dong Hwan Park [2] and DoHyeun Kim [1,*]

[1] Computer Engineering Department, Jeju National University, Jeju-si 63243, Korea; serrym29@gmail.com (S.M.); shabir@jejunu.ac.kr (S.A.); israrullahkk@yahoo.com (I.U.)
[2] Electronics and Telecommunications Research Institute, Daejeon-si 34129, Korea; dhpark@etri.re.kr
* Correspondence: kimdh@jejunu.ac.kr; Tel.: +82-64-754-3658

Received: 6 March 2019; Accepted: 9 April 2019; Published: 12 April 2019

Abstract: Industrial revolution is advancing, and the augmented role of autonomous technology and embedded Internet of Things (IoT) systems is at its vanguard. In autonomous technology, real-time systems and real-time computing are of core importance. It is crucial for embedded IoT devices to respond in real-time; along with fulfilling all the constraints. Many combinations for existing approaches have been proposed with different trade-offs between the resources constraints and tasks dropping rate. Hence, it highlights the significance of a task scheduler which not only takes care of complex nature task input; but also maximizes the CPU throughput. A complex nature task input is when combinations of hard real-time tasks and soft real-time tasks, with different priorities and urgency measures, arrive at the scheduler. In this work, we propose a custom tailored adaptive and intelligent scheduling algorithm for the efficient execution and management of hard and soft real time tasks in embedded IoT systems. The proposed scheduling algorithm aims to distribute the CPU resources fairly to the possibly starving, in overloaded cases, soft real-time tasks while focusing on the execution of high priority hard real-time tasks as its primary objective. The proposal is achieved with the help of two intelligent measures; Urgency Measure (UM) and Failure Measure (FM). The proposed mechanism reduces the rate of tasks missed and the rate of tasks starved, by utilizing the free CPU units for maximum CPU utilization and quick response times. We have performed comparisons of our proposed scheme based on performance metrics as percentage of task instances missed, number of tasks with missed instances, and tasks starvation rate to evaluate the CPU utilization. We first compare our proposed approach with multiple traditional and combined scheduling approaches, and then we evaluate the effect of intelligent modules by comparing the intelligent FEF with non-intelligent FEF. We also evaluate the proposed algorithm in contrast to the most commonly-used hybrid scheduling scheme in embedded systems. The results show that the proposed algorithm out performs the other algorithms, by significantly reducing the task starvation rate and increasing the CPU utilization.

Keywords: Real-time tasks; task scheduling; embedded IoT systems; periodic tasks; event-driven tasks

1. Introduction

Lately, manufacturing has been observed to be alive with quite a lot of new fields e.g., the 4th industrial revolution, connected devices, industry 4.0, connected factories, smart factories, and Internet of Things (IoT) embedded devices. Nowadays we are standing on the verge of technological insurgency that is going to change our ways of living and working. This change, with the revolution in industry 4.0, will be dissimilar to anything human beings have experienced earlier in terms of its complexity and efficiency. The number of IoT applications is accelerating in many areas; mainly smart cities, smart factories, smart homes, healthcare surveillance, smart industrial control, and smart environment monitoring. With the drastic increase in smart applications and the generation of tons of data every second, decision making is of vital importance in smart machines and systems [1]. Recent advancements in autonomous technology and embedded systems make real-time systems and embedded IoT devices a prominent area of research and development. In real-time embedded devices, software and hardware systems are subjected to many constraints and these embedded devices need to respond within specified time constraints or deadlines. Hence, the embedded devices should be facilitated with smart decision-making capabilities at the task scheduler; so that scheduler can best decide the order of tasks to be run and the best use of available resources.

Real time systems are either hard real-time systems or soft-real time systems. If the tasks in the system have firm deadlines the system is considered a hard real-time system with the strict constraint of tasks executing before their deadline. If the tasks in the system have soft/flexible deadline then the system is referred to as a soft real-time system. Some examples of hard real-time system are avionics, nuclear power plants systems, and anti-lock braking systems for automobiles; while some examples for soft-real time system are multimedia streaming and automated windshield wipers. [2].

1.1. Task Scheduler and Scheduling Policy

A task scheduler, for a hard or soft real-time embedded device system, arranges algorithms according to a stated order of tasks execution. A scheduling algorithm determines the way tasks are to be processed by the scheduling system. For a real-time scheduling system, in generic conditions, a deadline, description, and an identifier are attached to each task. The elected scheduling algorithm decides how to assign priorities to a particular task. Preemptive and non-preemptive scheduling algorithms are commonly used algorithms in scheduling analysis. A real-time system might consist of event-driven tasks, periodic tasks or a combination of both. Periodic tasks have regular arrival times while event-driven tasks have irregular arrival times.

The objective of the real-time task scheduling is to determine an order of running these tasks on the processor. Scheduling is performed by a module called scheduler, which mainly aims to maximize the throughput and fairness while minimizing the mean waiting and response time of the tasks. When implementing a scheduler, it becomes very hard to ensure all these requirements at a single time. If the real-time system is a hard real-time system, then the task's deadline becomes one of the most important factors, as a task must complete before its deadline. Other priorities might include a task's period, arrival time, slack, or any user-defined priority. Hence, in the real-time systems, the task scheduling mainly focuses on scheduling the tasks based on their priorities. To achieve this purpose, a scheduling policy is designed for any given scenario [3,4]. It is not always possible to meet the required deadline; hence further verification of the scheduling algorithm must be conducted or a scheduling policy must be defined. A scheduling policy is an approach which delves into the arrived set of tasks at the processor and selects the best candidate for execution; depending on different priority factors. The goal of scheduling the algorithms can diverge from situation to situation; there are many scheduling algorithms proposed in the literature, each having its own set of goals. These goals include fairness, efficiency, response time, and throughput. Fairness among tasks can be designated as giving the equal share of CPU's unit time to all the current or existing tasks queued to be scheduled. Allocating CPU resources to the tasks, with respect to total workload and priorities associated with the tasks at the current time, are also some of the factors considered under fairness of the task scheduler.

In priority-based scheduling policies, an important concept is to avoid task starvation. Starvation can be defined as preventing a task from completing its execution for a longer time because the CPU resources are allocated to some other task. A best scheduler is one, which also competes to save the lower priority tasks from the starvation along with taking care of all the priorities. Hence, we can conclude that, the primary objective of a scheduling algorithm is to decide the order in which tasks are to be executed at the processor so that minimum number of tasks is deprived of the CPU resources while maximum of the defined goal is achieved. A scheduler may aim at one or more of many goals, depending on the scenarios and scheduling priorities.

1.2. Challenges in Embedded IoT Systems

The term RT-IoT is getting recognition in embedded IoT systems [2,5], as it refers to real-time systems in IoT. RT-IoT devices often have limited resources and require control tasks to be executed within milliseconds. A RT-IoT system can be soft real-time system, hard real-time system or a combination of both. In scenarios where an RT-IoT embedded system has input tasks with deadlines of both natures, as hard deadline tasks and soft deadline tasks, then the system can be termed as a hybrid of hard and soft real-time IoT embedded systems.

When the sensing interval is too short and scheduler gets bombarded with periodic sensing data tasks or a series of events occur back to back and the scheduler gets bombarded with loads of event driven tasks. In such cases, in embedded IoT systems the tasks with soft deadlines are often starved due to heavy load at the scheduler end. Since IoT systems have limitations in terms of processing, memory and storage. As IoT systems are constrained with limited resources, hence allocation of separate scheduling servers is not always an affordable option.

Therefore, a scheduler, that allocates its resources in a best possible manner in emergency cases and puts best effort in avoidance of unnecessary starvations of tasks with soft deadline, is very crucial.

1.3. Solution Approach

We aim to design a scheduling algorithm for such hybrid RT-IoT system, where our proposed scheduling mechanism aims to meet the deadline for hard real-time tasks efficiently while making its best effort to minimize any delay/loss in execution of soft-real time tasks in heavy load scenarios.

In this paper work, we present a scheduling algorithm which proposes a novel solution for the real-time task input in embedded IoT systems, giving flexible parameter setting options in UM (Urgency Measure) and FM (Failure Measure), in order to consider all the system constraints for embedded IoT systems of different nature, and minimizing the starvation rate of the tasks. The proposed scheduling algorithm is adaptive and flexible; it makes best effort to meet the real-time deadlines, gives priority to the high-priority tasks, while also focusing to avoid starvation of the low priority tasks. Our proposed algorithm makes use of basic ANNs for optimal prediction of next step of processor in task scheduling and for using available resources wisely.

The rest of the paper is structured as follows. In Section 2, we present the literature review. In Section 3 we present the proposed methodology for the hybrid real-time task scheduler for embedded IoT systems. Section 4 we present the simulation and implementation of our scheduling visualization toolkit. Section 5 performs the comparisons analysis and Section 6 presents the discussions section.

2. Related Work

2.1. Traditional Scheduling Approaches for Real-Time and Non-Real Time Systems

There are many basic traditional scheduling algorithms with each offering some priority-based scheduling. The most simple and fair policy for non-complicated scenarios is First Come First Serve (FCFS) or First In First Out (FIFO). It queues the processes in the order they arrive at the processor and are executed accordingly. Many human services follow this approach, e.g. on cashier counters in department stores or shops, serving of food at the restaurants, or taking orders [6]. Shortest Job First/Next (SJF or SJN) is another simple approach, its main idea is to queue the tasks with one with the least estimated execution time first and so on. To implement this strategy, we should have the knowledge of execution time of each job in advance [7]. Highest priority first policy is a way to first assign priorities to the jobs and then execute accordingly. The one with the highest priority will be executed first. The scheduler will arrange the processes in the ready queue in order of their priority and also the lower-priority processes get interrupted by new arrived higher-priority processes. Least Laxity First (LFF) is a dynamic priority assignment scheme. It executes the jobs by calculating the remaining times and relative deadline of each job. It calculates the laxity for each job, which can be defined as the difference between deadline and remaining execution time. The Modified Least-Laxity-First (MLLF) scheduling algorithm proposes to reduce the number of context switches in the LLF, aiming to achieve high system performance. The one with minimum remaining difference at a given unit time will be executed [8]. Round Robin (RR) is a scheduling policy in which each job is executed for a small amount of time and then the CPU is given to the next job and this continues in a cycle. The small amount of time to run this cycle is pre-defined [9].

Usually the inputs to real-time system are of either periodic nature or event-driven nature. The most common approach for scheduling periodic tasks is Rate-Monotonic (RM) where tasks are scheduled with respect to their periods while the most common approach adopted for the event-driven tasks is Earliest Deadline First (EDF) where a task's deadline is taken as priority; the nearer a task's deadline is, the earlier it should be executed. For the implementation of rate-monotonic, each job must have a period value assigned to it as rate monotonic is a scheduling algorithm for periodic tasks. Deadline monotonic (DM) is an alteration of rate-monotonic, in rate-monotonic the deadline value is equal to the period value but in deadline monotonic the deadline can be smaller than the period and deadline monotonic follows the fixed priority depending on the deadline of jobs [10–13].

2.2. Customized Scheduling Approaches for Real Time Systems

Maximum Urgency First (MUF) executes the tasks based on their criticality, giving high priority to the tasks with maximum urgency. Urgency is defined by criticality, user priority and laxity of a task. All the tasks in the critical set are given high critical level and executed first and tasks out of critical set are given low critical level and executed later [14]. The behavior of classical RM and EDF is equated in [15], as these two have the most renowned scheduling algorithms for real-time applications. Comparison results have shown that EDF provides better CPU utilization and quick responsiveness as compared to RM and hence a better option for embedded systems.

Time-stepped load balancing (TLS) is a real-time scheduling algorithm that generates a load balancing schedule table, based on execution periods, for input set of time stepped simulations. With each change in the input model set, the table is changed dynamically. TLS saves around 4% processor resources over EDF with four times less jitter [16]. Smoothed Least Laxity First (sLLF) is an extension of LLF which finds a threshold for a task to be completed by minimizing its laxity and maximizing the feasibility edge. It has been proposed for deciding the charging rates for EVs (Electric Vehicles) [17].

The problem of scheduling stabilized control tasks on embedded devices is being revisited in [18]. A simple event triggered scheduler is being explored which is based on the feedback criterion, it presents a guaranteed execution of tasks and better performance keeping the periodic execution requirements at the same time. In embedded devices, leakage energy consumption is observed to be an accelerating concern. In this study [19], a special tasks scheduling technique, known as procrastination scheduling, is being introduced. As name suggests here, tasks' execution is being delayed in order to expand the duration of idle intervals. It also minimizes the leakage energy drain. The objective of minimization of energy consumption for both static and dynamic; has been achieved by focusing on the dynamic slack reclamation methods under procrastinating scheduling. The idle intervals between tasks are elongated through dynamic procrastination.

A scheduling policy for reliable execution in autonomic systems is proposed in [20]; the proposed technique uses different decision models and provides a demonstration of Markov decision model's application on a multi-threaded system model.

The work presented in [21] surveys data traffic scheduling techniques and provides a comparison among priority queuing (PQ), first-in-first-out (FIFO) and weighted fair queuing (WFQ). The aim of the work is to find most suitable data traffic scheduling scheme which ensures QoS (Quality of Service) over 5G mobile networks. The work in [22] presents a data traffic slicing model, and provides comparison among various packet traffic scheduling techniques such as PQ, FIFO, and WFQ in the 5G mobile based on the data traffic slicing model.

An autonomic fault tolerant scheduling approach for scientific workflows in cloud computing is presented in [23]. The proposed solution provides a hybrid heuristic for scheduling problem and a fault tolerant technique using virtual machine migration approach for tackling the task failure problem.

Many other scheduling policies are proposed in the previous works by altering the existing ones and adding new constraints to enhance performance [24–29]. A best scheduling policy can be one, which maximizes the benefits of all the existing scheduling approaches while minimizing the drawbacks. In the next section we propose our designed general-purpose scheduling algorithm, which aims to combine the best of all in order to maximize the CPU utilization.

3. Methodology for Intelligent Scheduling Algorithm

In this section, we present our proposed methodology for the proposed real-time task scheduling in hybrid of hard real-time and soft real-time embedded IoT systems. We name our proposed scheduler as FEF (Fair Emergency First) scheduler.

The proposed scheduling algorithm considers scenarios where some of the input tasks are of hard deadline and some are of soft deadline. It aims to schedule IoT systems in scenarios where the system is loaded with tasks of different nature of deadlines and different priorities.

The proposed scheduler is designed for different scenarios in IoT systems. In IoT environments, mostly tasks data is of two types as periodic and event driven data. The periodic tasks in IoT environment get the sensor readings, after set interval time, from the sensors installed in the smart place and pass the readings onto the system for further processing. While the event driven tasks are triggered in response to some set behavior. In our supposed scenarios, the event driven tasks are of highest priority as they are one-time tasks and triggered to indicate some set abnormal behavior by the system. In table below, we provide examples of event driven tasks from smart homes, smart vehicles and smart health management scenarios (see Table 1).

Table 1. Example scenarios of event-driven tasks in Internet of Things (IoT) environments.

IoT Environment	Emergency Nature Event Driven Tasks
Smart Homes	Smoke Alert, Fire Alert, Security threat Alert
Smart Vehicles	Hurdle detected, Route change alert
E-Health	Fall detection, abnormal BP, abnormal heart rate

We have two input type tasks as event-driven tasks and periodic tasks sub-categorized into four types as shown in Table 2 below along with their deadline types. Since in smart IoT environments, some of the event-driven tasks are of emergency nature as mentioned in Table 1. These emergency nature tasks are tagged as urgent event-driven tasks with the hard deadline. Next are the non-emergency nature event driven tasks, which can either have soft deadline or hard deadline. Priority periodic tasks are the tasks which arrive at the scheduler at regular interval and hold importance as they might carry some important flow of the system. The priority periodic tasks have hard deadlines. Normal periodic task is a task which has less priority as compared to the priority periodic tasks and has a soft deadline most of the times but can also have a hard deadline in some scenarios. Dividing the tasks into different types, though two sub-types might have same deadline type; helps in understanding the nature and context of incoming task in the smart environments. Also, context-based division of task type brings ease of understanding task dependencies based on contextual scenario learning.

Table 2. Input tasks type and deadline Categorization for IoT environments.

Task Type	Deadline Type
Urgent Event driven task	Hard deadline
Normal Event driven task	Soft/Hard Deadline
Priority Periodic Tasks	Hard Deadline
Normal Periodic Tasks	Soft/Hard Deadline

When the sensing interval is too short and scheduler gets bombarded with periodic sensing data tasks or a series of events occur back to back and the scheduler gets bombarded with loads of event driven tasks. For such cases, in IoT systems the soft deadlines are often missed, and with our proposed scheduler, we aim to best allocate the CPU resources to the available tasks while trying to minimize high loss of soft deadlines, or starvation of any low priority task with soft deadline.

3.1. Fair Emergency First Task Scheduler

The proposed FEF scheduling algorithm, strives to run the high priority and urgent tasks first while utilizing the free CPU chunks for other low priority tasks hence consuming the CPU time as efficiently as possible in the given input scenarios.

We have four types of tasks defined in our system as normal event-driven tasks, urgent event-driven tasks, normal periodic tasks and priority periodic tasks. Event-driven tasks are either normal event-driven tasks or with urgency tags. The urgent event-driven tasks (ETs) have a default priority, as they are triggered in high emergency cases. Normal event driven tasks are ones which should be executed any time before deadline, while urgent event driven tasks are the tasks which should be executed as soon as they arrive. Periodic tasks have a priority with respect to their period. The priority periodic tasks must execute before their deadline as they have hard deadlines. Also, any task with additional urgency or a priority task is considered with a hard deadline, and must execute on time. In ideal scenarios the scheduler must meet all deadlines, while in overloaded scenarios the tasks with soft deadlines might starve or miss. The proposed algorithm, FEF also makes sure to keep the priorities and urgency measures as flexible for the user as possible; as the periodic tasks can also be prioritized over event-driven tasks, if required. Our algorithm focuses on execution of the task by set priorities, obedience of deadlines, lowering the starvation rate for low priority tasks with soft deadlines, and maximizing the CPU utilization (Figure 1).

```
 Extract all arrived tasks in the queues at time Ti.
   IF urgent event-driven tasks exist
       Run the task with nearest deadline
   ELSIF (non-urgent event-driven task exists AND there's no periodic task) THEN
       Run the non-urgent event-driven task with nearest deadline
       IF (non-urgent event-driven tasks exist AND periodic tasks exist) THEN
           IF (priority periodic task exists) THEN run the priority periodic task
           Calculate UM
           IF (UM = 0) THEN run the periodic task
           ELSE run the event-driven task
           ENDIF
       ELSIF (only periodic tasks are there to run) THEN
         Calculate FM
           IF (FM = 0) THEN
               Run starving task/about to miss task
           ELSE
               Run the high priority periodic task
           ENDIF
       ENDIF
   ENDIF
Update History_Log
```

Figure 1. Pseudo-code for Fair Emergency First (FEF) algorithm.

In the Figure 2 below, the running state of the tasks at the scheduler is shown. During the task scheduling our system will have to choose which task to run depending on many existing factors at the given time. The system can have four types of tasks and run them based on their priority and availability order. The priority order followed is same as explained in FEF algorithm above.

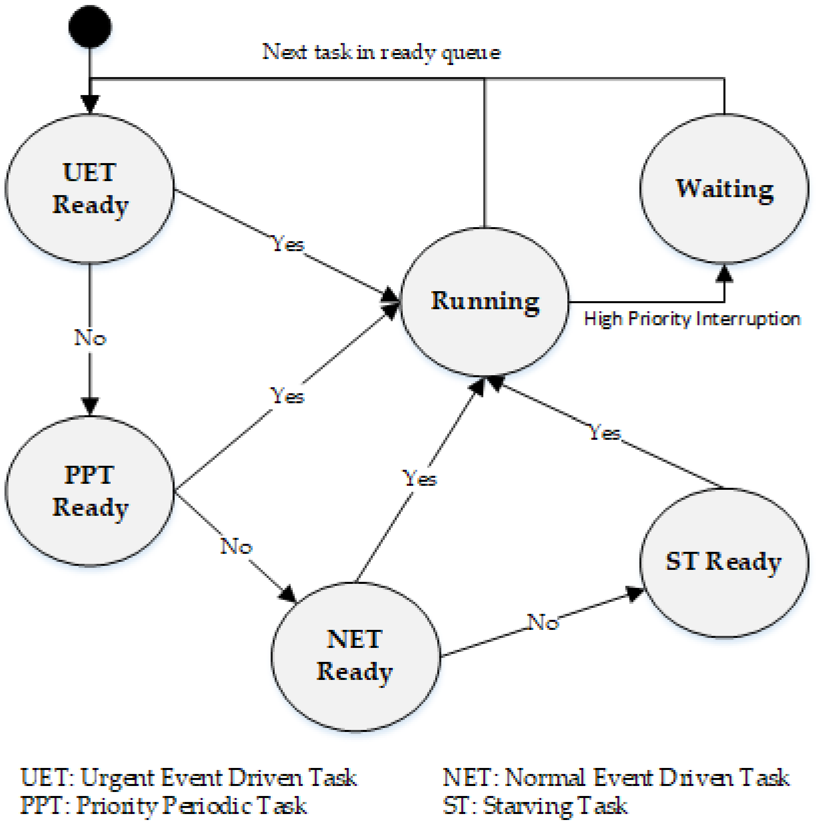

Next task in ready queue
UET Ready
Waiting
Running
High Priority Interruption
Yes
No
PPT Ready
NET Ready
ST Ready
UET: Urgent Event Driven Task
PPT: Priority Periodic Task
NET: Normal Event Driven Task
ST: Starving Task

Figure 2. State diagram for task running states of Fair Emergency First (FEF) algorithm.

We have two ANN learning based components, Failure Measure (FM) and Urgency Measure (UM), which is calculated during the running of online scheduling to help in making an informed and learned decision by the scheduler. Figure 1 shows the flow for the proposed task scheduler FEF algorithm.

3.2. Urgency Measure (UM)

UM is calculated between the normal event-driven tasks and periodic tasks, trying to decide whether to execute normal event-driven task first or the periodic task first.

Urgency means priority of a task to be executed at first, assuming that the task is of such urgent nature, that it should be executed as soon as it arrives. Such tasks are to be tagged with the mark of urgency so that scheduler can distinguish them from other tasks. The first thing to check at scheduler is, whether the event-driven task arrived is with the urgency tag or not. If an urgent event-driven task is arrived then scheduler runs it right away. If no urgent event-driven task is arrived then slack for the available periodic tasks and normal event-driven tasks is calculated. Slack is the difference between required execution time and deadline of a task. Equation 1 below is used to evaluate whether normal event driven task should run first or a starving periodic task.

$$Slack\ (ET) >= x \times (Slack(PT)) \tag{1}$$

Where, x is set as 2 initially and learned gradually using ANNs with passage of time as the tasks flow in the system and data history gets created. The aim is to predict a value for x which results in a safe and beneficial distance between the slack of non-urgent event driven tasks and periodic tasks.

In the initial phase of the deployment, the system is recommended to use the value of x as 2, which is set as a safe start after testing different starting values of x. Once the system starts receiving data tasks, and executes them on the scheduler, the system in parallel builds its history containing time stamp, number of tasks at current time stamp along with type and priorities, slack associated with each task at current time stamp, and the missing rate of tasks at the current time stamp (Figure 3). The purpose of building history log is to enable system to learn from its previous decisions. It is an effort to make the scheduler intelligent. So, initially the system would start like any regular scheduler but with the growth of system, the system will learn from history log based on the value x in Equation (1).

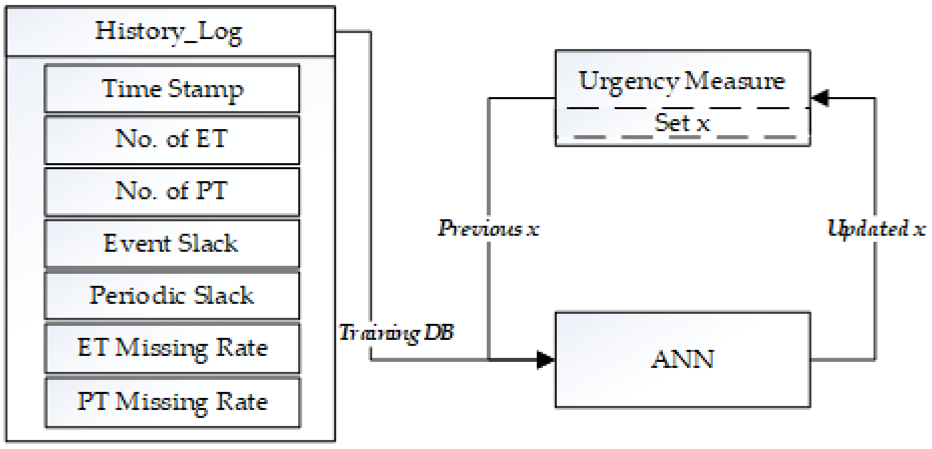


Figure 3. Learning of X in Urgency Measure (UM). ET: event-driven task; PT: Preemption Threshold.

If Equation (1) is true then UM is set as 0, indicating no urgency of executing normal event-driven tasks; else it is set as 1, indicating that normal event-driven task should be executed first else it might miss the deadline in the longer run. When the value of UM is set to 0, gives a fair chance for some low priority tasks to run in between, as those might miss the deadline in the longer run due to task starvation.

3.3. Failure Measure (FM)

FM is calculated between the available periodic tasks; it helps to decide whether to run the high priority periodic tasks first or if it is safe to run some starving low priority task. Failure measure is calculated to make sure that if at a time "t", scheduler runs a low priority starving task first, then it will not cause any high priority periodic task miss its deadline eventually. For calculation of failure measure, the scheduler will use the knowledge of all the arrived tasks at the processor along with history logs, and does not require any information about future tasks.

High Priority Task Execution Prediction is a function which predicts, the safe execution possibility, for all the high priority tasks in case of running the starving task first. The function uses the history log in order to make predictions using ANNs (Figure 4).

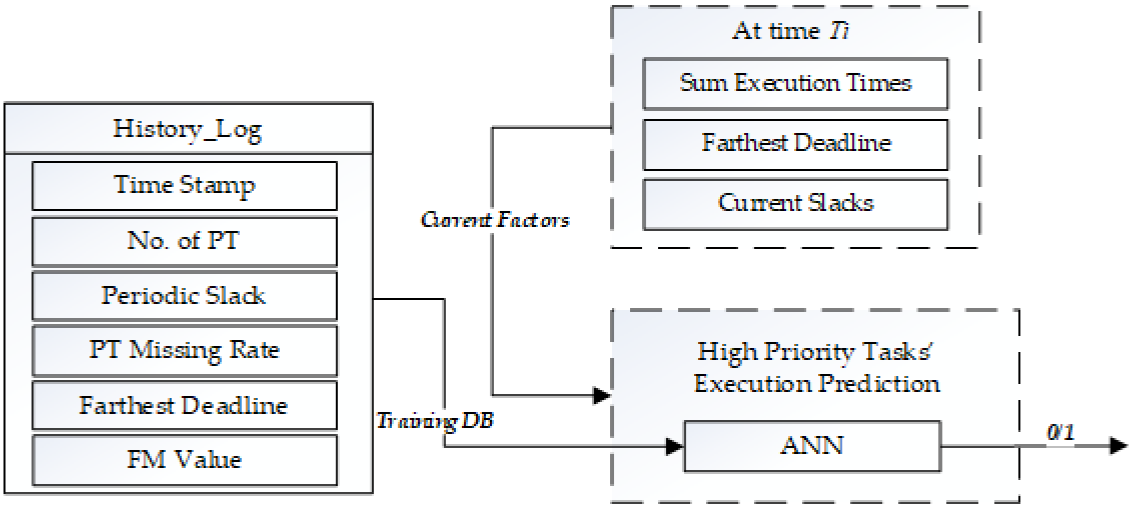


Figure 4. Prediction for high priority tasks' safe execution.

It calculates each task's slack, checks each task's execution time status (execution time completed, execution time left), and removes overlapping times to predict safe execution of each task in the future based on the current information and history log. When considering a low priority task to run, this function learns from history log whether running the low priority task in given constraints might fail some high priority task in the future or not. If the function predicts that it is safe to run the low priority task based on available information at current time ***t***, then the function returns 1 else it returns 0. If the function returns 1 then the value for FM is set to 0 and low priority starving tasks are executed as a result, else the value for FM is set to 1 and high priority periodic task is executed as a result. Hence FM value set as 0 indicates that CPU has enough resources to be allocated to the low priority starving tasks at the given moment (Figure 5).

```
Extract all arrived tasks in the queues at time Ti
Sum up the execution time for all the available tasks at the current time.
Calculate the difference between current time and farthest deadline in all available tasks.

   IF ((HighPriorityTaskExecutionPrediction () == 1))
       FM == 0    // Chances of failure are near to 0Learning results positive
   ELSE
       FM == 1    // Chances of failure are near to 1Learning results negative
   ENDIF

   IF (FM == 0)
       Execute low priority (starving) task
                            // Chances of failure are near to 0Learning results positive
   ELSE
       Execute high priority periodic task
                            // Chances of failure are near to 1Learning results negative
   ENDIF
```

Figure 5. Pseudo-code for Failure Measure (FM).

The output predicted by the function based on current information can also fail in case of the arrival of some unexpected urgent tasks in the future, since, the complete knowledge of the future tasks is not available in the real-time scheduling except for the ones already at the scheduler, periodic tasks which repeat themselves after certain period of time and history log. For some cases it is wise to keep some CPU chunks free; if system can't afford any such loss at all. For such scenarios we have defined a variable named "Reserved".

Reserved is an integer value aiming to reserve a number of CPU units for an unexpected urgent task arrival. The value is learned as the tasks in the system grow.

3.4. Preemption Bit (PB) and Preemption Threshold (PT) (Worst Case Scenario)

Now at last, we consider the worst-case scenario where majority of the tasks arriving are high priority tasks, then according to the priority policy, a low priority task cannot preempt an urgent event-driven tasks. So we introduce a PB (Preemption Bit) for the cases where system is overloaded with the urgent event-driven tasks, and assuming that it is mandatory to run a periodic task after certain amount of time even if it means interrupting the urgent event-driven task. The PB is added to the periodic tasks, and a PT (Preemption Threshold) value is set to limit maximum starvation time in such cases. If PB of a periodic task is 1, then once the PT is reached, the periodic task can preempt the running task and execute itself. The value for PT can be varied depending on the load of the CPU and nature of the periodic tasks.

4. Simulation of Proposed Scheduling Algorithm Based on Embedded Environments

In this section we present the implementation of our task scheduler and visualization tool. In Section 4.1, we discuss the implementation setup. Section 4.2 describes the input task model. In the Section 4.3, we present the overview of our implemented task simulation and visualization tool.

4.1. Implementation Setup

We have used python for implementing the core programming logic of the proposed task scheduling algorithm. Python is a very popular general-purpose programming language; widely used for developing desktop based and web-based applications. We have designed a web-based task simulation visualization tool using flask, which is an MVC-based framework (Section 4.3). In the experimentation phase, we have tested our built task scheduler on both, a general PC system and an embedded IoT system (Tables 3 and 4).

Table 3. Implementation environment for Windows.

System Component	Value
Operating System	Windows
CPU	Intel ®Core ™ i5-4570 CPU at 3.20 GHz
Primary Memory	8 GB
Programming Language	Python 3

Table 4. Implementation environment for raspberry PI-embedded system.

System Component	Value
Hardware	Raspberry Pi 3 Model B
Operating System	Raspbian
Memory	1 GB
Server	Flask Webserver
Resources	LED, Fan, Temperature Sensor, Humidity Sensor, Motion Sensor, Breadboard, Expansion Board, Connecting wires, e-Health Sensor Platform
Libraries	GPIO, CSVReader, Jinja Template, Bootstrap 3, HTML 5/CSS3
IDE	Vim, PyCharm (Remote Access)
Programming Language	Python 3

4.2. Input Tasks Notation

One of the most important steps is to design and implement the input model. We plan to perform our performance analysis based on two phases. In the first phase, we custom design the input task model, which is a combination of periodic and event-driven tasks. While for the second phase, we generate the input task model randomly consisting on both periodic and event-driven tasks. Basic task parameters in the input task model are arrival time, execution time, deadline, and period (for periodic tasks). Arrival time is the time when the task arrives at the processor. Execution time is the time, which a task needs from CPU to complete itself. Deadline is the time limit within which a task should complete its execution. Period is a time cycle after which a task will arrive again. Table 5 shows the criteria for the implementation of input tasks.

Table 5. Basic task parameter's selection criteria.

Task Parameter	Criteria
Period	Should not be prime number ideally except a feasible one like 5, so that hyper-period is not too long. Can pre-define a list and assign one by one or randomly e.g., {4, 5, 6, 10, 12, 15}
Deadline	D=P or D=P/2
Execution Time	Rand (0, min (D, P)). Selecting a number between 0 and deadline of task. Execution time is greater than 0 and less than deadline.
Start time	0

4.2.1. Periodic Tasks Set Notation

In the case of periodic tasks, the instances of a periodic task regularly arrive after a set period. Periodic tasks are real-time tasks with a constraint of having the period greater than zero, which means that after a certain amount of time the tasks instance must repeat. Usually, periodic tasks have two states; inactive and runnable. Inactive is the state when the task has not yet arrived at the processor and runnable is the state when the task has arrived again after a certain period and is waiting to run.

A periodic task with its *i*th periodic execution is denoted as following.

$$T_P\ (i) = [ID_i,\ AT_i,\ ET_i,\ D_i,\ P_i,\ PB] \tag{2}$$

Where, T_P denotes the periodic task instance, *ID* is the identifier of a periodic task, *AT* is the arrival time of a periodic task, *ET* is the execution time of a periodic task, *D* is the deadline of a periodic task, *P* is the period of a periodic task, *PB* is the preemption bit associated to a periodic task, $PB = 1$ indicates periodic task can preempt the high priority tasks in case of starvation for a set PT (Preemption Threshold), and $PB = 0$ indicates periodic task cannot preempt any high priority task.

4.2.2. Event-Driven Tasks Set Notation

An event-driven task is programmed to activate when an event occurs, it can handle any input at any moment. Event-driven tasks have two sub-categories of urgent event-driven tasks and flexible event-driven tasks, these sub-categories help in making the system more flexible. In case of urgent event-driven tasks, tasks should be executed as soon as they arrive at the processor, they cannot wait in the queue. On the other hand, flexible event-driven tasks can afford to wait in the queue but they must also be executed before the deadline. Event-driven tasks have three basic states; inactive, runnable and suspended. Inactive state is when the event to generate the task has not occurred yet, runnable state is when the event is generated and the task is waiting to run, and suspended state is when the event source is triggered off.

An event-driven task with its *i*th execution is denoted as following.

$$T_e\ (i) = [ID_i,\ e,\ AT_i,\ ET_i,\ D_i,\ UB_i] \tag{3}$$

Where, T_e denotes the event driven task instance, *ID* is the identifier of an event-driven task, *e* is the event that triggers an event-driven task, *AT* is the arrival time of an event-driven task, *ET* is the execution time of an event-driven task, *D* is the deadline of an event-driven task, *UB* is the urgency bit of an event-driven task, $UB = 1$ indicates task is urgent and should be executed ASAP and $UB = 0$ indicates non-urgent event-driven tasks.

4.3. Tasks Set

We have gathered tasks sensing data from temperature sensor, humidity sensor, motion sensor, pressure sensor, and e-Health sensor platform for values of ECG and Pulse. The e-Health sensor platform allows user to monitor and record real-time health data for ten different sensors installed on it [30]. The control tasks are generated for two actuators in the system as fan and LED. In the Table 6 below, we give the tasks along with their priority tag and description.

Table 6. Tasks set.

Task Name	Task Tag	Description
Get Temperature	Normal Periodic Task	Reads the temperature value from temperature sensor.
GET Humidity	Normal Periodic Task	Reads the humidity value from humidity sensor.
Motion Status	Priority Periodic Task	Gets the motion status of set area in smart space.
Pressure Status	Priority Periodic Task	Gets the pressure value from sensor.
Get ECG	Priority Periodic Task	Gets the ECG value from e-Health platform.
Get Pulse	Priority Periodic Task	Gets the Pulse value from e-Health platform.
Control Fan	Normal Event Driven Task	Control command task for fan (E.g. turn on/ turn off)
Control LED	Urgent Event Driven Task	Control command task for LED (E.g. turn on turn off/ Blink for set time period)

As shown in the Figure 6, the tasks are divided into three main categories as sensing tasks, actuator tasks and system tasks. In sensing tasks, we have tasks for getting the sensing values from sensors which are periodic tasks. In sensing tasks, we have two normal periodic tasks as GetTemp, GetHumid and four priority periodic tasks as MotionStatus, PressureStatus, GetECG, and GetPulse. The two control tasks are LED control and fan control, which are generated in result to different sensing thresholds. Fan control is normal event driven task for the given two cases and LED control is urgent event driven task for the given four cases. The only system task is notification message, which is normal event driven task when generated for normal event driven control tasks and urgent event driven task when generated for urgent event driven control tasks.

4.4. Real-Time Tasks SchedulingSimulation and Visualization Toolkit

In order to simulate the scheduling and visualize the output, we have reused our previously built task scheduling simulation and visualization toolkit [5]. It is an IoT task simulator, proved to be one of best among the existing state-of-art scheduling and visualization tools. Our proposed scheduling mechanism's logic is implemented at the back end while using the referred visualization tool at front end user interface. Figure 7 below shows the CPU timeline visualization output screen for the proposed scheduling algorithm.

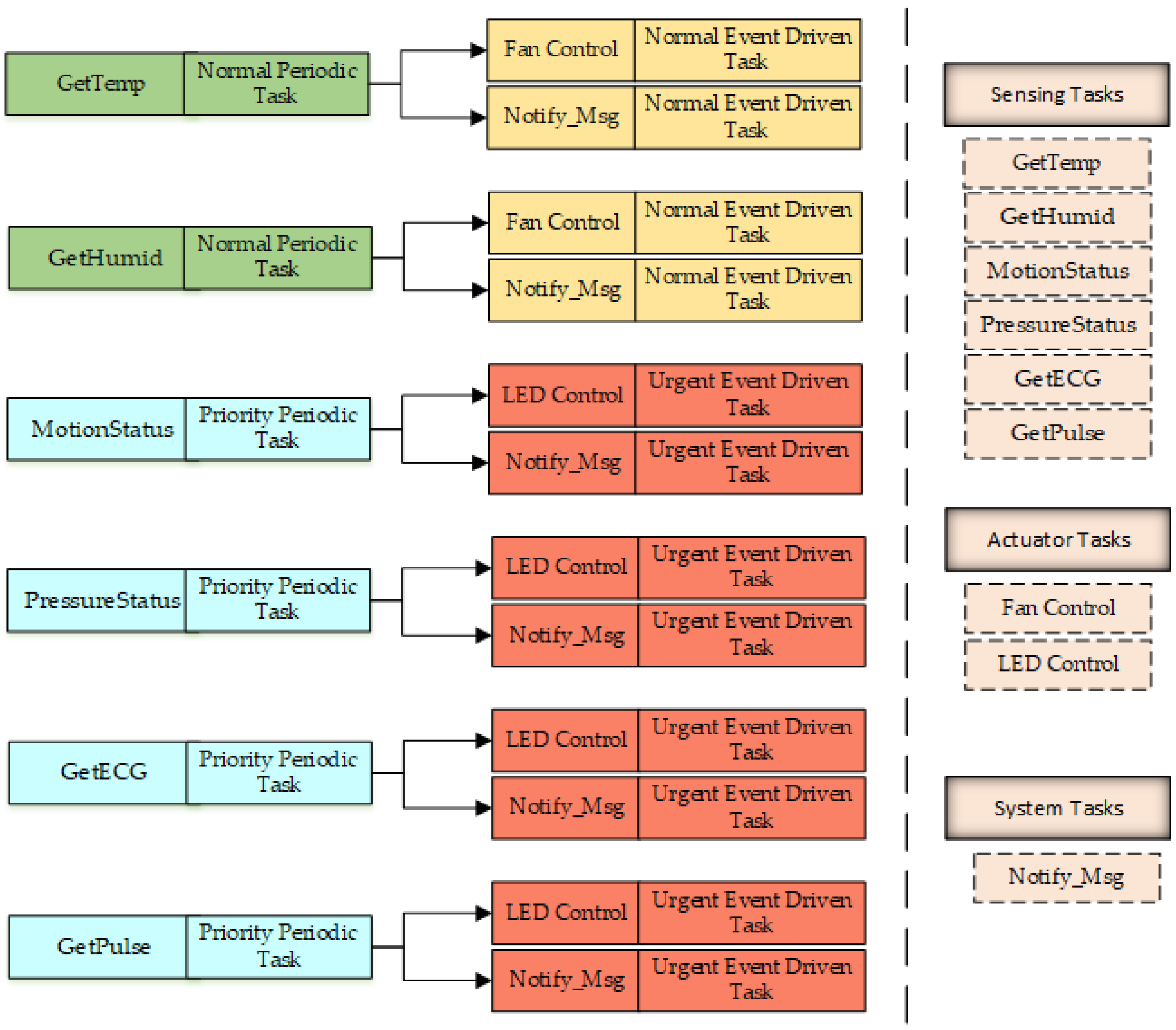
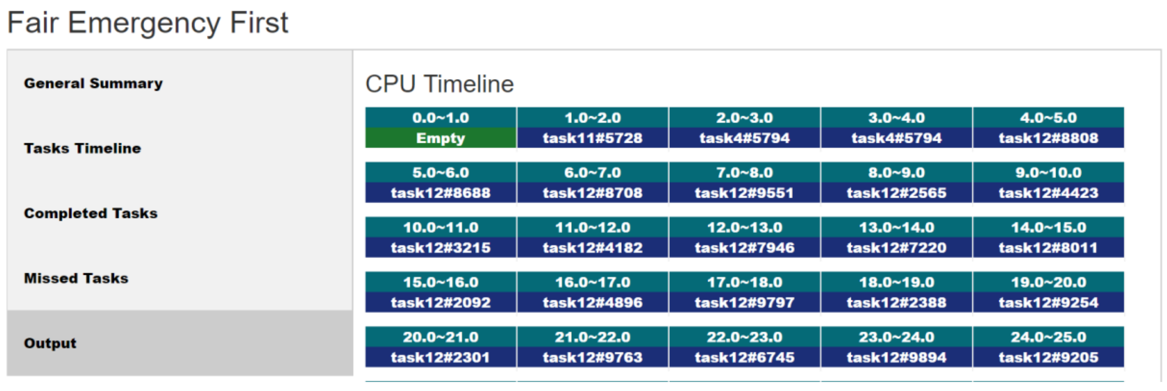


Figure 6. Tasks flow along with their category and priority tags.


Fair Emergency First

General Summary

Tasks Timeline

Completed Tasks

Missed Tasks

Output

CPU Timeline

0.0~1.0 Empty	1.0~2.0 task11#5728	2.0~3.0 task4#5794	3.0~4.0 task4#5794	4.0~5.0 task12#8808
5.0~6.0 task12#8688	6.0~7.0 task12#8708	7.0~8.0 task12#9551	8.0~9.0 task12#2565	9.0~10.0 task12#4423
10.0~11.0 task12#3215	11.0~12.0 task12#4182	12.0~13.0 task12#7946	13.0~14.0 task12#7220	14.0~15.0 task12#8011
15.0~16.0 task12#2092	16.0~17.0 task12#4896	17.0~18.0 task12#9797	18.0~19.0 task12#2388	19.0~20.0 task12#9254
20.0~21.0 task12#2301	21.0~22.0 task12#9763	22.0~23.0 task12#6745	23.0~24.0 task12#9894	24.0~25.0 task12#9205

Figure 7. Web-powered visualization toolkit—CPU timeline.

5. Results Analysis

In this section we present results analysis of our proposed task scheduling algorithm in detail. Section 5.1 is evaluated both on general purpose PC system and embedded IoT system while Sections 5.2 and 5.3 is performed only for embedded IoT system.

Our tested scenario for the smart environment task scheduling is combination of smart home scenario and e-health scenario for the smart home residents (Figure 6). The data is taken from sensors after set intervals, based on which the tasks are generated. The considered intervals are 10 seconds, 5 seconds, 1 second, 500 milliseconds, and 100 milliseconds. For the extensive testing, we have tested the scheduler system with 100 and 500 milliseconds interval of tasks load along with event generation of 2 seconds and 3 seconds for multiple scenarios. Our proposed algorithm FEF has two main variable parameters as Reserved and PT, to be set depending on the system conditions or requirements. In Section 5.1, we set *Reserved* = 1 for our simulations; as we wanted to use maximum slack time in order to increase CPU utilization. While in Section 5.2, we set *Reserved* = 3; as here our focus is to compare constrained embedded IoT system scenarios and we aim to keep our preemption rate lower. For the same reasons, we have kept PT = 10 for the Section 5.1 and PT = 20 for the Sections 5.2 and 5.3.

All the tasks scheduling simulations are performed, in Sections 5.1–5.3 are, under overloaded scenarios of tasks load; which mean the scheduler is bombarded with heavy loads of tasks arriving at the scheduler. The aim is to evaluate the best allocation of the resources by the scheduler, in order to execute the hard-deadline tasks within deadline and in parallel also accommodate a larger number of soft deadline tasks in heavy load scenarios.

5.1. Comparison Analysis with Traditional Algorithms

In this section, we compare our proposed FEF scheduling algorithm with some of the traditional and combined scheduling algorithms such as EDF, RR, LLF, DM, Quantum based (Sharing time based on wait time, sharing time based on CPU time) scheduling, EDF based on zero laxity, POSIX 1003 Highest Priority, and MUF.

In Figure 8, the output comparisons of missed instances for the performance analysis on the proposed scheduling algorithm are shown. The task set is run on ten different algorithms, and our proposed FEF scheduling algorithm. The output graph in the figure shows the percentage of task instances missed by each algorithm for the given input. Event-driven tasks arrive once with no repeating instances; while in the case of periodic tasks, instances are repeated after certain period and hence percentage of total instances missed, during the simulation time, is taken out for the comparisons. FEF shows the best results, in comparison to other algorithms, with maximum CPU utilization as it misses minimum number of task instances in the given scenario. Some of the basic scheduling policies seem to have very high missing rate, as they only focus on one or two parameters, instead of taking into account all the scheduling parameters and developing scenario. For example, time sharing and CPU sharing, strictly allocate the resources on turns without paying attention to deadlines and priorities. Hence, in an overloaded scenario where maximum tasks' instances missing rate could be about 68%, as seen for some scheduling approaches in the figure below, the proposed mechanism managed to reduce it to 21% with an intelligent handling of CPU resources. Since the scenario is of tasks' overload at the scheduler; the loss of instances to some extent is inevitable as load is greater than total available capacity. A better management of tasks at the scheduler can accommodate the load in a best possible manner; and save many tasks from missing deadlines.

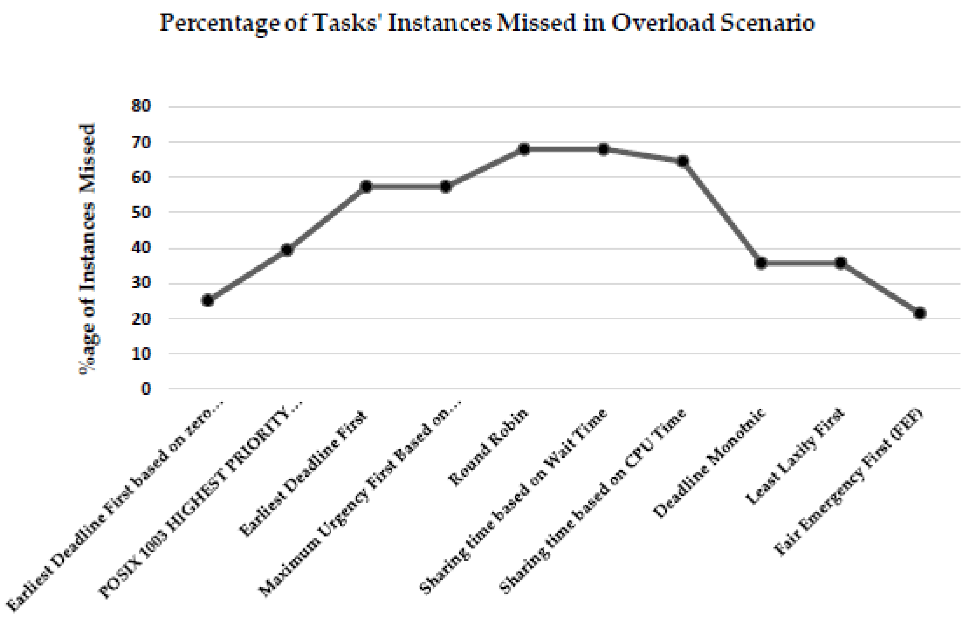


Figure 8. Percentage of total tasks' instances missed.

In Figure 9, the comparisons of tasks' starvation rates between the proposed FEF scheduler and other scheduling algorithms are depicted. The starvation rate, shown in the figure above for multiple algorithms counts the total weightage of the urgent and non-urgent event-driven tasks missed, priority periodic tasks and normal periodic tasks which never ran even a single instance and priority periodic and normal periodic tasks with many missed instances. Whereas, for our proposed algorithm the starvation rate comprised of only soft deadline tasks as non-urgent event driven tasks and normal periodic tasks. FEF shows the lowest starvation rate with an average of around 10% while the next closest is almost double the proposed FEF scheduling algorithm. The results indicate that proposed algorithm uses CPU time in the most careful manner, putting its best efforts to maximize output.

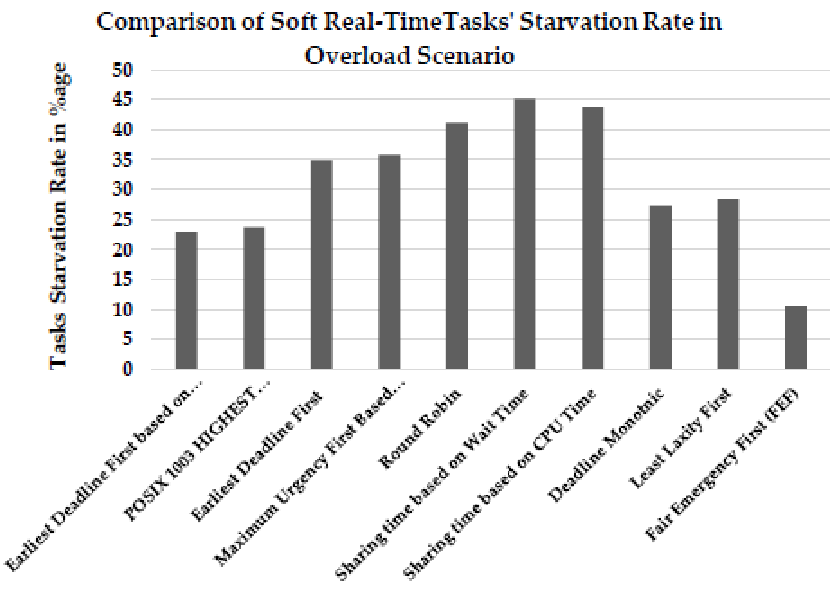


Figure 9. Average starvation rate comparison between FEF and other algorithms.

In the overload scenario results provided in the Figures 8 and 9, the proposed FEF scheduler, though missed 10% of the tasks, did not miss any hard-real time task. The proposed FEF scheduler makes a tradeoff between soft real time tasks and hard real time tasks in such overload scenarios, as hard real time tasks are crucial to run on time in real-time systems.

5.2. Priority Bit's Effect for Periodic Tasks

In this section, we observe the effect of priority bit's addition to the periodic tasks. We test and compare the scenarios where the processor is loaded with majority of urgent event-driven tasks. Since our proposed algorithms gives best results in comparison to other algorithms, but worst case can be when the processor is flooded with the urgent event-driven tasks. In such case according to basic definitions, urgent event-driven task should be executed right away without giving any other task a chance to execute itself. However, in the scenarios where some periodic task has to be executed after certain time period, and has some maximum limit of being missed or delayed, we introduced PB (Preemption Bit) for the periodic tasks. The periodic task with its PB value as 1 can run after a starvation period of specified PT (Preemption Threshold) value.

The Figure 10a shows the output of the scenario when PB is set to 0, in comparison to the case shown in Figure 10b where PB is set to 1. When PB is set as 0, the urgent event-driven tasks will keep running one after another and the periodic tasks will be left starving for the CPU resources. On the other hand, when PB is set to 1 then once the periodic task passes the threshold of starvation, processor will preempt the urgent event-driven task and allocate the CPU resources to the periodic task with preemption bit set as 1. Hence, for any case where some important period update is must to be made after certain amount of time, FEF gives better options and results.

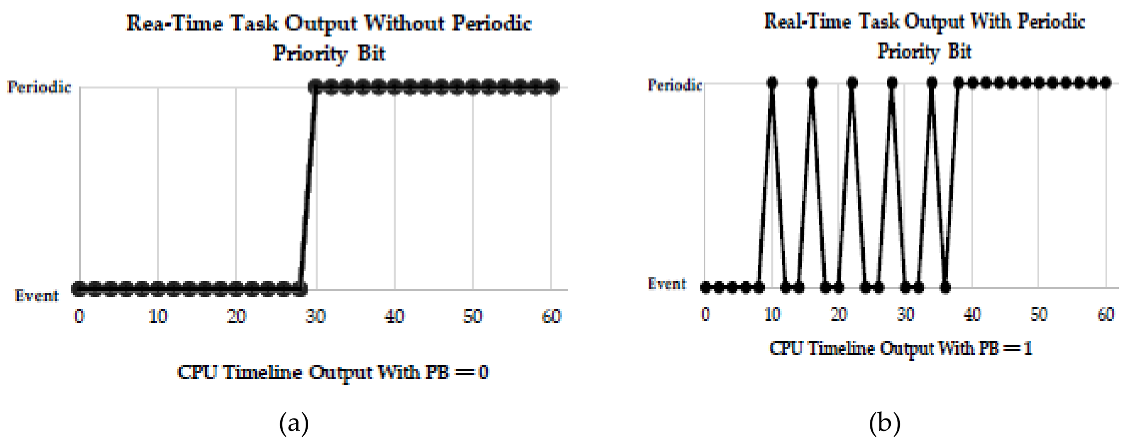


Figure 10. (**a**) Event-driven loaded scenario without preemption bit (**b**) event-driven loaded scenario with preemption bit.

5.3. Comparison Analysis with Scheduling Algorithm for Embedded Systems

In this section, we make the comparisons between our proposed intelligent FEF scheduling algorithms with a non-intelligent FEF implementation. The proposed algorithm without machine learning modules in UM and FM is referred as non-intelligent FEF. For the non-intelligent implementation, we set the x in UM constant at 2 and the FM between the periodic tasks is eliminated resulting in execution of high priority periodic task first.

The comparison of intelligent FEF with non-intelligent FEF is presented in Figure 11. It shows the comparison of task starvation rate and average instances missed rate for task instances. We can observe a significant increase in the starvation rate and instances missed rate for the non-intelligent FEF. Hence, the learning modules inclusion in the proposed methodology plays a vital role in the overall performance of the scheduler for overloaded scenarios.

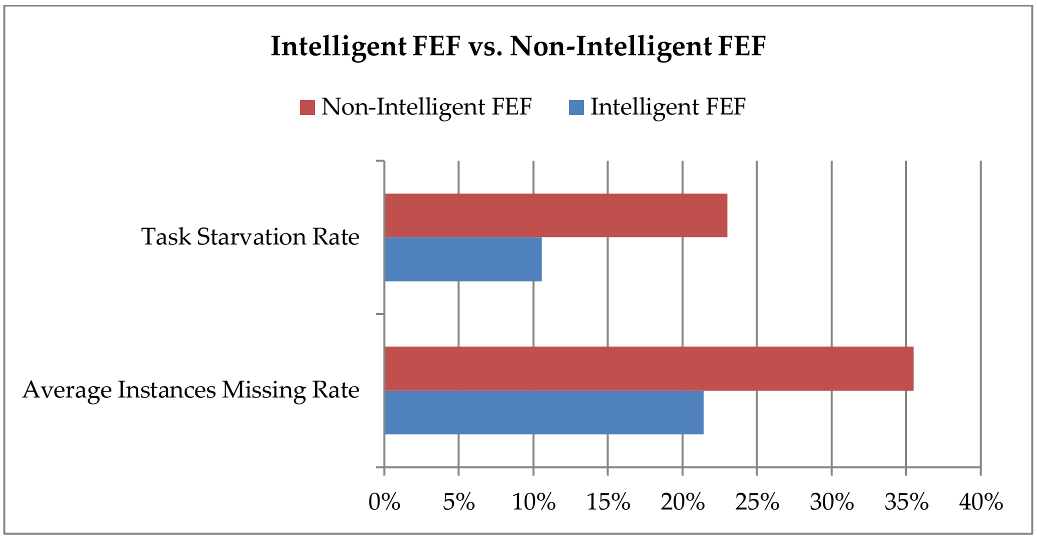


Figure 11. Performance comparison of FEF with and without ANN learning module.

We also compare the proposed scheduler with commonly used combined scheduling policy for embedded IoT systems. Most of the real-time embedded IoT systems use priority-based scheduling algorithms such as LINUX and QNX based real-time embedded IoT systems follow scheduling algorithms with high priority first with FIFO and RR [31]. VxWorks, a real-time operating system designed for embedded IoT systems uses pre-emptive priority-based scheduler as default scheduler along with option of RR scheduler and user defined custom scheduler [32]. The priority scheduler follows 256 priority levels, same as used in Windows CE too [31]. We have implemented a combined approach of scheduling scheme followed in embedded systems to make comparisons with our proposed FEF scheduling scheme. The combined approach implemented is a combination of preemptive priority-based time sliced scheduling (8–256 priority levels) and in case of same priority it applies FIFO and RR scheduling.

In Table 7 we present the comparison results, performed under heavy loads scenarios, between proposed scheduler and hybrid priority-based scheme. The number of tasks completing within deadline shows that how many tasks out of total successfully executed before deadline while success rate shows the percentage of successful completion. Starvation rate minimized shows the difference of success rate between the combined implemented approach and the proposed FEF scheduling approach. It is evident from the comparisons that the proposed FEF scheduling scheme makes a fair effort to increase CPU utilization and throughput.

Table 7. Comparison between combined scheduling scheme and FEF.

No. of Tasks	**Priority based (8–256 priority levels) followed by FIFO and RR Scheduling**		**Fair Emergency First (FEF) Scheduling**		**Starvation Rate Minimized (%)**
	No. of tasks Completing within deadline	**Success Rate (%)**	**No. of tasks Completing within deadline**	**Success Rate (%)**	
40	31	77.5	34	77.5	7.5
80	55	68.75	61	68.75	7.5
120	83	69.16	92	69.16	7.51
160	121	75.62	129	75.62	5
200	154	77	166	77	6
240	188	78.33	204	78.33	6.67
280	223	79.6	242	79.6	6.82
320	249	77.8	272	77.8	7.2
360	293	81.3	321	81.3	7.86
400	315	78.75	349	78.75	8.5

Figure 12 shows the result comparisons for worst case execution times intelligent FEF, non-intelligent FEF and hybrid priority algorithm. Hybrid priority in the figure above refers to the priority-based (8–256 priority levels) followed by FIFO and RR scheduling algorithm used in the embedded systems. We can observe that the WCET for non-intelligent version is very close to the WCET for hybrid priority, whereas the intelligent FEF scheduling algorithm has very low execution time in such scenarios.

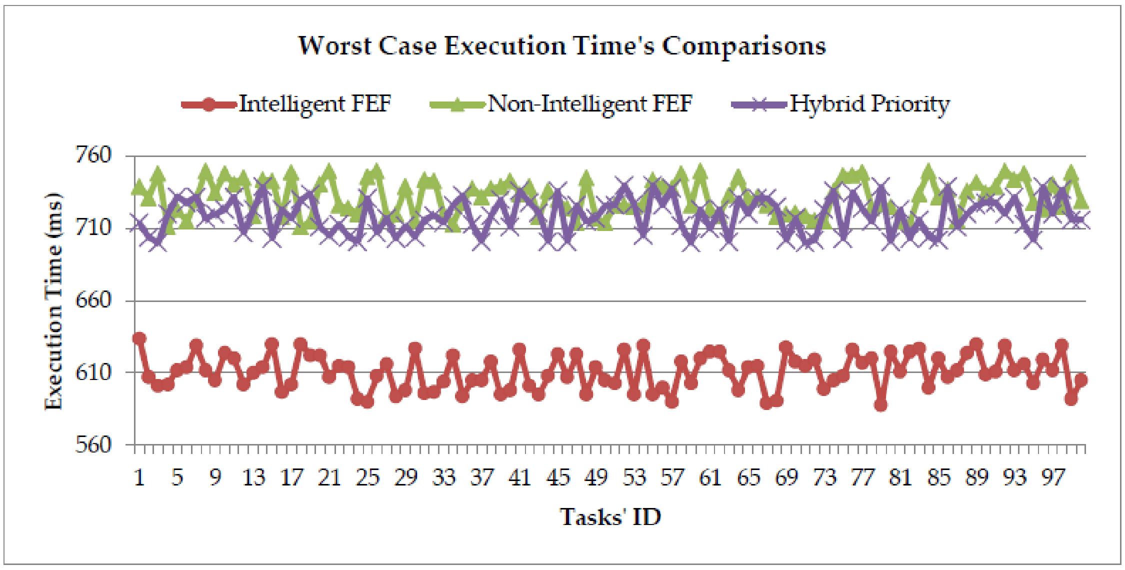


Figure 12. Worst case execution time (WCET).

6. Discussions

As in this work, we aimed to propose an intelligent real-time scheduling algorithm which best suits the combination hard real-time and soft real-time embedded IoT systems. The proposed scheduling algorithm aims to schedule the hard real-time tasks efficiently while makes its best effort to save soft real time tasks from starvation in heavy load scenarios. Hence, the proposed scheduler is strict hard real-time system for the hard real-time tasks input and a best effort soft real-time system for the soft real-time tasks input.

Through our extensive literature review, we realized the importance of an adaptive and flexible scheduling algorithm, which can take combination (hard deadline and soft deadline) real-time tasks' set as input with different priorities and urgency factors and give the best possible solution for the general-purpose systems as well as for the embedded IoT systems. We aim to best allocate the CPU resources to the available tasks while trying to minimize high loss of soft deadlines, or starvation of any low priority task with soft deadline. Hence, we proposed an algorithm, named as Fair Emergency First (FEF) consisting of two intelligent components as Urgency Measure (UM) and Failure Measure FM). The proposed algorithm gives first preference to the urgent and high priority tasks and at the same time tries to save the low priority tasks from unnecessary starvation by implementing fair division of CPU resources among all the tasks. In our proposed scheduling scheme, both the measures introduced (UM and FM), use ANNs to best predict the next move of the scheduler. Moreover, in our proposed approach we also introduced factors such as Reserved, PB and PT; giving a full independence across different system environments and constraints, to tune scheduling according to different scenarios.

For the performance analysis, we first compared our proposed approach with multiple traditional and combined scheduling approaches, and then we evaluated the effect of intelligent modules by comparing the intelligent FEF with non-intelligent FEF. We also evaluated the proposed algorithm in contrast to the most commonly used hybrid scheduling scheme in embedded systems. The results of the performance analysis show a significant reduction in the number of tasks starved in FEF as compared to other algorithms as well as in the number of task instances missed in FEF as compared to algorithms. In comparison with hybrid approach for embedded IoT systems; under strict constraint of less preemption, a significant reduction in task starvation is observed. The proposed algorithm provides the maximum utilization of available resources along with high performance. In scenarios, where only the hard real-time tasks are loaded on the scheduler, the proposed would act as a firm real-time scheduler; whereas when the proposed scheduler will be given combination inputs then it will act as a savior for soft deadline tasks too and also put efforts in lowering the starvation rates. Hence, our proposed intelligent scheduling algorithm is a best fit for real-time tasks scheduling in embedded IoT systems with a combination of both hard real-time and soft-real time scenarios.

Our proposed approach uses ANNs for better resource allocation prediction by the scheduler. One of the limitations of the proposed approach is that it needs some time to gather enough history logs for the training purposes. Though, the scheduler performs reasonably best at the initial stages too, as seen in the non-intelligent implementation of the FEF algorithm. Applications where the scheduling system is to be deployed for long periods; the addition of learning modules will be of great use with the passage of time as the system will learn from its scheduling decisions. On the other hand, if scheduling system is to be deployed for a brief amount of time then the system might not get enough chance to make use of learning modules to their full potential.

Author Contributions: Data Curation, S.M.; Formal Analysis, S.M.; Funding Acquisition, D.K.; Investigation, S.A. and I.U.; Methodology, S.M.; Resources, D.K.; Software, S.A.; Supervision, D.K.; Validation, S.M.; Visualization, S.M. and S.A.; Project management, D.H.P.; Writing—original draft, S.M.; Writing—review and editing, I.U. and D.K.

Funding: This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) and ITRC (Information Technology Research Center) support program supervised by the IITP.

Acknowledgments: This work was supported by Institute for Information & communications Technology Promotion (IITP) grant funded by the Korea government (MSIT) (No.2017-0-00526, Development of Intelligent IoT System capable of cooperation and learning between things in movement-free environment), and this research was supported by the MSIT(Ministry of Science and ICT), Korea, under the ITRC(Information Technology Research Center) support program(IITP-2019-2014-1-00743) supervised by the IITP(Institute for Information & communications Technology Planning & Evaluation). Any correspondence related to this paper should be addressed to DoHyeun Kim.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Lytras, M.D.; Raghavan, V.; Damiani, E. Big data and data analytics research: From metaphors to value space for collective wisdom in human decision making and smart machines. *Int. J. Semant. Web Inf. Syst.* **2017**, *13*, 1–10. [CrossRef]
2. Chen, C.Y.; Hasan, M.; Mohan, S. Securing real-time internet-of-things. *Sensors* **2018**, *18*, 4356. [CrossRef] [PubMed]
3. Audsley, N.; Burns, A. Real-time System Scheduling. 1990. Available online: http://citeseerx.ist.psu.edu/viewdoc/download;jsessionid=82FC9A6552D388204F29755AA44C970F?doi=10.1.1.29.4929&rep=rep1&type=pdf (accessed on 20 January 2019).
4. Mohammadi, A.; Akl, S.G. *Scheduling Algorithms for Real-Time Systems*; Tech. Rep; School of Computing Queens University: Kingston, ON, Canada, 2005.
5. Ahmad, S.; Malik, S.; Ullah, I.; Park, D.H.; Kim, K.; Kim, D. Towards the Design of a Formal Verification and Evaluation Tool of Real-Time Tasks Scheduling of IoT Applications. *Sustainability* **2019**, *11*, 204. [CrossRef]
6. Schwiegelshohn, U.; Yahyapour, R. Analysis of first-come-first-serve parallel job scheduling. *SODA* **1998**, *98*, 629–638.
7. Arpaci-Dusseau, R.H.; Arpaci-Dusseau, A.C. Operating Systems: Three Easy Pieces. Available online: https://www.usenix.org/system/files/login/articles/login_spring17_02_arpaci-dusseau.pdf (accessed on 11 April 2019).

8. Ramamritham, K.; Stankovic, J.A. Scheduling algorithms and operating systems support for real-time systems. *Proc. IEEE* **1994**, *82*, 55–67. [CrossRef]
9. Oh, S.-H.; Yang, S.-M. A modified least-laxity-first scheduling algorithm for real-time tasks. In Proceedings of the Fifth International Conference on Real-Time Computing Systems and Applications, Hiroshima, Japan, 27–29 October 1998.
10. Lehoczky, J.; Sha, L.; Ding, Y. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In Proceedings of the Real-time Systems Symposium, Santa Monica, CA, USA, 5–7 December 1989; pp. 166–171.
11. Audsley, N.C.; Burns, A.; Richardson, M.F.; Wellings, A.J. *Deadline Monotonic Scheduling*; University of York, Department of Computer Science: York, UK, 1990.
12. Stankovic, J.A.; et al. *Deadline Scheduling for Real-Time Systems: EDF and Related Algorithms*; Springer Science & Business Media: Berlin/Heidelberg, Germany, 2012; Volume 460.
13. Chetto, H.; Chetto, M. Some results of the earliest deadline scheduling algorithm. *IEEE Trans. Softw. Eng.* **1989**, *15*, 1261. [CrossRef]
14. Stewart, D.B.; Khosla, P. Real-time scheduling of sensor-based control systems. *IFAC Proc. Vol.* **1991**, *24*, 139–144. [CrossRef]
15. Buttazzo, G.C. Rate monotonic vs. EDF: Judgment day. *Real-Time Syst.* **2005**, *29*, 5–26. [CrossRef]
16. Wu, Y.; Song, X.; Gong, G. Real-time load balancing scheduling algorithm for periodic simulation models. *Simul. Model. Pract. Theory* **2015**, *52*, 123–134. [CrossRef]
17. Nakahira, Y.; Chen, N.; Chen, L.; Low, S.H. Smoothed Least-laxity-first Algorithm for EV Charging. In Proceedings of the Eighth International Conference on Future Energy Systems, Hong Kong, China, 16–19 May 2017; pp. 242–251.
18. Tabuada, P. Event-triggered real-time scheduling of stabilizing control tasks. *Ieee Trans. Autom. Control* **2007**, *52*, 1680–1685. [CrossRef]
19. Jejurikar, R.; Gupta, R. Dynamic slack reclamation with procrastination scheduling in real-time embedded systems. In Proceedings of the 42nd annual Design Automation Conference, Anaheim, CA, USA, 13–17 June 2005.
20. Tidwell, T.; Glaubius, R.; Gill, C.; Smart, W.D. Scheduling for reliable execution in autonomic systems. In *International Conference on Autonomic and Trusted Computing*; Springer: Berlin/Heidelberg, Germany, 2008; pp. 149–161.
21. Dighriri, M.; Alfoudi, A.S.D.; Lee, G.M.; Baker, T.; Pereira, R. Comparison data traffic scheduling techniques for classifying QoS over 5G mobile networks. In Proceedings of the 2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA), Taipei, Taiwan, 27–29 March 2017; pp. 492–497.
22. Dighriri, M.; Lee, G.M.; Baker, T. Applying Scheduling Mechanisms Over 5G Cellular Network Packets Traffic. In *Third International Congress on Information and Communication Technology*; Springer: Singapore, 2019; pp. 119–131.
23. Bala, A.; Chana, I. Autonomic fault tolerant scheduling approach for scientific workflows in Cloud computing. *Concurr. Eng.* **2015**, *23*, 27–39. [CrossRef]
24. Eker, J.; Hagander, P.; Årzén, Ka. A feedback scheduler for real-time controller tasks. *Control Eng. Pract.* **2000**, *8*, 1369–1378. [CrossRef]
25. Marzario, L.; Lipari, G.; Balbastre, P.; Crespo, A. Iris: A new reclaiming algorithm for server-based real-time systems. In Proceedings of the Real-Time and Embedded Technology and Applications Symposium, Toronto, ON, Canada, 25–28 May 2004; pp. 211–218.
26. Cho, H.; Ravindran, B.; Jensen, E.D. An optimal real-time scheduling algorithm for multiprocessors. In Proceedings of the Real-Time Systems Symposium, Rio de Janeiro, Brazil, 5–8 December 2006.
27. Buttazzo, G.C.; Bertogna, M.; Yao, G. Limited preemptive scheduling for real-time systems. a survey. *IEEE Trans. Ind. Inform.* **2013**, *9*, 3–15. [CrossRef]
28. Huang, W.-H.; Chen, Ji.; Zhou, H.; Liu, C. PASS: Priority assignment of real-time tasks with dynamic suspending behavior under fixed-priority scheduling. In Proceedings of the 52nd ACM/EDAC/IEEE Design Automation Conference (DAC), San Francisco, CA, USA, 8–12 June 2015; pp. 1–6.

29. Ayele, A.A.; Rao, V.S.; Dileep, K.G.; Bokka, R.K. Combining EDF and LST to enhance the performance of real-time task scheduling. In Proceedings of the International Conference on ICT in Business Industry & Government (ICTBIG), Indore, India, 18–19 November 2016; pp. 1–6.
30. Cooking Hacks. e-Health Sensor Platform. 2015. Available online: https://www.cooking-hacks.com/documentation/tutorials/ehealth-biometric-sensor-platform-arduino-raspberry-pi-medical (accessed on 15 December 2018).
31. Urunuela, R.; Déplanche, A.-M.; Trinquet, Y. Storm a simulation tool for real-time multiprocessor scheduling evaluation. In Proceedings of the IEEE Conference on IEEE Emerging Technologies and Factory Automation (ETFA), Bilbao, Spain, 13–16 September 2010; pp. 1–8.
32. Chéramy, M.; Hladik, P.-E.; Déplanche, A.-M. SimSo: A simulation tool to evaluate real-time multiprocessor scheduling algorithms. In Proceedings of the 5th International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS), Madrid, Spain, 8 July 2014.