<u>**PROJECT REPORT**</u>

Done under the guidance of

Dr. Biju K. Raveendran

Submitted by

Iyer Rajkumar Varsha

(2017A3PS0292G)

In partial fulfilment for the award of degree in

Bachelor of Engineering:

Electrical and Electronics Engineering

Course: Design Project

Course Code: EEE F376

Design of a Mixed Criticality Scheduler with improved QoS

BIRLA INSTITUTE OF TECHNOLOGY AND SCIENCE, PILANI

May 2020

ACKNOWLEDGEMENT

I would like to deeply express my gratitude towards Dr. Biju K. Raveendran, Professor in the Computer Science Department at BITS Pilani, KK Birla Goa Campus, for giving me the opportunity to undertake this project. His sincerity and his guidance has helped me countless times and his unwavering support has made my insights into the field of systems possible.

I would also like to thank the members of BITS Pilani, KK Birla Goa campus, for their encouragement and co-operation while I was doing this project.

Last but not least, I would like to thank my family and friends for their constant support. I am very grateful to them.

ABSTRACT

This report discusses mixed criticality systems, the various issues related to scheduling tasks in such systems, models and different approaches towards scheduling. We look at Earliest Deadline First(EDF) algorithm in particular, and then extend it to EDF-VD(EDF-Virtual Deadline) algorithm.

INDEX

CHAPTER 1

INTRODUCTION

1.1 Introduction

This project, titled “Design of a Mixed Criticality Scheduler with improved QoS”, is situated in the research field of real-time mixed criticality scheduling. Scheduling analysis, until some years back, had been done with the assumption that all tasks in the system have same level of criticality. This approach has been extended by introducing what is called a “mixed criticality system".

A mixed criticality system is a system with two or more distinct levels of criticality. In some domains, it is necessary to certify the system in order to ensure that it will operate in a safe and secure way. This certification requires providing evidence that the system will behave as expected. Perhaps up to five levels of criticality can be identified, with typical names for levels being ASILs (Automotive Safety and Integrity Levels), DALs (Design Assurance Levels or Development Assurance Levels) and SILs (Safety Integrity Levels) etc. Different levels of assurance criticality are present for different fields like robotics, automobiles etc. The worst-case execution times(WCET) of tasks also become dependent on criticality.

Traditionally, components with different criticality levels were present on separate processors. However, since this results in unwanted resource consumption, nowadays components with different criticality levels coexist in the same processor as well.

In a simplified case, there are only two criticality levels – high-criticality (HI) and low-criticality (LO). HI criticality tasks have to be guaranteed to meet their deadlines, while low criticality tasks should not cause a high criticality task to miss its deadline. QoS, or quality of service, depends on the performance of all of these tasks. While a low criticality task should not interfere with high-criticality tasks, disregarding them will affect the overall performance of the system and lower the QoS. The low criticality tasks can be discarded, however they do not always have to be discarded.

A task should not overrun the WCET of its own criticality level. If it does, the system must shift to a higher criticality mode.

Please note that throughout the report, the assumption is that there is only a single processor.

1.2 Aim and Motivation

The motivation to undertake this project was to get more insights into the field of mixed criticality systems and try to apply the knowledge of real-time systems in this field. It was also to get a flavour of the different scheduling algorithms using in the field of mixed criticality systems.

1.3 Related Work

The papers that have been referred to specifically are:

- “Mixed Criticality Systems - A Review” by Alan Burns and Robert I. Davis [1], which covers research on the topic of mixed criticality systems. We look at mixed criticality models and single processor analysis in this paper, delving into fixed priority scheduling and EDF scheduling in particular.
- “Preemptive Uniprocessor Scheduling of Mixed-Criticality Sporadic Task Systems” by Sanjoy Baruah et al.[2], which introduces EDF-VD for mixed criticality systems and elaborates on its capabilities when it comes to scheduling a mixed criticality taskset.

CHAPTER 2

MIXED CRITICALITY SYSTEMS

2.1 Example of a Mixed Criticality System

A typical example is an aircraft's software system, or a car's software system. Let us mention only a few tasks in the car's software system – the speedometer and the airbag function. The system designer is interested in ensuring that all parts of the system work at all times. However, that is not always possible. There are many issues that pop up – when faced with a choice between two functions to be executed, which has to be chosen? Criticality of the airbag function is intuitively much higher than the speedometer. However, the more tasks there are, the harder it becomes to clearly distinguish which criticalities each should belong to, and to decide how many criticality levels should be present.

Also, what if the localised parts necessary to run the airbag function is damaged? The only way to reduce such chances is to have duplicate units adhering to this function elsewhere. In other words, how do we resolve the conflicting requirements between partitioning for safety assurance (in case some units fail), and sharing for efficient resource usage?

2.2 Issues to Overcome

For mixed criticality systems, there are two mainly distinct issues – the first one is run-time robustness and the second is static verification.

Run-time robustness is a form of fault tolerance such that even when all components cannot run satisfactorily, the higher-criticality tasks must still meet their deadlines. Lower-criticality tasks may need to sacrifice their requested level and quality of service in order to keep the higher-criticality tasks running satisfactorily.

Static verification is mostly related to the issue of certification of safety-critical systems. How do we validate these systems? What are the worst-case scenarios to assume to determine the worst-case behaviour of the system? Generally the Certification Authority(CA) has rather pessimistic assumptions about the worst-case, so that the system's safely-critical parts can be deemed correct. The CA is only concerned with the correctness of the safely-critical part of the system, while the system designer would wish to ensure that the entire system, inclusive of the non-critical parts, are correct, so that the degradation of the quality of service of the entire system is less.

2.3 Formal Mixed Criticality Model Description

A system is defined as a finite set of components K. Formally, each component is defined using a finite taskset, where each task τi is defined by a 4-tuple of parameters: τi = (Pi , Di , Ci , Li), where:

- Pi ∈ R + is its period (minimum interarrival time i.e. the minimum time the task will repeat again. It can also come any time after that.)
- Di ∈ R + is its deadline. Generally, Di is greater than or equal to Pi.
- Ci ∈ R + is WCET for each criticality level, where L is number of criticality levels
- Li ∈ N + is criticality level of the task. The larger i is, the higher the criticality is.

These parameters are however not independent. Tasks give rise to a potentially unbounded sequence of jobs.

Each job's actual computation time is not known – it depends on various factors like available hardware and how deterministic the hardware is.

2.4 Issues Arising while Defining a Model

While defining a model, there are issues regarding separation of tasks – tasks from different components must not interfere with each other. This is because this might cause changes in execution times and also tasks cannot exceed its specified computation time then. This is a much more significant issue when components are of different criticality levels.

The higher the criticality level, the more stringently the tasks have to be verified. This increases computation time Ci, which lowers the efficiency.

For systems executing on more deterministic hardware platforms, WCET cannot be known with full certainty. For example, in a system, if a job occurs all cache miss, or if I/O performs erratically, we cannot determine WCET. To make a platform deterministic, we require many things like TLB, cache etc to be deterministic. Without that, uncertainty will always arise due to interferences. For these systems with time-randomised components, generally probabilistic WCET is used.

2.5 New Definition of Model for a Mixed Criticality System

Each task Ti is defined by a four-tuple of parameters (T->, D, C-> , χ), where C-> and T-> are vectors of values – one per criticality level, with the constraints:

- L1 > L2 ⇒ C(L1) ≥ C(L2). Higher criticality means higher computation.
- L1 > L2 ⇒ T(L1) ≤ T(L2) for any two criticality levels L1 and L2. Lower period is higher criticality.

Tasks can have different kinds of deadlines – like the safely critical deadline and QoS deadline. A system will execute in a number of criticality modes, starting from the lowest criticality. If jobs in the system behave according to the criticality mode at that time, system remains in the lowest criticality. However, if any job oversteps the boundaries set by the current criticality mode of the system, i.e if any job tries to take up more computation or execution time, or executes more frequently than in accordance with the criticality mode of the system, criticality mode change occurs.

In most papers, criticality is restricted to only increase and not decrease. Many papers also restrict themselves to 2 criticality levels, High(HI) and low(LO) – also called a dual-criticality system. Such systems will have tasks with parameters of minimum interarrival time, deadline, worst-case execution times of HI and LO levels (Ci(HI) and Ci(LO)), and task criticality.

In the case of a dual criticality system, a scheduling algorithm for such a model is said to be *correct* if it is able to schedule any system such that:

- When system is in LO-criticality, all jobs achieve required level of execution between release time and deadline.
- When system is in HI-criticality, all HI-criticality jobs get the required level of execution between release time and deadline.

2.6 Fixed Priority Scheduling Schemes

We will first look at RTA-based approaches, then we will move on to slack scheduling and period transformations.

2.6.1 Response-time based approaches

We will discuss two approaches in this, one by Vestal and one by Baruah.

2.6.1.1 Vestal's Approach

In one approach(by Vestal) formalised by Dorin et al. [3], release jitter was considered which allowed priorities of high and low criticality tasks to be interleaved. However, all tasks had to be evaluated as if they were of highest criticality(that is, during validation).

In the extended version of this approach, better use of the processor was made and also, many systems guaranteed by other approaches could be scheduled, plus many that were not. However, the number of criticality modes were restricted to two. This means that when the system was at low criticality, execution times were lower (within low criticality bounds) and all deadlines were guaranteed to be met .When it was at high criticality, this rely condition of "all jobs meet deadlines" are weaker, and high-criticality jobs can have larger execution times. In other words, in high-criticality, the system ensures that high-criticality jobs execute satisfactorily even if it may be at the expense of low-criticality jobs. However, mode change does not happen instantaneously, and what happens during the mode change is still rather unclear.

2.6.1.2 Baruah's Approach

In another approach (By Baruah et al.[4]), the priority of high-criticality tasks are maximised and tasks are ordered via deadline-monotonic algorithm. Simplified Audsley's[5] priority assignment algorithm(which is a scheme with n^2 complexity) is used to assign priorities from lowest to highest level.

It first identifies some tasks which may be assigned the lowest priority. Then this task is removed from the task system and then this recursion continues.

More formally, at each priority level, the lowest priority task from the low criticality task set is tried first:

- If it is schedulable, then the algorithm moves up to the next priority level
- If it is not schedulable, then the lowest priority task from the high criticality set is tested. If it is schedulable then again the algorithm moves on to the next level. But if neither of these two tasks are schedulable then the search can be abandoned as the task set is un-schedulable.

Maximum 2N-1 tests are needed, where N is the number of tasks. Highest priority tasks are anyway schedulable as computation time of those are less than deadline.

If the algorithm were not applicable it would have needed n! possible orderings to be searched.

The protocol of dropping all LO-criticality work if any task executes for more than its C(LO) value and moving the system to high-criticality mode is shown to outperform other schemes in terms of success ratio for randomly generated tasksets.

2.6.2 Slack Scheduling

Instead of dropping all LO-criticality jobs in the event of a mode change, LO-criticality jobs are run in the slack generated by HI-criticality jobs, only using their low criticality execution budgets. Slack is generated by tasks not using their full budget, and also by job arrivals being lesser than anticipated. There are certain issues with this method. When is it safe to allocate slack – how can we be sure that high criticality jobs still operate correctly without missing their deadlines? When should ‘slack’ of a non-appearing high-criticality sporadic task be allocated to low-criticality jobs? After all, we only know minimum interarrival times, and the task may come any time after that. If a low-criticality task executes beyond its deadline, high-criticality task could miss its deadline. Therefore, the low-criticality task must either be aborted at its deadline, or its priority must be reduced to a background level to derive a safe analysis. Otherwise, it could cause cascaded failures, where the low criticality task overrunning would cause another task to overrun and so on.

2.6.2.1 Sensitivity analysis and scaling

A scaling factor F (F > 1) is introduced such that the system remains schedulable with all C(LO) values replaced by F * C(LO). The same is done for C(HI) calues.

This increases robustness of system, as LO criticality jobs can execute for longer (scaled) before mode change is triggered.

2.6.3 Period Transformation

Period transformation splits task with period T and C into n parts, such that each of the split tasks have parameters T/n and C/n. If all tasks have deadlines equal to periods (such systems are called *implicit* systems), then the application of rate monotonic algorithm (where lesser period means higher priority) would increase relative priority of all the transformed tasks. If we transform high-criticality tasks this way such that the transformed periods are shorter than the ones of low criticality tasks, then rate monotonic algorithm would deliver critically monotonic priorities, where transformed high criticality tasks have higher priority. However, there will obviously be increased number of context switches, which is detrimental for low-criticality tasks with small periods. The context switch time also depends on how the tasks have been split.

If we ignore the context-switching overhead, this works well in theory, due to the inherent property of period transformation to deliver tasksets with harmonic periods (Baruah and Burns [6]), which are more likely to be schedulable.

Without period transformation, low criticality task execution times must be monitored since they may have high priorities. High criticality tasks must also be monitored in case they trigger a criticality change (if they execute for more than C(LO)). With period transformation, low-criticality jobs need not be monitored since they have lower priorities, while high-criticality tasks must still be monitored to make sure they do not overrun their budgets and affect another high-criticality task.

The higher the number of criticality levels, the lesser the benefit of period transformation becomes.

2.7 EDF Scheduling

Earliest Deadline First (EDF) is an optimal dynamic priority scheduling algorithm which allows pre-emption. It uses absolute deadlines to assign priorities to tasks. The task whose deadline is closest gets the highest priority. Therefore, priorities are changed dynamically. In EDF, if CPU usage is lesser than 100%, it means that all tasks have met their deadlines (an optimal feasible schedule is found). A feasible schedule is one in which all the tasks in the system are executed within the deadline. This can be used to schedule real-time systems.

The C-code for this algorithm has been given in the appendix A. In the implementation, an implicit system has been chosen, where period of tasks is equal to their respective deadlines.

2.7.1 Algorithm

1. Read input for the taskset parameters
2. Find the hyperperiod of the taskset in order to schedule the taskset for that hyperperiod

3. Call the runtime function. Timer is initialised as -1. Next closest job arrival of all the tasks is found. This also saves the jobs to be added at that moment with parameters in accordance to the related task.
4. While timer is less than hyperperiod, steps 5 to 8 are repeated:
5. If timer is equal to next job arrival time, the jobs arriving at that time is added to the job queue. If job queue is empty or null, this will be the first job to be added. Then, next job arrival is calculated again.
6. In case the queue is empty, the processor is idle.
7. If any job has completed its execution time, then the job is removed from the job queue and that memory is freed.
8. Timer is increased by whatever granularity is chosen.
9. The schedule is printed out and the program exits.

2.7.2 Pseudo Code for Algorithm

```
1. Globally, the timer is initialised to -1. Job_queue_node structure's linked list hed
   and tail are initialised to NULL.
2. Program starts.
3. Input entered. Get_tasks function called to populate the dynamic array of
   structures (structure task) t.
4. Initialise hyperperiod by Hyperperiod_calc function.
5. Runtime function is called:
6. Initialise next_job_arrival by new_job_arrival function (main parameter timer).
   This also changes jobs_to_add dynamic array.
7. While timer is less than hyper_period:
   a. If timer is equal to hyper_period:
           i. Job *temp is initialised with jobs_to_add
          ii. Iterate through temp and insert jobs properly in the job linked list using
              insert_job(temp).
         iii. Next_job_arrival is calculated again using new_job_arrival function
              with timer+1.
   b. If head of job linked list is null and timer>=0(to avoid time -1 prints):
           i. The processor is idle
   c. Timer is incremented by 1
   d. If head is not null:
```

i. The execution time of job in head is decremented by 1

ii. If the execution time of job in head reaches 0:

    1. Head is removed from list and list is updated
    2. The memory of the head is freed

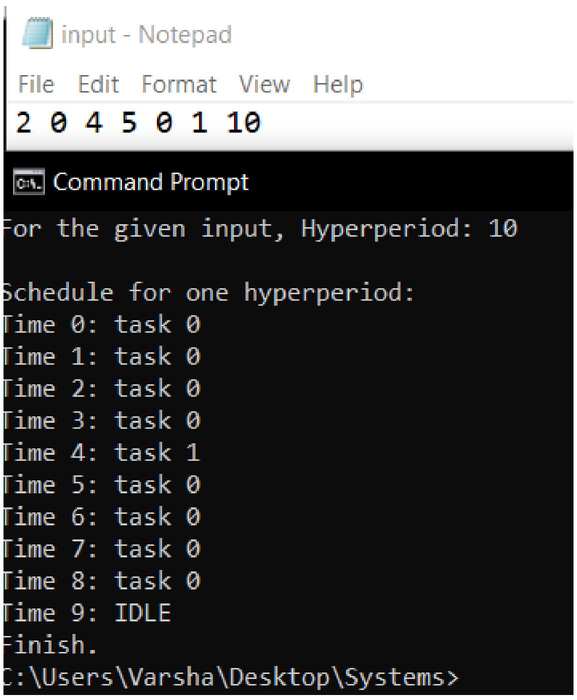
8. Program finish

2.7.3 Input and Output of program

The input format is: (with spaces in between)

Number_of_tasks arrival_time_task_0 execution_time_task_0 period_task_0 …

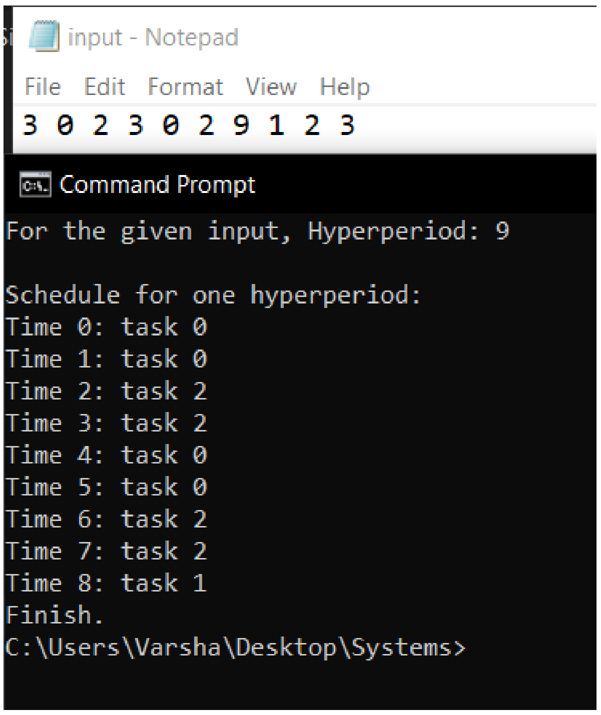
Here two cases have been provided as the example for input and output for code in appendix A. Please note that tasks are numbered starting from 0, and that schedule is given for one hyperperiod. Time granularity has been taken as 1 here.



```
2 0 4 5 0 1 10
```

```
For the given input, Hyperperiod: 10

Schedule for one hyperperiod:
Time 0: task 0
Time 1: task 0
Time 2: task 0
Time 3: task 0
Time 4: task 1
Time 5: task 0
Time 6: task 0
Time 7: task 0
Time 8: task 0
Time 9: IDLE
Finish.
C:\Users\Varsha\Desktop\Systems>
```



```
3 0 2 3 0 2 9 1 2 3
```

```
For the given input, Hyperperiod: 9

Schedule for one hyperperiod:
Time 0: task 0
Time 1: task 0
Time 2: task 2
Time 3: task 2
Time 4: task 0
Time 5: task 0
Time 6: task 2
Time 7: task 2
Time 8: task 1
Finish.
C:\Users\Varsha\Desktop\Systems>
```

Figure 2.1: Input and output of EDF code.

2.7.4 Adapting EDF algorithm for Mixed Criticality Systems

If we use EDF algorithm without any modification in mixed criticality systems, criticalities are really not considered and HI-criticality tasks may also execute after LO-criticality tasks. There are many modified versions that have been introduced.

For example, a slack-based scheme for EDF scheduling jobs for dual-criticality systems called the criticality-based EDF(CB-EDF) was introduced by Park and Kim[7]. This runs HI-criticality tasks as late as possible while making LO-criticality tasks run in that slack.

Another scheme by for dual-criticality systems assigned two relative deadlines to each high criticality task – one the real deadline and the other is the artificial earlier deadline so that high-criticality tasks may run earlier than low-criticality tasks in EDF. This artificial earlier deadline is obtained by reducing the actual deadline by some factor(which may not be the same for all the tasks). When a task overruns the bounds of the low-criticality mode, then the system changes into high-criticality mode and all low-criticality tasks are abandoned, while high-criticality tasks now run with their actual deadlines.

A similar scheme was presented by Baruah et al.[2], called EDF-VD (Earliest deadline first with virtual deadlines). HI-criticality tasks have their deadlines reduced, if necessary, during LO-criticality mode execution. All deadlines are reduced by the same factor, known as the "x factor". More on this scheme will be elaborated in chapter 3.

A different approach for dual-criticality systems was using a reservation-based approach, where sufficient budget was reserved for high criticality tasks. If they only use what is assumed by their low-criticality requirements, then a set of low-criticality tasks can be guaranteed. Therefore, in the capacity reclaimed from high-criticality tasks, low criticality tasks execute. The capacity reclaiming is changed by the deadlines chosen for high-criticality tasks.

CHAPTER 3

EARLIEST DEADLINE FIRST – VIRTUAL DEADLINES

3.1 Introduction

This algorithm is used to schedule mixed-criticality systems in a single preemptive processor, where only some of its functionality is safely critical (and needs proper validation) and the rest is non-safety critical and thus needs only a lower level of assurance(or even no certification at all). An implicit system where each task's deadline is equal to its period has been assumed.

In the paper "Preemptive Uniprocessor Scheduling of Mixed-Criticality Sporadic Task Systems" (Baruah et al. [2]), this algorithm has been described in length. In the paper, it is also proved that any 2-level implicit-deadline task system that can be scheduled by a clairvoyant exact algorithm on a single processor can be scheduled by EDF-VD on a processor 4/3 times as fast.

3.2 Model Used in Algorithm

The model used is: a mixed criticality sporadic taskset of a K-level system ($K \in N+$), a collection of tasks τi ($i \in N+$), is characterized by criticality level $\chi i \in [K]$ and a tuple(ci, di,pi)) $\in Q\chi i+ \times Q+ \times Q+$, where:

- ci = (ci(1), ci(2)…..ci(χi)) is a vector of WCETs, one for each criticality level from 1 to χi. Assume ci(1) ≤ci(2) ≤….. ≤ci(χi).
- di is relative deadline of tasks in τi
- pi is period (or rather, minimum interarrival time) between two jobs of task τi.

Task τi generates a sequence of jobs (Ji1, Ji2….), where a job Jij of task τi has the parameters Ji j = (aij, γij), where:

aij $\in$ R+ is the arrival time of the job,

γij $\in$ (0, ci(χi)] is the execution requirement of the job.

The absolute deadline of the job will obviously be aij + di.

The task τi gives rise to a potentially unbounded sequence of jobs Ji1, Ji2…with successive jobs being released atleast pi time units apart.

3.3 Related Semantics

The utilization of task τi at level k is:

$$u_i(k) \stackrel{\text{def}}{=} \frac{c_i(k)}{p_i} \qquad (3.1)$$

The total utilization at level k of tasks that are of criticality level l is:

$$U_l(k) \stackrel{\text{def}}{=} \sum_{i \in [n]:\, \chi_i = l} u_i(k) \qquad (3.2)$$

For example, taking the following taskset:

τ_i	χ_i	$c_i(1)$	$c_i(2)$	p_i
τ_1	1	2	2	4
τ_2	2	1	5	6

U1(1) = 2/4, U2(1) = 1/6. Therefore, total utilization at level 1 is the sum of these, 2/3.

At level 2, the total utilization is U2(2) = 5/6.

3.4 Algorithm

Algorithm EDF-VD consists of an offline preprocessing phase and a runtime scheduling phase. The first phase is done before runtime and it executes a schedulability test to determine whether the MC implicit sporadic task system τ is actually schedulable or not. If it is, the phase also gives two output values: an integer parameter k (where 1<= k <=K), and also an "x factor" (<=1) in order to determine virtual deadlines of tasks. In the second phase of actual runtime, there are K variants called EDF-VD(1), EDF-VD(2), ..., EDF-VD(K).Each of these is related to a different value of the parameter k that was provided by the first phase. At runtime, the variant EDF-VD(k) is applied based on the k value provided in the preprocessing phase.

3.4.1 Offline preprocessing phase

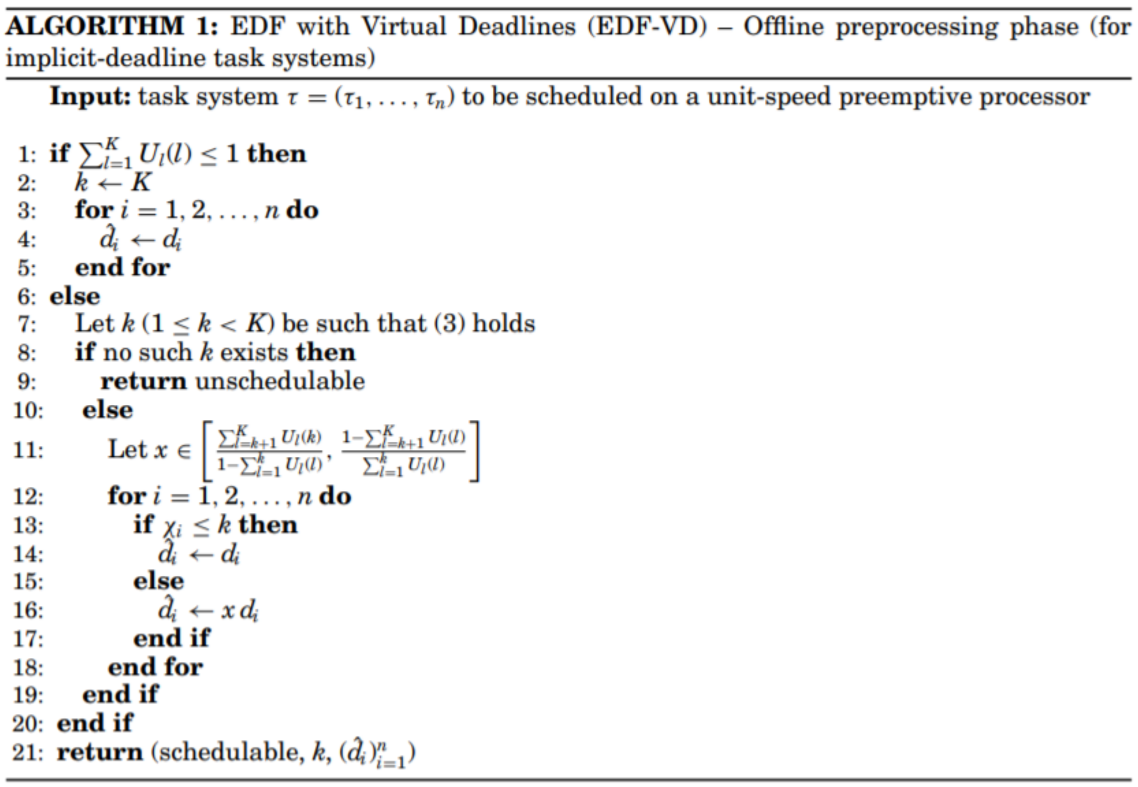
The algorithm is given in figure 3.1. Taking taskset τ. First, if

$$\begin{aligned} &\textstyle\sum_{l=1}^{K} U_l(l) \le 1 \\ &k \leftarrow K \end{aligned} \qquad (3.3)$$

Then all jobs can be scheduled for their worst-case execution times at their own criticality levels. This is illustrated in step 1 to 5.

Otherwise, it is tested whether a k exists such that :

$$1 - \sum_{l=1}^{k} U_l(l) > 0 \quad \text{and} \quad \frac{\sum_{l=k+1}^{K} U_l(k)}{1 - \sum_{l=1}^{k} U_l(l)} \leq \frac{1 - \sum_{l=k+1}^{K} U_l(l)}{\sum_{l=1}^{k} U_l(l)}, \tag{3.4}$$



ALGORITHM 1: EDF with Virtual Deadlines (EDF-VD) – Offline preprocessing phase (for implicit-deadline task systems)

Input: task system $\tau = (\tau_1, \ldots, \tau_n)$ to be scheduled on a unit-speed preemptive processor

```
1:  if $\sum_{l=1}^{K} U_l(l) \leq 1$ then
2:    $k \leftarrow K$
3:    for $i = 1, 2, \ldots, n$ do
4:      $\hat{d}_i \leftarrow d_i$
5:    end for
6:  else
7:    Let $k$ $(1 \leq k < K)$ be such that (3) holds
8:    if no such $k$ exists then
9:      return unschedulable
10:   else
11:     Let $x \in \left[\frac{\sum_{l=k+1}^{K} U_l(k)}{1-\sum_{l=1}^{k} U_l(l)}, \frac{1-\sum_{l=k+1}^{K} U_l(l)}{\sum_{l=1}^{k} U_l(l)}\right]$
12:     for $i = 1, 2, \ldots, n$ do
13:       if $\chi_i \leq k$ then
14:         $\hat{d}_i \leftarrow d_i$
15:       else
16:         $\hat{d}_i \leftarrow x\, d_i$
17:       end if
18:     end for
19:   end if
20: end if
21: return (schedulable, $k$, $(\hat{d}_i)_{i=1}^{n}$)
```

Figure 3.1: Algorithm for offline pre-processing phase in EDF-VD

If no such k exists, the taskset τ is deemed unschedulable (line 9). Otherwise, we find a x-factor "x" <1 such that:

$$x \in \left[\frac{\sum_{l=k+1}^{K} U_l(k)}{1-\sum_{l=1}^{k} U_l(l)}, \frac{1-\sum_{l=k+1}^{K} U_l(l)}{\sum_{l=1}^{k} U_l(l)}\right] \tag{3.5}$$

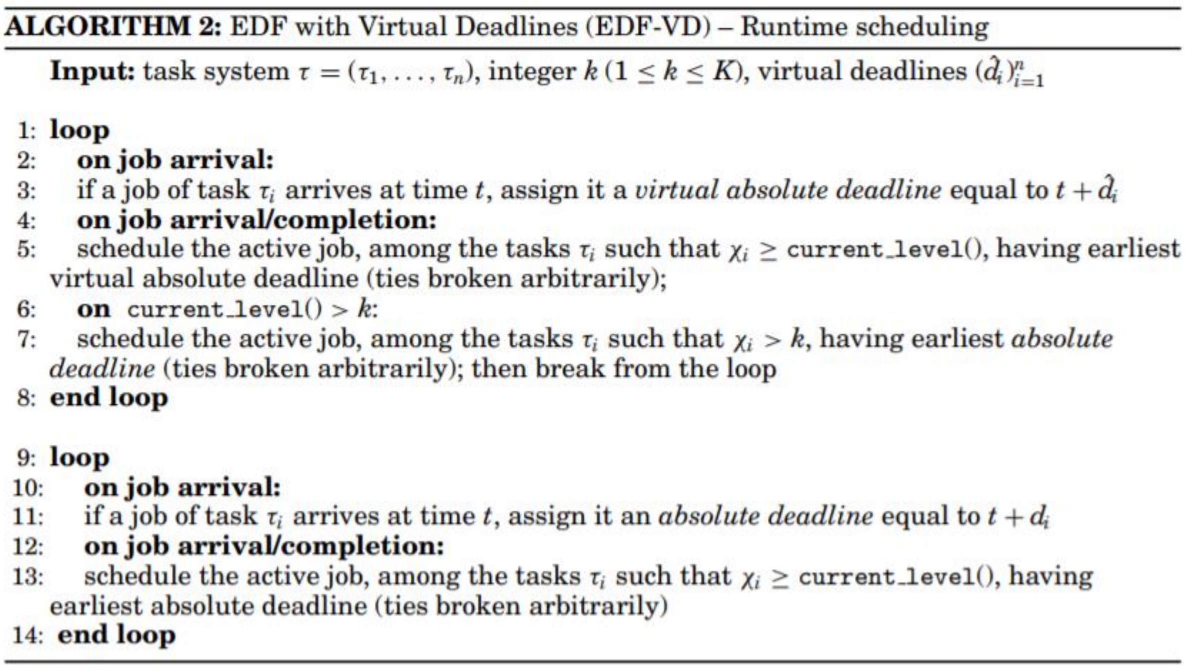
Then, the virtual deadlines of all tasks having criticality level 1 to k are set to be equal to original deadlines, as shown in line 14. The virtual deadlines of all tasks having criticality level k+1 to K are scaled by the x-factor.

3.4.2 Runtime scheduling phase

The pseudocode is described in figure 3.2. The function current_level() returns the level exhibited by the scenario so far, that is,

current_level() = $\min\{l \in [K]: \tilde{\gamma}_{ij} \leq c_i(l)$ for all jobs J_{ij} (partially) executed so far$\}$,

where $\tilde{\gamma}_{ij}$ is the part of γ_{ij} that has been observed thus far.



ALGORITHM 2: EDF with Virtual Deadlines (EDF-VD) – Runtime scheduling

```
Input: task system $\tau = (\tau_1, \ldots, \tau_n)$, integer $k$ $(1 \leq k \leq K)$, virtual deadlines $(\hat{d}_i)_{i=1}^n$

loop
    on job arrival:
    if a job of task $\tau_i$ arrives at time $t$, assign it a virtual absolute deadline equal to $t + \hat{d}_i$
    on job arrival/completion:
    schedule the active job, among the tasks $\tau_i$ such that $\chi_i \geq$ current_level(), having earliest virtual absolute deadline (ties broken arbitrarily);
    on current_level() $> k$:
    schedule the active job, among the tasks $\tau_i$ such that $\chi_i > k$, having earliest absolute deadline (ties broken arbitrarily); then break from the loop
end loop

loop
    on job arrival:
    if a job of task $\tau_i$ arrives at time $t$, assign it an absolute deadline equal to $t + d_i$
    on job arrival/completion:
    schedule the active job, among the tasks $\tau_i$ such that $\chi_i \geq$ current_level(), having earliest absolute deadline (ties broken arbitrarily)
end loop
```

Figure 3.2: Pseudocode for runtime scheduling phase in EDF-VD.

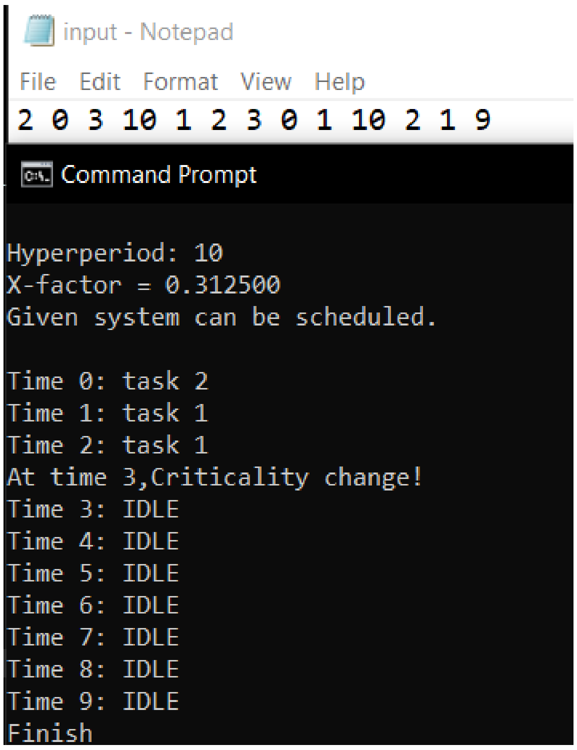
This algorithm uses k as an input and the virtual deadlines computed during preprocessing phase. As long as system is in level 1 to k, the tasks having level >= current_level() are scheduled according to EDF with respect to virtual deadlines. The second the system reaches level >k, all tasks with task criticality k or below are discarded, while all with tasks of criticality k+1 or higher are restored and executed with their original deadlines.

The proof of these algorithms are present in the paper.

3.4.3 Input and output of code

The C code is there in appendix-B. Task numbering starts from 1. Criticality starts from 1. Time granularity has been taken as 1. Input format: (with spaces between inputs)

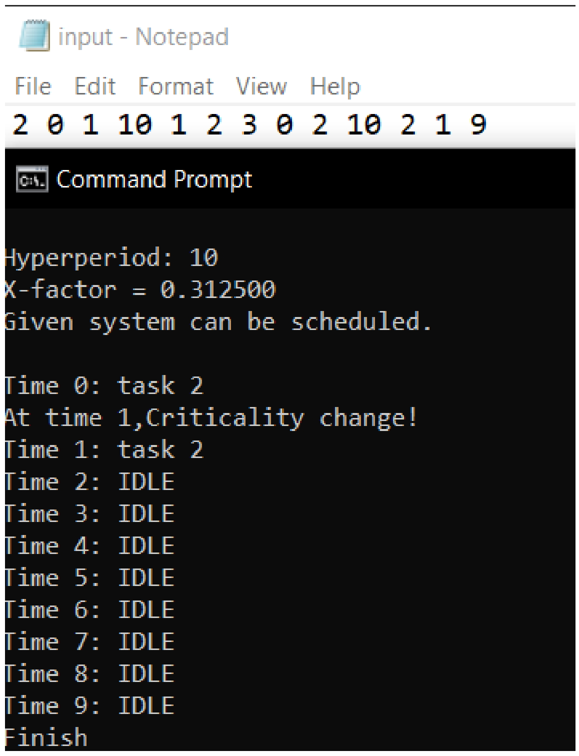
number_of_tasks task1_arrival task1_execution task1_deadline task1_criticalitylevel task1_wcetlevel1 task1_wcetlevel2….



```
2 0 3 10 1 2 3 0 1 10 2 1 9
```

```
Hyperperiod: 10
X-factor = 0.312500
Given system can be scheduled.

Time 0: task 2
Time 1: task 1
Time 2: task 1
At time 3,Criticality change!
Time 3: IDLE
Time 4: IDLE
Time 5: IDLE
Time 6: IDLE
Time 7: IDLE
Time 8: IDLE
Time 9: IDLE
Finish
```



```
2 0 1 10 1 2 3 0 2 10 2 1 9
```

```
Hyperperiod: 10
X-factor = 0.312500
Given system can be scheduled.

Time 0: task 2
At time 1,Criticality change!
Time 1: task 2
Time 2: IDLE
Time 3: IDLE
Time 4: IDLE
Time 5: IDLE
Time 6: IDLE
Time 7: IDLE
Time 8: IDLE
Time 9: IDLE
Finish
```

Figure 3.3: Different inputs and outputs in EDF-VD.

CONCLUSION

The field of mixed criticality systems, covering various aspects, was discussed. EDF-algorithm was implemented and EDF-VD for mixed criticality for a single processor was also discussed, and implemented for a dual-criticality system.

REFERENCES

1. Alan Burns and Robert I.Davis , *Mixed Criticality Systems - A Review*, twelfth edition, March 2019
2. S.K. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. *Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems*. Journal of the ACM (JACM), 62(2):14, 2015. 16
3. F. Dorin, P. Richard, M. Richard, and J. Goossens. *Schedulability and sensitivity analysis of multiple criticality tasks with fixed-priorities*. Real-Time Systems Journal, 46(3):305–331, 2010. 9
4. S.K. Baruah, A. Burns, and R. I. Davis. *Response-time analysis for mixed criticality systems*. In Proc. IEEE Real-Time Systems Symposium (RTSS), pages 34–43, 2011. 9, 10, 12, 22, 33, 34
5. N.C. Audsley. *On priority assignment in fixed priority scheduling*. Information Processing Letters, 79(1):39–44, 2001. 5, 9, 22
6. S.K. Baruah and A. Burns. *Fixed-priority scheduling of dual-criticality systems*. In Proc. 21st RTNS, pages 173–182. ACM, 2013. 14
7. T. Park and S Kim. *Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems.* In Proc. ACM EMSOFT, pages 253–262, 2011. 8, 15
8. S.K. Baruah, V. Bonifaci, G. D'Angelo, H. Li, A. Marchetti-Spaccamela, S. Van Der Ster, and L. Stougie. *The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task system.*
9. Felix Wermelinger. *Implementation and Evaluation of Mixed Criticality Scheduling Approaches.* February to June 2013.
10. Science direct, *Mixed Criticality in control systems.* https://www.sciencedirect.com/science/article/pii/S1474667016435664
11. A.Burns, *Is Audsley's Scheme the Most Expressive Optimal Priority Assignment Algorithm?* https://www.cs.york.ac.uk/rts/publications/R_Burns_2013e.html

FIGURES

APPENDIX-A

CODE FOR EDF SCHEDULING

Implemented for an implicit system.

Header file containing function declarations and data structures, “edf_functions.h”

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<stdbool.h> //bool operator needs this header file.

typedef struct
{
      int task_id;
      int task_release_time;
      int task_execution_time;  //given execution time
      int task_actual_deadline;
      int task_virtual_deadline;

} task;

typedef struct
{
      int job_id;
      int job_release_time; //absolute release time
      int job_execution_time; //remaining execution time
      int job_actual_deadline; //absolute deadline
      int job_virtual_deadline; //absolute deadline (kept here to
extend to edf-vd later)

}job;




struct job_queue_node
{
            job *job_node;
            struct job_queue_node *next;
};

typedef struct job_queue_node node;



//functions
int gcd(int a, int b); //used to find greatest common divisor
int lcm(int *a, int n); //used to find least common multiple
extern int hyperperiod_calc(task *t1,int n); //used to find hyperperiod
of tasks
int new_job_arrival(task *t1,int tmr,int n); //used to find next job
arrival
bool insert_job(job *add); //used to insert job at proper place in job
linked list
```

```
extern void runtime(int n); //runtime function
```

Function definitions, “edf_functions.c”

```
#include "edf_functions.h"
#include <stdio.h>

int timer=-1;
int hyper_period=0;
task* t=NULL; //dynamic array of task structures
job *jobs_to_add; //jobs to be added at next closest job arrival
int number_of_jobs=0; //number of jobs to be added
node *head = NULL;
node *tail = NULL;

int gcd(int a, int b)
{
      /*Function to calculate greatest common divisor. Used in lcm to
calculate hyperperiod*/
  if (b == 0)
        return a;
  else
  return gcd(b, a%b);
}

int lcm(int *a, int n)
{
      /*Function to calculate least common multiple. Used in
hyperperiod_calc*/
  int res = 1, i;
  for (i = 0; i < n; i++)
  {
    res = res*a[i]/gcd(res, a[i]);
  }
  return res;
}

int hyperperiod_calc(task *t1,int n)
{
      /*Function to calculate hyperperiod of tasks*/
      int i=0,ht,a[10];
      while(i<n)

            {
            a[i]=t1->task_actual_deadline;
            t1++;
            i++;
            }
      ht=lcm(a,n);
      return ht;
}

//A TASK RELEASES A JOB AT EVERY RELEASE_TIME + K*PERIOD
int new_job_arrival(task *t1,int tmr,int n)
{
      int i=0;
      int next_arrival=2*hyper_period;
      job *to_be_added = malloc(sizeof(job));
      job *iterate = to_be_added;
      jobs_to_add=to_be_added;
      number_of_jobs=0;
      while(i<n)
      {
            int job_arrival_of_task;
            int k;
            for(k=0;k<=hyper_period/(t1->task_actual_deadline);k++)
            {

                  if(timer< t1->task_release_time + k*t1-
>task_actual_deadline)
                  {
                        job_arrival_of_task=t1->task_release_time +
k*t1->task_actual_deadline;
                        goto cont;
                  }
            }
            printf("\nexpecting no job");
            return 0;

            cont:
            ;
            if(next_arrival>job_arrival_of_task)
            {
                  number_of_jobs=1;
                  //replace to_be_added with a job with parameters
job_arrival_of_task, t1->task_execution_time, (k+1)*t1-
>task_actual_deadline
                  free(to_be_added);
                  to_be_added = (job*) malloc((size_t)
number_of_jobs*sizeof(job));
                  to_be_added->job_id=t1->task_id;

                  to_be_added->job_release_time = job_arrival_of_task;
                  to_be_added->job_execution_time = t1-
>task_execution_time;
                  to_be_added->job_actual_deadline = (k+1)*t1-
>task_actual_deadline + t1->task_release_time;
                  to_be_added->job_virtual_deadline = k*t1-
>task_actual_deadline + t1->task_virtual_deadline + t1-
>task_release_time;
                  next_arrival=job_arrival_of_task;
            }
            else if(next_arrival==job_arrival_of_task)
            {
                  number_of_jobs++;
                  //add extra job to to_be_added with parameters
job_arrival_of_task, t1->task_execution_time, (k+1)*t1-
>task_actual_deadline
                  to_be_added = (job*) realloc(to_be_added,(size_t)
number_of_jobs*sizeof(job));
                  to_be_added++;
                  to_be_added->job_id=t1->task_id;

                  to_be_added->job_release_time = job_arrival_of_task;
```

```
                to_be_added->job_execution_time = t1-
>task_execution_time;
                to_be_added->job_actual_deadline = (k+1)*t1-
>task_actual_deadline+ t1->task_release_time;
                to_be_added->job_virtual_deadline = k*t1-
>task_actual_deadline + t1->task_virtual_deadline+ t1-
>task_release_time;

           }
           t1++;
           i++;
      }

      return next_arrival;
}


bool insert_job(job *add)
{
      node *node_to_add = malloc(sizeof(node));
      node_to_add->job_node = add;
      node_to_add->next = NULL;

      if(head==NULL) //empty
      {
           head = node_to_add;
           head->next = NULL;
           tail=head;
           return true;
      }
      else
      {

           node *temp = head;
           //starting condition. Equal to condition to not cause
context switch - job added second.
           if(add->job_virtual_deadline == (head->job_node)-
>job_virtual_deadline)
           {
                node_to_add->next = head->next;
                head->next = node_to_add;
                return true;
           }
           else if(add->job_virtual_deadline < (head->job_node)-
>job_virtual_deadline) //What if job to be added at the start?
           {

                node_to_add->next = head;
                head = node_to_add;
                return true;
           }
           //what if add at the end?
           else if(add->job_virtual_deadline >= (tail->job_node)-
>job_virtual_deadline)
           {
                tail->next = node_to_add;
                tail = node_to_add;
                tail->next =NULL;
```

```
                    return true;
              }
              else //to be added somewhere in the middle
              {
                    while(temp->next!=NULL)
                    {
                          if((add->job_virtual_deadline >= (temp-
>job_node)->job_virtual_deadline)&&(add->job_virtual_deadline <=
((temp->next)->job_node)->job_virtual_deadline))
                          {
                                node_to_add->next = temp->next;
                                temp->next = node_to_add;
                                return true;
                          }
                          temp=temp->next;
                    }
              }
        }
        return false;

}

void runtime(int n)
{

        int next_job_arrival = new_job_arrival(t,timer,n);

        while(timer< hyper_period)
          {

               //if job arrival:

               if(timer==next_job_arrival)
               {

                    job *temp = jobs_to_add;
                    int x=0;
                    while(x<number_of_jobs)
                    {
                          insert_job(temp);
                          x++;
                          temp++;
                    }

                    next_job_arrival = new_job_arrival(t,timer+1,n);

               }
              if((head==NULL)&&(timer>=0))
                     printf("\nTime %d: IDLE",timer);
              else if(timer>=0)
                          printf("\nTime %d: task %d", timer,(head-
>job_node)->job_id);
              timer++;
               if(head!=NULL)
               {
                    (head->job_node)->job_execution_time--;
```

```
                  if((head->job_node)->job_execution_time ==0) //if job
at head has finished execution
                  {
                        //remove head from queue
                        node* to_delete = head;
                        if(head->next==NULL)
                              head=NULL;
                        else
                              head=head->next;
                        free(to_delete);
                  }
             }
        }

}
```

Driver code, "edf_driver.c"

```
#include "edf_functions.h"

extern int hyper_period;
extern task *t; //dynamic array of task structures

int main(int argc, char *argv[])
{
      int n;
      FILE *fp;
      fp = fopen("input.txt","r");
      if(!fp)
            perror("fopen");

      fscanf(fp, "%d",&n);
      if(n==0) return 0;
      t=malloc(n*sizeof(task));
      int i=0;
      task *t1 = t;
      while(i<n)
      {
            fscanf(fp, "%d %d %d", &t1->task_release_time,&t1-
>task_execution_time,&t1->task_actual_deadline);
            t1->task_id=i;
            t1->task_virtual_deadline=t1->task_actual_deadline; //for
now, normal edf
            t1++;
            i++;
      }
      fclose(fp);

      hyper_period=hyperperiod_calc(t,n);
      printf("For the given input, Hyperperiod: %d\n",hyper_period);
      printf("\nSchedule for one hyperperiod:");
      runtime(n);

      printf("\nFinish.");
      return 0;
}
```

APPENDIX-B

CODE FOR EDF-VD SCHEDULING

Implemented for an implicit system. The number of criticality levels has been restricted to 2 for now.

Header file containing function declarations and data structures, “edfvd_functions.h”

```
#include<stdio.h>
#include<math.h>
#include<stdlib.h>
#include<stdbool.h>

typedef struct
{
      int task_id;
      int task_release_time;
      int task_execution_time;  //given execution time
      int task_actual_deadline; //used in level 2. HERE period =
task_actual_deadline
      float task_virtual_deadline; //used in level 1

      int task_criticality;
      int task_wcet_levels[2]; //STARTING FROM 0
} task;

typedef struct
{
      int job_id;
      int job_release_time; //absolute
      int job_execution_time; //remaining
      int job_actual_deadline; //absolute
      float job_virtual_deadline; //absolute

      int job_criticality;
      int job_wcet_levels[2];
      int job_execution_time_copy;
}job;

struct job_queue_node
{
            job *job_node;
            struct job_queue_node *next;
};

typedef struct job_queue_node node;

int gcd(int a, int b);
int lcm(int *a, int n);
extern int hyperperiod_calc(task *t1,int n);
float ulkcalculator(task *t1, int l, int k);
float ulksummer(task *t1, int llt, int hlt, int k);
float ullsummer(task *t1, int llt, int hlt);
```

```
bool schedulable_offline_preprocessing(task *t1);
int new_job_arrival(task *t1,int tmr,int n);
bool insert_job(job *add);
void criticality_change_taskparameter_change(task *t1);
bool criticality_change_job_removal();
extern void runtime(int n);
```

Function definitions, “edfvd_functions.c”

```
#include "edfvd_functions.h"
#include <stdio.h>


int timer=-1;
int hyper_period;
int number_of_tasks=0;
int no_levels=2;
task *t;
job *jobs_to_add;
int number_of_jobs=0; //number of jobs to be added


int system_level=0;
node *head = NULL;
node *tail = NULL;
;

bool criticality_change_flag=false;

int gcd(int a, int b)
{
  if (b == 0)
        return a;
  else
  return gcd(b, a%b);
}

int lcm(int *a, int n)
{
  int res = 1, i;
  for (i = 0; i < n; i++)
  {
    res = res*a[i]/gcd(res, a[i]);
  }
  return res;
}

int hyperperiod_calc(task *t1,int n)
{
      int i=0,ht,a[10];
      while(i<n)

            {
            a[i]=t1->task_actual_deadline;
            t1++;
            i++;
            }
      ht=lcm(a,n);
      return ht;
}



float ulkcalculator(task *t1, int l, int k)
{
      float ulk=0.0;
      int i=0;
      while(i<number_of_tasks)
      {
            if(t1->task_criticality == l)
            {

                  //printf("\ntask %d: %f",i,(float)(t1-
>task_wcet_levels[k]) / (float)(t1->task_actual_deadline));
                  ulk = ulk + (float)(t1->task_wcet_levels[k]) /
(float)(t1->task_actual_deadline);
            }
            t1++;
            i++;
      }
      return ulk;
}

float ulksummer(task *t1, int llt, int hlt, int k)
{
      float ulksum=0.0;

      int i=0;
      for(int l=llt;l<=hlt; l++)
                  ulksum=ulksum + ulkcalculator(t1,l,k);
      return ulksum;
}

float ullsummer(task *t1, int llt, int hlt)
{
      float ullsum=0.0;

      for(int l=llt;l<=hlt; l++)
            ullsum=ullsum + ulkcalculator(t1,l,l);

      return ullsum;
}

bool schedulable_offline_preprocessing(task *t1)
{
      //printf("\nt1->task_wcet_levels %d %d\n",t1-
>task_wcet_levels[0],t1->task_wcet_levels[1]);
      float x_factor=1;
      if(ullsummer(t1,0,no_levels-1) <=1.0)
      {
            int i=0;
                  while(i<number_of_tasks)
                  {
                        ;
                        t1->task_virtual_deadline = t1-
>task_actual_deadline;
```

```
                        t1++;
                        i++;
                  }
                  printf("\nx_factor = 1");
                  return true;
      }
      else
      {
            int k=0;
            for(k=0;k<no_levels-1;k++)
            {

                  //printf("\nk=%d: %f / %f      %f /
%f",k,ulksummer(t1,k+1,no_levels-1,k),(1-ullsummer(t1,0,k)),(1-
ullsummer(t1,k+1,no_levels-1)),ullsummer(t1,0,k));
                  float expr1 = ulksummer(t1,k+1,no_levels-1,k)/(1-
ullsummer(t1,0,k));
                  float expr2=(1-ullsummer(t1,k+1,no_levels-
1))/ullsummer(t1,0,k);
                  //printf("\n%f %f", expr1,expr2);
                  //printf("\nexpr1<=expr2?
%s",expr1<=expr2?"true":"false");
                  if ((1-ullsummer(t1,0,k) >0) &&(expr1<=expr2))
                  {

                        //printf("\nk is %d, expr1 is %f, expr2 is
%f",k,expr1,expr2);
                        x_factor=(float)(expr1+expr2)/(float)2.0;
                        printf("\nX-factor = %f",x_factor);
                        int i=0;
                        while(i<number_of_tasks)
                        {
                              if(t1->task_criticality <= k)
                                    t1->task_virtual_deadline =
(float)t1->task_actual_deadline;
                              else
                                    t1->task_virtual_deadline =
x_factor*(float)(t1->task_actual_deadline);

                              //printf("\ntask %d new vd = %f",i,t1-
>task_virtual_deadline);
                              i++;
                              t1++;
                        }
                        return true;

                  }
                  //printf("\nCondition not true for k = %d",k);
            }

      }

      return false;
}



int new_job_arrival(task *t1,int tmr,int n)
```

```
{
      int i=0;
      int next_arrival=2*hyper_period;
      job *to_be_added = malloc(sizeof(job));
      job *iterate = to_be_added;
      jobs_to_add=to_be_added;
      number_of_jobs=0;
      while(i<n)
      {
            int job_arrival_of_task;
            int k;
            for(k=0;k<=hyper_period/(t1->task_actual_deadline);k++)
            {

                  if((timer< t1->task_release_time + k*t1-
>task_actual_deadline)&&(t1->task_criticality>=system_level))
                  {
                        job_arrival_of_task=t1->task_release_time +
k*t1->task_actual_deadline;
                        goto cont;
                  }
            }
            //printf("\nexpecting no job");
            return 0;

            cont:
            ;
            if(next_arrival>job_arrival_of_task)
            {
                  number_of_jobs=1;
                  //replace to_be_added with a job with parameters
job_arrival_of_task, t1->task_execution_time, (k+1)*t1-
>task_actual_deadline
                  free(to_be_added);
                  to_be_added = (job*) malloc((size_t)
number_of_jobs*sizeof(job));
                  to_be_added->job_id=t1->task_id;

                  to_be_added->job_execution_time_copy = t1-
>task_execution_time;
                  to_be_added->job_wcet_levels[0] = t1-
>task_wcet_levels[0];
                  to_be_added->job_wcet_levels[1] = t1-
>task_wcet_levels[1];
                  to_be_added->job_criticality = t1->task_criticality;


                  to_be_added->job_release_time = job_arrival_of_task;
                  to_be_added->job_execution_time = t1-
>task_execution_time;
                  to_be_added->job_actual_deadline = (k+1)*t1-
>task_actual_deadline + t1->task_release_time;
                  to_be_added->job_virtual_deadline = (float)(k*t1-
>task_actual_deadline) + t1->task_virtual_deadline + (float)t1-
>task_release_time;
                  next_arrival=job_arrival_of_task;
            }
            else if(next_arrival==job_arrival_of_task)
            {
                  number_of_jobs++;
                  //add extra job to to_be_added with parameters
job_arrival_of_task, t1->task_execution_time, (k+1)*t1-
>task_actual_deadline
                  to_be_added = (job*) realloc(to_be_added,(size_t)
number_of_jobs*sizeof(job));
                  to_be_added++;
                  to_be_added->job_id=t1->task_id;

                  to_be_added->job_execution_time_copy = t1-
>task_execution_time;
                  to_be_added->job_wcet_levels[0] = t1-
>task_wcet_levels[0];
                  to_be_added->job_wcet_levels[1] = t1-
>task_wcet_levels[1];
                  to_be_added->job_criticality = t1->task_criticality;

                  to_be_added->job_release_time = job_arrival_of_task;
                  to_be_added->job_execution_time = t1-
>task_execution_time;
                  to_be_added->job_actual_deadline = (k+1)*t1-
>task_actual_deadline+ t1->task_release_time;
                  to_be_added->job_virtual_deadline = (float)(k*t1-
>task_actual_deadline) + t1->task_virtual_deadline+ (float)t1-
>task_release_time;

            }
            t1++;
            i++;
      }

      return next_arrival;
}

bool insert_job(job *add)
{
      node *node_to_add = malloc(sizeof(node));
      node_to_add->job_node = add;
      node_to_add->next = NULL;

      if(head==NULL) //empty
      {
            head = node_to_add;
            head->next = NULL;
            tail=head;
            //printf("\nJob %d: %d %d %d %f added to queue via
nothingness",add->job_id, add->job_release_time, add-
>job_execution_time, add->job_actual_deadline, add-
>job_virtual_deadline);
            return true;
      }
      else
      {

            node *temp = head;
            //starting condition. What if add at the start?
            if(add->job_virtual_deadline == (head->job_node)-
>job_virtual_deadline)
            {
                  node_to_add->next = head->next;
                  head->next = node_to_add;
                  //printf("\nJob %d: %d %d %d %f added to queue via
start equal",add->job_id,add->job_release_time, add-
>job_execution_time, add->job_actual_deadline, add-
>job_virtual_deadline);
                  return true;
            }
            else if(add->job_virtual_deadline < (head->job_node)-
>job_virtual_deadline)
            {

                  node_to_add->next = head;
                  head = node_to_add;
                  //printf("\nJob %d: %d %d %d %f added to queue via
start",add->job_id,add->job_release_time, add->job_execution_time, add-
>job_actual_deadline, add->job_virtual_deadline);
                  return true;
            }
            //what if add at the end?
            else if(add->job_virtual_deadline >= (tail->job_node)-
>job_virtual_deadline)
            {
                  tail->next = node_to_add;
                  tail = node_to_add;
                  tail->next =NULL;
                  //printf("\nJob %d: %d %d %d %f added to queue via
end",add->job_id,add->job_release_time, add->job_execution_time, add-
>job_actual_deadline, add->job_virtual_deadline);
                  return true;
            }
            else //somewhere in the middle
            {
                  while(temp->next!=NULL)
                  {
                        if((add->job_virtual_deadline >= (temp-
>job_node)->job_virtual_deadline)&&(add->job_virtual_deadline <=
((temp->next)->job_node)->job_virtual_deadline))
                        {
                              node_to_add->next = temp->next;
                              temp->next = node_to_add;
                              //printf("\nJob %d: %d %d %d %f added to
queue via middle",add->job_id,add->job_release_time, add-
>job_execution_time, add->job_actual_deadline, add-
>job_virtual_deadline);
                              return true;
                        }
                        temp=temp->next;
                  }
            }
      }
      return false;

}
```

```
void criticality_change_taskparameter_change(task *t1)
{
      int i=0;
      while(i<number_of_tasks)
      {
            t1->task_virtual_deadline = t1->task_actual_deadline;
            i++;
            t1++;
      }
}


bool criticality_change_job_removal()
{
      node *temp = head;
      node *temp_previous = NULL;
      if((head==tail)&&((head->job_node)->job_criticality
<system_level))
      {
            node* to_delete = head;
            head = NULL;
            tail = NULL;
            //printf("\nDeleted job %d: %d %d %d %f",(to_delete-
>job_node)->job_id,(to_delete->job_node)->job_release_time,(to_delete-
>job_node)->job_execution_time,(to_delete->job_node)-
>job_actual_deadline,(to_delete->job_node)->job_virtual_deadline);
            free(to_delete);
            return true;
      }
      if((head==tail)&&((head->job_node)->job_criticality
>=system_level))
            (head->job_node)->job_virtual_deadline = (head->job_node)-
>job_actual_deadline;

      while(temp->next!=NULL)
      {
            // printf("\nJob criticality: %d",(temp->job_node)-
>job_criticality);
            if((temp->job_node)->job_criticality <system_level) //1
            {
                  //if delete from head
                  if(temp==head)
                  {
                        node* to_delete = head;
                        //printf("\nDeleted job %d: %d %d %d
%f",(to_delete->job_node)->job_id,(to_delete->job_node)-
>job_release_time,(to_delete->job_node)->job_execution_time,(to_delete-
>job_node)->job_actual_deadline,(to_delete->job_node)-
>job_virtual_deadline);
                        if(head->next==NULL)
                              head=NULL;
                        else
                              head=head->next;
                        free(to_delete);
                  }
                  else
                  {
                        node *to_delete = temp;
                        //printf("\nDeleted job %d: %d %d %d
%f",(to_delete->job_node)->job_id,(to_delete->job_node)-
>job_release_time,(to_delete->job_node)->job_execution_time,(to_delete-
>job_node)->job_actual_deadline,(to_delete->job_node)-
>job_virtual_deadline);
                        temp_previous->next = temp->next;
                        free(to_delete);

                  }
            }
            else
            {
                  (temp->job_node)->job_virtual_deadline = (temp-
>job_node)->job_actual_deadline;
            }

            temp_previous = temp;
            temp = temp->next;

      }


      if((tail->job_node)->job_criticality <system_level)
                  {
                        node *to_delete = tail;
                        //printf("\nDeleted job %d: %d %d %d
%f",(to_delete->job_node)->job_id,(to_delete->job_node)-
>job_release_time,(to_delete->job_node)->job_execution_time,(to_delete-
>job_node)->job_actual_deadline,(to_delete->job_node)-
>job_virtual_deadline);
                        tail = temp_previous;
                        tail->next=NULL;
                        free(to_delete);
                  }
      else (tail->job_node)->job_virtual_deadline = (tail->job_node)-
>job_actual_deadline;
      return true;

}





void runtime(int n)
{

      int next_job_arrival = new_job_arrival(t,timer,n);

      while(timer< hyper_period)
        {

             //if job arrival:
             if(timer==next_job_arrival)
             {
                  // printf("\nat time %d",timer);
                  //add jobs from to_be_added into job queue
```

```
                    job *temp = jobs_to_add;
                    int x=0;
                    while(x<number_of_jobs)
                    {
                          insert_job(temp);
                          x++;
                          temp++;
                    }

                    next_job_arrival = new_job_arrival(t,timer+1,n);

               }
               if(head!=NULL)
               {
                    (head->job_node)->job_execution_time--;

                    if(((head->job_node)->job_execution_time_copy - (head-
>job_node)->job_execution_time > (head->job_node)-
>job_wcet_levels[system_level])&&(system_level==0))
                    {
                          criticality_change_flag=true;

                          printf("\nAt time %d,Criticality
change!",timer);
                          criticality_change_flag=true;
                          system_level++;
                          criticality_change_job_removal();
                          criticality_change_taskparameter_change(t);
                          //printf("\nremoval done!");

                    }
               }

              if((head==NULL)&&(timer>=0))
                     printf("\nTime %d: IDLE",timer);
              else if(timer>=0)
                    printf("\nTime %d: task %d", timer,(head->job_node)-
>job_id + 1);
              timer++;

                    if((head!=NULL)&&((head->job_node)->job_execution_time
==0))
                    {
                          //remove head from queue
                          //printf("\nat time %d",timer);
                          node* to_delete = head;
                          //printf("\nJob %d: %d %d %d %f removed from
queue",(head->job_node)->job_id,(head->job_node)->job_release_time,
(head->job_node)->job_execution_time, (head->job_node)-
>job_actual_deadline, (head->job_node)->job_virtual_deadline);
                          if(head->next==NULL)
                                head=NULL;
                          else
                                head=head->next;
                          free(to_delete);
                    }
```

```
        }

}
```

Driver code, “edfvd_driver.c”

```
#include "edfvd_functions.h"

extern int hyper_period;
extern task *t; //dynamic array of task structures
extern int number_of_tasks;

int main(int argc, char *argv[])
{

      int n;
      FILE *fp;
      fp = fopen("input.txt","r");
      if(!fp)
            perror("fopen");

      fscanf(fp, "%d",&n);
      if(n==0) return 0;
      number_of_tasks=n;
      t=malloc(n*sizeof(task));

      int i=0;
      task *t1 = t;
      while(i<n)
      {
            fscanf(fp, "%d %d %d %d %d %d ", &t1->task_release_time,&t1-
>task_execution_time,&t1->task_actual_deadline,&t1-
>task_criticality,&t1->task_wcet_levels[0],&t1->task_wcet_levels[1]);
            t1->task_criticality--;
            t1->task_id=i;
            t1->task_virtual_deadline=(float)(t1->task_actual_deadline);
            t1++;
            i++;
      }
      fclose(fp);


      hyper_period=hyperperiod_calc(t,n);
      printf("\nHyperperiod: %d",hyper_period);
      bool is_schedulable;
      is_schedulable = schedulable_offline_preprocessing(t);
      if(is_schedulable== false)
      {
            printf("\nGiven system cannot be scheduled!");
            return 0;
      }
      else printf("\nGiven system can be scheduled.\n");
      runtime(n);
      printf("\nFinish");
      return 0;
}
```