Time-triggered Scheduling Algorithms for Mixed-criticality Systems

Lalatendu Behera

Time-triggered Scheduling Algorithms for Mixed-criticality Systems

Thesis submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy

by

Lalatendu Behera

Under the supervision of

Prof. Purandar Bhaduri

Department of Computer Science and Engineering

Indian Institute of Technology Guwahati
Guwahati 781039, India

May 2019

ॐ

नीलाचलनिवासाय नित्याय परमात्मने।

बलभद्रसुभद्राभ्यां जगन्नाथाय ते नमः॥

जगदानन्दकन्दाय प्रणतार्तहराय च।

नीलाचलनिवासाय जगन्नाथाय ते नमः॥

Dedicated to my mother, brother and wife

Declaration

I, Lalatendu Behera, confirm that:

a. The work contained in this thesis is original and has been done by myself and the general supervision of my supervisor.

b. The work has not been submitted to any other Institute for any degree or diploma.

c. Whenever I have used materials (data, theoretical analysis, results) from other sources, I have given due credit to them by citing them in the text of the thesis and giving their details in the references.

d. Whenever I have quoted written materials from other sources, I have put them under quotation marks and given due credit to the sources by citing them and giving required details in the references.

Place: IIT Guwahati
Date:

Lalatendu Behera
Research Scholar
Department of Computer Science and Engineering,
Indian Institute of Technology Guwahati,
Assam-781039, India

Certificate

This is to certify that this thesis entitled **"Time-triggered Scheduling of Mixed-criticality Systems"** submitted by **Lalatendu Behera**, to the Indian Institute of Technology Guwahati, for partial fulfillment of the award of the degree of Doctor of Philosophy, is a record of bona fide research work carried out by him under my supervision and guidance.

The thesis, in my opinion, is worthy of consideration for award of the degree of Doctor of Philosophy in accordance with the regulations of the institute. To the best of my knowledge, the results embodied in the thesis have not been submitted to any other university or institute for the award of any other degree or diploma.

Place : IIT Guwahati, India
Date:

(Purandar Bhaduri)
Professor,
Dept. of Computer Science and Engineering,
Indian Institute of Technology, Guwahati

Acknowledgements

I wish to express my most sincere gratitude and appreciation to my supervisor, Prof. Purandar Bhaduri, for his support, help and guidance throughout the research. His continued support led me the right way to bring forth this thesis successfully. I would like to extend my appreciation to my doctoral committee members, Prof. Hemangee K. Kapoor, Dr. Arnab Sarkar and Dr. Aryabartta Sahu for providing constructive suggestions related to my work. I especially appreciate Dr. Sarkar's feedback in improving the quality of the thesis. My sincere appreciation also goes to Prof. Sanjoy Baruah for his invaluable suggestions that helped to fine-tune the thesis.

I wish to thank Prof. S. V. Rao, Head and other faculty members from the Department of Computer Science and Engineering for their support and help. I am grateful to a number of people of IIT Guwahati who, over the last few years, have helped me by providing valuable ideas and suggestions. They include Prof. Gautam Biswas (Director), Prof. Gautam Barua (former Director), Prof. Sukumar Nandi, Dr. Santosh Biswas and Dr. R. Inkulu. I also sincerely acknowledge the efforts devoted by all the teachers starting from my school days. I would also like to take this opportunity to thank all my friends, only to name a few, Shounak, Shirshendu, Ranajit, Amit, Mrityunjay, Basant, Ramanuj, Nandi, Abinash, Biswaranjan, Sawan, Rupak, Sandeep who directly and indirectly helped in finishing my thesis. I am thankful to Pradeep bhai and Badri bhai for their moral support and love which helped me to overcome all the tough situations in my life at IIT Guwahati.

Last but most important are my parents and other family members whose blessings and love made my path of success. I am grateful to my mother (Mrs. Premalata Behera), brother (Mr. Jajatendu Behera), wife (Mrs. Shubhashree Behera) and sister-in-law (Mrs. Rajashree Behera) who have always supported me at every course of my life and offered me constant encouragement and inspiration.

Abstract

Real-time and embedded systems are moving from the traditional design paradigm to integration of multiple functionalities onto a single computing platform. Some of the functionalities are safety-critical and subject to certification. The rest of the functionalities are non-safety critical and do not need to be certified. Designing efficient scheduling algorithms which can be used to meet the certification requirement is challenging because the requirements of both the system designers and certification authorities have to be met within given time budgets.

Our research considers the time-triggered approach to scheduling of mixed-criticality jobs with two criticality levels. In the first contribution, we propose an algorithm for uniprocessor mixed-criticality systems which directly constructs two scheduling tables for the two criticality levels without using a priority order. Furthermore, we show that our algorithm schedules a strict superset of instances which can be scheduled by two current approaches to the time-triggered scheduling of such systems – the OCBP-based algorithm as well as by MCEDF. We show that our algorithm outperforms both the OCBP-based algorithm and MCEDF in terms of the number of instances scheduled in a randomly generated set of instances. We generalize our algorithm for jobs with m-criticality levels. Subsequently, we extend our algorithm to find scheduling tables for periodic and dependent jobs.

Apart from schedulability, it is also important to consider some of the non-functional properties of mixed-criticality systems, for example, energy consumption. In this work, we propose a time-triggered dynamic voltage and frequency scaling (DVFS) algorithm for uniprocessor mixed-criticality systems and show that our algorithm outperforms the predominant existing algorithms which use DVFS for such systems with respect to minimization of energy consumption. We prove an optimality result for the proposed algorithm with respect to energy consumption. Then we extend our algorithm for tasks with dependency constraints.

Finally, we propose a time-triggered scheduling algorithm for both independent and dependent mixed-criticality jobs on an identical multiprocessor platform. We show that our algorithm is more efficient than the Mixed-criticality Priority Improvement (MCPI) algorithm, the only existing such algorithm for a multiprocessor platform, although the set of instances scheduled by the two algorithms are identical.

Contents

List of Figures

List of Tables

Nomenclature

AMC	Adaptive Mixed-criticality
ASIL	Automotive Safety and Integrity Levels
CA	Certification Authority
CBEDF	Criticality-based Earliest Deadline First
CPS	Cyber-physical Systems
DMS	Deadline Monotonic Scheduling
DVFS	Dynamic Voltage and Frequency Scaling
EDF	Earliest Deadline First
FAA	Federal Aviation Authority
MCEDF	Mixed-criticality Earliest Deadline First
MCPI	Mixed-criticality Priority Improvement
MCRTS	Mixed-criticality Real-time Systems
MCS	Mixed-criticality Systems
OCBP	Own Criticality Based Priority
PC	Partitioned Criticality
RMS	Rate Monotonic Scheduling
SD	System Designer
SIL	Safety Integrity Levels
SMC	Static Mixed-criticality
TT	Time-triggered
UAV	Unmanned Aerial Vehicle
WCET	Worst-case Execution Time

Chapter 1

Introduction

In recent times there has been a rapid increase in the use of real-time and embedded systems in day-to-day life. A real-time system is required to produce not only correct results but also produce them within the stipulated time. Typical applications of real-time systems are in the field of defense and space systems, networked multimedia systems, embedded automotive and avionics systems etc. Many of these real-time systems are safety-critical in nature.

The correctness requirements of safety-critical systems must be met at any cost. To meet this goal designers perform a priori verification and also try to ensure run-time robustness [Bar18] of the systems. In the verification process, formal models are constructed to assess the run-time behaviors of the systems. On the other hand, run-time robustness checks that any behavior not included in the system specification must not appear at run-time. One of the most important features to verify by modeling is the execution time of a piece of code. In traditional safety-critical real-time systems, a piece of code was verified using simple processors and deterministic timing properties, leading to predictable run-time behavior during the verification processes. However, safety-critical systems have become more complicated with increasing size and complexity. Also, they are being implemented on advanced processors which are much less predictable with respect to the run-time behavior of such systems. Upon such platforms, the timing behavior of a piece of code varies considerably during different runs. The longest time taken by a piece of code in different runs is called the worst-case execution time (WCET).

In real-time systems, we assume that no task will exceed its computed worst-case execution time (WCET) in order to ensure that every task meets its deadline. This assumption may not always hold. In practice, it is very difficult to predict the WCET

of a task [PB00], a fact summarized by the observation that "the more confidence one needs in a task execution time bound, the larger and more conservative that bound tends to be in practice" [Ves07]. The increasing complexity of safety-critical system functionalities and the enforced non-determinism due to the complex architecture of today's platforms, determination of WCET becomes very challenging. WCETS are estimated by some methodologies and tools which compute the upper bound for a functionality. Due to the non-determinism of the platform and the complexity of the codes, there can be a significant difference between the WCETs computed by two different tools for the same functionality.

In real-time systems, satisfying the timing specifications for a given set of tasks by determining an appropriate order among task executions boils down to a challenging scheduling problem. Traditional scheduling schemes have primarily dealt with scenarios in which all tasks belong to a single criticality level. In these systems, tasks at distinct criticality levels are typically handled by allocating a dedicated server for each criticality level. However, such federated schemes often lead to severe resource under-utilization. As a result, real-time systems are moving towards integrating various functionalities onto a single platform such that the under-utilization of the system resources can be alleviated. The Integrated Modular Avionics (IMA) [Pri92] initiative for aerospace and AUTomotive Open System ARchitecture (AUTOSAR) [SB08, FBH+06, FMB+09] for the automotive industry are examples of frameworks for integrated functionalities. The task of making timing guarantees for such systems is very challenging.

In various applications, the severity of missing a deadline varies from task to task [Ves07]. For example, DO-178B [RB92], a software development process standard (*Software Considerations in Airborne Systems and Equipment Certification*), is accepted by the United States Federal Aviation Authority (FAA) to certify the software used in the avionics industry. DO-178B assigns five different criticality levels to a task based on the use of commercial aircraft. The criticality levels are presented in Table 1.1. A failure of assurance level E can cause a sub-optimal behavior of the system, while one of assurance level A can damage the system.

Safety functionalities are more difficult to verify. A major practice in some industries is to increase the use of software to increase the safety of the system [Bow00]. But this leads to the added burden of verifying the reliability of the software. A defect in software may impact the timing aspects of a real-time system. Safety functionalities require certification by *certification authorities* (CAs). The tools used by the certification authorities for estimating

Table 1.1: Criticality levels of DO-178B standard [BV08]

Level	Failure Condition	Interpretation
E	No Effect	Failure has no impact on the safety of an aircraft
D	Minor	Failure is noticeable, but has less impact on the safety of an aircraft
C	Major	Failure is significant but not hazardous
B	Hazardous	Failure has very large impact on the safety of the aircraft
A	Catastrophic	Total system failure and crash

WCETs of tasks are very pessimistic although they are concerned only about the safety-critical functionalities. On the other hand, the *system designers* (SDs) use less pessimistic tools for WCET estimation but are concerned about all functionalities. For example, a UAV (Unmanned Aerial Vehicle) consists of two functionalities, viz., safe operation of the vehicle above the ground (safety-critical) and capturing an image and sending to the base station (mission-critical). In this case, the certification authorities assess the reliability of only the safety-critical functionality whereas the system designers need to worry about both functionalities, albeit with relaxed assumptions about WCETs. This is the crux of the scheduling problem for mixed-criticality systems.

1.1 Overview of Mixed-criticality Real-time Systems

The main goal of studying mixed-criticality real-time systems is to build *safety-critical cyber-physical systems* (CPS) in a resource-efficient manner. As mentioned above, safety-critical systems must satisfy some correctness constraints and in many cases, for example, for aircrafts and automotive vehicles, need to be certified by some certification authority. In order to verify the correctness of the system, it must be guaranteed that the run-time behavior of the system is *predictable*. For example, an anti-lock braking system has multiple forms of predictability requirements, viz., functional predictability and timing predictability. Functional predictability deals with the physical components of the anti-lock braking system, whereas timing predictability deals with the time elapsed between the triggered event and the actual action. Both requirements are vital with respect to the correctness of the system.

Here we mainly deal with cyber-physical systems which are *reactive*, i.e., the system regularly interacts with the *environment* or the *physical world*. There are mainly three components of a cyber-physical system, *viz.*, *program*, *platform* and *environment*. We

write some *programs* which are executed on a given *platform* and which interact with the *environment*. The resulting behavior of these three components defines the functionality of a CPS. To achieve the timing predictability of a CPS, we must follow these guidelines, viz., (1) the programs must behave in a deterministic fashion during run-time, (2) resource under-utilization must be avoided and (3) the behavior of the platform must be deterministic. Apart from these, the environment component is also necessary for timing predictability. Since we cannot predict the behavior of the environment in general, we cannot represent the environment exactly. Hence, if any aspect of a CPS is event-triggered (i.e., triggered by an event in the environment) then such a model has to be conservative and must incorporate pessimism by taking into account a range of possible environment behaviors. From the above discussion, we can say that as cyber-physical systems become larger and more complex, they become computationally more demanding. The fact that CPS are inherently interactive results in pessimism and conservative assumptions about timing guarantees. The degree of pessimism also depends on the criticality level of the task. This leads us into the realm of mixed-criticality systems.

A *mixed-criticality real-time system* (MCRTS) [BBD+12a, BBB+09, LB10, BD13, BB11, Ves07, BV08] is one that has two or more distinct levels of criticality, such as, safety-critical, mission-critical, non-critical, etc. Typical names of the criticality levels used in industries are ASIL (Automotive Safety and Integrity Levels) [fS11, aHCJPH11, HSK+09] and SIL (Safety Integrity Level) [Com10, Gal08], etc. All the run-time behaviors of a CPS are not equally important. Therefore the important behaviors must be validated to a greater level of assurance or higher degrees of confidence. That means to validate or certify a very critical property, we need to make very conservative assumptions. We introduce the mixed-criticality scheduling problem with an example [BBD+12a] from the domain of UAVs. The functionalities of UAVs may be classified into two categories, viz., *mission-critical* (LO-criticality) and *flight-critical* (HI-criticality).

- **Mission-critical** functionalities include capturing images from the ground and transmitting those to the base station, etc.
- **Flight-critical** functionalities include safe operation while performing the mission.

It is mandatory that the flight-critical functionalities be certified to be correct because failure of these functionalities could be catastrophic. There are different certification authorities (CAs) for different functionalities. The CAs for flight-critical jobs tend to be

very conservative. During the certification process, the CAs focus mainly on the run-time behavior of the systems. The analytical tools, techniques and methodologies used by the CAs estimate more pessimistic results than the system designers. System designers are interested in both flight-critical and mission-critical functionalities, but are not as rigorous as the CAs when estimating timing parameters. As mentioned above, the computation of the exact worst-case execution time (WCET) of a non-trivial piece of code is extremely difficult due to the complex architecture of today's systems. A safe upper bound on the actual WCET requires great effort. A CA may estimate the WCET of a piece of code to be far higher than that of the system designer. This leads to two different WCET estimates, i.e., one by the CA which is very pessimistic and the other one by the system designer which is much lower. The gaps between the CAs and the system designers are more likely to increase in future as pointed out in [HS06]. On the other hand, it is very unlikely that a system would realize the higher WCET estimated by the CA. As a result, most of the resources which are provided to run the piece of code go unused if the pessimistic estimates are adhered to.

Example 1.1.1: Consider the instance given in Table 1.2 which consists of three jobs.

Table 1.2: An example instance to explain MCS

Job	Arrival time	Deadline	Criticality	WCET estimated by SDs	WCET estimated by CAs
j_1	0	3	LO	1	1
j_2	0	6	HI	3	4
j_3	1	5	HI	1	2

The given instance is EDF schedulable from the perspective of system designers, as shown in Fig. 1.1. The CAs use very conservative tools and their estimates are given in the last column of Table 1.2. When we consider the WCET estimated by the CAs, we find that EDF fails to schedule the instance as is shown in Fig. 1.2. The symbol $\bigotimes$ indicates a deadline miss.

But we know that CAs do not care about the schedulability of the LO-criticality jobs. So they can certify the system only if the HI-criticality jobs are schedulable. We can verify that the HI-criticality jobs are schedulable if j_2 is scheduled in $[0, 1]$ and $[3, 6]$ and j_3 is scheduled in $[1, 3]$ which is shown in Fig. 1.3. The schedule is obtained by the *criticality-monotonic scheduling* with EDF policy, where a higher criticality job with earliest deadline

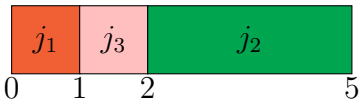


Figure 1.1: EDF Schedulable according to the SDs

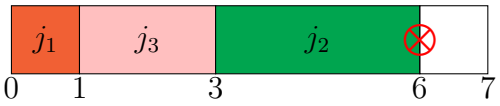


Figure 1.2: EDF fails to schedule according to the CAs

is scheduled first. We can see that j_1 misses its deadline but this fact is not important to the CAs. But the schedule thus obtained does not satisfy the system designers even with the lower WCET estimates. Suppose both j_2 and j_3 runs for 3 and 1 units of execution time as estimated by the system designers. We can see in Fig. 1.4 that j_1 has missed its deadline which can lead to mission failure even if it is not catastrophic.

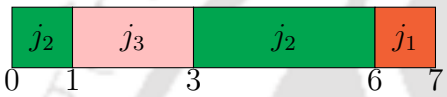


Figure 1.3: Scheduling of HI-criticality jobs according to the CAs

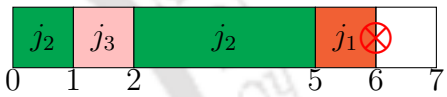


Figure 1.4: Criticality-monotonic fails to schedule according to the CAs

So the main problem in scheduling mixed-criticality jobs/tasks is to satisfy both the CAs and SDs. Hence we need to construct deterministic algorithms where both the CAs and SDs can certify the system correct with respect to their perspectives. We now present a schedule in Fig. 1.5 which can satisfy both the CAs and SDs. In Fig. 1.5., if job j_3 finishes

Figure 1.5: A schedule which satisfies both the CAs and SDs

its execution at time instant 2, then j_2 can be assigned in $[2, 3]$. We then need to schedule the 3 remaining units of execution time of j_2 in $[3, 6]$ which is possible.

1.2 Time-triggered Scheduling

In a *time-triggered* (TT) system, all activities in a distributed real-time system are initiated by the progression of time [Kop11] and the information about the run-time of each task is known at design time. All activities are scheduled at predefined points in time even before the system is deployed. The scheduler dispatches the jobs using this prior information. In such

architectures, it is essential to synchronize the actions of all participating nodes to a global clock. A time-triggered schedule is deterministic [BF11] and hence is very easy to verify and certify. With the advantage of timing predictability, there are also some disadvantages. Once a time-triggered schedule is prepared, it cannot be changed during run-time. If a job finishes its execution in less time than the actual worst-case execution time, then the allotted system resources for the job cannot be reallocated at run-time. Hence, under-utilization of the system resources is a serious drawback of time-triggered scheduling. Since our work is motivated by verification and certification, we focus on the time-triggered paradigm. There are many variations of the paradigm, such as, *slot shifting*, *mode change*, etc. In the slot shifting paradigm [Foh95, IF00], the time-triggered schedule is partially computed off-line. Then some additional scheduling decision is taken depending upon the run-time events. On the other hand, there are more than one pre-computed scheduling strategies available in the mode change paradigm [Foh93]. The decision to change from one scheduling strategy to another is triggered by the occurrence of a run-time event. These scheduling strategies are designed in such a way that the ongoing activities run uninterruptedly. Our main focus is in the mode change paradigm. We prepare more than one scheduling table for a task set and switch between the tables to assert the correctness depending upon the requirement.

1.3 Outline of the Thesis

Traditional real-time scheduling algorithms are not able to schedule mixed-criticality task sets efficiently as shown in [Ves07]. According to Vestal's model, the high criticality tasks must be guaranteed to meet their deadlines if estimates of CAs are to be considered and all tasks must be guaranteed to meet their deadlines if estimates of SDs are to be considered. Our thesis has the goal of finding time-triggered schedules for mixed-criticality task sets considering Vestal's model.

1.3.1 Time-triggered Scheduling of Uniprocessor Mixed-criticality Systems

In this thesis, we focus on the **time-triggered scheduling** of mixed-criticality systems. We will construct two predefined scheduling tables $\mathcal{S}_{\text{HI}}$ and $\mathcal{S}_{\text{LO}}$ for a given instance I which will be used at run time. MCEDF [SPBB13] and OCBP [BF11] are two existing fixed-priority

job scheduling algorithms for mixed-criticality systems, which are used to construct time-triggered scheduling tables. Both the MCEDF and OCBP algorithms fail to schedule some instances due to a fixed priority assignment to the jobs. If the algorithms do not find a priority order then they will not be able to construct the time-triggered scheduling tables. We propose an algorithm called TT-Merge which can directly construct the time-triggered scheduling tables without using priorities. We prove that the TT-Merge algorithm schedules a strict superset of instances scheduled by the OCBP-based time-triggered scheduling algorithm and MCEDF. We first propose the TT-Merge algorithm for independent mixed-criticality jobs. We then generalize the problem for m-criticality levels. Then TT-Merge is extended for dependent mixed-criticality jobs (abbreviated as DP-TT-Merge), periodic jobs and synchronous reactive systems. Finally, we present results of experiments based on randomly generated job instances.

1.3.2 Energy-efficient Time-triggered Scheduling of Uniprocessor Mixed-criticality Systems

Huang et al. [HKGT14] proposed an energy-efficient algorithm for scheduling mixed-criticality systems based on the EDF-VD algorithm. They showed the algorithm to be optimal in finding processor frequencies for each job with respect to EDF-VD. We show that our algorithm TT-Merge from Section 1.3.1 schedules more instances (i.e., a strict superset of instances) than the EDF-VD algorithm. We also prove that TT-Merge will find more energy-efficient schedules than the existing algorithm. Here we focus on dual-criticality task sets. Since the probability of a scenario change from the LO-criticality scenario to HI-criticality is very low, minimizing energy consumption in HI-criticality scenarios is of less importance. Our objective is to minimize the system energy consumption by slowing down the tasks in the LO-criticality scenario using the DVFS method while ensuring that they do not miss their deadlines. Without loss of generality, we calculate the energy consumption minimization up to the hyperperiod P of the task set. The idea is to find the energy-efficient LO-criticality WCET $\tilde{C}_{ik}(\text{LO})$ and the corresponding frequency f_{ik}^{LO} for each job j_{ik} of the task set in the hyperperiod which will minimize the energy consumption in the LO-criticality scenario. We also prove the processor frequencies computed by our proposed algorithm are optimal with respect to the TT-Merge algorithm.

Then we extend the algorithm to find the energy-efficient LO-criticality WCET $\tilde{C}_{ik}(\text{LO})$

and the corresponding frequency f_{ik}^{LO} for dependent jobs such that the DP-TT-Merge algorithm can schedule them without any deadline miss. Finally, we compare the results with the existing work [HKGT14, ASK15] using random task sets.

1.3.3 Time-triggered Scheduling of Multiprocessor Mixed-criticality Systems

There exists only one time-triggered mixed-criticality scheduling algorithm for multiprocessor systems by Socci et al. [SPBB15]. They showed that the computational complexity of their algorithm MCPI is $O(|E|n^2 + mn^3 \log n)$ for dependent jobs, where n is the number of jobs in the instance I, m is the number of processors and E is the dependency relation between jobs. In this thesis, we propose an algorithm which is easier to understand and schedules exactly the instances that are schedulable by MCPI. We also prove that the computational complexity of our algorithm is $O(|E|n+mn^3)$ for dependent jobs and $O(mn^3)$ for independent jobs.

1.4 Organization of the Thesis

The thesis is organized as follows.

- **Chapter 2:** We present a brief introduction to traditional real-time systems followed by mixed-criticality real-time systems. We then discuss different existing scheduling algorithms for mixed-criticality real-time systems.
- **Chapter 3:** We propose a time-triggered scheduling algorithm for non-recurrent independent mixed-criticality jobs on uniprocessor systems and then extend it to handle m-criticality levels, dependent jobs, periodic tasks and synchronous reactive tasks.
- **Chapter 4:** We propose a time-triggered energy-efficient scheduling algorithm and show it to be optimal with respect to the TT-Merge algorithm.
- **Chapter 5:** We propose a time-triggered scheduling algorithm for independent non-recurrent mixed-criticality jobs on multiprocessor systems which is extended for dependent non-recurrent mixed-criticality jobs.
- **Chapter 6:** We summarize the work done in this thesis and the scope for future work.

Chapter 2

Background and Related Work

In this chapter, we present the basic notations and definitions used in this thesis which include the background and a survey of *mixed-criticality systems*. We begin the chapter with the description of a generic task model. We then describe the classification of scheduling algorithms and the associated results. Finally, we present a brief introduction of mixed-criticality systems and work done in this area.

2.1 Real-time Task Model

A *real-time task* is one that has to perform some amount of computation within a specified time limit. Based on the occurrence of real-time tasks, they are classified into three classes, i.e., *periodic*, *sporadic* and *aperiodic*. The tasks which recur at a constant interval of time are called *periodic* tasks [LL73]. On the other hand, a task is called *sporadic* [Liu00] when it arrives at arbitrary times but the minimum inter-arrival time between any two consecutive tasks is fixed. An *aperiodic* task [Liu00] can arrive at any time. All these tasks are called *recurrent tasks*. A real-time task can recur indefinitely and generates a sequence of jobs. Each instance of a task arrival is called a *job*. If after one instance a task does not recur then it is called a *non-recurrent task*. The non-recurrent task sets are called an *instance* or a *job set*. In the case of periodic tasks, the arrival time of instances is known due to the constant inter-arrival time. This is also called as a time-triggered task [The15]. In this thesis, we handle both recurrent and non-recurrent tasks. In the case of recurrent tasks, we use only periodic tasks, where the arrival time of each job is known off-line. Note that if our scheduling algorithms work on a job set, that means this is a single instance release of the

task set and no other instance of the task set is available.

We now present a periodic task model. A periodic task (τ_i) is characterized by a 3-parameter tuple $\tau_i = (C_i, P_i, D_i)$, where

- $C_i \in \mathbb{N}^+$ denotes the worst-case *computation time* or *execution time*.
- $P_i \in \mathbb{N}^+$ denotes the *period*, i.e., the inter-arrival time between two tasks.
- $D_i \in \mathbb{N}^+$ denotes the *relative deadline*, i.e., the maximum time permissible for the task to complete its execution.

Based on the relation between deadlines and periods, task sets have three types of constraints on deadlines, i.e., *implicit deadline*, *constrained deadline* and *arbitrary deadline*. A task set is said to have implicit deadline constraints, if all task periods are equal to their deadlines, i.e., $P_i = D_i$. If $D_i \leq P_i$, then the task set has constrained deadlines. All other constraints are known as arbitrary deadline constraints. In some cases, periodic tasks are assigned an initial *arrival time* or *offset*. This is the delay in the arrival of the task. If the offsets of all the tasks are same, then the task set is called *synchronous*. In this thesis, we consider the offset to be zero, unless stated explicitly.

In some cases, our work is based on jobs. A task can generate an infinite sequence of jobs. The job or an instance j_{ik} of task τ_i, i.e., the k^{th} job of the i^{th} task is characterized by a 3-parameter tuple $j_{ik} = (a_{ik}, C_{ik}, D_{ik})$, where

- $a_{ik} \in \mathbb{N}$ denotes the *arrival time*, i.e., the release time of the job.
- $C_{ik} \in \mathbb{N}^+$ denotes the worst-case *computation time* or *execution time* of the job.
- $D_{ik} \in \mathbb{N}^+$ denotes the *relative deadline*, i.e., the maximum time permissible for the job to complete its execution.

Since our algorithms are restricted to non-recurrent jobs and periodic tasks, we do not focus on sporadic and aperiodic tasks here. In real-time systems, jobs or tasks need to complete their execution with not just *correct results*, but also *on time*. There are various algorithms which can schedule jobs such that all the tasks complete their execution correctly on time, i.e., before their deadlines. We next present some definitions regarding the schedulability of a task or job set .

Definition 2.1.1: A scheduling algorithm which schedules the tasks or jobs and/or allocates resources to the tasks or jobs is called a *scheduler* [Liu00]. An assignment of all the jobs in the system to the available processors is called a *schedule*.

Definition 2.1.2: If a schedule works correctly, then we call it a *valid schedule* [Liu00]. In a valid schedule, all the jobs must complete their execution before their deadlines, no job is assigned to a processor before its arrival time, all jobs must satisfy their resource allocation and precedence constraints.

Definition 2.1.3: A task set is said to be *feasible* [Liu00] in a system if there exists an algorithm (scheduler) which can correctly schedule all the jobs generated by a task set before their deadlines.

Definition 2.1.4: An algorithm is said to be *optimal* [DB11] with respect to a system and task model, if it can correctly schedule all task sets that are feasible.

Definition 2.1.5: A scheduling algorithm is said to be *clairvoyant* [DB11] if the algorithm has prior knowledge about the events of the tasks, (such as, arrival time or actual execution time) which is unknown until run-time.

2.1.1 Overview of Real-time Scheduling

Here we discuss the basics of real-time task scheduling. Generally, scheduling algorithms are divided into two categories:

- Dynamic-priority scheduling
- Fixed-priority scheduling

2.1.2 Dynamic-priority Scheduling

In dynamic priority scheduling, the tasks are assigned priorities at every scheduling instant. In Earliest Deadline First (EDF) scheduling [LL73], the task having the shortest time to its deadline is assigned the highest priority. EDF scheduling is very intuitive and proven to be an optimal [Hor74] uniprocessor scheduling algorithm. The schedulability test for EDF is

$$\sum_{i=1}^{n} \frac{c_i}{p_i} = \sum_{i=1}^{n} u_i \leq 1 \tag{2.1}$$

where c_i is the worst case execution time and p_i is the period of the i^{th} task, n is the number of tasks to be scheduled and u_i is the CPU utilization due to the i^{th} task.

2.1.3 Fixed-priority Scheduling

In fixed priority scheduling, the tasks are assigned priorities before execution. There are two well-known fixed priority algorithms in wide use, viz., Rate Monotonic Scheduling (RMS) [LL73] and Deadline monotonic scheduling (DMS) [ABW93].

Rate Monotonic scheduling (RMS) [LL73]: This algorithm assigns priorities to tasks based on their periods, i.e., the shorter the period, the higher the priority. We know that the rate of a task is the inverse of its period. Hence, higher the rate, higher its priority. The following conditions are important criteria which can be used to check the schedulability of a task set under RMS [Liu00].

1. **Necessary Condition:** A set of periodic real-time tasks would not be RMS schedulable unless they satisfy the following necessary condition:

$$\sum_{i=1}^{n} \frac{c_i}{p_i} = \sum_{i=1}^{n} u_i \leq 1 \tag{2.2}$$

2. **Sufficient Condition: [LL73]** A set of n real-time periodic tasks are schedulable under RMS, if

$$\sum_{i=1}^{n} u_i \leq n(2^{\frac{1}{n}} - 1) \tag{2.3}$$

Deadline Monotonic scheduling (DMS) [ABW93]: This algorithm is similar in concept to rate monotonic algorithm. Here the tasks are assigned priorities according to their deadline. The task with the shortest deadline gets the highest priority and the one with the longest deadline with the lowest priority.

The following are important criteria which can be used to check the schedulability of a task set under DMS

$$\forall i : 1 \leq i \leq n : \frac{C_i}{D_i} + \frac{I_i}{D_i} \leq 1 \tag{2.4}$$

where $I_i = \sum_{j=1}^{i-1} \left\lceil \frac{D_i}{P_j} \right\rceil \times C_j$, P_j is period of the j^{th} task, C_i is execution time of the i^{th} task, D_i is deadline of the i^{th} task and n is the number of tasks in the task set.

2.1.4 Real-time Multiprocessor Scheduling

The scheduling algorithms discussed in the previous sections are based on uniprocessor real-time systems. There are multiple algorithms proposed for scheduling jobs in a multiprocessor

Table 2.1: Different types of real-time multiprocessor algorithms

Job	Dynamic	Job fixed	Task fixed
Global	Fluid Schedule Pfair	Global EDF	Global RM
Partitioned	Partitioned EDF	Partitioned EDF	Partitioned RM

real-time system. As in the uniprocessor case, the scheduling algorithms are divided into different categories as given in Table 2.1.

Here we mainly focus on the *homogeneous multiprocessor* systems or multiprocessor with identical processors. In the above table, when we move from left to right, the priority assignments by the schemes move from unrestricted to restricted with respect to a task set. In a *dynamic scheduling policy* the priorities are given at each instant of time. So there is no restriction on a task in the beginning with respect to priority. On the other hand, each job is given a fixed priority in its life time in the *job fixed policy*. In *task fixed policy*, each task is given a single priority in its life time prior to their execution which cannot be changed later. A *global scheduling policy* means a job can be executed in any processor after a preemption, i.e., an inter-processor migration for jobs is allowed. On the other hand, a *partitioned scheme* assigns a set of jobs to a particular processor and those jobs will be executed only in their assigned processors from beginning to end.

In 1997, Phillips et al. [PSTW97, PSTW02] investigated the *global EDF* scheduling approach (which is a job fixed policy) to schedule sporadic tasks in multiprocessor systems. A given sporadic task set is called global EDF schedulable if EDF meets all deadlines for every collection of jobs that may be generated by the task system. There were no exact feasibility tests for sporadic tasks in this work. Srinivasan and Baruah [SB02], Goossens et al. [GFB03] and Baruah [Bar04a] extended this work to find a sufficient condition. In 2008, Baruah and Baker [BB08] proposed a sufficient schedulability test for global EDF. Baruah et al. [BG03] proposed a global rate-monotonic scheduling algorithm for multiprocessor real-time systems where a task is assigned the highest priority if it has the lowest period. In 2006, Cho et al. [CRJ06] proposed an optimal scheduling algorithm for multiprocessor systems known as LLREF based on the *fluid scheduling model* and a notion of fairness. The notion of fairness was introduced by Baruah et al. [BCPV96] in 1996. They proposed the proportionate progress (*P-fairness*) notion which is used to solve the periodic scheduling problems in real-

time systems. Anderson et al. [AS00] proposed a variant of fair scheduling known as *ER-fair schedule* or early-release fair scheduling which is work conservative and optimally schedules periodic tasks on a multiprocessor system. Levin et al. [LFS+10] proposed an algorithm based on deadline partitioning which guarantees optimality and improves performance over all other algorithms in multiprocessor real-time systems.

On the other hand, the *partitioned EDF* algorithm [Bar13, LDG04, BCA08] is known to be *intractable*. But many polynomial time algorithms have been proposed to solve the problem in *approximation*.

2.2 Mixed-criticality System Model

In this section, we present a general mixed-criticality periodic task model. A mixed-criticality task is characterized by a 4-tuple of parameters: $\tau_i = (a_i, p_i, \chi_i, C_i)$, where

- $a_i \in \mathbb{N}$ denotes the *arrival time* of the first job of task τ_i (also known as the offset).
- $p_i \in \mathbb{N}^+$ denotes the *period*.
- $\chi_i \in \mathbb{N}^+$ denotes the *criticality* level.
- $C_i \in \mathbb{N}^L$ is a vector, where the l^{th} coordinate denotes the *worst-case execution time* ($C_i(l)$) of l^{th} criticality level of task τ_i. We represent the worst-case execution times of a task for the L criticality levels as the tuple $(C_i(1), C_i(2), \ldots, C_i(L))$.

A task τ_i generates an unbounded sequence of mixed-criticality jobs released at intervals of time p_i whose relative deadlines are after p_i units of time after release. A job j_k of task τ_i is characterized by a 4-tuple parameters: $j_k = (a_k, d_k, \chi_i, C_k)$, where a_k and d_k are arrival time and deadline of job j_k, respectively. We assume that the worst-case execution time estimates are monotonically increasing with the criticality levels, i.e., $C_k(1) \leq C_k(2) \leq \cdots \leq C_k(L)$ for each job j_k. This is because the execution time estimates represent upper bounds for the respective criticality levels and we know that higher criticality levels correspond to more conservative estimates. A task set $\mathscr{T}$ is a collection of n tasks. For simplicity, we assume that each task can generate only one job between its arrival and its next period. In case of non-recurrent tasks, an *instance* is a set of n jobs.

To understand the mixed-criticality scheduling problem, we need to define the notion of a *scenario* [BBD+12a]. We know that a job j_k arrives at its arrival time a_k and must

finish its execution before its deadline d_k. The execution times defined above are worst-case estimates. The actual execution time $c_i(l)$ is not known until a job signals its completion. The collection of all such actual execution times of tasks/jobs is called a *scenario*, i.e., a tuple $(c_1, c_2, \ldots, c_n)$, where c_k is the actual execution time of job j_k. A scenario can be categorized into L types depending upon the criticality levels.

Definition 2.2.1: An l-criticality *scenario* [BBD+12a] is defined as the smallest integer l, such that $\forall j_k \in \mathscr{T}.c_k \leq C_k(l)$

Again a *scenario* is divided into two types, i.e., *erroneous* and *non-erroneous*. If a smallest l cannot be found according to Def. 2.2.1, then the scenario is called erroneous, else it is called non-erroneous.

Definition 2.2.2: [BBD+12a] A schedule for a scenario $(c_1, c_2, \ldots, c_n)$ of criticality l is feasible if every job j_k with $\chi_k \geq l$ receives execution time c_k during its time window $[a_k, d_k]$.

2.3 Related Work

In 2007, Vestal identified and formalized the mixed-criticality concept in his seminal work [Ves07]. He established the necessity of conservative worst-case execution time parameters in safety-critical systems with respect to fixed-priority preemptive uniprocessor scheduling of recurrent task systems. Baruah and Vestal [BV08] presented a thorough study of feasibility and schedulability for multi-criticality real-time systems using Audsley's algorithm [Aud01] when implemented upon preemptive uniprocessor platforms. They showed that the earliest deadline first (EDF) algorithm [LL73] does not outperform fixed-priority schemes in the presence of criticality levels. They also showed that some feasible systems are not schedulable by EDF.

Burns and Baruah [BB11] proposed three schedulability algorithms based on the response time analysis of the task set, i.e., *Partitioned Criticality* (PC), *Static Mixed-Criticality* (SMC) and *Adaptive Mixed-Criticality* (AMC). In PC, the priorities are assigned according to criticality, whereas SMC and AMC use run-time monitoring to assign the priorities. They proved that the proposed algorithms dominate the existing fixed-priority algorithms for traditional real-time systems. These algorithms are applicable to periodic tasks.

Baruah et al. [BBD+12a,BBD+10] proved the MC-schedulability problem to be NP-hard

in the strong sense even for two criticality levels and for identical arrival times. The strong NP-hardness of MC-schedulability indicates that neither polynomial nor pseudo-polynomial time algorithms are likely to exist to exactly decide whether there is a scheduling policy for a mixed-criticality job instance or task set [Li13].

So most schedulability research on mixed-criticality systems revolves around finding efficient approximation algorithms. In [BBD+12a, BBD+10, BLS10a, BLS10b], Baruah et al. proposed the *own-criticality based priority* (OCBP) algorithm. OCBP is a job based fixed priority algorithm which is described in Chapter 3 in more detail. They also proved that the speedup factor of OCBP is optimal. In 2010, Baruah et al. [BLS10b] enhanced the OCBP algorithm for sporadic tasks by obtaining the initial priority list and updating the list on-line to maintain the correctness of the priority. Park and Kim [PK11a] proposed the *Criticality Based Earliest Deadline First* (CBEDF) algorithm which is an extension of the traditional EDF algorithm. Baruah et al. [BBD+12b, BBD+11, BBD+15] proposed an algorithm like the traditional EDF algorithm based on the sporadic task model known as *Earliest Deadline First - Virtual Deadline* (EDF-VD). The *speedup factor* of an algorithm A is the smallest real number α such that any task system τ that is schedulable on a unit-speed processor by a hypothetical optimal clairvoyant algorithm (where a clairvoyant algorithm for scheduling mixed-criticality systems is one that knows prior to run-time whether the system is going to exhibit LO-criticality or HI-criticality scenario) is successfully scheduled on a speed-α processor by algorithm A. The smaller the speedup factor, the closer the behavior of the algorithm A to that of a clairvoyant optimal algorithm. The speedup factor of EDF-VD was given as 1.618 in [BBD+11] whereas an improved analysis in [BBD+12b] showed it to be 1.33. We describe the algorithm in detail in Chapter 4.

In 2011, Baruah and Fohler [BF11] introduced a technique to schedule dual-criticality mixed-criticality jobs using the *time-triggered* framework. Their objective was to ensure that adequate resources are reserved for each application to be able to guarantee the timing requirements. They used the OCBP algorithm to assign priorities to the jobs. These priorities are used to find two scheduling tables which are used at the run-time to schedule the jobs. They also showed that the speedup factor of the algorithm is 1.62.

Subsequently, Baruah [Bar14], [Bar12] proposed a schedule-generation algorithm for mixed-criticality synchronous programs upon uniprocessor platforms. He proved that the proposed algorithm for single-rate synchronous programs is optimal. He then proved that an efficient and optimal schedule generation problem for multi-rate synchronous program is

NP-hard in the strong sense. He also proposed a schedule generation algorithm based on OCBP for multi-rate synchronous programs. We describe the algorithm in detail in Chapter 3.

In 2013, Socci et al. [SPBB13] proposed a fixed priority scheduling approach called *Mixed-Criticality Earliest Deadline First* (MCEDF) for mixed-criticality jobs. In this paper, they assigned a priority to each job and then constructed two priority tables, i.e., PT_{LO} and PT_{HI}. The scheduling of jobs starts with the table PT_{LO}, while the table PT_{HI} is used after a mode change occurs. We discuss this algorithm in more detail in Chapter 3. In [TFB13] Theis et al. present a backtracking based iterative deepening algorithm for the generation of the scheduling tables.

Zhao et al. [ZGZ14, ZGYZ16] proposed a new algorithm for scheduling of mixed-criticality jobs with resource sharing. They extended the traditional resource sharing scheduling algorithms such as PIP (*Priority Inheritance Protocol*) and PCP (*Priority Ceiling Protocol*) for mixed-criticality systems. There are many properties of a mixed-criticality system which should be taken care of other than functional properties. Huang et al. [HKGT14] introduced the energy consumption problem for mixed-criticality systems. They focused on *dynamic voltage and frequency scaling* (DVFS) to reduce the energy consumption. They constructed an algorithm which computes a LO-criticality processor frequency for all the LO-criticality tasks and two processor frequencies for all the HI-criticality tasks, i.e., one for LO-criticality execution time and one for HI-criticality execution time after mode change. They showed that if the mixed-criticality jobs run with the calculated processor frequencies, then EDF-VD consumes optimum energy. In 2016 [NHG+16], Narayana et al. extended it to consider both static and dynamic power consumption. The proposed method in [NHG+16] is based on a more generalized system model to reduce energy consumption in multicore mixed-criticality systems.

Many other papers discuss the energy efficient problem in mixed-criticality systems, but using different platforms, models and other considerations. In 2015, Ali et al. [ASK15] proposed an algorithm based on DVFS to reduce energy consumption in mixed-criticality real-time systems. They claimed that their algorithm dominates the work of Huang et al. [HKGT14] based on their experiments. Vincent et al. [LJP13a] proposed a method to find an appropriate trade-off between the number of missed deadlines and their energy consumption. This method is not based on DVFS. Asyaban et al. [AKTM16a] discussed the energy uncertainty scheduling problem of a battery-less mixed-criticality systems which is

not based on DVFS.

All the mixed-criticality scheduling algorithms discussed above are based on uniprocessor mixed-criticality systems. In 2011, Baruah et al. [BBD+11] proposed an OCBP like algorithm for multiprocessor systems consisting of m identical machines. They found the speedup bound for a partitioned scheduling approach for the dual-criticality case. In the process, they proved the existence of a polynomial-time approximation scheme (PTAS) that partitions dual-criticality sporadic tasks to multiprocessors with the EDF-VD scheduler. In 2012, Pathan [Pat12] proposed methods to schedule mixed-criticality sporadic tasks based on both global and fixed-priority schemes. He also derived a sufficient schedulability test based on the response time analysis for the proposed algorithm. Li et al. [LB12] proposed a global mixed-criticality scheduling algorithm for multiprocessor systems. This algorithm is the same as the EDF-VD algorithm and uses the FpEDF algorithm [Bar04b, BCLS14a] to check the schedulability of HI-criticality tasks. They found the speedup bound of the algorithm to be no larger than $\sqrt{5}+1$. In [BCLS14a] Baruah et al. proposed a partitioned algorithm for mixed-criticality multiprocessor systems. They compared both the approaches and concluded that the partitioned algorithm outperforms the global scheduling algorithm. Socci et al. [SPBB15] proposed the Mixed-criticality Priority Improvement (MCPI) algorithm to find the priorities for jobs with precedence constraints. The priority order is used to construct time-triggered scheduling tables. They showed the worst-case time complexity of the algorithm to be $O(mn^3 \log n)$ for independent jobs and $O(|E|n^2 + mn^3 \log n)$ for dependent jobs, where n is the number of jobs in the instance I, m is the number of processors and E depicts the dependency between jobs.

Giannopoulou et al. [GSHT13b] proposed a flexible time-triggered criticality-monotonic scheduling scheme to schedule tasks with shared-resources in multi-core mixed-criticality systems. They combined the scheduling strategy with a mapping optimization technique to achieve better resource utilization. Apart from resource sharing, there exists some work on energy efficient scheduling policies on multiprocessor mixed-criticality systems. Awan et al. [AMT15a] proposed a task allocation method in a heterogeneous multiprocessor mixed-criticality platform which is energy efficient but not based on DVFS.

Chapter 3

Time-triggered Scheduling of Uniprocessor Mixed-criticality Systems

3.1 Introduction

Time-triggered scheduling is an off-line scheduling strategy. The scheduling activities in a time-triggered paradigm are triggered by the progression of time. and are predetermined at each time instant for each job. Generally, this precalculated schedule is kept in a table format. The scheduler takes the scheduling decisions according to this precalculated scheduling table. Generally, scheduling tables are generated off-line which are used to dispatch jobs on-line. Hence we need m tables for m criticality levels of a mixed-criticality system. Here we present a time-triggered scheduling algorithm for mixed-criticality jobs that is an improvement over the ones proposed by Baruah and Fohler [BF11] and Socci et al [SPBB13] by showing that it can schedule a superset of instances that can be scheduled by their algorithms.

The rest of the chapter is organized as follows. In Section 3.2, we introduce the system model used for this chapter. We then introduce the work done related to the time-triggered scheduling of mixed-criticality jobs for uniprocessor systems. In Section 3.2.2, we present our proposed algorithm (TT-Merge) for independent jobs. We then extend the TT-Merge algorithm for m-criticality level instances in Section 3.4. In Section 3.5, we proposed the TT-Merge-DEP algorithm for dependent jobs, an extension of the algorithm for independent jobs. In Section 3.6 and 3.7, we extend our algorithm for periodic jobs and synchronous

reactive systems. Finally, we discuss the results from our experiments and conclude the chapter in Sections 3.8 and 3.9, respectively.

3.2 System Model

The mixed-criticality model used in this chapter is based on non-recurrent tasks with at most two levels of criticality, LO and HI. A job is characterized by a 5-tuple of parameters: $j_i = (a_i, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}))$, where

- $a_i \in \mathbb{N}$ denotes the *arrival time*.
- $d_i \in \mathbb{N}^+$ denotes the *absolute deadline*.
- $\chi_i \in \{LO, HI\}$ denotes the *criticality* level.
- $C_i(\text{LO}) \in \mathbb{N}^+$ denotes the LO-criticality *worst-case execution time*.
- $C_i(\text{HI}) \in \mathbb{N}^+$ denotes the HI-criticality *worst-case execution time*.

We assume that the system is *preemptive* and $C_i(\text{LO}) \leq C_i(\text{HI})$ for $1 \leq i \leq n$. Note that in this chapter, we consider arbitrary arrival times of jobs.

An ***instance*** of mixed-criticality (MC) [BBD+12a] job set can be defined as a finite collection of MC jobs, i.e., $I = \{j_1, j_2, ..., j_n\}$. The job j_i in the instance I is available for execution at time a_i and should finish its execution before d_i. The job j_i must execute for c_i amount of time which is the actual execution time between a_i and d_i, but this can be known only at the time of execution. The scenarios in this model can be of two types, i.e., *LO-criticality scenarios* and *HI-criticality scenarios*. When each job j_i in instance I executes c_i units of time and signals completion before its $C_i(\text{LO})$ execution time, it is called a LO-criticality scenario. If any job j_i in instance I executes c_i units of time and does not signal its completion after it completes the $C_i(\text{LO})$ execution time, then such a scenario is called a HI-criticality scenario.

Each mixed-criticality instance needs to be scheduled by a scheduling strategy where both kinds of scenarios (LO and HI) can be scheduled. If we have prior knowledge about the scenario, then the scheduling strategy is known as a *clairvoyant scheduling strategy*. If we do not have prior knowledge about the scenario, then the scheduling strategy is called an *online scheduling strategy*. Here we assume that if any job continues its execution without

signaling its completion at $C_i(\text{LO})$ then no LO-criticality jobs are required to complete by their deadlines. Now we define the notion of MC-schedulability.

Definition 3.2.1: An instance I is MC-schedulable if it admits a correct online scheduling policy.

Here we focus on the **time-triggered schedules** [BF11] of MC instances. We will construct two tables $\mathcal{S}_{\text{HI}}$ and $\mathcal{S}_{\text{LO}}$ for a given instance I for use at run time. The length of the tables is the length of the interval $[\min_{j_i \in I}\{a_i\}, \max_{j_i \in I}\{d_i\}]$. The rules to use the tables $\mathcal{S}_{\text{HI}}$ and $\mathcal{S}_{\text{LO}}$ at run-time, (i.e., the *scheduler*) are as follows:

- The criticality level indicator Γ is initialized to LO.
- While (Γ = LO), at each time instant t the job available at time t in the table $\mathcal{S}_{\text{LO}}$ will execute.
- If a job executes for more than its LO-criticality WCET without signaling completion, then Γ is changed to HI.
- While (Γ = HI), at each time instant t the job available at time t in the table $\mathcal{S}_{\text{HI}}$ will execute.

Definition 3.2.2: A dual-criticality MC instance I is said to be **time-triggered schedulable** [BF11] if it is possible to construct the two schedules $\mathcal{S}_{\text{HI}}$ and $\mathcal{S}_{\text{LO}}$ for I, such that the run-time scheduler algorithm described above schedules I in a correct manner.

3.2.1 Related Work

Baruah *et al* [BBD+12a] proposed a priority-based scheduling technique known as OCBP (Own Criticality Based Priority) scheduling for mixed-criticality jobs. The OCBP algorithm chooses a job j_i and assigns it the lowest priority if there is at least $C_i(\chi_i)$ time units available between its arrival time and its deadline when every other job j_k is executed with higher priority than j_i for $C_k(\chi_i)$ time units. The authors proved that an instance I is *OCBP-schedulable* on a given processor, if and only if all the jobs in I is assigned a priority by the OCBP algorithm.

Baruah and Fohler [BF11] introduced a technique to schedule MC jobs using the *time-triggered* framework. Their objective was to ensure that adequate resources are reserved for each application to be able to guarantee the timing requirements. They used the OCBP

algorithm to assign priorities to the jobs. Using this priority, they constructed two tables S^{oc}_{LO} and S^{oc}_{HI} which are used by the dispatch algorithm [BF11] to schedule the jobs. We show in Section 3.3 that our algorithm can schedule a strict superset of instances schedulable by the OCBP-based algorithm. In Section 3.8 we quantify the number of instances scheduled by the two algorithms on a set of randomly generated instances and show that our algorithm has better performance.

Socci et al [SPBB13] proposed a fixed priority scheduling approach called MCEDF for mixed-criticality jobs. In this paper, they construct two priority tables, i.e., PT_{LO} and PT_{HI}. The scheduling of jobs starts with the table PT_{LO}, while the table PT_{HI} is used after a mode change occurs. The authors proved that an instance I is *MCEDF-schedulable* on a given processor, if and only if all the jobs in I is assigned a priority by the MCEDF algorithm. In Section 3.8 we quantify the number of instances scheduled by MCEDF and our algorithm and show that the latter performs better.

In [TFB13] Theis et al present a backtracking based iterative deepening algorithm for the generation of the scheduling tables. We were not able to compare this algorithm with ours because of the absence of implementation details.

Baruah [Bar14], [Bar12] proposed a schedule-generation algorithm for mixed-criticality synchronous programs upon uniprocessor platforms. He proved that proposed algorithm for single-rate synchronous programs is optimal. He then proved that an efficient and optimal schedule generation problem for multi-rate synchronous program is NP-hard in the strong sense. He also proposed a schedule generation algorithm based on OCBP for multi-rate synchronous programs. In Section 3.7, we show that our algorithm can schedule a strict superset of instances of this OCBP-based algorithm.

3.2.2 Our Work

In this section, we present an algorithm which can schedule not only the instances which are schedulable by the OCBP-based algorithm [BF11] and MCEDF algorithm [SPBB13] but additional ones as well. We then generalize the algorithm to the m criticality case. Subsequently we extend the algorithm to construct scheduling tables for periodic and dependent jobs.

Example 3.2.1: Consider the MC instance of 6 jobs given in Table 3.1.

The above MC instance is not OCBP schedulable because we will not be able to assign

Table 3.1: Example instance scheduled by TT-Merge and not by the OCBP-based algorithm or MCEDF

Job	Arrival time	Deadline	Criticality	C_i(LO)	C_i(HI)
j_1	0	14	HI	1	8
j_2	0	3	LO	1	1
j_3	0	8	LO	2	2
j_4	0	8	LO	2	2
j_5	8	13	HI	2	3
j_6	0	12	HI	2	3

a priority order as shown below.

- If j_1 is assigned the lowest priority, then j_2, j_3, j_4 and j_6 could consume 8 units of time (i.e., C_2(HI) + C_3(HI) + C_4(HI) + C_6(LO)) over [0, 8) as j_1 is a HI-criticality job. In the interval $[8, 11)$, j_5 execute its C_5(HI) units of execution, thus leaving no time for j_1 to execute its C_1(HI) before its deadline.

- If j_2 is assigned the lowest priority, then j_1, j_3, j_4 and $j_6$could consume 7 units of time (i.e., C_1(LO) + C_3(LO) + C_4(LO) + C_4(LO)) over [0, 7). This leaves no time for j_2 to execute its C_2(LO) time to finish by its deadline.

- If j_3 is assigned the lowest priority, then j_1, j_2, j_4 and j_6 could consume 6 units of time (i.e., C_1(LO) + C_2(LO) + C_4(LO) + C_6(LO)) over [0, 6). Job j_5 execute its C_5(LO) units of execution over [8, 11), thus leaving two units of space over [6, 8) for j_3 to execute its C_3(LO) units of execution before its deadline. So, j_3 can be assigned the lowest priority.

- Similarly, job j_4 can also be assigned as the lowest priority jobs among $\{j_1, j_2, j_4, j_5, j_6\}$ after removing job j_3.

Next, we remove the job j_4 and consider $\{j_1, j_2, j_5, j_6\}$ and try to assign the next lowest priority.

- If j_1 is assigned the lowest priority, then j_2 and j_6 could consume 4 units of time (i.e., C_2(HI) + C_6(HI)) over [0, 4) and j_5 could consume 3 units of C_2(HI) execution time

over [8, 11), thus leaving 7 units of time for j_1 to execute its C_1(HI) units of execution before its deadline which is not possible.

- If j_2 is assigned the lowest priority, then j_1 and j_6 could consume 3 units of time (i.e., C_1(LO) + C_6(LO)) over [0, 3), thus leaving no time for j_2 to execute its C_2(LO) units of execution before its deadline which is not possible.

- If j_5 is assigned the lowest priority, then j_1, j_2 and j_6 could consume 12 units of time (i.e., C_1(HI) + C_2(HI) + C_6(HI)) over [0, 12), thus leaving 1 unit of time for j_5 to execute its C_5(HI) units of execution before its deadline which is not possible.

- If j_6 is assigned the lowest priority, then j_1, j_2 and j_5 could consume 12 units of time (i.e., C_1(HI) + C_2(HI) + C_5(HI)) over [0, 12), thus leaving no time for j_6 to execute its C_6(HI) units of execution before its deadline which is not possible.

Since, no other job can be assigned the lowest priority, we declare the MC instance is not OCBP-schedulable. So due to the unavailability of an OCBP order, we cannot construct a time-triggered schedule. □

Now we try to schedule the same instance with the MCEDF algorithm [SPBB13], [SPBB15]. We find the two priority tables PT_{LO} and PT_{HI} and check the schedulability. According to the MCEDF algorithm, if the instance is schedulable in the LO scenario, then it generates a priority tree. The nodes of the priority tree are sorted using topological sort [CLRS09]. The table PT_{LO} is constructed from the order generated by the topological sort. The table PT_{HI} is nothing but a simple EDF order of HI-criticality jobs. The algorithm checks for each possible HI scenario failure. If it does not get any HI scenario failure, then the algorithm declares success, otherwise it declares failure.

The EDF order of the above instance given in Table 3.1 is $(2, 3, 4, 6, 5, 1)$. The MCEDF algorithm generates the priority tree shown in Fig. 3.1. The instance is schedulable in LO scenario. The instance has one busy interval, i.e., $[0, 10]$. This means the lowest priority job of this interval will be the root of the priority tree. In this busy interval, j_{LO}^{Late} is job j_4 and j_{HI}^{Late} is job j_1. Clearly, j_1 is chosen to be the lowest priority job as the deadline of j_4 is less than 10. Next, j_1 is removed which splits the busy interval into two, i.e., $[0, 7]$ and $[8, 10]$. Job j_5 is the single job in the busy interval $[8, 10]$. So it can be assigned as one of the children of the root, i.e., job j_5 is removed from the interval. In the busy interval $[0, 7]$, j_{LO}^{Late} is job j_4 and j_{HI}^{Late} is job j_6. Here the MCEDF algorithm chooses job j_4 as one of the lowest

priority job as its deadline is greater than 7. After removal of j_4, the busy interval splits into two intervals, i.e., $[0, 3]$ and $[5, 7]$. Now the priority tree generation steps are trivial. The resulting priority tree is given in Fig. 3.1.

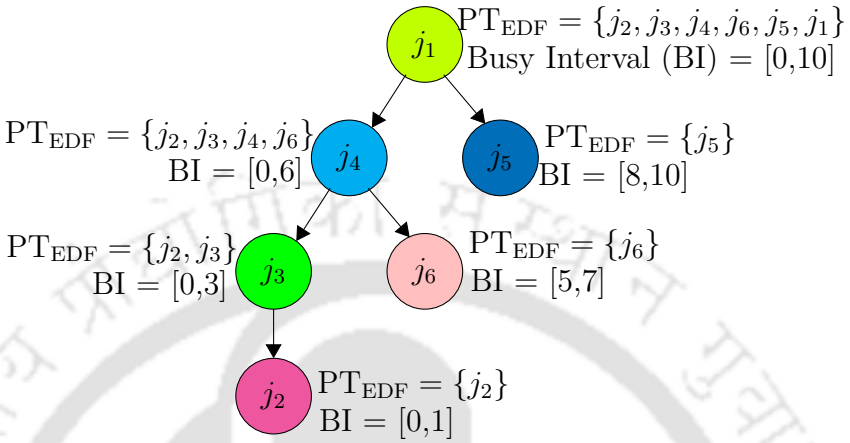


Figure 3.1: Priority tree of the instance given in Table 3.1

Now the MCEDF algorithm uses topological sort to find a priority order of the instance which in this case could be chosen to be $\{j_2, j_3, j_6, j_4, j_5, j_1\}$. The table PT_{LO} according to the priority order is given in Fig. 3.2.

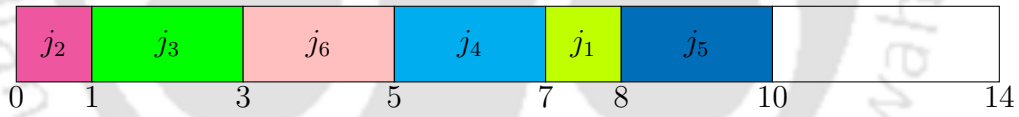


Figure 3.2: Table PT_{LO} of the instance given in Table 3.1

Then the MCEDF algorithm checks all possible HI-criticality scenarios for a deadline miss. When the job j_6 at time instant 5 does not signal its completion, there must be sufficient time for $1, 3$ and 8 units of execution for jobs j_6, j_5 and j_1 respectively before time instant 14. But, we have only 9 units of time left to complete these 12 units of execution. So, MCEDF cannot schedule the given instance.

We propose an algorithm called *TT-Merge* which can construct a time-triggered schedule for this instance and is an improvement over OCBP in terms of the set of instances that can be scheduled. We show through experiments that the number of instances schedulable by our algorithm exceeds those schedulable by OCBP and MCEDF by a significant amount on randomly generated instances. We describe the algorithm in detail in the next section. The two scheduling tables generated by our algorithm for the instance

in Table 3.1 are shown in Fig. 3.3. □

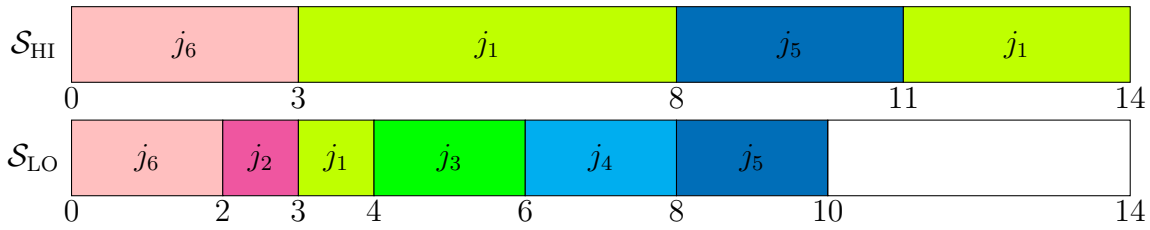


Figure 3.3: Tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$ constructed by our algorithm for the instance given in Table 3.1

3.3 The Proposed Algorithm: TT-Merge

From Section 3.2.2, it is clear that both the MCEDF and OCBP algorithms fail to schedule some instances due to a fixed priority assignment to the jobs. Both these algorithms construct the scheduling tables from the priority order of the jobs. That means, if the algorithms do not find a priority order then they will not be able to construct the scheduling tables. We propose an algorithm which can directly construct the scheduling tables without using priorities. We also focus on scheduling more number of instances than the OCBP and MCEDF algorithms. The main insight behind our algorithm is as follows.

- We want to find a time-triggered schedule not based on a priority order.
- We want to find the exact time to run a job in a scheduling table by merging two tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ containing jobs of the two different criticality levels.
- The LO-criticality execution time of HI-criticality jobs must be completed at a time instant t such that there is sufficient time to complete the remaining execution before its deadline.
- We want to construct the table $\mathcal{S}_{\text{LO}}$ by filling the vacant time slots of $\mathcal{T}_{\text{HI}}$ by the available jobs of $\mathcal{T}_{\text{LO}}$ at those time slots.

3.3.1 The Algorithm

In this section, we propose TT-Merge, an algorithm which can schedule more instances than the OCBP-based algorithm as well as MCEDF. Our algorithm has a pseudo-polynomial time

complexity. The proposed algorithm constructs two tables $\mathcal{S}_{\text{HI}}$ and $\mathcal{S}_{\text{LO}}$ for the given MC instance, if possible. Our intention is to find $\mathcal{S}_{\text{LO}}$ and then construct $\mathcal{S}_{\text{HI}}$ keeping the same starting time for all the jobs as in $\mathcal{S}_{\text{LO}}$.
We define D_{max} which is the maximum deadline of the MC instance I.

$$D_{max} = max\{d_i\} \tag{3.1}$$

We construct $\mathcal{S}_{\text{LO}}$ from two temporary tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. Algorithm 1 and 2 describe the construction processes of $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. The length of the two temporary tables $\mathcal{T}_{\text{HI}}$ and $\mathcal{T}_{\text{LO}}$ is the same as the length of $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$.

Algorithm 1 Construct-$\mathcal{T}_{\text{LO}}(I)$

Input : $I = \{j_1, j_2, ..., j_n\}$, where $j_i =< a_i,\ d_i,\ \chi_i,\ C_i(\text{LO}),\ C_i(\text{HI}) >$.
Output : $\mathcal{T}_{\text{LO}}$
Assume earliest arrival time is 0.

1: Find the maximum deadline (D_{max}) of the jobs;
2: Prepare a temporary table $\mathcal{T}_{\text{LO}}$ of maximum length D_{max};
3: Let Ψ be the set of LO-criticality jobs of instance I;
4: Let O be the EDF order of the jobs of Ψ on the time-line using $C_i(\text{LO})$ units of execution for job j_i ;
5: **if** (any job cannot be scheduled) **then**
6: Declare failure;
7: **end if**
8: Starting from the rightmost job segment of the EDF order of Ψ, move each segment of a job j_i as close to its deadline as possible in $\mathcal{T}_{\text{LO}}$.

Algorithm 1 constructs the temporary table $\mathcal{T}_{\text{LO}}$. This algorithm chooses the LO-criticality jobs from the instance I and orders them in EDF order [LL73]. Then, all the job segments of the EDF schedule are moved as close to their deadline as possible so that no job misses its deadline in $\mathcal{T}_{\text{LO}}$. For example, we have an EDF order of three jobs as in Fig. 3.4 whose arrival times are $0, 2, 5$, execution times $6, 4, 2$ and deadlines $16, 11, 10$, respectively. The up and down arrows in the figure refer to the release and completion times respectively. Then, starting from the right end of the schedule, we shift each job segment as close to its

deadline as possible so that no job misses its deadline. Here we move the rightmost job segment, i.e., j_1's segment as close to its deadline, i.e., from [8,12] to [12,16]. We then move the next job segment of j_2 from [7,8] to [10,11]. Then the job segment of j_3 is moved right from [5,7] to [8,10] as the deadline of j_3 is 10. Then the job segment of j_2 is moved right from [2,5] to [5,8]. Finally, j_1's segment in the interval [0,2] is moved as close to its deadline as possible. Since at this stage there is an empty space at [11,12], j_1's segment in the interval [0,2] is distributed over [4,5] and [11,12]. The resulting table $\mathcal{T}_{\text{LO}}$ is given in Fig. 3.5. Note that, if the arrival times of the jobs are not the same, then the jobs may execute in more than one segment, in general. If the arrival times of all the jobs are the same then, the jobs will execute in one segment.

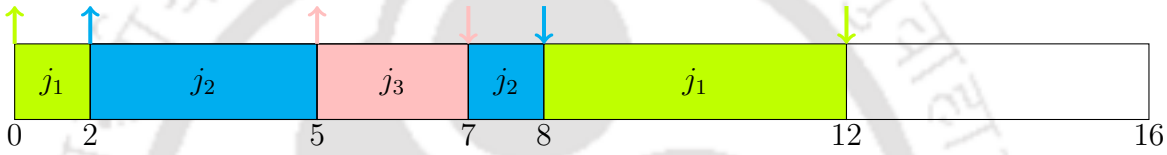


Figure 3.4: EDF order of three jobs. Up arrows indicate arrival and down arrows indicate completion times

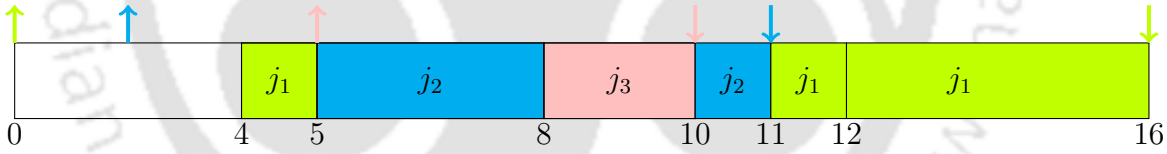


Figure 3.5: After the shifting of jobs

Algorithm 2 constructs the temporary table $\mathcal{T}_{\text{HI}}$. This algorithm chooses the HI-criticality jobs from the instance I and orders them in EDF order. Then, all the job segments of the EDF schedule are moved as close to their deadline as possible so that no job misses its deadline in $\mathcal{T}_{\text{HI}}$. Then, out of the total allocation so far, the algorithm allocates $C_i(\text{LO})$ units of execution of job j_i in $\mathcal{T}_{\text{HI}}$ from the beginning of its slot and leaves the rest of the execution time of j_i unallocated in $\mathcal{T}_{\text{HI}}$. Suppose, there is an instance I which contains three HI-criticality jobs j_1, j_2 and j_3 with arrival times $0, 2, 5$, execution times $(2, 6), (2, 4), (2, 2)$ and deadlines $16, 11, 10$, respectively. This instance is arranged in EDF order and then each job segment is shifted as close to its deadline as possible. The resulting allocation is given at the top of Fig. 3.6, which happens to be the same as in the earlier example for $\mathcal{T}_{\text{LO}}$. Then algorithm 2 allocates $C_i(\text{LO})$ units of execution and leaves $(C_i(\text{HI}) - C_i(\text{LO}))$ units of

Algorithm 2 Construct-$\mathcal{T}_{\text{HI}}(I)$

Input : $I = \{j_1, j_2, ..., j_n\}$, where $j_i =< a_i, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}) >$.

Output : $\mathcal{T}_{\text{HI}}$

Assume earliest arrival time is 0.

1: Find the maximum deadline (D_{max}) of the jobs;
2: Prepare a temporary table $\mathcal{T}_{\text{HI}}$ of maximum length D_{max};
3: Let Ψ be the set of HI-critical jobs of instance I;
4: Let O be the EDF order of the jobs of Ψ on the time-line using $C_i(\text{HI})$ units of execution for job j_i ;
5: **if** (any job cannot be scheduled) **then**
6: Declare failure;
7: **end if**
8: Starting from the rightmost job segment of the EDF order of Ψ, move each segment of a job j_i as close to its deadline as possible in $\mathcal{T}_{\text{HI}}$.
9: **for** $i := 1$ to m **do**
10: Allocate $C_i(\text{LO})$ units of execution to job j_i from its starting time in $\mathcal{T}_{\text{HI}}$ and leave the rest unallocated;
11: **end for**

execution unallocated. The resulting allocation is shown at the bottom of Fig. 3.6.

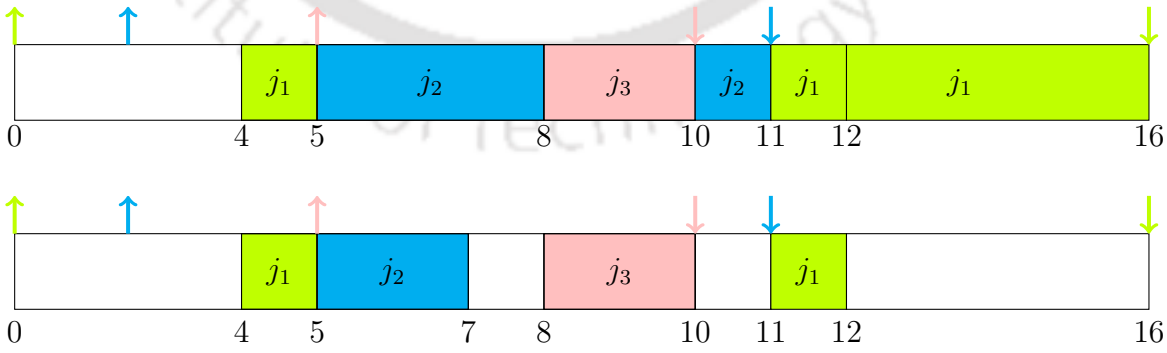


Figure 3.6: Allocating $C_i(\text{LO})$ units of execution only

Now, we use Algorithm 3 to construct the table $\mathcal{S}_{\text{LO}}$ from $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. The algorithm starts the construction of $\mathcal{S}_{\text{LO}}$ from time 0 and checks the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ simultaneously.

Algorithm 3 TT-Merge($I, \mathcal{T}_{\text{LO}}, \mathcal{T}_{\text{HI}}$)

Input : $I = \{j_1, j_2, ..., j_n\}$, where $j_i =< a_i, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}) >, \mathcal{T}_{\text{LO}}, \mathcal{T}_{\text{HI}}$
Output : Tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$
Assume earliest arrival time is 0.

Construction of $\mathcal{S}_{\text{LO}}$.
Find the maximum deadline (D_{max}) of the jobs;
The maximum length of tables $\mathcal{S}_{\text{HI}}$ and $\mathcal{S}_{\text{LO}}$ are both D_{max};
$t := 0$;
while ($t \leq D_{max}$) **do**
  if ($\mathcal{T}_{\text{LO}}[t] = NULL$ & $\mathcal{T}_{\text{HI}}[t] = NULL$) **then**
    Search the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ simultaneously from the beginning to find the first available job at time t;
    Let k be the first occurrence of a job j_i in $\mathcal{T}_{\text{LO}}$ or $\mathcal{T}_{\text{HI}}$;
    if (Both LO-criticality & HI-criticality job are found) **then**
      $\mathcal{S}_{\text{LO}}[t] := \mathcal{T}_{\text{LO}}[k]$;
      $\mathcal{T}_{\text{LO}}[k] := NULL$;
    else if (LO-criticality job is found) **then**
      $\mathcal{S}_{\text{LO}}[t] := \mathcal{T}_{\text{LO}}[k]$;
      $\mathcal{T}_{\text{LO}}[k] := NULL$;
    else if (HI-criticality job is found) **then**
      $\mathcal{S}_{\text{LO}}[t] := \mathcal{T}_{\text{HI}}[k]$;
      $\mathcal{T}_{\text{HI}}[k] := NULL$;
    else if (NO job is found) **then**
      $\mathcal{S}_{\text{LO}}[t] := NULL$
      $t := t + 1$;
    end if
  else if ($\mathcal{T}_{\text{LO}}[t] = NULL$ & $\mathcal{T}_{\text{HI}}[t]$!= $NULL$) **then**
    $\mathcal{S}_{\text{LO}}[t] := \mathcal{T}_{\text{HI}}[t]$;
    $\mathcal{T}_{\text{HI}}[t] := NULL$;
    $t := t + 1$;
  else if ($\mathcal{T}_{\text{LO}}[t]$!= $NULL$ & $\mathcal{T}_{\text{HI}}[t] = NULL$) **then**
    $\mathcal{S}_{\text{LO}}[t] := \mathcal{T}_{\text{LO}}[t]$;
    $\mathcal{T}_{\text{LO}}[t] := NULL$;
    $t := t + 1$;
  else if ($\mathcal{T}_{\text{LO}}[t]$!= $NULL$ & $\mathcal{T}_{\text{HI}}[t]$!= $NULL$) **then**
    Declare failure;
  end if
end while
This is the table $\mathcal{S}_{\text{LO}}$;

Construction of $\mathcal{S}_{\text{HI}}$
Copy all the jobs from table $\mathcal{S}_{\text{LO}}$ to table $\mathcal{S}_{\text{HI}}$;
Scan the table $\mathcal{S}_{\text{HI}}$ from left to right:
for each HI-criticality job j_i, allocate an additional $C_i(\text{HI}) - C_i(\text{LO})$ time units immediately after the rightmost segment of job j_i, recursively pushing all the overlapping HI-criticality job segments in $\mathcal{S}_{\text{HI}}$ (except those whose allocation time is same as in $\mathcal{T}_{\text{HI}}$) to the right and overwriting any LO-criticality jobs in the process.

There are four possibilities while merging the two temporary tables to construct $\mathcal{S}_{\text{LO}}$. At time slot t, one of the following situations can occur.

1. Both $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ are empty.
2. Both $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ are not empty.
3. $\mathcal{T}_{\text{LO}}$ is empty and $\mathcal{T}_{\text{HI}}$ is not empty.
4. $\mathcal{T}_{\text{LO}}$ is not empty and $\mathcal{T}_{\text{HI}}$ is empty.

If situation 1 occurs, then the algorithm will allocate the nearest ready job to the right at time slot t where a LO-criticality job gets higher priority over a HI-criticality job. In this case, the place of the ready job in $\mathcal{T}_{\text{LO}}$ or $\mathcal{T}_{\text{HI}}$ is marked as empty. In case of situation 2, the algorithm declares failure to schedule. In situation 3, the algorithm allocates the HI-criticality job from $\mathcal{T}_{\text{HI}}$, whereas in situation 4, the algorithm allocates the LO-criticality job from $\mathcal{T}_{\text{LO}}$. Once an instant of a job is allocated in $\mathcal{S}_{\text{LO}}$, the place where it was scheduled in $\mathcal{T}_{\text{LO}}$ or $\mathcal{T}_{\text{HI}}$ is emptied.

We then construct the table $\mathcal{S}_{\text{HI}}$ from $\mathcal{S}_{\text{LO}}$. We first copy the jobs of table $\mathcal{S}_{\text{LO}}$ to $\mathcal{S}_{\text{HI}}$. Then the HI-criticality jobs are allocated $C_i(\text{HI}) - C_i(\text{LO})$ units of HI-criticality execution time after their $C_i(\text{LO})$ units of execution in $\mathcal{S}_{\text{HI}}$. These additional time units are allocated by pushing all overlapping HI-criticality jobs in $\mathcal{S}_{\text{HI}}$ to the right and overwriting any LO-criticality job in the process. An exception to this is when the allocation time of an overlapping HI-criticality job is the same in both the tables $\mathcal{S}_{\text{HI}}$ and $\mathcal{T}_{\text{HI}}$, in which case the additional time units are allocated after this job. A LO-criticality job j_k present in table $\mathcal{S}_{\text{LO}}$ will not appear in table $\mathcal{S}_{\text{HI}}$ if and only if the additional $C_i(\text{HI}) - C_i(\text{LO})$ time units of allocation of any HI-criticality job overlaps with the allocation of j_k in table $\mathcal{S}_{\text{LO}}$.

3.3.2 Intuition Behind the Algorithm

In the following subsections, we show that TT-Merge dominates both the existing mixed-criticality time-triggered scheduling algorithms by being able to schedule a larger subset of instances. Here we briefly explain the working of TT-Merge, contrasting it with the existing algorithms.

The OCBP algorithm fails to find a priority order for instance I, if it is unable to choose a lowest priority job from I. For example, a HI-criticality job j_i is assigned the lowest priority

if all other jobs can finish their HI-criticality execution times before their deadlines and still leave sufficient time for j_i to finish its execution. But this is too strong a requirement, since in a HI-criticality scenario the LO-criticality jobs need not meet their deadlines. Since it is not possible for OCBP to check the worst-case starting and completion time of each job separately at each criticality level, it fails to assign priorities in some cases. We construct an algorithm which does not depend on any priority, while finding a time-triggered schedule. We construct two separate schedules for the two different criticality levels. We merge the two tables to find a LO-criticality schedule and then find the HI-criticality schedule using this LO-criticality schedule.

The core idea behind TT-Merge is to allocate jobs at each instant of the time-triggered schedule without depending on any priority such that both the scenarios (HI-criticality and LO-criticality) can be successfully scheduled. To this end, we find the worst-case starting and completion times of each job of the same criticality for the LO-criticality scenario separately in the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. Algorithms 1 and 2 find the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ by shifting the job segments of the EDF order of jobs as close to their deadlines as possible considering $C_i(\text{LO})$ and $C_i(\text{HI})$ units of executions, respectively. Then Algorithm 2 keeps $C_i(\text{LO})$ units of execution for each HI-criticality job in $\mathcal{T}_{\text{HI}}$ and empties the rest of the slots. From table $\mathcal{T}_{\text{HI}}$, we know the worst-case completion time of a LO-criticality execution of a HI-criticality job. These two tables are identical to the OCBP order for the jobs of the same criticality, which we prove later in Lemma 3.3.4 and 3.3.5. Algorithm 3 merges the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ to construct the table $\mathcal{S}_{\text{LO}}$, where all the tables have the same schedule length, i.e., D_{max}. Algorithm 3 keeps the jobs of table $\mathcal{T}_{\text{HI}}$ at their assigned slots and fills the empty places of this table with the jobs of the table $\mathcal{T}_{\text{LO}}$. This guarantees the timely execution of HI-criticality jobs in both the scenarios which is not always possible in the case of the OCBP-based and MCEDF algorithms. Since jobs of the table $\mathcal{T}_{\text{LO}}$ fill the empty spaces of the table $\mathcal{T}_{\text{HI}}$, we prefer a LO-criticality job to be allocated at time t, if both the tables are empty at time t. If a LO-criticality job is not available at t and a HI-criticality job is available, then that HI-criticality job segment is chosen to be allocated at t.

We illustrate the operation of this algorithm by an example.

Example 3.3.1: Consider the MC instance given in Table 3.2.
Let us first find the two temporary tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ in which the LO-criticality and HI-criticality jobs are allocated respectively.

Table 3.2: An example instance to explain the TT-Merge algorithm

Job	Arrival time	Deadline	Criticality	C_i(LO)	C_i(HI)
j_1	1	8	HI	1	2
j_2	1	6	HI	1	2
j_3	2	4	HI	1	2
j_4	0	4	LO	1	1
j_5	0	4	LO	2	2

- $D_{max} = 8$.
- The maximum length of $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ is 8.
- According to Algorithm 1, we choose the LO-criticality jobs and allocate them in $\mathcal{T}_{\text{LO}}$ in EDF order. Then, each segment of the jobs in EDF order are shifted as close to their deadlines as possible according to their C_i(LO) units of execution. So j_4 is allocated in the interval [1,2] and j_5 is allocated in the interval [2,4]. The resulting table $\mathcal{T}_{\text{LO}}$ is given in Fig. 3.7.

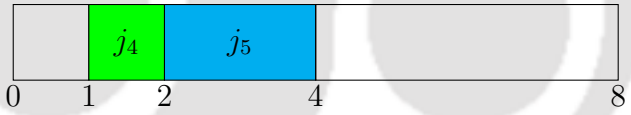


Figure 3.7: Temporary table $\mathcal{T}_{\text{LO}}$

- According to Algorithm 2, we choose the HI-criticality jobs to allocate them in $\mathcal{T}_{\text{HI}}$ in EDF order. Then, each segment of the jobs in EDF order are shifted as close to their deadlines as possible according to their C_i(HI) units of execution. So j_3 is allocated in the interval [2,4], j_2 is allocated in the interval [4,6] and j_1 is allocated in the interval [6,8]. The resulting table $\mathcal{T}_{\text{HI}}$ is given in Fig. 3.8.

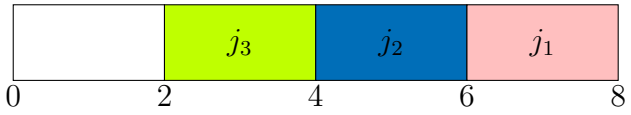


Figure 3.8: Intermediate temporary table $\mathcal{T}_{\text{HI}}$

- Then, we allocate $C_i(\text{LO})$ units of execution of j_i and leave the $(C_i(\text{HI}) - C_i(\text{LO}))$ units of execution unallocated. Here j_3 has been allocated its $C_i(\text{LO})$ units of execution time in the interval [2,3]. So we empty the occurrence of j_3 in the interval [3,4]. We repeat the same process for both j_2 and j_1. After this modification of $\mathcal{T}_{\text{HI}}$, the resulting table $\mathcal{T}_{\text{HI}}$ is given in Fig. 3.9.

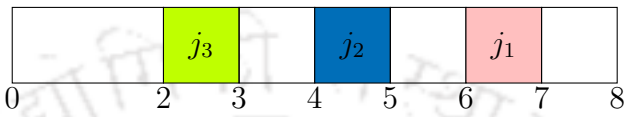


Figure 3.9: Temporary table $\mathcal{T}_{\text{HI}}$

- Finally, we construct $\mathcal{S}_{\text{LO}}$ from these two temporary tables.

We construct the table $\mathcal{S}_{\text{LO}}$ according to Algorithm 3.

- We start from time $t = 0$.
- At $t = 0$, both $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ are empty. So we allocate the LO-criticality job from $\mathcal{T}_{\text{LO}}$ which is ready at $t = 0$, i.e., j_4. We empty the interval [1,2] in $\mathcal{T}_{\text{LO}}$ from where the first occurrence of j_4 is found.
- At $t = 1$, both $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ are empty. So we allocate the LO-criticality job from $\mathcal{T}_{\text{LO}}$ which is ready at $t = 1$, i.e., j_5. We empty the interval [2,3] in $\mathcal{T}_{\text{LO}}$ from where the first occurrence of j_5 is found.
- At $t = 2$, $\mathcal{T}_{\text{LO}}$ is empty and $\mathcal{T}_{\text{HI}}$ contains j_3. So we allocate j_3 from $\mathcal{T}_{\text{HI}}$ and empty the interval [2,3] of $\mathcal{T}_{\text{HI}}$.
- At $t = 3$, $\mathcal{T}_{\text{LO}}$ contains j_5 and $\mathcal{T}_{\text{HI}}$ is empty. So we allocate j_5 from $\mathcal{T}_{\text{LO}}$ and empty the interval [3,4] of $\mathcal{T}_{\text{LO}}$.
- At $t = 4$, $\mathcal{T}_{\text{LO}}$ is empty and $\mathcal{T}_{\text{HI}}$ contains j_2. So we allocate j_2 from $\mathcal{T}_{\text{HI}}$ and empty the interval [4,5] of $\mathcal{T}_{\text{HI}}$.
- At $t = 5$, both $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ are empty. So we allocate a ready LO-criticality job from $\mathcal{T}_{\text{LO}}$. But, no LO-criticality jobs are there to be allocated. So we allocate the remaining jobs of $\mathcal{T}_{\text{HI}}$.

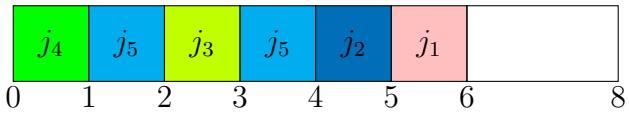


Figure 3.10: Table $\mathcal{S}_{\text{LO}}$

- The resulting table $\mathcal{S}_{\text{LO}}$ is given in Fig. 3.10.

Now, we construct the table $\mathcal{S}_{\text{HI}}$ from $\mathcal{S}_{\text{LO}}$ using the steps shown in Fig. 3.11.

- We copy the table $\mathcal{S}_{\text{LO}}$ to table $\mathcal{S}_{\text{HI}}$.
- For the first HI-criticality job j_3, $C_3(\text{HI}) - C_3(\text{LO})$ units of execution time are allocated in the interval $[3, 4]$. In this process, we overwrite job j_5 which was present in the interval $[3, 4]$. This is shown in the top table of Fig. 3.11.
- Then $C_2(\text{LO})$ units of execution time of j_2 are allocated in the interval $[4, 5]$ followed by $C_2(\text{HI}) - C_2(\text{LO})$ units of execution time in the interval $[5, 6]$. In this process, we push job j_1 to its right, i.e., to the interval $[6, 7]$ from $[5, 6]$. Finally, j_1 is allocated in the interval [6,8]. This is shown in the middle table of Fig. 3.11.
- The resulting table $\mathcal{S}_{\text{HI}}$ is given in the table at the bottom of Fig. 3.11.

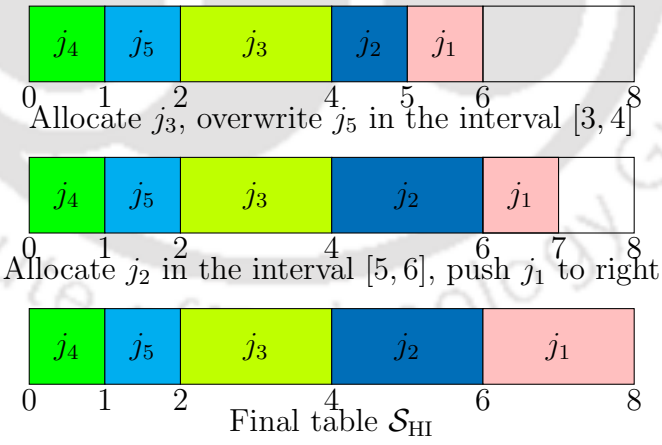


Figure 3.11: Construction of table $\mathcal{S}_{\text{HI}}$

□

Now we present an example to show the tables constructed by different existing algorithm and TT-Merge for the same instance.

Example 3.3.2: The point of this example is to show how the tables constructed by TT-Merge differ from the ones constructed by the OCBP-based algorithm, when both the algorithms are successful. Consider the MC instance given in Table 3.3. Fig. 3.12 shows

Table 3.3: An instance where both TT-Merge and OCBP-based algorithms are successful

Job	Arrival time	Deadline	Criticality	C_i(LO)	C_i(HI)
j_1	0	2	LO	1	1
j_2	0	7	HI	2	3
j_3	2	10	LO	4	4
j_4	5	10	HI	2	5

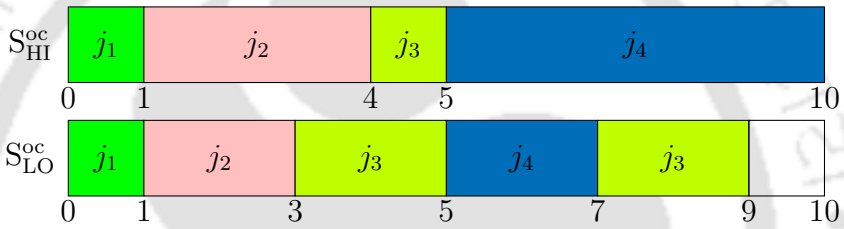


Figure 3.12: Scheduling tables according to OCBP-based algorithm

the tables constructed using the OCBP-based algorithm. The MCEDF algorithm computes the same priority order as OCBP. So it constructs the same tables as OCBP. Fig. 3.13 shows the scheduling tables according to TT-Merge.

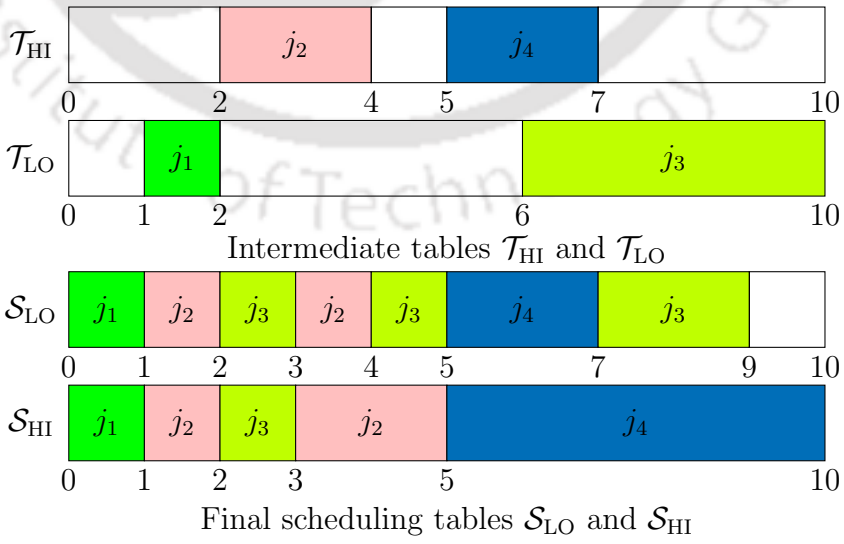


Figure 3.13: Scheduling tables according to TT-Merge

3.3.3 Correctness Proof

For correctness, we have to show that if TT-Merge finds the two scheduling tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$, then these two tables will give a correct scheduling strategy. We start with the proof of some properties of the schedule.

Observation 3.3.1: The table $\mathcal{T}_{\text{HI}}$ shows the latest possible allocation of the initial (LO-criticality) segment of a HI-criticality job that can still meet its deadline in a schedule. To see this, recall that the table $\mathcal{T}_{\text{HI}}$ is constructed from the EDF order of the HI-criticality jobs. Each job segment in the EDF order is pushed as close to its deadline as possible. Then the initial $C_i(\text{LO})$ time units of each job are kept and the rest are unallocated. By the construction, no job segment can be pushed further to the right and still meet its deadline.

Remark: We know that the table $\mathcal{S}_{\text{LO}}$ allocates each HI-criticality job on or before its allocation in $\mathcal{T}_{\text{HI}}$. Then no job can be pushed to the right in the table $\mathcal{S}_{\text{HI}}$ after its allocation in $\mathcal{T}_{\text{HI}}$ as it will miss its deadline. This follows from Observation 3.3.1.

Lemma 3.3.1: If Algorithm 3 does not declare failure, then each job j_i receives $C_i(\text{LO})$ units of execution in $\mathcal{S}_{\text{LO}}$ and each HI-criticality job j_k receives $C_k(\text{HI})$ units of execution in $\mathcal{S}_{\text{HI}}$ by its deadline.

Proof. First, we show that any job j_i receives $C_i(\text{LO})$ units of execution in $\mathcal{S}_{\text{LO}}$. We construct $\mathcal{S}_{\text{LO}}$ from the temporary tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. Each job j_i can be scheduled in $\mathcal{S}_{\text{LO}}$ on or before its scheduled time in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. If TT-Merge finds the table $\mathcal{S}_{\text{LO}}$ then each job must receive $C_i(\text{LO})$ units of execution.

Next we show that any HI-criticality job j_k receives $C_k(\text{HI})$ units of execution in $\mathcal{S}_{\text{HI}}$. We start constructing $\mathcal{S}_{\text{HI}}$ by copying the jobs in $\mathcal{S}_{\text{LO}}$. But according to TT-Merge, the HI-criticality jobs are allocated their remaining $C_k(\text{HI}) - C_k(\text{LO})$ units of allocation in $\mathcal{S}_{\text{HI}}$ after they complete their $C_k(\text{LO})$ units of allocation in $\mathcal{S}_{\text{HI}}$ by pushing recursively all the following HI-criticality job segments to the right except those whose allocation is the same as in table $\mathcal{T}_{\text{HI}}$. This means we can push a job segment to the right in $\mathcal{S}_{\text{HI}}$ only if it is allocated before its allocation in $\mathcal{T}_{\text{HI}}$ and moreover, no job is pushed beyond its allocation in $\mathcal{T}_{\text{HI}}$, because if the construction of $\mathcal{T}_{\text{HI}}$ does not declare failure then it allocates enough time for the execution of all the HI-criticality jobs. In this case, all the jobs can get sufficient time to schedule their $C_k(\text{HI}) - C_k(\text{LO})$ units of execution as they are allocated on or before the allocation in table $\mathcal{T}_{\text{HI}}$. This is clear from the remark following Observation 1. If a HI-criticality job j_h cannot be pushed to the right then it will get its remaining $C_h(\text{HI}) - C_h(\text{LO})$ units of execution

time in table $\mathcal{S}_{\text{HI}}$ by a similar reasoning as above. □

Lemma 3.3.2: At any time t, if a job j_i is present in $\mathcal{S}_{\text{HI}}$ but not in $\mathcal{S}_{\text{LO}}$, then the job j_i has finished its LO-criticality execution before time t in $\mathcal{S}_{\text{LO}}$.

Proof. We use the same order of jobs in $\mathcal{S}_{\text{LO}}$ to construct $\mathcal{S}_{\text{HI}}$. We know the HI-criticality jobs are allocated their $C_i(\text{HI}) - C_i(\text{LO})$ units of execution after the allocation of $C_i(\text{LO})$ units of execution in $\mathcal{S}_{\text{HI}}$. In $\mathcal{S}_{\text{HI}}$, the HI-criticality jobs are preferred over the LO-criticality jobs, i.e., a HI-criticality job is chosen to be allocated in table $\mathcal{S}_{\text{HI}}$ if a LO-criticality job is found in $\mathcal{S}_{\text{LO}}$ while allocating $C_i(\text{HI}) - C_i(\text{LO})$ units of execution in table $\mathcal{S}_{\text{HI}}$. This means each of the job segments present in table $\mathcal{S}_{\text{HI}}$ is either at the same position in $\mathcal{S}_{\text{LO}}$ or to the right of it. When a job j_i is present in $\mathcal{S}_{\text{HI}}$ and not in $\mathcal{S}_{\text{LO}}$ at time t, it means this has already completed its LO-criticality execution in $\mathcal{S}_{\text{LO}}$. □

Lemma 3.3.3: At any time t, when a mode change occurs, each HI-criticality job still has $C_i(\text{HI}) - c_i$ units of execution in $\mathcal{S}_{\text{HI}}$ after time t to complete its execution, where c_i is the execution time already completed by job j_i before time t in $\mathcal{S}_{\text{LO}}$.

Proof. Suppose a mode change occurs at time t. This means all the HI-criticality jobs scheduled before time t have either signaled their completion or the current HI-criticality job is the first one to complete its $C_i(\text{LO})$ units of execution without signaling its completion. We know that all the HI-criticality jobs are allocated their $C_i(\text{HI}) - C_i(\text{LO})$ units of execution in $\mathcal{S}_{\text{HI}}$ after the completion of their $C_i(\text{LO})$ units of execution in both $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$. If a job j_i has already executed its $C_i(\text{LO})$ units of execution in $\mathcal{S}_{\text{LO}}$, then it requires $C_i(\text{HI}) - C_i(\text{LO})$ units of time to be completed in $\mathcal{S}_{\text{HI}}$. When job j_i initiates the mode change, this is the first job which does not signal its completion after completing its $C_i(\text{LO})$ units of execution. Before time t, the scheduler uses the table $\mathcal{S}_{\text{LO}}$ to schedule the jobs, while subsequently the scheduler uses table $\mathcal{S}_{\text{HI}}$ due to the mode change. If a job j_i has already executed its c_i units of execution in $\mathcal{S}_{\text{LO}}$, then it requires $C_i(\text{HI}) - c_i$ units of time to be completed in $\mathcal{S}_{\text{HI}}$ its execution. We know that the tables $\mathcal{S}_{\text{HI}}$ and $\mathcal{S}_{\text{LO}}$ have same order and according to lemma 3.3.1 and 3.3.2, each job will get sufficient time to complete its $C_i(\text{HI})$ units of execution. Hence, each HI-criticality job will get $C_i(\text{HI}) - c_i$ units of time in $\mathcal{S}_{\text{HI}}$ to complete its execution after the mode change at time t. □

Theorem 3.3.1: If the scheduler dispatches the jobs according to $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$, then it will be a correct scheduling strategy.

Proof. For LO-criticality scenarios, all jobs can be correctly scheduled by the table $\mathcal{S}_{\text{LO}}$ as proved in Lemma 3.3.1. Now, we need to prove that in a HI-criticality scenario, all the HI-criticality jobs can be correctly scheduled by the table $\mathcal{S}_{\text{HI}}$. In Lemma 3.3.1, we have already proved that all the HI-criticality jobs get sufficient units of time in $\mathcal{S}_{\text{HI}}$ to complete their execution. In Lemma 3.3.3, we have proved that when the mode change occurs at time t, all the HI-criticality jobs can be scheduled without missing their deadline. So from Lemma 3.3.1 and Lemma 3.3.3, it is clear that if the scheduler uses the tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$ to dispatch the jobs then it will be a correct scheduling strategy. □

3.3.4 Dominance Over OCBP-based Algorithm

We know that the algorithm proposed by Baruah and Fohler [BF11] is based on the OCBP order [BBD+12a] and constructs the tables $\text{S}^{\text{oc}}_{\text{LO}}$ and $\text{S}^{\text{oc}}_{\text{HI}}$ based on this order. We show that if the OCBP-based algorithm constructs the tables $\text{S}^{\text{oc}}_{\text{LO}}$ and $\text{S}^{\text{oc}}_{\text{HI}}$ for an instance then TT-Merge will also construct the two tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$ for the same instance.

Notation: We use $\text{S}^{\text{oc}}_{\text{LO}}$ and $\text{S}^{\text{oc}}_{\text{HI}}$ for the tables constructed by the OCBP-based algorithm and $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$ for the tables constructed by TT-Merge. Further, we use $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ for the two temporary tables in TT-Merge.

Lemma 3.3.4: If OCBP chooses a latest deadline job as the lowest priority job at each stage, then the OCBP priority order of jobs of the same criticality is the same as that assigned by EDF.

Proof. See observation 1 of Lemma 2 from Park and Kim [PK11b] for the proof of this lemma. □

Lemma 3.3.5: If OCBP finds a priority order for an instance I, then there exists an OCBP priority order for I in which all jobs of the same criticality are in EDF order.

Proof. Suppose there exists an OCBP priority order for instance I. Let j_i and j_k be two jobs of the same criticality, where j_i is assigned higher priority than j_k by OCBP and $d_i \geq d_k$. OCBP assigns lower priority to j_k because all other jobs including j_i finish their $C(\chi_k)$ units of execution and there is sufficient time in the interval $[a_k, d_k]$ for j_k to finish its $C(\chi_k)$ units of execution. If we swap the priority levels of j_i and j_k, then j_k certainly meets its deadline and even though the execution segments of j_i are shifted to the right, its deadline d_i is not

violated, since $d_i \geq d_k$. So we can exchange their priority which means there exists a priority order for I in which all jobs of the same criticality are in EDF order. □

Without loss of generality, by Lemma 3.3.5 all the jobs in the table S^{oc}_{LO} constructed by the OCBP-based algorithm of the same criticality are in EDF order.

Lemma 3.3.6: If OCBP finds a priority order for an instance I, then Algorithms 1 and 2 can construct the tables $\mathcal{T}_{LO}$ and $\mathcal{T}_{HI}$ and these are obtained from the OCBP order by moving the job segments to the right starting from the right end of the schedule for the LO-criticality and HI-criticality jobs respectively.

Proof. Follows from Lemma 3.3.4 and 3.3.5. □

Theorem 3.3.2: If an instance I is schedulable by the OCBP-based scheduling algorithm, then it is also schedulable by TT-Merge.

Proof. OCBP generates a priority order for an instance I. Then the OCBP-based algorithm finds the tables S^{oc}_{LO} and S^{oc}_{HI} for the instance I using this priority order. We need to show that if there exists tables S^{oc}_{LO} and S^{oc}_{HI} constructed by the OCBP-based algorithm, then TT-Merge will not encounter a situation where at time slot t $\mathcal{T}_{LO}$ and $\mathcal{T}_{HI}$ are non-empty, for any t.

We know that $C_i(\text{LO})$ units of execution are allocated to each job j_i for constructing the tables $\mathcal{T}_{LO}$ and $\mathcal{T}_{HI}$. Each job in $\mathcal{T}_{LO}$ and $\mathcal{T}_{HI}$ is allocated as close to its deadline as possible. That means no job can execute after its allocation time in $\mathcal{T}_{LO}$ and $\mathcal{T}_{HI}$ without affecting the schedule of any other job and still meet its deadline. Algorithm 3 declares failure if it finds a non-empty slot at any time t in both the tables $\mathcal{T}_{LO}$ and $\mathcal{T}_{HI}$. This means the two jobs which are found in the tables $\mathcal{T}_{LO}$ and $\mathcal{T}_{HI}$ respectively cannot be scheduled with all other remaining jobs from this point, because all the jobs to the right have already been moved as far to the right as possible.

Suppose there is an OCBP priority order of the jobs of instance I and the LO-criticality table S^{oc}_{LO} follows this priority order.

Let j_l and j_h be two jobs in $\mathcal{T}_{LO}$ and $\mathcal{T}_{HI}$ respectively found at time t during the construction of table $\mathcal{S}_{LO}$ by TT-Merge, which means all job segments in the interval $[0, t-1]$ from $\mathcal{T}_{LO}$ and $\mathcal{T}_{HI}$ have already been assigned in $\mathcal{S}_{LO}$. But, we know that OCBP has assigned priorities to these jobs j_l and j_h. There are two cases to consider.

In the first case, assume j_l is assigned lower priority than j_h by OCBP. Let a_l be the arrival time of job j_l and t_l and t_l' be the starting and completion times of j_l in $\mathcal{T}_{\text{LO}}$ computed by Algorithm 1. Since job j_l can be scheduled only on or after the arrival time a_l, we need to show that the job segment of j_l found at time t cannot be scheduled in the interval $[a_l, t-1]$ by the OCBP-based algorithm. We know that Algorithm 3 can allocate a job in table $\mathcal{S}_{\text{LO}}$ on or before its allocation in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. But Algorithm 3 has not allocated the job segments found in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ at time t in the interval $[a_l, t-1]$ of the table $\mathcal{S}_{\text{LO}}$ and by Lemma 3.3.6 this is due to the presence of equal or higher priority job segments of the OCBP priority order in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. We know that all the jobs in $\mathcal{T}_{\text{LO}}$ in the interval $[a_l, t]$ and the jobs in $\mathcal{T}_{\text{HI}}$ including job j_h in the interval $[a_l, t]$ are of priority greater or equal to that of j_l according to OCBP since, by moving job segments to the right starting from the OCBP schedule the jobs to the left of j_l are of priority greater than or equal to that of j_l. This means the jobs in the interval $[a_l, t-1]$ of table $\mathcal{S}_{\text{LO}}$ are either equal or higher priority jobs than j_l according to OCBP. So both the algorithms, the OCBP-based one and ours, allocate higher or equal priority jobs (or, job segments according to Algorithm 3) before time t. Then it is clear that after the jobs of higher priority than j_l finish their $C(\text{LO})$ units of execution by time t, there will not be sufficient time for j_l to finish its $C_l(\text{LO})$ units of execution in the interval $[a_l, t_l']$ in the OCBP schedule. This is because at time t, the OCBP-based algorithm has already allocated all ready jobs with higher or equal priority than j_l (according to OCBP) in the interval $[a_l, t]$ with no vacant slot for further allocation of j_l's segment found at time slot t, which is the case for Algorithm 3 as well. A similar statement holds for j_h. Therefore j_h and j_l cannot be simultaneously scheduled to meet their deadlines in the remaining time, according to the OCBP-based algorithm.

In the second case, assume j_h is assigned lower priority than j_l by OCBP. Let a_h be the arrival time of job j_h and let the starting and completion times of the LO-criticality execution of j_h be t_h and t_h' respectively, and the completion time of the HI-criticality execution be t_e. As in the previous case, all the jobs in $\mathcal{T}_{\text{HI}}$ in the interval $[a_h, t]$ and the jobs in $\mathcal{T}_{\text{LO}}$, including job j_l, in the interval $[a_h, t]$ are of priority (according to OCBP) greater than or equal to that of j_h. OCBP considers $C(\text{HI})$ units of execution time to assign a priority to a HI-criticality job. As seen above, it is clear that after the jobs of higher priority than j_h finish their $C(\text{LO})$ units of execution by time t, there will not be sufficient time for j_h to finish its $C_h(\text{LO})$ units of execution in the interval $[a_h, t_h']$ according to OCBP. We know that $C(\text{LO}) \leq C(\text{HI})$. If job j_h does not get sufficient time to execute its $C_h(\text{LO})$ units of

execution in the interval $[a_h, t_h']$, then it will not get sufficient time to execute its C_h(HI) units of execution in the interval $[a_h, t_e]$ either.

From the above two cases, it is clear that OCBP cannot assign priorities to job j_l and j_h, which is a contradiction. This means if there exists an OCBP priority order for instance I, then TT-Merge will not encounter a situation where both the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ are non-empty at any time t for the instance I.

Note that we need to consider only the LO-criticality scenarios in the proof since Lemma 3.3.3 implies that if $\mathcal{S}_{\text{LO}}$ can be constructed, then so can $\mathcal{S}_{\text{HI}}$. □

3.3.5 Dominance Over MCEDF Algorithm

Now we show the dominance of TT-Merge over the MCEDF algorithm [SPBB13].

Lemma 3.3.7: If MCEDF finds a priority order for an instance I, then there exists an MCEDF priority order for I in which all jobs of the same criticality are in EDF order.

Proof. This can be derived directly from the priority assignment to the jobs by the MCEDF algorithm.

□

Theorem 3.3.3: If an instance I is schedulable by the MCEDF algorithm, then it is also schedulable by TT-Merge.

Proof. The MCEDF algorithm generates a priority order for an instance I. This priority order is used to find the table PT_{LO}. We need to show that if there exists a table PT_{LO} and the $anyHIscenarioFailure()$ subroutine in Algorithm MCEDF on page 95 of [SPBB13] does not fail, then TT-Merge will not encounter a situation where $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ are non-empty at any time slot t.

We know that C_i(LO) units of execution are allocated to each job j_i for constructing the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. Each job in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ is allocated as close to its deadline as possible. That means no job can execute after its allocation time in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ without affecting the schedule of any other job and still meet its deadline. Algorithm 3 declares failure if it finds a non-empty slot at any time t in both the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. This means the two jobs which are found in the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ respectively cannot be scheduled with all other remaining jobs from this point, because all the jobs to the right have already been moved as far to the right as possible.

By Lemma 3.3.7, without loss of generality, the MCEDF order is the same as the EDF orders for jobs of the same criticality. So the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ are obtained from the MCEDF order by moving the job segments to the right starting from the right end of the schedule for the LO-criticality and HI-criticality jobs respectively.

Suppose there is an MCEDF priority order of the jobs of instance I and a table PT_{LO} according to this priority order and suppose the $anyHIscenarioFailure()$ subroutine does not fail.

Let j_l and j_h be two jobs in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ respectively found at time t during the construction of table $\mathcal{S}_{\text{LO}}$ by TT-Merge, which means all the job segments in the interval $[0, t-1]$ from $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ have already been assigned in $\mathcal{S}_{\text{LO}}$. But, we know that MCEDF has assigned priorities to these jobs j_l and j_h. Now there are two cases.

In the first case, assume j_l is assigned lower priority than j_h by MCEDF. Let a_l be the arrival time of job j_l and t_l and $t_l{}'$ be the starting and completion times of j_l in $\mathcal{T}_{\text{LO}}$ computed by Algorithm 1. Since job j_l can be scheduled only on or after the arrival time a_l, we need to show that the job segment of j_l found at time t cannot be scheduled in the interval $[a_l, t-1]$ by the MCEDF algorithm. We know that Algorithm 3 can allocate a job in table $\mathcal{S}_{\text{LO}}$ on or before its allocation in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. But Algorithm 3 has not allocated the job segments found in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ at time t in the interval $[a_l, t-1]$ of the table $\mathcal{S}_{\text{LO}}$, and by Lemma 3.3.7, this is due to the presence of equal or higher priority job segments of the MCEDF priority order in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. We know that all the jobs in $\mathcal{T}_{\text{LO}}$ in the interval $[a_l, t]$ and the jobs in $\mathcal{T}_{\text{HI}}$ including job j_h in the interval $[a_l, t]$ are of priority greater or equal to that of j_l according to the MCEDF algorithm since, by moving job segments to the right starting from the EDF schedule, the jobs to the left of j_l are of priority greater than or equal to that of j_l. This means the jobs in the interval $[a_l, t-1]$ of table $\mathcal{S}_{\text{LO}}$ are either equal or higher priority jobs than j_l according to MCEDF. So both the algorithms, MCEDF and ours, allocate higher or equal priority jobs (or, job segments according to Algorithm 3) before time t. Then it is clear that after the jobs of higher priority than j_l finish their $C(LO)$ units of execution, there will not be sufficient time for j_l to finish its $C_l(\text{LO})$ units of execution in the interval $[a_l, t_l{}']$ in the MCEDF schedule. This is because at time t, the MCEDF algorithm has already allocated all ready jobs with higher or equal priority than j_l (according to MCEDF) in the interval $[a_l, t]$ with no vacant slot for further allocation of j_l's segment found at time slot t, which is the case for Algorithm 3 as well. A similar statement holds for j_h. Therefore j_h and j_l cannot be simultaneously scheduled to meet their deadlines

in the remaining time, according to MCEDF. In the second case, assume j_h is assigned lower priority than j_l by MCEDF. Let a_h be the arrival time of job j_h and let the starting and completion times of the LO-criticality execution of j_h be t_h and t_h' respectively, and the completion time of the HI-criticality execution be t_e. As in the previous case, all the jobs in $\mathcal{T}_{\text{HI}}$ in the interval $[a_h, t]$ and the jobs in $\mathcal{T}_{\text{LO}}$, including job j_l, in the interval $[a_h, t]$ are of priority (according to MCEDF) greater or equal to that of j_h. MCEDF considers $C(\text{HI})$ units of execution time to assign a priority to a HI-criticality job. As seen above, it is clear that after the jobs of higher priority than j_h finish their $C(\text{LO})$ units of execution, there will not be sufficient time for j_h to finish its $C_h(\text{LO})$ units of execution in the interval $[a_h, t_h']$ according to MCEDF. We know that $C(\text{LO}) \leq C(\text{HI})$. If job j_h does not get sufficient time to execute its $C_h(\text{LO})$ units of execution in the interval $[a_h, t_h']$, then it will not get sufficient time to execute its $C_h(\text{HI})$ units of execution in the interval $[a_h, t_e]$ either.

From the above two cases, it is clear that MCEDF may assign priorities to job j_l and j_h but the $anyHIscenarioFailure()$ subroutine will fail, which is a contradiction. This means if the MCEDF algorithm finds a schedule for instance I, then TT-Merge will not encounter a situation where both the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ are non-empty at any time t for the instance I.

Note that we need to consider only the LO-criticality scenarios in the proof since Lemma 3 implies that if $\mathcal{S}_{\text{LO}}$ can be constructed, then so can $\mathcal{S}_{\text{HI}}$. □

3.4 Extension for m Criticality Levels

The algorithm discussed in Section 3.3 constructs two scheduling tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$ for the dual-criticality instances which can be used by the scheduler to dispatch the jobs. Now we extend TT-Merge for instances with m criticality levels. Here we need to create m different tables for m criticality levels which can be used by the scheduler to dispatch the jobs.

3.4.1 Model

A job is characterized by a 5-tuple of parameters: $j_i = (a_i, d_i, \chi_i, \{C_i(1), C_i(2), \ldots, C_i(m)\})$, where

- $a_i \in \mathbb{N}$ denotes the *arrival time*.
- $d_i \in \mathbb{N}^+$ denotes the *absolute deadline*.

- $\chi_i \in \mathbb{N}^+$ denotes the *criticality* level.
- $\{C_i(1), C_i(2), \ldots, C_i(m)\}$ denotes the *worst-case execution time* at each criticality level.

We assume that $C_i(k)$ is monotonically increasing with increasing k, i.e., $\forall i : C_i(1) \leq C_i(2) \leq \ldots \leq C_i(m)$, where $1 \leq i \leq n$.

Definition 3.4.1: An m *criticality* MC instance I is said to be *time-triggered schedulable* if it is possible to construct m tables $\mathcal{S}_1$, $\mathcal{S}_2$, ..., $\mathcal{S}_m$ such that the scheduler can schedule any non-erroneous scenario of instance I.

The following scheduler algorithm is used to dispatch the jobs using the m tables at run-time.

- Initially $\chi_i = 1$
- The criticality level indicator Γ is initialized to χ_i.
- Repeat
  - While ($\Gamma = \chi_i$), at each time instant t the job available at time t in the table $\mathcal{S}_{\chi_i}$ will execute.
  - If a job executes for more than its χ_i-criticality WCET without signaling completion, then Γ is changed to $\chi_i + 1$.

3.4.2 Algorithm

Here we need to construct m tables to find a time triggered schedule. Each table is of length D_{max} as in Equation 3.1.

Algorithm 4 constructs m temporary tables $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}_m$. For each table $\mathcal{T}_{\chi_i}$ where $\chi_i \in \{1, 2, \ldots, m\}$, Algorithm 4 chooses jobs with χ_i-criticality level and orders them in EDF order. Then, all the job segments of the EDF order are moved as close to their deadline as possible so that no job misses its deadline in $\mathcal{T}_{\chi_i}$. Then out of the total allocation so far, the algorithm allocates $C_i(1)$ units of execution of j_i in $\mathcal{T}_{\chi_i}$ from the beginning of its slot and leaves the rest of the execution time of j_i unallocated in $\mathcal{T}_{\chi_i}$. This is similar to the dual criticality case.

Now, we use Algorithm 5 to construct the table $\mathcal{S}_1$ from tables $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}_m$. The algorithm starts the construction of $\mathcal{S}_1$ from time 0 and checks all m tables simultaneously.

Algorithm 4 Construct-TT-m-crit-$\mathcal{T}_{\chi_i}(I)$

Input : $I = \{j_1, j_2, ..., j_n\}$, where $j_i =< a_i, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}) >$.
Output : $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_m$
Assume earliest arrival time is 0.

```
1: Find the maximum deadline (D_max) of the jobs;
2: for χ_i := 1 to m do
3:    Prepare a temporary table T_χi of maximum length D_max;
4:    Let Ψ be the set of χ_i-criticality jobs of instance I;
5:    Let O be the EDF order of the jobs of Ψ on the time-line using C_i(χ_i) units of execution for job j_i ;
6:    if (any job cannot be scheduled) then
7:       Declare failure;
8:    end if
9:    Starting from the rightmost job segment of the EDF order of Ψ, move each segment of a job j_i as
      close to its deadline as possible in T_χi.
10:   for k := 1 to |Ψ| do
11:      Allocate C_k(1) units of execution to job j_k from its starting time in T_χi and leave the rest unallocated;
12:   end for
13: end for
```

There will be three situations while merging these tables to construct $\mathcal{S}_1$. At time slot t, one of the following can occur:

1. All m tables are empty.
2. Two or more tables from the m tables are not empty.
3. Exactly one table from the m tables is not empty.

If situation 1 occurs, then the algorithm will allocate the nearest ready job to the right at time slot t where a lower criticality job gets higher priority than a higher criticality job. After the allocation of the job j_i in $\mathcal{S}_1$, that instant of j_i in $\mathcal{T}_{\chi_i}$ is marked empty. In case of situation 2, the algorithm declares failure to schedule. In situation 3, the algorithm allocates the first available job from the table which is non-empty at time t in $\mathcal{S}_1$.

We then construct the table $\mathcal{S}_2$ from $\mathcal{S}_1$. We first copy the jobs of table $\mathcal{S}_1$ to table $\mathcal{S}_2$. Then all the jobs whose criticality are greater than 1 need to be allocated $C_i(2) - C_i(1)$ units of execution time immediately after their $C_i(1)$ units of execution in $\mathcal{S}_2$. These additional time units is allocated by pushing all overlapping jobs whose criticality is greater than or equal to 2 to the right and overwriting any job with criticality 1 in the process. If the

Algorithm 5 TT-Merge-m-crit(I, $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}_m$)

Input : $\mathcal{T}_1$, $\mathcal{T}_2$, ..., $\mathcal{T}_m$ and $I = \{j_1, j_2, ..., j_n\}$, where $j_i =< a_i, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}) >$.
Output : Tables $\mathcal{S}_1$, $\mathcal{S}_2$, ..., $\mathcal{S}_m$

Construction of $\mathcal{S}_1$.
Find the maximum deadline (D_{max}) of the jobs;
The maximum length of tables $\mathcal{S}_1$, $\mathcal{S}_2$,..., $\mathcal{S}_m$ are D_{max} each;
$t := 0$;
while ($t \leq D_{max}$) **do**
  if ($|\{\chi_i | T_{\chi_i}[t] \neq NULL\}| = 0$) **then**
    Search the tables $\mathcal{T}_{\chi_i}$ simultaneously from the beginning to find the first available job at time t;
    Let k be the first occurrence, if any, of such a job j_i in $\mathcal{T}_{\chi_i}$;
    if (more than one job is found) **then**
      $LC :=$ the lowest criticality such that a job j_i is found in $\mathcal{T}_{LC}$;
      $\mathcal{S}_1[t] := \mathcal{T}_{LC}[k]$;
      $\mathcal{T}_{LC}[k] := NULL$;
    else if (only job of χ_i-criticality level is found) **then**
      $\mathcal{S}_1[t] := \mathcal{T}_{\chi_i}[k]$;
      $\mathcal{T}_{\chi_i}[k] := NULL$;
    else if (no job is found) **then**
      $\mathcal{S}_1[t] := NULL$
      $t := t + 1$;
    end if
  else if ($|\{\chi_i | T_{\chi_i}[t] \neq NULL\}| = 1$) **then**
    $\mathcal{S}_1[t] := \mathcal{T}_{\chi_i}[t]$;
    $\mathcal{T}_{\chi_i}[t] := NULL$;
    $t := t + 1$;
  else if ($|\{\chi_i | T_{\chi_i}[t] \neq NULL\}| > 1$) **then**
    Declare failure;
  end if
end while
This is the table $\mathcal{S}_1$;

Construction of $\mathcal{S}_{\chi_i}$ where $2 \leq \chi_i \leq m$
for $\chi_i := 2$ to m **do**
  Copy all the jobs from table $\mathcal{S}_{\chi_i - 1}$ to table $\mathcal{S}_{\chi_i}$;
  Scan the table $\mathcal{S}_{\chi_i}$ from left to right:
  for each χ_i-criticality job j_l, allocate an additional $C_l(\chi_i) - C_l(\chi_i - 1)$ time units after the rightmost segment of job j_l, recursively pushing all the overlapping job segments with criticality greater or equal to χ_i-criticality in $\mathcal{S}_{\chi_i}$ (except those whose allocation time is same as in $\mathcal{T}_{\chi_i}$) to the right and overwriting any jobs with criticality $(\chi_i - 1)$-criticality or lesser in the process.
end for

allocation time of a job whose criticality is 2 or more which needs to be pushed is same in both the tables $\mathcal{S}_2$ and $\mathcal{T}_2$, then the additional time units are allocated after this job.

Similarly, we construct the table $\mathcal{S}_{\chi_i}$ from $\mathcal{S}_{\chi_i-1}$. We first copy the jobs of table $\mathcal{S}_{\chi_i-1}$ to table $\mathcal{S}_{\chi_i}$. Then the χ_i-criticality jobs are allocated $C_i(\chi_i) - C_i(\chi_i - 1)$ units of χ_i-criticality execution time immediately after their $C_i(\chi_i - 1)$ units of execution in $\mathcal{S}_{\chi_i}$. These additional time units is allocated by pushing all overlapping jobs whose criticality is greater than or equal to χ_i to the right and overwriting any job with criticality less than or equal to $(\chi_i - 1)$ in the process. If the allocation time of a χ_i-criticality job which needs to be pushed is same in both the tables $\mathcal{S}_{\chi_i}$ and $\mathcal{T}_{\chi_i}$, then the additional time units are allocated after this job.

3.4.3 Correctness Proof

Theorem 3.4.1: If the scheduler dispatches the jobs according to tables $\mathcal{S}_1$, $\mathcal{S}_2$, ..., $\mathcal{S}_m$, then it will be a correct scheduling strategy.

Proof. We prove the theorem by strong induction.

Let $S(i)$ be the statement "If the scheduler dispatches the jobs according to tables $\mathcal{S}_1$, $\mathcal{S}_2$, ..., $\mathcal{S}_i$, then it will be a correct scheduling strategy up to criticality level i."
BASE STEP ($i = 2$): Since $i = 2$ is a dual criticality instance for which the correctness has already been proved in the previous section, $S(2)$ is true.
INDUCTIVE STEP: Fix some $i \geq 2$, and assume that for every t satisfying $2 \leq t \leq i$, the statement $S(t)$ is true.
Now we need to show that $S(i + 1)$ is true, i.e., if the algorithm finds a correct online scheduling policy up to the i-criticality level using the first i scheduling tables, then there exists an online scheduling policy for $(i+1)$-criticality levels using the first $(i+1)$ tables. We know that $S(i)$ is true which means for the i-criticality level, the scheduler dispatches the job according to the first i tables which is a correct online scheduling strategy for i-criticality levels. Algorithm 5 starts constructing the table $\mathcal{S}_{(i+1)}$ from the table $\mathcal{S}_i$ keeping the same order of the jobs. According to Algorithm 5, after $C_l(i)$ units of execution for each job j_l of $\chi_{(i+1)}$-criticality level is allocated, the remaining $\{C_l(i + 1) - C_l(i)\}$ units of execution has been allocated to them immediately after the rightmost job segment in the table $\mathcal{S}_{(i+1)}$ while following the job order of table $\mathcal{S}_i$. So, each job j_l of $\chi_{(i+1)}$-criticality gets sufficient time to execute their $C_l(i + 1)$ units of execution in $\mathcal{S}_{(i+1)}$. This proof is similar to the dual criticality case. Hence, we get a correct online scheduling policy. □

3.5 Extension for Dependent Jobs

In previous sections we have considered instances with independent jobs. Now we consider the case of dual-criticality instances with dependent jobs. In this section we design algorithms to find two scheduling tables such that if the scheduler discussed in Section 3.2 dispatches the jobs according to these two tables then it will be a correct online scheduling strategy without violating the dependencies between them. To the best of our knowledge, there is no existing algorithm which can schedule the jobs of an instance I with dependencies, although a similar type of problem is discussed in Baruah [Bar14] based on synchronous programs. First we discuss the case of non-recurrent jobs and we then extend it for recurrent or periodic jobs.

3.5.1 Model

A job is characterized by a 5-tuple of parameters: $j_i = (a_i, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}))$, where

- $a_i \in \mathbb{N}$ denotes the *arrival time*.
- $d_i \in \mathbb{N}^+$ denotes the absolute *deadline*.
- $\chi_i \in \{LO, HI\}$ denotes the *criticality* level.
- $C_i(\text{LO}) \in \mathbb{N}^+$ denotes the LO-criticality *worst-case execution time*.
- $C_i(\text{HI}) \in \mathbb{N}^+$ denotes the HI-criticality *worst-case execution time*.

We assume that $\forall i : C_i(\text{LO}) \leq C_i(\text{HI})$, where $1 \leq i \leq n$ and $\chi_i \in \{LO, HI\}$.

An instance of a mixed-criticality system with dependent jobs can be defined as a *directed acyclic graph* (DAG). An instance I is represented in the form of $I(V, E)$, where V represents the set of jobs $\{j_1, j_2, \ldots, j_n\}$ and E represents the dependencies between the jobs. We also assume that no HI-criticality job can depend on a LO-criticality job. This means, there will be no instance where an outward edge from a LO-criticality job becomes an inward edge to a HI-criticality job.

Definition 3.5.1: A dual-criticality MC instance I with job dependencies is said to be **time-triggered schedulable** if it is possible to construct the two schedules $\mathcal{S}_{LO}$ and $\mathcal{S}_{HI}$ for I without violating the dependencies, such that the run-time scheduler algorithm described above schedules I in a correct manner.

3.5.2 The Algorithm

Here we propose an algorithm which can construct two scheduling tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$ for a dual-criticality instance with dependent jobs. If the scheduler discussed in Section 3.2 dispatches job according to these two tables, then this will be a correct scheduling strategy.

We construct the tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$ from two temporary tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. The length of all these tables are D_{max}, i.e., the length of the maximum deadline among all the jobs in the instance.

Algorithm 6 constructs a subgraph Ψ which consists of all the LO-criticality jobs and the edges between them. Then it finds a job j_i with the smallest deadline and no inward edges and allocates its $C_i(\text{LO})$ units of execution in $\mathcal{T}_{\text{LO}}$. After $C_i(\text{LO})$ units of execution of the job is allocated, the job and all its outward edges are removed from Ψ. The process continues until all the jobs in Ψ are scheduled. Then all job segments in $\mathcal{T}_{\text{LO}}$ are shifted as close to their deadlines as possible without violating the dependencies between them so that no job misses their deadline. For an example see Fig. 3.4 and Fig. 3.5

Algorithm 6 Construct-Dependency-$\mathcal{T}_{\text{LO}}(I)$

Input : $I = \{j_1, j_2, ..., j_n\}$, where $j_i =< a_i, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}) >$.
Output : $\mathcal{T}_{\text{LO}}$
Assume earliest arrival time is 0.

1: Find the maximum deadline (D_{max}) of the jobs;
2: Prepare a temporary table $\mathcal{T}_{\text{LO}}$ of maximum length D_{max};
3: Let Ψ be the subgraph of DAG I containing LO-criticality jobs and the edges between them;
4: **repeat**
5: Choose an available job j_i from Ψ with the earliest deadline that does not have an inward edge.
6: Allocate j_i's execution time at the next available slot in the temporary table $\mathcal{T}_{\text{LO}}$;
7: **if** (j_i's $C_i(\text{LO})$ units of execution is allocated) **then**
8: delete j_i and its outward edges from Ψ;
9: **end if**
10: **if** (job j_i misses its deadline) **then**
11: Declare failure and exit;
12: **end if**
13: **until** (all the jobs in Ψ are allocated)
14: Let O be the final order of the jobs of Ψ on the time-line of $\mathcal{T}_{\text{LO}}$ using $C_i(\text{LO})$ units of execution for job j_i;
15: Starting from the rightmost job segment of the order O, move each segment of a job j_i as close to its deadline as possible in $\mathcal{T}_{\text{LO}}$ without violating the dependency;

Algorithm 7 constructs a subgraph Ψ which consists of all the HI-criticality jobs and

the edges between them. Then it finds a job j_i with the smallest deadline and no inward edges and allocates C_i(HI) units of execution to it in $\mathcal{T}_{\text{HI}}$. After C_i(HI) units of execution of the job is allocated, the job and all its outward edges are removed from Ψ. The process continues until all the jobs in Ψ are scheduled. Then all job segments in $\mathcal{T}_{\text{HI}}$ are shifted as close to their deadlines as possible without violating the dependencies between them so that no job miss their deadline. Then out of the total allocation so far, it allocates C_i(LO) units of execution of job j_i in $\mathcal{T}_{\text{HI}}$ from the beginning of its slot and leaves the rest of the execution time of j_i unallocated in $\mathcal{T}_{\text{HI}}$, as in Fig. 3.5 and Fig. 3.6.

Algorithm 7 Construct-Dependency-$\mathcal{T}_{\text{HI}}(I)$

Input : $I = \{j_1, j_2, ..., j_n\}$, where $j_i =< a_i, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}) >$.
Output : $\mathcal{T}_{\text{HI}}$
Assume earliest arrival time is 0.

1: Find the maximum deadline (D_{max}) of the jobs;
2: Prepare a temporary table $\mathcal{T}_{\text{HI}}$ of maximum length D_{max};
3: Let Ψ be the subgraph of DAG I containing HI-criticality jobs and the edges between them;
4: **repeat**
5: Choose an available job j_i from Ψ with the earliest deadline and does not have an inward edge.
6: Allocate j_i's execution time at the next available slot in the temporary table $\mathcal{T}_{\text{HI}}$;
7: **if** (j_i's C_i(HI) units of execution is allocated) **then**
8: delete j_i and its outward edges from Ψ;
9: **end if**
10: **if** (job j_i misses its deadline) **then**
11: Declare failure and exit;
12: **end if**
13: **until** (all the jobs in Ψ are allocated)
14: Let O be the final order of the jobs of Ψ on the time-line of $\mathcal{T}_{\text{HI}}$ using C_i(HI) units of execution for job j_i ;
15: Starting from the rightmost job segment of the order O, move each segment of a job j_i as close to its deadline as possible in $\mathcal{T}_{\text{HI}}$ without violating the dependency.
16: **for** $i := 1$ to m **do**
17: Allocate C_i(LO) units of execution to job j_i from its starting time in $\mathcal{T}_{\text{HI}}$ and leave the rest unallocated;
18: **end for**

Now, we use Algorithm 8 to construct the table $\mathcal{S}_{\text{LO}}$ from $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ and then construct $\mathcal{S}_{\text{HI}}$ from $\mathcal{S}_{\text{LO}}$. The algorithm starts the construction of $\mathcal{S}_{\text{LO}}$ from time 0 and checks the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ simultaneously at each instant. There are four possibilities while merging the two temporary tables to construct $\mathcal{S}_{\text{LO}}$.

At time t, one of the following situations can occur.

1. Both $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ are empty.
2. Both $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ are not empty.
3. $\mathcal{T}_{\text{LO}}$ is empty and $\mathcal{T}_{\text{HI}}$ is not empty.
4. $\mathcal{T}_{\text{LO}}$ is not empty and $\mathcal{T}_{\text{HI}}$ is empty.

If situation 1 occurs, then the algorithm will search both the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ to find the first available job in both the table. Then, it allocates one of the available jobs at time t where a LO-criticality job gets higher priority over a HI-criticality job. If a LO-criticality job is chosen to be allocated, then all the predecessor of that job must be finished allocation. Then the place of the ready job in $\mathcal{T}_{\text{LO}}$ or $\mathcal{T}_{\text{HI}}$ is marked as empty. In case of situation 2, the algorithm declares failure to schedule. In situation 3, the algorithm allocates the HI-criticality job from $\mathcal{T}_{\text{HI}}$ whereas in situation 4, it allocates the LO-criticality job from $\mathcal{T}_{\text{LO}}$ if and only if all the predecessor of the job has already finished allocation. Once an instant of a job is allocated in $\mathcal{S}_{\text{LO}}$, the place where it was scheduled in $\mathcal{T}_{\text{LO}}$ or $\mathcal{T}_{\text{HI}}$ is emptied.

We then construct the table $\mathcal{S}_{\text{HI}}$ from $\mathcal{S}_{\text{LO}}$. We first copy the jobs of table $\mathcal{S}_{\text{LO}}$ to $\mathcal{S}_{\text{HI}}$. Then the HI-criticality jobs are allocated their $C_i(\text{HI}) - C_i(\text{LO})$ units of HI-criticality execution time immediately after their $C_i(\text{LO})$ units of execution in $\mathcal{S}_{\text{HI}}$. These additional time units are allocated by recursively pushing all overlapping HI-criticality jobs in $\mathcal{S}_{\text{HI}}$ to the right and overwriting any LO-criticality job in the process. An exception to this is when the allocation time of an overlapping HI-criticality job is the same in both the tables $\mathcal{S}_{\text{HI}}$ and $\mathcal{T}_{\text{HI}}$, in which case the additional time units are allocated after this job without violating the dependency constraints.

We illustrate the algorithm by an example.

Example 3.5.1: Consider the instance shown in Fig. 3.14. This is an instance with five jobs j_1, j_2, j_3, j_4 and j_5 with dependencies between them. The properties of these jobs can be seen from Table 3.4.

We find the two temporary tables as in Example 3.3.1. But, here we need to take care of the job dependencies. So, we apply Algorithm 6 and 7 to construct the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ shown in Fig. 3.15 and 3.16.

We then use Algorithm 8 to find the table $\mathcal{S}_{\text{LO}}$ which is shown in Fig. 3.17. Finally, we construct the table $\mathcal{S}_{\text{HI}}$ from the table $\mathcal{S}_{\text{LO}}$ which is shown in Fig. 3.18. □

Algorithm 8 TT-Merge-DEP($I, \mathcal{T}_{\text{LO}}, \mathcal{T}_{\text{HI}}$)

Input : $\mathcal{T}_{\text{LO}}, \mathcal{T}_{\text{HI}}$ and $I = \{j_1, j_2, ..., j_n\}$, where $j_i =< a_i, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}) >$.
Output : Tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$

Construction of $\mathcal{S}_{\text{LO}}$.
Find the maximum deadline (D_{max}) of the jobs;
The maximum length of tables $\mathcal{S}_{\text{HI}}$ and $\mathcal{S}_{\text{LO}}$ are both D_{max};
$t := 0$;
while ($t \leq L$) **do**
  if ($\mathcal{T}_{\text{LO}}[t] = NULL$ & $\mathcal{T}_{\text{HI}}[t] = NULL$) **then**
    Search the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ simultaneously from the beginning to find the first available job at time t;
    Let k be the first occurrence of a job j_i in $\mathcal{T}_{\text{LO}}$ or $\mathcal{T}_{\text{HI}}$;
    if (Both LO-criticality & HI-criticality job are found) **then**
      if (Predecessors of $\mathcal{T}_{\text{LO}}[k]$ has been allocated its C_i(LO) execution time) **then**
        $\mathcal{S}_{\text{LO}}[t] := \mathcal{T}_{\text{LO}}[k]$;
        $\mathcal{T}_{\text{LO}}[k] := NULL$;
      else
        $\mathcal{S}_{\text{LO}}[t] := \mathcal{T}_{\text{HI}}[k]$;
        $\mathcal{T}_{\text{HI}}[k] := NULL$;
      end if
    else if (LO-critical job is found) **then**
      if (Predecessors of $\mathcal{T}_{\text{LO}}[k]$ has been allocated its C_i(LO) execution time) **then**
        $\mathcal{S}_{\text{LO}}[t] := \mathcal{T}_{\text{LO}}[k]$;
        $\mathcal{T}_{\text{LO}}[k] := NULL$;
      else
        $\mathcal{S}_{\text{LO}}[t] := NULL$;
        $t := t + 1$;
      end if
    else if (HI-criticality job is found) **then**
      $\mathcal{S}_{\text{LO}}[t] := \mathcal{T}_{\text{HI}}[k]$;
      $\mathcal{T}_{\text{HI}}[k] := NULL$;
    else if (NO job is found) **then**
      $\mathcal{S}_{\text{LO}}[t] := NULL$
      $t := t + 1$;
    end if
  else if ($\mathcal{T}_{\text{LO}}[t] = NULL$ & $\mathcal{T}_{\text{HI}}[t]$!= $NULL$) **then**
    $\mathcal{S}_{\text{LO}}[t] := \mathcal{T}_{\text{HI}}[t]$;
    $\mathcal{T}_{\text{HI}}[t] := NULL$;
    $t := t + 1$;
  else if ($\mathcal{T}_{\text{LO}}[t]$!= $NULL$ & $\mathcal{T}_{\text{HI}}[t] = NULL$) **then**
    $\mathcal{S}_{\text{LO}}[t] := \mathcal{T}_{\text{LO}}[t]$;
    $\mathcal{T}_{\text{LO}}[t] := NULL$;
    $t := t + 1$;
  else if ($\mathcal{T}_{\text{LO}}[t]$!= $NULL$ & $\mathcal{T}_{\text{HI}}[t]$!= $NULL$) **then**
    Declare failure;
  end if
end while
This is the table $\mathcal{S}_{\text{LO}}$;

Construction of $\mathcal{S}_{\text{HI}}$
Copy all the jobs from table $\mathcal{S}_{\text{LO}}$ to table $\mathcal{S}_{\text{HI}}$;
Scan the table $\mathcal{S}_{\text{HI}}$ from left to right:
for each HI-criticality job j_i, allocate an additional $C_i(\text{HI}) - C_i(\text{LO})$ time units immediately after the rightmost segment of job j_i, repeatedly pushing all the following HI-criticality jobs in $\mathcal{S}_{\text{HI}}$ (except those whose allocation time is same as in $\mathcal{T}_{\text{HI}}$) to the right and overwriting any LO-criticality jobs in the process.

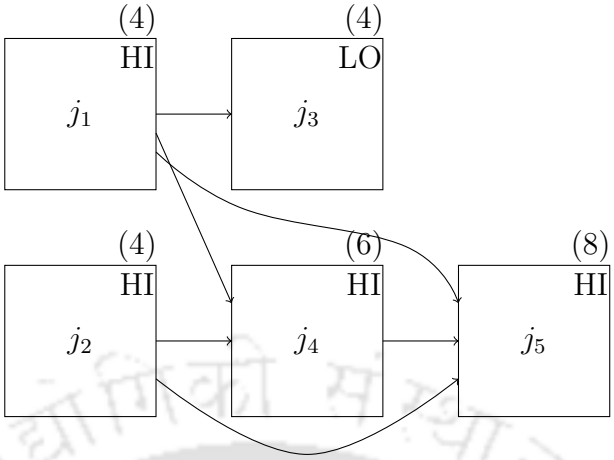


Figure 3.14: A DAG showing job dependencies. The numbers in parentheses indicates deadline

Table 3.4: An example instance to explain the TT-Merge-DEP algorithm

Job	Arrival time	Deadline	Criticality	C_i(LO)	C_i(HI)
j_1	0	4	HI	1	2
j_2	0	4	HI	1	2
j_3	0	4	LO	1	1
j_4	0	6	HI	1	2
j_5	3	8	HI	1	2

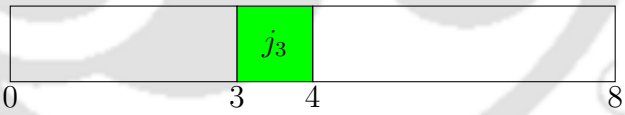


Figure 3.15: Temporary table $\mathcal{T}_{\text{LO}}$

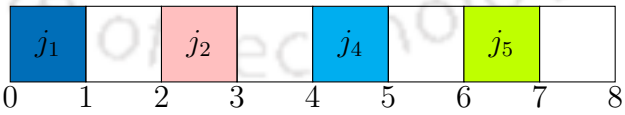


Figure 3.16: Temporary table $\mathcal{T}_{\text{HI}}$

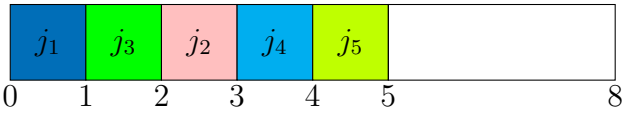


Figure 3.17: Final table $\mathcal{S}_{\text{LO}}$

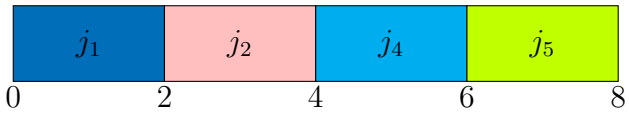


Figure 3.18: Final table $\mathcal{S}_{\text{HI}}$

3.5.3 Correctness Proof

We need to show that if TT-Merge-DEP finds the two tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$, then the scheduler can find an online scheduling strategy using these tables.

Lemma 3.5.1: If Algorithm 8 does not declare failure, then each job j_i receives $C_i(\text{LO})$ units of execution in $\mathcal{S}_{\text{LO}}$ and each HI-criticality job j_k receives $C_k(\text{HI})$ units of execution in $\mathcal{S}_{\text{HI}}$ without violating the dependency constraints.

Proof. The table $\mathcal{S}_{\text{LO}}$ is constructed from the two temporary tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. Each LO-criticality job j_i can be allocated in $\mathcal{S}_{\text{LO}}$ on or before its scheduled time instant in $\mathcal{T}_{\text{LO}}$ if and only if all of its predecessor jobs have completed their allocation, which does not violate the dependencies. We have already assumed that no HI-criticality job depends on any LO-criticality job. We know that each job in $\mathcal{T}_{\text{HI}}$ is allocated according to its dependency constraints. So each HI-criticality job j_i can be allocated in $\mathcal{S}_{\text{LO}}$ on or before its scheduled time instant in $\mathcal{T}_{\text{HI}}$. If TT-Merge-DEP finds a table $\mathcal{S}_{\text{LO}}$ then each job must receives $C_i(\text{LO})$ units of execution time.

Next we show that any HI-criticality job j_k receives $C_k(\text{HI})$ units of execution in $\mathcal{S}_{\text{HI}}$. We start constructing $\mathcal{S}_{\text{HI}}$ by copying the jobs in $\mathcal{S}_{\text{LO}}$. But according to TT-Merge-DEP, the HI-criticality jobs are allocated their remaining $C_k(\text{HI}) - C_k(\text{LO})$ units of allocation in $\mathcal{S}_{\text{HI}}$ after they complete their $C_k(\text{LO})$ units of allocation in $\mathcal{S}_{\text{HI}}$ by pushing recursively all the following HI-criticality job segments to the right except those whose allocation is the same as in table $\mathcal{T}_{\text{HI}}$ and without violating the dependency constraints. This means we can push a job segment to the right in $\mathcal{S}_{\text{HI}}$ only if it is allocated before its allocation in $\mathcal{T}_{\text{HI}}$ and, moreover, no job is pushed beyond its allocation in $\mathcal{T}_{\text{HI}}$, because if $\mathcal{T}_{\text{HI}}$ does not declare failure then it allocates enough time for the execution of all the HI-criticality jobs without violating the dependency constraints. In this case, all the jobs can get sufficient time to schedule their $C_k(\text{HI}) - C_k(\text{LO})$ units of execution as they are allocated on or before the allocation in table $\mathcal{T}_{\text{HI}}$. This is clear from the remark following Observation 1 which holds for dependent jobs as well. If a HI-criticality job j_h cannot be pushed to the right then it

will get its remaining $C_h(\text{HI}) - C_h(\text{LO})$ units of execution time in table $\mathcal{S}_{\text{HI}}$ by a similar reasoning as above. □

Theorem 3.5.1: If the scheduler dispatches the jobs according to $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$, then it will be a correct scheduling strategy without violating the dependency constraints.

Proof. Algorithms 6 and 7 take care of all the dependencies between LO-criticality and HI-criticality jobs respectively. We know that Algorithm 8 checks the dependencies of the LO-criticality jobs on HI-criticality jobs before allocating the LO-criticality jobs. We have assumed that no HI-criticality job depends on a LO-criticality job. So the construction of the tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$ does not violate the dependency constraints of instance I. From Lemma 3.5.1, it is clear that each job in $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$ receives $C(\text{LO})$ and $C(\text{HI})$ units of execution respectively. The rest of the proof is similar to that of Theorem 3.3.1. □

3.5.4 Generalizing the Algorithm for m Criticality Levels

We know that Algorithm 8 can find two tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$ which can be used by the scheduler for correct online scheduling policy. In Section 3.4, we have already proved that the dual-criticality algorithm can be modified to find m number of tables which can be used by the scheduler to find a correct online scheduling strategy for m criticality levels. Thus, we can say that the algorithm discussed in this section can be extended to find m tables which can be used by the scheduler to find a correct online scheduling strategy.

3.6 Extension for Periodic Jobs

Now we extend TT-Merge for periodic or recurrent jobs. Here, a job is characterized by a 5-tuple of parameters: $j_i = (a_i, p_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}))$, where

- $a_i \in \mathbb{N}$ denotes the *arrival time*.
- $p_i \in \mathbb{N}^+$ denotes the *period*.
- $\chi_i \in \{LO, HI\}$ denotes the *criticality* level.
- $C_i(\text{LO}) \in \mathbb{N}^+$ denotes the LO-criticality *worst-case execution time*.
- $C_i(\text{HI}) \in \mathbb{N}^+$ denotes the HI-criticality *worst-case execution time*.

We assume that $\forall i : C_i(\text{LO}) \leq C_i(\text{HI})$, where $1 \leq i \leq n$ and $\chi_i \in \{LO, HI\}$. Note that in this chapter, we also assume that $p_i = d_i$, where d_i is the deadline and $1 \leq i \leq n$.

As we can see that the job model is very much similar to the non-recurrent jobs except the periods which initiate the new instance of the job. The process of constructing a time-triggered schedule for the jobs having the above dual-criticality model will be very similar to the one we have discussed in Section 3.3. Here we follow the same algorithms as in Section 3.3 to find the two tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$. These two tables will be used by the scheduler to dispatch the jobs at each instant of time.

All algorithms discussed in Section 3.3 constructed the tables of length D_{max}. But, in this case, all the tables will have length equals to the lcm or hyper-period L of periods of all the jobs. Here we need to modify Algorithms 1 and 2 only. We find the EDF order of the LO-criticality and HI-criticality jobs up to the hyper-period L in the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ respectively. We then can use Algorithm 3 to find tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$.

3.7 Comparison with Mixed-criticality Synchronous Programs

Baruah [Bar14] proposed a technique to schedule mixed-criticality synchronous programs on a uniprocessor system. He showed that the scheduling of mixed-criticality *single-rate* synchronous program is polynomial time solvable whereas the optimal and efficient scheduling of mixed-criticality *multi-rate* synchronous program is NP-hard in the strong sense. He also proved that the schedule generation algorithm which finds a schedule for single-rate synchronous programs is optimal. Baruah used graphs to represent the reactive blocks and their dependencies. The multi-rate graph of a synchronous program is unrolled to find a *directed acyclic graph (DAG)* in which each invocation of each block within an interval, of length equal to the lcm of the periods, is explicitly represented as a separate node. Each node is then assigned a priority according to the OCBP algorithm. From the above priorities, two tables can be constructed which can be used by the scheduler to dispatch the blocks. We present an algorithm which can construct two tables with which we can schedule a strict superset of OCBP-schedulable mixed-criticality multi-rate programs.

3.7.1 Model

We follow the same model of synchronous program as suggested in [Bar14]. The model is described as follows.

- The synchronous program is represented as a *directed acyclic graph* $G(V, E)$, where V is the set of vertices and E is the set of edges. The blocks $(B_1, \ldots, B_n)$ of the synchronous program are represented as the vertices of the graphs, i.e., $B_i \in V$. The dependencies between the blocks (B_i, B_j) is represented by the directed edges, i.e., $(B_i, B_j) \in E$.
- Some of the blocks are designated as *output* blocks and *input* blocks; these generate output and input values of the synchronous program. Other blocks are called *internal* blocks.
- Each block is characterized by a 5-tuple of parameters: $B_i = (a_i, p_i, \chi_i, C_i(LO), C_i(HI))$, where
  - $a_i \in \mathbb{N}$ denotes the *arrival time*.
  - $p_i \in \mathbb{N}^+$ denotes the *period*.
  - $\chi_i \in \{LO, HI\}$ denotes the *criticality* level.
  - $C_i(\text{LO}) \in \mathbb{N}^+$ denotes the LO-criticality *worst-case execution time*.
  - $C_i(\text{HI}) \in \mathbb{N}^+$ denotes the HI-criticality *worst-case execution time*.
- We assume that $\forall i : C_i(\text{LO}) \leq C_i(\text{HI})$, where $1 \leq i \leq n$ and $\chi_i \in \{LO, HI\}$. Note that in this chapter, we also assume that $p_i = d_i$, where d_i is the deadline.
- Each output block can either be a HI-criticality or a LO-criticality block. We assume that a HI-criticality block cannot depend upon a LO-criticality block. This means if a block B_i is a HI-criticality block, then all the preceding blocks of B_i will be HI-criticality blocks.

As discussed earlier, the CAs are interested in the certification of the values of the HI-criticality output blocks only, whereas the system designers want to verify the correctness of all the blocks in a synchronous program.

Table 3.5: An example instance to explain the application of our algorithm on synchronous reactive systems

Block	Arrival time	Period	Criticality	C_i(LO)	C_i(HI)
B_1	0	14	HI	3	5
B_2	2	14	HI	1	2
B_3	0	7	LO	3	3
B_4	0	14	HI	3	7

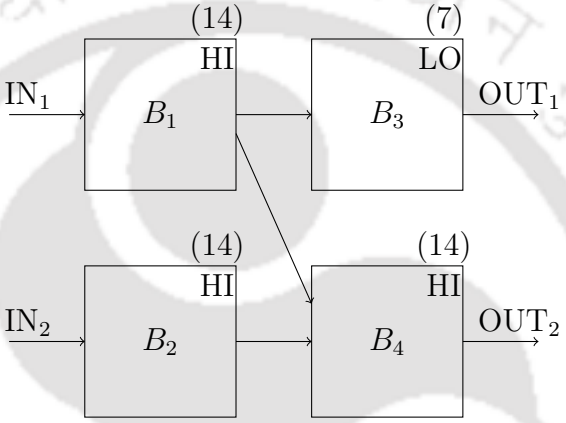


Figure 3.19: DAG of instance I given in Table 3.5

Example 3.7.1: Let us consider an instance I given in Table 3.5 and its corresponding DAG in Fig. 3.19. Since we are considering a periodic instance, the instance I is unrolled according to the method given in [Bar14]. The resulting DAG is given in Fig. 3.20. We try to apply the OCBP algorithm to find the priority from which the tables S^{oc}_{LO} and S^{oc}_{HI} are constructed. As the procedure shown in [Bar14], the blocks $B_3(1)$ is chosen to be assigned the lowest priority block. Since $B_3(1)$ is a LO-criticality block, we need to consider C(LO) units of execution of each block. Now we can see that block B_1 can execute over $[0, 3]$, block $B_3(0)$ can execute over $[3, 6]$, block B_2 can execute over $[6, 7]$ and block B_4 can execute over $[7, 10]$. So there is sufficient time for $B_3(1)$ to execute its three units of execution. Thus, block $B_3(1)$ is assigned lowest priority. Now, we can see that no more blocks can be assigned a priority. Since, there is no OCBP priority order, the algorithm discussed in [Bar14] cannot construct the two scheduling tables S^{oc}_{LO} and S^{oc}_{HI}.

Now we apply our algorithm 8 on the synchronous program given in Example 3.7.1 to construct the two scheduling tables $\mathcal{S}_{LO}$ and $\mathcal{S}_{HI}$. We consider the unrolled synchronous

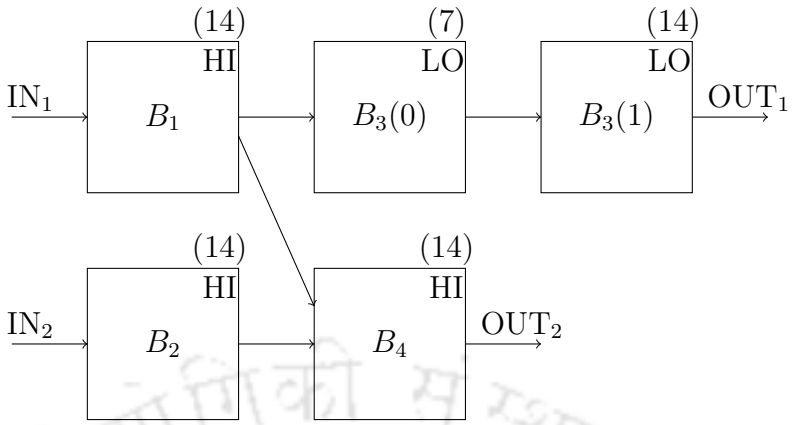


Figure 3.20: DAG after unroll

program given in Fig. 3.20 to find the scheduling tables. As we know, we need two temporary tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ to construct the scheduling table $\mathcal{S}_{\text{LO}}$. Then $\mathcal{S}_{\text{HI}}$ will be constructed using $\mathcal{S}_{\text{LO}}$.

First, Algorithm 6 and 7 construct the two temporary tables as shown in Fig. 3.21. Then Algorithm 8 constructs the table $\mathcal{S}_{\text{LO}}$ as shown in Fig. 3.22 from which the table $\mathcal{S}_{\text{HI}}$

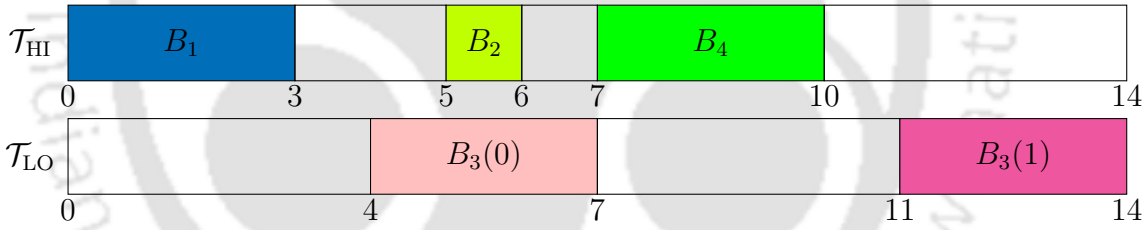


Figure 3.21: Tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$

is constructed as shown in Fig. 3.23.

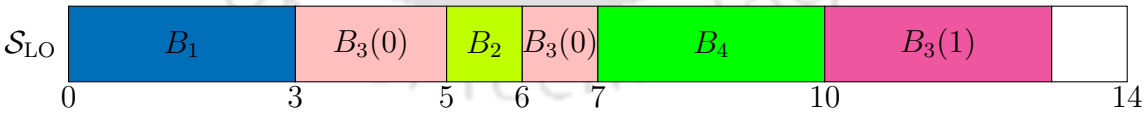


Figure 3.22: Table $\mathcal{S}_{\text{LO}}$

We follow all the Lemmas from Section 3.3.4 to prove Theorem 3.7.1.

Theorem 3.7.1: If a mixed-criticality synchronous program is schedulable by the OCBP-based algorithm, then it is also schedulable by our algorithm.

Proof. The OCBP algorithm generates priority orders for the synchronous programs. Then the OCBP-based algorithm finds tables $\text{S}^{\text{oc}}_{\text{LO}}$ and $\text{S}^{\text{oc}}_{\text{HI}}$ for the synchronous programs using this

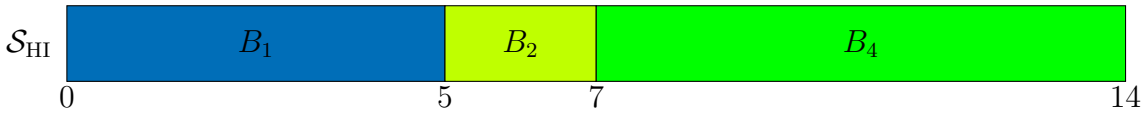


Figure 3.23: Table $\mathcal{S}_{\text{HI}}$

priority order. We need to show that if the OCBP-based algorithm constructs the tables $\text{S}^{\text{oc}}_{\text{LO}}$ and $\text{S}^{\text{oc}}_{\text{HI}}$ for an instance, then our algorithm will not encounter a situation where at time slot t the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ are non-empty, for any t.

We know that $C_i(\text{LO})$ units of execution is allocated to each block B_i for constructing the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. Each block in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ is allocated as close to its deadline as possible without violating the dependency constraints. That means no block can execute after its allocation time in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ without affecting the schedule of any other block and still meet its deadline. Algorithm 8 never allocates a block in $\mathcal{S}_{\text{LO}}$ whose predecessors have not completed its $C(\text{LO})$ units of execution in $\mathcal{S}_{\text{LO}}$. Because, Algorithm 6 and 7 take care of the dependencies between the LO-criticality and HI-criticality blocks respectively and Algorithm 8 takes care the dependencies of a LO-criticality block on HI-criticality block. Algorithm 8 declares failure if it finds a non-empty instant at any time t in both the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. This means the two blocks which are found in the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ respectively cannot be scheduled with all other remaining blocks from this point, because all the blocks to the right have already been moved as far to the right as possible.

Let there be an OCBP priority order of the blocks of synchronous program and a table S_{LO} according to this priority order.

Let B_l and B_h be two blocks in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ respectively found at time t during the construction of $\mathcal{S}_{\text{LO}}$, by our algorithm which means all job segments in the interval $[0, t-1]$ from $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ have already been assigned in $\mathcal{S}_{\text{LO}}$. But, we know that OCBP has assigned priorities to these blocks B_l and B_h. Now there are two cases.

In the first case, assume B_l is assigned lower priority than B_h by OCBP. Let a_l be the arrival time of B_l and the starting and completion times of B_l in $\mathcal{T}_{\text{LO}}$ be t_l and t_l' respectively. Since block B_l can be scheduled only on or after the arrival time a_l, we need to show that the block segment of B_l found at time t cannot be scheduled in the interval $[a_l, t-1]$ by the OCBP-based algorithm. We know that Algorithm 8 can allocate a block in table $\mathcal{S}_{\text{LO}}$ on or before its allocation in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ without violating the dependency constraints. But Algorithm 8 has not allocated the block segments found in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ at time t in

the interval $[a_l, t-1]$ of the table $\mathcal{S}_{\text{LO}}$, and by Lemma 3.3.6, this is due to the presence of equal or higher priority block segments of the OCBP priority order in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. We know that all the blocks in $\mathcal{T}_{\text{LO}}$ in the interval $[a_l, t]$ and the blocks in $\mathcal{T}_{\text{HI}}$ including block B_h in the interval $[a_l, t]$ are of priority greater or equal to that of B_l according to OCBP since, by moving block segments to the right starting from the OCBP schedule the blocks to the left of B_l are of priority greater or equal to that of B_l. This means the blocks in the interval $[a_l, t-1]$ of table $\mathcal{S}_{\text{LO}}$ are either equal or higher priority blocks than B_l according to OCBP. So both the algorithms, the OCBP-based one and ours, allocate higher or equal priority jobs (or, block segments according to Algorithm 8) before time t. Then it is clear that after the blocks of higher priority than B_l finish their $C(LO)$ units of execution, there will not be sufficient time for B_l to finish its $C_l(\text{LO})$ units of execution in the interval $[a_l, t_l']$ in the OCBP schedule. This is because at time t, the OCBP-based algorithm has already allocated all ready blocks with higher or equal priority than B_l (according to OCBP) in the interval $[a_l, t]$ with no vacant slot for further allocation of B_l's segment found at time slot t which is the case for Algorithm 8 as well. A similar statement holds for B_h. Therefore B_h and B_l cannot be simultaneously scheduled to meet their deadlines in the remaining time, according to the OCBP-based algorithm.

In the second case, assume B_h is assigned lower priority than B_l by OCBP. Let a_h be the arrival time of block B_h and let the starting and completion times of the LO-criticality execution of B_h be t_h and t_h', respectively and the completion time of the HI-criticality execution be t_e. As in the previous case, all the blocks in $\mathcal{T}_{\text{HI}}$ in the interval $[a_h, t]$ and the blocks in $\mathcal{T}_{\text{LO}}$, including block B_l, in the interval $[a_h, t]$ are of priority (according to OCBP) greater than or equal to that of B_h. OCBP considers $C(\text{HI})$ units of execution time to assign a priority to a HI-criticality block. As seen above, it is clear that after the blocks of higher priority than B_h finish their $C(\text{LO})$ units of execution, there will not be sufficient time for B_h to finish its $C_h(\text{LO})$ units of execution in the interval $[a_h, t_h']$ according to OCBP. We know that $C(\text{LO}) \leq C(\text{HI})$. If block B_h does not get sufficient time to execute its $C_h(\text{LO})$ units of execution in the interval $[a_h, t_h']$, then it will not get sufficient time to execute its $C_h(\text{HI})$ units of execution in the interval $[a_h, t_e]$ either.

From the above two cases, it is clear that OCBP cannot assign priorities to job B_l and B_h, which is a contradiction. This means if there exists an OCBP priority order for a synchronous program, then our algorithm will not encounter a situation where both the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ are non-empty at any time t for the same synchronous program.

Note that we need to consider only the LO-criticality scenarios since Lemma 3.3.3 implies that if $\mathcal{S}_{\text{LO}}$ can be constructed, then so can $\mathcal{S}_{\text{HI}}$. □

3.8 Results and Discussion

In this section we present the experiments conducted to evaluate TT-Merge for the dual-criticality case for non-recurrent jobs (Algorithm 3). The experiments show the impact of utilization on TT-Merge versus the OCBP-based and MCEDF algorithms. The comparison is done over a large number of instances with randomly generated parameters.
The job generation policy may have significant effect on the experiments. The details of the job generation policy are as follows.

- The utilization (u_i) of the jobs of instance I are generated according to the UUniFast algorithm [BB05].
- We use the exponential distribution proposed by Davis *et al* [DZB08] to generate the deadline (d_i) of the jobs of instance I.
- The C_i(LO) units of execution time of the jobs are calculated as $u_i \times d_i$.
- The C_i(HI) units of execution time of the jobs are calculated as $C_i(\text{HI}) = \text{CF} \times C_i(\text{LO})$ where CF is the criticality factor which varies between 2 and 6 for each HI-criticality job j_i.
- Each instance I contains at least one HI-criticality job and one LO-criticality job.
- For each point on the X-axis, we have plotted the average result of 10 runs.

In the first experiment, we fix the utilization at LO-criticality level of each instance at 0.9 and let the deadline of the jobs vary between 1 and 2000. The number of jobs in each instance is set to 10. The graph in Fig. 3.24 shows the number of schedulable instances out of different numbers of randomly generated instances.
From the graph in Fig. 3.24, it is clear that TT-Merge schedules more instances successfully than both the OCBP-based algorithm and the MCEDF algorithm. As can be seen from Fig. 3.24, for an utilization of 0.9 about 620 instances out of 1000 instances are successfully scheduled by TT-Merge which is two times more than the OCBP-based algorithm and 1.25

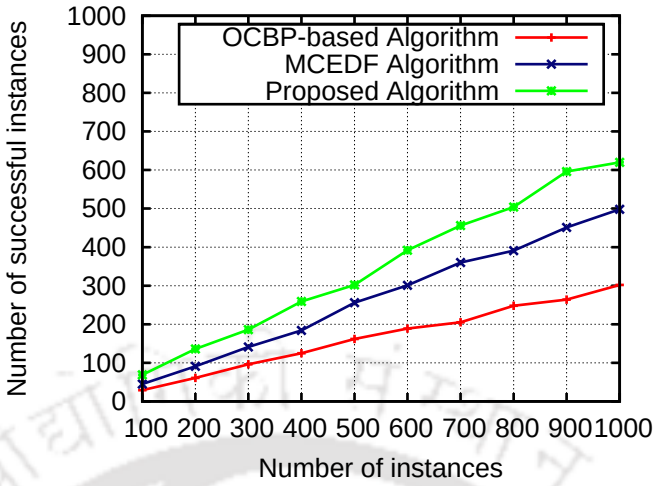


Figure 3.24: Comparison of number of MC-schedulable instances at an utilization of 0.9

times more than the MCEDF algorithm. As the number of instances increases, the success ratio is more or less stable.

The next experiment checks the impact of the utilizations on the schedulable instances. Here the number of jobs in an instance is fixed at 20. The deadlines of the jobs in an instance range between 1 and 2000. The utilizations at LO-criticality level of the instances are varied between 0.1 and 0.9. The graph in Fig. 3.25 shows the number of schedulable instances from 1000 randomly generated instances.

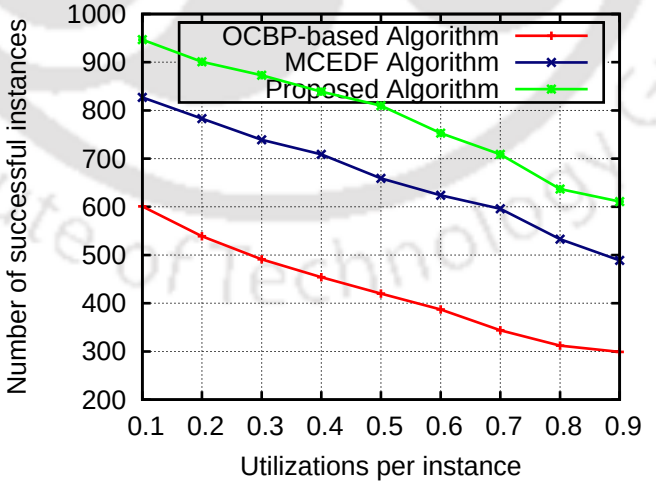


Figure 3.25: Comparison of number of MC-schedulable instances with different utilizations

From the graph, it is clear that TT-Merge constructs more tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$ successfully than the OCBP-based scheduling algorithm. We can see that TT-Merge also

schedules more instances than MCEDF by a factor of 1.25. Typically TT-Merge is successful in scheduling twice the number of instances than the OCBP-based algorithm. We can see that the number of schedulable instances decrease with the increase in the utilization.

We have done another experiment where the number of jobs in an instance varied between 5 and 100. For this experiment, we fix the utilization at LO-criticality level of each instance at 0.9 and let the deadline of the jobs vary between 1 and 2000. We plot the result from 1000 randomly generated instances in Fig. 3.26.

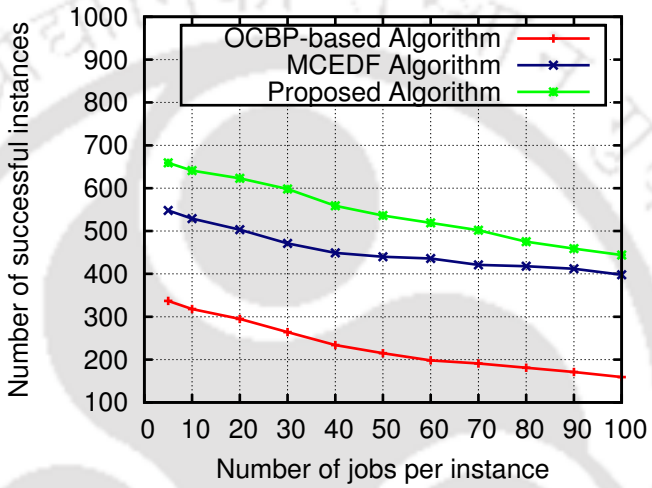


Figure 3.26: Comparison of number of MC-schedulable instances with different number of jobs per instance

From the graph in Fig. 3.26 it is clear that TT-Merge successfully schedules significantly more (by a factor of two) instances successfully than the OCBP-based algorithm and also schedules more instances than the MCEDF algorithm.

3.9 Conclusion

In this chapter, we proposed a new algorithm for the time-triggered scheduling of mixed-criticality systems. We proved that our algorithm can schedule a bigger set of instances than the previous algorithm based on OCBP. We also showed that our algorithm schedules more instances than MCEDF. The experiments show the differences in number of schedulable instances between our algorithm and the OCBP-based algorithm and MCEDF. We have also extended the work to handle periodic and dependent jobs. Finally, we proved that our algorithm for dependent jobs can be used to schedule the blocks of a synchronous program

and for which it schedules a bigger set of instances than the algorithm based on OCBP.

In the next chapter, we plan to investigate the non-functional properties of mixed-criticality systems, e.g., energy consumption. We extend the TT-Merge algorithm for the optimization of the energy consumption of mixed-criticality systems.

Chapter 4

Energy-efficient Time-triggered Scheduling of Uniprocessor Mixed-criticality Systems

4.1 Introduction

Besides schedulability, researchers are beginning to look at various other aspects of mixed-criticality systems, such as energy consumption minimization. In the energy consumption minimization problem, task executions are slowed down by using dynamic voltage and frequency scaling (DVFS) and/or dynamic power management (DPM) such that the system energy consumption is minimized without affecting the mixed-criticality schedulability requirement. The energy consumption minimization problem is as hard as the mixed-criticality scheduling problem which has been proved to be NP-hard in the strong sense. This is easy to see, as the problem of finding minimized energy schedule for a mixed-criticality system with just one processor frequency is the same as a general mixed-criticality scheduling problem.

In Chapter 3, we proposed a time-triggered scheduling algorithm for a uniprocessor mixed-criticality real-time system. Now we investigate the energy consumption minimization problem with respect to the algorithm of Chapter 3. The work closest to ours is [HKGT14], where the proposed method is based on EDF-VD [BBD+12b] and successful only if the task set is schedulable by EDF-VD. We try to find a method which can schedule more number of task sets compared to [HKGT14] as well as minimize the energy consumption. Our method is

to integrate the mixed-criticality energy-efficient problem with the time-triggered scheduling algorithm TT-Merge of Chapter 3. As in [HKGT14], we consider minimizing the energy consumption only in the LO-criticality scenarios as the probability of occurrence of a HI-criticality scenario is extremely low from the designer's viewpoint. The energy optimization problem for HI-criticality scenarios is part of future work. We show that our algorithm outperforms that of [HKGT14, NHG+16], the predominant existing algorithm which uses DVFS for mixed-criticality systems with respect to minimization of energy consumption. We then prove the optimality of the proposed algorithm with respect to energy consumption minimization for scheduler produced by the TT-Merge algorithm. Ours is the first energy-efficient time-triggered algorithm for scheduling of mixed-criticality systems. Extending our algorithm to multiprocessor mixed-criticality systems is part of future work.

The rest of the chapter is organized as follows: Section 4.2 describes the system model and definitions. In Section 4.3, we formulate the energy consumption minimization problem for mixed-criticality systems. Section 4.4 presents a new algorithm which finds the processor frequency and the execution time to be taken after DVFS for each job. We extend the proposed algorithm for dependent jobs in Section 4.5. Section 4.6 presents the evaluation of the proposed algorithm. Section 4.7 concludes the chapter.

4.2 System Model and Literature Survey

4.2.1 Mixed-criticality Task Model

In this chapter, we consider periodic mixed-criticality task systems. A mixed-criticality (MC) periodic task system $\mathscr{T}$ consists of a number of tasks $\tau_1, \ldots, \tau_n$. A task τ_i is characterized by a 4-tuple $(\chi_i, c_i(\text{LO}), c_i(\text{HI}), p_i)$, where

- $\chi_i \in \{LO, HI\}$ denotes the *criticality* level.
- $C_i(\text{LO}) \in \mathbb{N}^+$ denotes the LO-criticality *worst-case execution time*.
- $C_i(\text{HI}) \in \mathbb{N}^+$ denotes the HI-criticality *worst-case execution time*.
- $p_i \in \mathbb{N}^+$ denotes the *period*.

We assume that $C_i(\text{LO}) \leq C_i(\text{HI})$ for all tasks τ_i and the deadline of each task is the same as its period. Each of these tasks may generate an unbounded number of dual-

criticality jobs, either of LO-criticality or HI-criticality. A job j_{ik} of task τ_i is characterized by a 5-tuple of parameters: $j_{ik} = (a_{ik}, d_{ik}, \chi_i, C_i(\text{LO}), C_i(\text{HI}))$, where

- $a_{ik} \in \mathbb{N}$ denotes the *arrival time*, $a_{ik} \geq 0$.
- $d_{ik} \in \mathbb{N}^+$ denotes the *relative deadline*, $d_{ik} = p_i$.

We assume that the system is *preemptive*. Generally, a job in the task set is available for execution at time a_{ik} and should finish its execution before $a_{ik} + d_{ik}$. The job j_{ik} must execute for c_i amount of time which is the actual execution time between a_{ik} and $a_{ik} + d_{ik}$, but this can be known only at the time of execution. Now we define the schedulability condition for a mixed-criticality task set.

Definition 4.2.1: A scheduling strategy is *feasible or correct* if and only if the following conditions are true:

1. If all the jobs finish their $C_i(\text{LO})$ units of execution time on or before their deadlines.
2. If any job does not declare its completion after executing its $C_i(\text{LO})$ units of execution time, then all the HI-criticality jobs must finish their $C_i(\text{HI})$ units of execution time on or before their deadlines.

Here we focus on **time-triggered schedules** [BF11] of the MC task set. Two time-triggered scheduling tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$ are constructed for a given task set. These tables are used to schedule the task set at run time. The length of the tables is the length of the least common multiple (lcm) of the periods of the task set. The rules to use the tables $\mathcal{S}_{\text{HI}}$ and $\mathcal{S}_{\text{LO}}$ at run time, (i.e., the *scheduler*) are as follows:

- The criticality level indicator Γ is initialized to LO.
- While (Γ = LO), at each time instant t the job available at time t in the table $\mathcal{S}_{\text{LO}}$ will execute.
- If a job executes for more than its LO-criticality WCET without signaling completion, then Γ is changed to HI.
- While (Γ = HI), at each time instant t the job available at time t in the table $\mathcal{S}_{\text{HI}}$ will execute.

Definition 4.2.2: A dual-criticality MC task set is said to be **time-triggered schedulable** [BF11] if it is possible to construct the two schedules $\mathcal{S}_{\text{HI}}$ and $\mathcal{S}_{\text{LO}}$ for $\mathscr{T}$, such that the run-time scheduler algorithm described above schedules $\mathscr{T}$ in a correct manner.

4.2.2 Power Model and DVFS

Here we consider the state-of-the-art power model [CK07, PC14, ZMM04]:

$$P(f) = P_s + P_d(f) = P_s + \beta.f^{\alpha} \tag{4.1}$$

where f is the processor frequency, $P(f)$ is the power consumption at frequency f, P_s is the static power consumption due to leakage current, and P_d denotes the frequency-dependent active power. The quantity β is a circuit dependent positive constant and $\alpha \geq 2$ depends on the hardware. Since $\alpha \geq 2$, the power consumption is a convex increasing function of the processor frequency.

Since our target is to minimize energy consumption due to P_d by DVFS, we ignore the effect of the static power P_s. We also assume that the system runs at a base frequency f_b, $f_{\min} \leq f_b \leq f_{\max}$, where $f_{\min}$ and $f_{\max}$ are the minimum and maximum processor frequencies. Without loss of generality, we assume the frequency $f_{\max}$ to be 1.

4.2.3 Related Work

The papers [ASK15, AKTM16b, HKGT14, AMT15b, AMT16, LJP13b] have looked at energy-efficient scheduling of mixed-criticality systems. Out of these only [ASK15, HKGT14] are for uniprocessor systems. The work by Huang et al. [HKGT14] and Narayan et al. [NHG+16] are the only ones with which our work is comparable, because the most of the papers use different mixed-criticality real-time systems models and power management schemes.

In [HKGT14], Huang et al. studied the energy consumption minimization in uniprocessor mixed-criticality systems using the DVFS technique based on continuous frequency levels. They found the processor frequencies $f_{\text{LO}}^{\text{LO}}$ of LO-criticality tasks and $f_{\text{HI}}^{\text{LO}}$ of HI-criticality tasks which can be used in the EDF-VD algorithm [BBD+12b] to schedule the given task set successfully and which result in minimum energy consumption in the LO-criticality scenario with respect to EDF-VD. They also proved that the energy consumed in the system is optimal for the EDF-VD algorithm. In [ASK15], Ali et al. proposed an algorithm, hereafter abbreviated by PMC, which is based on EDF-VD and claimed that it

consumes less energy than the algorithm in [HKGT14] based on experimental evidence, but without a proof.

In 2016, Narayana et al. [NHG+16] proposed a method based on a more generalized system model to reduce energy consumption in multicore mixed-criticality systems. Since the search space for optimality condition is huge, Narayana et al. considered 3 separate processor frequency variables as in [HKGT14]. The optimal processor frequencies computed in [NHG+16] when restricted to unicore system turns out to be the same as in [HKGT14], i.e., the one which is optimal for EDF-VD. We show that our algorithm consumes less energy as compared to any energy-efficient algorithm based on EDF-VD.

4.3 Motivation and Problem Definition

The algorithm presented in [HKGT14] finds optimal processor frequencies for both LO-criticality and HI-criticality jobs in a LO-criticality scenario with respect to EDF-VD [BBD+12b] to get an energy efficient schedule. We first show that the TT-merge algorithm proposed in Chapter 3 can schedule more instances (i.e., a strict superset) than the EDF-VD algorithm [BBD+12b]. We then propose an energy-efficient adaptation of the TT-Merge algorithm (Algorithm 11 in Section 4.4) which gives more energy efficient schedules than the energy-efficient EDF-VD algorithm discussed in [HKGT14]. Here we prove the first claim, i.e., the TT-Merge algorithm schedules more task sets than the EDF-VD algorithm. In the second part of the chapter, we show that the energy consumption of the energy-efficient TT-Merge algorithm is less than that of the energy-efficient EDF-VD algorithm and PMC. We also prove an optimality result with respect to energy consumption for our algorithm.

Example 4.3.1: Consider the MC task set of 4 tasks given in Table 4.1. To be schedulable

Table 4.1: A task set which is not schedulable by EDF-VD

Task	Arrival time	Period	Criticality	C_i(LO)	C_i(HI)
τ_1	0	14	HI	3	5
τ_2	0	14	HI	1	2
τ_3	0	7	LO	3	3
τ_4	0	14	HI	3	7

under EDF-VD, it must satisfy the following condition [BBD+12b]:

$$xU_{\text{LO}}^{\text{LO}}(\mathscr{T}) + U_{\text{HI}}^{\text{HI}}(\mathscr{T}) \leq 1 \tag{4.2}$$

where $U_{\chi_1}^{\chi_2} = \sum_{\tau_i \in \mathscr{T}_{\chi_1}} \frac{C_i(\chi_2)}{p_i}$ and $x = \frac{U_{\text{HI}}^{\text{LO}}}{1-U_{\text{LO}}^{\text{LO}}}$. For the given task set in Table 4.1, $x = 0.875$, $U_{\text{LO}}^{\text{LO}} = 0.375$ and $U_{\text{HI}}^{\text{HI}} = 1$. Since the left-hand side of inequality 4.2 gives a value which is greater than 1. Hence, this task set is not schedulable under EDF-VD.

But the task set is schedulable under TT-Merge where the resulting tables are given in Fig. 4.1. Here we can see that each HI-criticality job can finish its execution at the time instant where a scenario change occurs in the table $\mathcal{S}_{\text{HI}}$. On the other hand, if a scenario change does not occur, then all jobs can finish their LO-criticality execution in the table $\mathcal{S}_{\text{LO}}$.

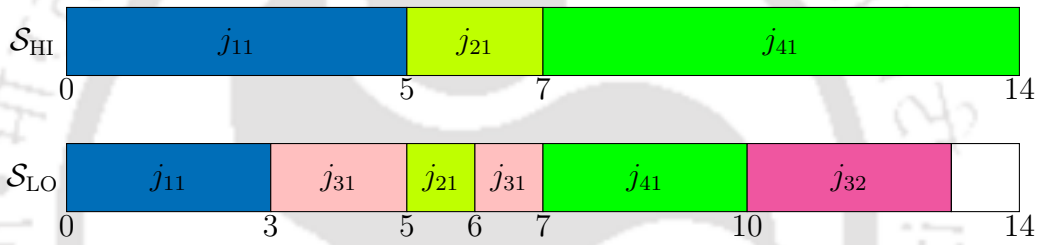


Figure 4.1: Tables constructed by the TT-Merge algorithm

Lemma 4.3.1: If a task set is schedulable under EDF-VD, then the task set is schedulable under TT-merge.

Proof. We need to show that if the EDF-VD algorithm finds a mixed-criticality schedule for a task set, then the TT-Merge algorithm will not encounter a situation where at time slot t, tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ are non-empty, for any t.

We know that $C_i(\text{LO})$ units of execution time are allocated to each job j_{ik} for constructing the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. Each job in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ is allocated as close to its deadline as possible. That means no job can be executed after its allocation time in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ without affecting the schedule of any other job and still meet its deadline. The TT-Merge algorithm declares failure if it finds a non-empty slot at any time t in both the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$. This means the two jobs which are found at time instant t in the tables $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ respectively cannot be scheduled with all other remaining jobs from this point, because all the jobs to the right have already been moved as far to the right as possible.

Suppose a task set satisfies the preprocessing phase of EDF-VD and therefore has a mixed-criticality schedule.

Let j_l and j_h be two jobs in $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ respectively found at time t during the construction of table $\mathcal{S}_{\text{LO}}$ by the TT-Merge algorithm, which means all job segments in the interval $[0, t-1]$ from table $\mathcal{T}_{\text{LO}}$ and $\mathcal{T}_{\text{HI}}$ have already been assigned in $\mathcal{S}_{\text{LO}}$. But, we know that these jobs can be scheduled by EDF-VD as the task set satisfies the required schedulability condition. The EDF-VD algorithm executes or dispatches the waiting jobs with earliest scheduling deadline. Without loss of generality, we can say that the processor is not idle in the interval $[0, t]$, i.e., there is no such time when a task is not available. Both the algorithm schedules all the LO-criticality jobs whose deadline is less than or equal to the deadline of job j_l in the interval $[0, t]$. This is because both the algorithms schedule the LO-criticality jobs in the EDF order. We know that EDF-VD schedules the HI-criticality jobs in the EDF order according to the LO-scenario deadline, i.e., $\hat{p}_i$ $(= x \cdot p_i)$ [BBD+12b]. We know that the LO-scenario deadline is computed using the factor $\frac{U_{\text{HI}}^{\text{LO}}}{1-U_{\text{LO}}^{\text{LO}}}$ for each HI-criticality job. That means the deadlines of HI-criticality job is modified with equal proportion which does not change the original EDF order, i.e., the EDF order with actual deadlines. So the EDF order of the HI-criticality jobs according to the LO-scenario deadline is the same as the EDF order of the HI-criticality jobs according to the actual deadlines. So all the HI-criticality jobs whose deadline is less than or equal to the deadline of job j_h are allocated between 0 and t by both the algorithms. We know that the LO-criticality execution times of each job is scheduled in the interval $[0, t]$ in a work conservative way by both the algorithms. According to our algorithm, there is no time to schedule j_l and j_h before time t, because there is no empty space in the interval $[0, t]$ after scheduling LO-criticality execution time of each job whose deadline is less than or equal to the deadlines of j_l and j_h. EDF-VD also schedules all the LO-criticality jobs whose deadline is less than or equal to that of job j_l and all the HI-criticality jobs whose LO-scenario deadline is less than or equal to that of job j_h. EDF-VD cannot schedule a HI-criticality job whose LO-scenario deadline is greater than that of j_h in the interval $[0, t]$, because the jobs whose deadline is less than that of j_l and j_h has already filled the interval $[0, t]$. Moreover, both j_l and j_h are close to their deadlines. Since our algorithm does not find a place to schedule j_h or j_l in the time interval $[0, t]$, EDF-VD cannot schedule j_h or j_l in that time interval. Hence either j_h or j_l will miss its deadline as they are close to their deadline at time t. Therefore EDF-VD cannot schedule the task set. This is a contradiction. Hence all task sets which can be scheduled by the EDF-VD

algorithm can also be scheduled by our algorithm. □

Theorem 4.3.1: The collection of task sets schedulable under EDF-VD is a strict subset of TT-Merge, i.e., $\mathscr{T}^{\text{EDF-VD}} \subsetneq \mathscr{T}^{\text{TT-Merge}}$ where $\mathscr{T}^{\text{EDF-VD}}$ and $\mathscr{T}^{\text{TT-Merge}}$ are the set of task sets schedulable by the EDF-VD and TT-Merge algorithms, respectively.

Proof. By Lemma 4.3.1, at least $\mathscr{T}^{\text{EDF-VD}} \subseteq \mathscr{T}^{\text{TT-Merge}}$ is satisfied. By Example 4.3.1, some task sets are not schedulable by EDF-VD, but schedulable by the TT-Merge algorithm. Hence, $\mathscr{T}^{\text{EDF-VD}} \subsetneq \mathscr{T}^{\text{TT-Merge}}$. □

4.3.1 Problem Formulation

Now we formally present our energy consumption minimization problem. Our objective is to minimize the system energy consumption by slowing down the tasks in the LO-criticality scenario while ensuring that they do not miss their deadlines using DVFS method. Without loss of generality, we calculate the energy consumption minimization up to the hyperperiod P of the task set. The idea is to find the energy-efficient LO-criticality WCET $\tilde{C}_{ik}(\text{LO})$ and the corresponding frequency f_{ik}^{LO} for each job j_{ik} of the task set in the hyperperiod which will minimize the energy consumption in the LO-criticality scenario.

Definition 4.3.1: We denote by $\tilde{C}_{ik}(\text{LO})$ the worst-case execution time at LO-criticality level of the job j_{ik} of task τ_i after DVFS using the processor frequency f_{ik}^{LO}, i.e.,

$$\tilde{C}_{ik}(\text{LO}) = C_i(\text{LO}) \cdot \frac{f_b}{f_{ik}^{\text{LO}}} \tag{4.3}$$

and by $\tilde{C}_{ik}(\text{HI})$ as the worst-case execution time at HI-criticality level of the job j_{ik} of task τ_i after DVFS, i.e.,

$$\tilde{C}_{ik}(\text{HI}) = \tilde{C}_{ik}(\text{LO}) + (C_i(\text{HI}) - C_i(\text{LO})) \tag{4.4}$$

Since we assume that in HI-criticality scenario every job runs at the base frequency f_b, i.e., DVFS is not used in HI-criticality scenario.

Problem 1: Mixed-criticality Time-triggered Energy Consumption Minimization (***MT-TEM***) Given a dual-criticality task set schedulable under TT-Merge, decide offline the frequency f_{ik}^{LO} for each job j_{ik} of the task set in the hyperperiod such that

- each j_{ik} of τ_i whether it is of LO-criticality or HI-criticality should execute for $\tilde{C}_{ik}(\text{LO})$ units of execution time instead of $C_i(\text{LO})$ with the frequency $f_{ik}^{\text{LO}} \leq f_b$ in a LO-criticality scenario, and

- each HI-criticality job j_{ik} of task τ_i should execute for $\tilde{C}_{ik}(\text{LO})$ units of execution time instead of $C_i(\text{LO})$ with the frequency $f_{ik}^{\text{LO}} \leq f_b$ and $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time with the frequency f_b in a HI-criticality scenario, respectively

and further the LO-criticality scenario system energy consumption is minimized without affecting the schedulability of the system under TT-Merge.

We want to minimize the normalized energy consumption over a hyperperiod (i.e., the actual energy consumption divided by the hyperperiod of the task set) in the LO-criticality scenario by varying the processor frequency. The normalized energy consumption in a hyperperiod is:

$$\frac{1}{P} \cdot \sum_{\tau_i \in \mathscr{T} \wedge 1 \leq k \leq N_i} C_i(\text{LO}) f_b \cdot \frac{1}{f_{ik}^{\text{LO}}} \cdot \beta \cdot (f_{ik}^{\text{LO}})^{\alpha} \tag{4.5}$$

where N_i is the number of times a job of task τ_i will run in a hyperperiod, i.e., $N_i = \frac{P}{p_i}$. The energy consumption minimization is constrained by:

- Bounds for the Frequency for each job:

$$f_{ik}^{\text{LO}} \in [f_{min}, f_{max}]. \tag{4.6}$$

- Construction of temporary tables:

  It should be possible to construct $\tilde{\mathcal{T}}_{\text{LO}}$ and $\tilde{\mathcal{T}}_{\text{HI}}$ using $\tilde{C}_{ik}(\text{LO})$ and $\tilde{C}_{ik}(\text{HI})$ units of execution time of job j_{ik}, respectively without missing the deadline, as TT-Merge of Chapter 3. (4.7)

- Construction of time-triggered table:

  It should be possible to construct $\tilde{\mathcal{S}}_{\text{LO}}$ while ensuring the failure situation (4) of TT-Merge does not occur, at any time t (i.e., $\tilde{\mathcal{T}}_{\text{LO}}[t]$ contains a LO-criticality job and $\tilde{\mathcal{T}}_{\text{HI}}[t]$ contains a HI-criticality job at time t, respectively). (4.8)

Finally, our energy consumption minimization problem is

$$\begin{gathered} \textbf{minimize}(4.5) \\ \textbf{s.t.}(4.6), (4.7), (4.8) \end{gathered} \tag{4.9}$$

Recall that the table $\tilde{\mathcal{S}}_{\text{HI}}$ is constructed using the table $\tilde{\mathcal{S}}_{\text{LO}}$. Also, the TT-Merge algorithm guarantees that if the table $\tilde{\mathcal{S}}_{\text{LO}}$ can be constructed, then so can the table $\tilde{\mathcal{S}}_{\text{HI}}$.

4.4 The Proposed Algorithm

In this section, we propose an algorithm to find a processor frequency f_{ik}^{LO} and an energy-efficient LO-criticality WCET $\tilde{C}_{ik}(\text{LO})$ for each job j_{ik} in the LO-criticality scenario. We also find $\tilde{C}_{ik}(\text{HI})$ of each HI-criticality job using $\tilde{C}_{ik}(\text{LO})$. We then use $\tilde{C}_{ik}(\text{LO})$ and $\tilde{C}_{ik}(\text{HI})$ to construct the tables $\tilde{\mathcal{S}}_{\text{LO}}$ and $\tilde{\mathcal{S}}_{\text{HI}}$ using the TT-Merge algorithm. The key ideas behind our algorithm are as follows.

- We find a processor frequency for each job to run in the LO-criticality scenario.
- We find a LO-criticality worst-case execution time $\tilde{C}_{ik}(\text{LO})$ of each job which gives a minimized energy consumption schedule in LO-criticality scenario using the processor frequency f_{ik}^{LO}.
- On a mode change from LO-criticality to HI-criticality, a HI-criticality job runs at the base frequency f_b.
- We achieve energy efficiency by reducing the idle time of the processor as much as possible.

Algorithm 9 constructs the table E_{LO} which includes only the LO-criticality tasks. This algorithm chooses the LO-criticality tasks from the task set and orders them in EDF order [LL73]. Then, all the job segments of the EDF schedule are moved as close to their deadline as possible so that no job misses its deadline in table E_{LO}. Note that, if the arrival times of the jobs are not the same, then the jobs may execute in more than one segment, in general. If the arrival times of all the jobs are the same then, the jobs will execute in one segment.

Algorithm 10 constructs the table E_{HI} which contains only the HI-criticality tasks. This algorithm chooses the HI-criticality tasks from the task set and orders them in EDF order. Then, all the job segments of the EDF schedule are moved as close to their deadline as possible so that no job misses its deadline in table E_{HI}. Then, out of the total allocation so far, the algorithm allocates $C_i(\text{LO})$ units of execution of job j_{ik} in table E_{HI} from the beginning of its slot and leaves the rest of the execution time of j_{ik} unallocated in table E_{HI}.

Algorithm 11 is the same as the TT-Merge algorithm of Chapter 3 except it calls the function SetFrequency(E_{LO},E_{HI},E_{FINAL}) at the end. Function SetFrequency(E_{LO},E_{HI},E_{FINAL}) in Algorithm 12 finds the energy efficient LO-criticality WCET $\tilde{C}_{ik}(\text{LO})$ after DVFS and the processor frequency f_{ik}^{LO} for each job j_{ik} which will

Algorithm 9 Construct-$E_{\text{LO}}(\mathscr{T})$

Input : A task set $\mathscr{T} = \{\tau_1, \tau_2, ..., \tau_n\}$, where j_{ik} is a job of $\tau_i =< a_{ik}, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}) >$.

Output : Temporary table E_{LO}

Assume the earliest arrival time is 0.

1: Let P denote the least common multiple of the periods of the task set $\mathscr{T}$: $P := lcm(p_1, p_2, \ldots, p_n)$.
2: The length of table E_{LO} is set to P;
3: Let L be the set of LO-criticality tasks of the task set;
4: Let O be the EDF order of the tasks of L on the time-line using $C_i(\text{LO})$ units of execution for each job j_{ik};
5: **if** (any job cannot be scheduled) **then**
6: Declare failure;
7: **end if**
8: Starting from the rightmost job segment of the EDF order of L, move each segment of a job j_{ik} as close to its deadline as possible in table E_{LO}.

be used by the TT-Merge algorithm to construct the scheduling tables $\tilde{\mathcal{S}}_{\text{LO}}$ and $\tilde{\mathcal{S}}_{\text{HI}}$. In step 1, all the job segments in table E_{FINAL} are moved to the right as close to their finishing time in table E_χ as possible, where χ is the criticality level of the job. After each job j_{ik} is moved to its right, the completion time of each job in the table E_{FINAL} is called its *finishing time* d_{ik}^{Δ}. In step 3, the function initializes $\tilde{C}_{ik}(\text{LO})$ to the value of $C_i(\text{LO})$. In step 6, the function finds all the empty intervals in the table E_{FINAL}. All the jobs found before the start of the first empty interval in the table E_{FINAL} have not been moved. So these jobs cannot be slowed down by lowering the processor frequency as they have no idle time between their arrival times and finishing times. In step 9, the function removes the set of jobs J' which cannot be slowed down from the set of jobs J, namely the jobs before the first empty interval, where J is the set of all jobs and J' is the set of all jobs which cannot be slowed down. In step 10, the possible expansion per unit of execution time r_{LO} for each job in J is calculated using

Algorithm 10 Construct-$E_{\text{HI}}(\mathscr{T})$

Input : A task set $\mathscr{T} = \{\tau_1, \tau_2, ..., \tau_n\}$, where job j_{ik} of $\tau_i =< a_{ik}, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}) >$.
Output : Temporary table E_{HI}

1: Let P denote the least common multiple of the periods of the task set $\mathscr{T}$: $P := lcm(p_1, p_2, \ldots, p_n)$.
2: The length of table E_{HI} is set to P;
3: Let H be the set of HI-critical tasks of the task set;
4: Let O be the EDF order of the tasks of H on the time-line using $C_i(\text{HI})$ units of execution for job j_{ik} ;
5: **if** (any job cannot be scheduled) **then**
6: Declare failure;
7: **end if**
8: Starting from the rightmost job segment of the EDF order of H, move each segment of a job j_{ik} as close to its deadline as possible in table E_{HI}.
9: **for** $i := 1$ to m **do**
10: Allocate $C_i(\text{LO})$ units of execution to job j_{ik} from its starting time in table E_{HI} and leave the rest unallocated;
11: **end for**

the following formula.

$$r_{\text{LO}} = \frac{P - EMPTY_1^S}{\sum_{\forall j_{ik} \in J} C_{ik}(\text{LO})} \tag{4.10}$$

where $EMPTY_1^S$ is the start time of the first empty interval in table E_{FINAL}.

The fraction r_{LO} is used as follows to compute the processor frequency for each job. If the processor frequency $\frac{f_b}{r_{\text{LO}}}$ is less than or equal to $f_{\min}$, then $\tilde{C}_{ik}(\text{LO})$ of each job $j_{ik} \in J$ is updated using the following formula:

$$\tilde{C}_{ik}(\text{LO}) = C_{ik}(\text{LO}) \cdot \frac{f_b}{f_{\min}} \tag{4.11}$$

On the other hand, if $\frac{f_b}{r_{\text{LO}}} > f_{\min}$, then $\frac{f_b}{r_{\text{LO}}}$ is the lowest possible frequency or each job. So $\tilde{C}_{ik}(\text{LO})$ of each job $j_{ik} \in J$ is updated using the following formula:

Algorithm 11 EE-TT-MERGE($\mathscr{T}$, table E_{LO}, table E_{HI})

Input : Table E_{LO}, table E_{HI}, $\mathscr{T} = \{\tau_1, \tau_2, ..., \tau_n\}$, where job j_{ik} of $\tau_i =< a_{ik}$, d_i, χ_i, C_i(LO), C_i(HI) >.
Output : $\tilde{C}_{ik}$(LO), $\tilde{C}_{ik}$(HI), f_{ik}^{LO}.

1: Copy table E_{LO} and E_{HI} to $\tilde{E}_{\text{LO}}$ and $\tilde{E}_{\text{HI}}$, respectively;
2: Let P denote the least common multiple of the periods of the task set $\mathscr{T}$: $P := lcm(p_1, p_2, \ldots, p_n)$.
3: The length of table E_{FINAL} is set to P;
4: $t := 0$;
5: **while** $(t \leq P)$ **do**
6: **if** $(\tilde{E}_{\text{LO}}[t] = NULL \ \& \ \tilde{E}_{\text{HI}}[t] = NULL)$ **then**
7: Search the tables $\tilde{E}_{\text{LO}}$ and $\tilde{E}_{\text{HI}}$ simultaneously from the beginning to find the first available job at time t;
8: Let k be the first occurrence of a job j_{ik} in $\tilde{E}_{\text{LO}}$ or $\tilde{E}_{\text{HI}}$;
9: **if** (Both LO-criticality & HI-criticality job are found) **then**
10: $E_{\text{FINAL}}[t] := \tilde{E}_{\text{LO}}[k]$;
11: $\tilde{E}_{\text{LO}}[k] := NULL$;
12: **else if** (LO-criticality job is found) **then**
13: $E_{\text{FINAL}}[t] := \tilde{E}_{\text{LO}}[k]$;
14: $\tilde{E}_{\text{LO}}[k] := NULL$;
15: **else if** (HI-criticality job is found) **then**
16: $E_{\text{FINAL}}[t] := \tilde{E}_{\text{HI}}[k]$;
17: $\tilde{E}_{\text{HI}}[k] := NULL$;
18: **else if** (NO job is found) **then**
19: $E_{\text{FINAL}}[t] := NULL$
20: $t := t + 1$;
21: **end if**
22: **else if** $(\tilde{E}_{\text{LO}}[t] = NULL \ \& \ \tilde{E}_{\text{HI}}[t] \neq NULL)$ **then**
23: $E_{\text{FINAL}}[t] := \tilde{E}_{\text{HI}}[t]$;
24: $\tilde{E}_{\text{HI}}[t] := NULL$;
25: $t := t + 1$;
26: **else if** $(\tilde{E}_{\text{LO}}[t] \neq NULL \ \& \ \tilde{E}_{\text{HI}}[t] = NULL)$ **then**
27: $E_{\text{FINAL}}[t] := \tilde{E}_{\text{LO}}[t]$;
28: $\tilde{E}_{\text{LO}}[t] := NULL$;
29: $t := t + 1$;
30: **else if** $(\tilde{E}_{\text{LO}}[t] \neq NULL \ \& \ \tilde{E}_{\text{HI}}[t] \neq NULL)$ **then**
31: Declare failure;
32: **end if**
33: **end while**
34: SetFrequency(E_{LO},E_{HI},E_{FINAL});

Algorithm 12 Function SetFrequency(E_{LO},E_{HI},E_{FINAL},$\mathscr{T}$)

Input : E_{LO}, E_{HI}, E_{FINAL} and a task set $\mathscr{T} = \{\tau_1, \tau_2, ..., \tau_n\}$, where $j_{ik} = (a_{ik}, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}))$ is the k^{th} job of τ_i.
Output : $\tilde{C}_{ik}(\text{LO})$, $\tilde{C}_{ik}(\text{HI})$, f_{ik}^{LO}.
/* $\tilde{C}_{ik}(\text{LO})$ and $\tilde{C}_{ik}(\text{HI})$ are the LO-criticality and HI-criticality execution time of each job after DVFS, respectively and f_{ik}^{LO} is the LO-criticality processor frequency of each job in DVFS */

1: Starting from the rightmost job segment of the table E_{FINAL}, move each segment of a job j_{ik} as close to its finishing time in E_χ as possible, where χ is the criticality of j_{ik};
2: Let J be the set of all jobs in table E_{FINAL};
3: **for** each job in J **do**
4: $\quad\tilde{C}_{ik}(\text{LO}) := C_i(\text{LO})$;
5: **end for**
6: Find all the empty intervals in the table E_{FINAL};
7: Let $EMPTY$ be the set of empty intervals in the table E_{FINAL};
8: Let J' be the set of jobs before the first empty interval;
9: $J := J \setminus J'$;
10: $r_{\text{LO}} := \frac{P - EMPTY_1^S}{\sum_{\forall j_{ik} \in J} C_{ik}(\text{LO})}$; /* $EMPTY_1^S$ is the beginning of the first empty interval */
11: **if** $(\frac{f_b}{r_{\text{LO}}} \leq f_{\min})$ **then**
12: $\quad$**for** each job in J **do**
13: $\quad\quad\tilde{C}_{ik}(\text{LO}) := C_{ik}(\text{LO}) \cdot \frac{f_b}{f_{\min}}$;
14: $\quad$**end for**
15: **else**
16: $\quad$**for** each job in J **do**
17: $\quad\quad\tilde{C}_{ik}(\text{LO}) := C_{ik}(\text{LO}) \cdot r_{\text{LO}}$;
18: $\quad$**end for**
19: **end if**
20: Find the finishing time (d_{ik}^Δ) of each job in J;
21: Sort the jobs in table E_{FINAL} according to their finishing times in non-decreasing order;
22: **for** each job j_{ik} in increasing order of the corresponding d_{ik}^Δ **do**
23: $\quad$**if** $(\tilde{C}_{ik}(\text{LO}) + \sum_{j_l \in J \wedge d_l^\Delta < d_{ik}^\Delta} \tilde{C}_l(\text{LO}) \geq d_{ik}^\Delta)$ **then**
24: $\quad\quad t^\Delta :=$ CheckEmptySpace(E_{FINAL},d_{ik}^Δ);
25: $\quad\quad\delta := d_{ik}^\Delta - \sum_{j_l \in J \wedge d_l^\Delta \leq d_{ik}^\Delta} \tilde{C}_l(\text{LO})$; /* compute the amount of time δ by which j_{ik} misses its
26: finishing time */
27: $\quad\quad$**for** each job $j_l \in J \wedge a_l \geq t^\Delta \wedge d_l^\Delta \leq d_{ik}^\Delta$ **do**
28: /* subtract a total of δ amount of execution time in equal parts from jobs whose arrival time is greater than or equal to t^Δ and finishing time is less than or equal to d_{ik}^Δ */
29: $\quad\quad\quad\tilde{C}_l(\text{LO}) := \tilde{C}_l(\text{LO}) - \frac{\delta}{\sum_{j_l \in J \wedge a_l \geq t^\Delta \wedge d_l^\Delta \leq d_{ik}^\Delta} C_l(\text{LO})} \cdot C_l(\text{LO})$;
30: $\quad\quad$**end for**
31: $\quad\quad$**for** each job $j_l \in J \wedge d_l^\Delta > d_{ik}^\Delta$ **do**
32: $\quad\quad\quad\tilde{C}_l(\text{LO}) := \tilde{C}_l(\text{LO}) + \frac{\delta}{\sum_{j_l \in J \wedge d_l^\Delta > d_{ik}^\Delta} C_l(\text{LO})} \cdot C_l(\text{LO})$;
33: $\quad\quad$**end for**

34: Goto step 22;
35: **end if**
36: **end for**
37: **for** each job $j_{ik} \in J$ **do**
38: $f_{ik}^{\text{LO}} = \max\{f_{\min}, \frac{C_{ik}(\text{LO})}{\tilde{C}_{ik}(\text{LO})} \cdot f_b\}$;
39: $\tilde{C}_{ik}(\text{HI}) := (C_i(\text{HI}) - C_i(\text{LO})) + \tilde{C}_{ik}(\text{LO})$;
40: **end for**

Algorithm 13 Function CheckEmptySpace(E_{FINAL},d^{Δ})

Output : Time instant (t^{Δ}) of an empty space

1: $\delta := d^{\Delta} - \sum_{j_l \in J \wedge d_l^{\Delta} \leq d^{\Delta}} \tilde{C}_l(\text{LO})$;
2: **for** each job $j_l \in J \wedge d_l^{\Delta} \leq d^{\Delta}$ **do**
3: $\tilde{C}_l(\text{LO}) := \tilde{C}_l(\text{LO}) - \frac{\delta}{\sum_{j_l \in J \wedge d_l^{\Delta} \leq d^{\Delta}} C_l(\text{LO})} \cdot C_l(\text{LO})$;
4: **end for**
5: Simulate each job $j_l \in J \wedge d_l^{\Delta} \leq d^{\Delta}$ using the EDF algorithm considering d_l^{Δ} as the deadline.
6: **if** (an empty space is found) **then**
7: return the end time (t^{Δ}) of the empty space;
8: **else**
9: return 0;
10: **end if**

$$\tilde{C}_{ik}(\text{LO}) = C_{ik}(\text{LO}) \cdot r_{\text{LO}} \tag{4.12}$$

In step 20, the function finds the finishing time (d_{ik}^{Δ}) of each job j_{ik} in J. Then all the jobs are sorted according to their finishing time in non-decreasing order. In step 22, the function checks that each job must meet its finishing time d_{ik}^{Δ} using $\tilde{C}_{ik}(\text{LO})$ units of execution time. The schedulability of each job can be verified by the following condition:

$$\tilde{C}_{ik}(\text{LO}) + \sum_{j_l \in J \wedge d_l^{\Delta} < d_{ik}^{\Delta}} \tilde{C}_l(\text{LO}) \leq d_{ik}^{\Delta} \tag{4.13}$$

If no job execution crosses its finishing time d_{ik}^{Δ}, then the total energy consumption found for the LO-criticality scenario is the minimum, as proved below in Lemma 4.4.2. On

the other hand, if a job j_{ik} crosses its finishing time in the above process, then the algorithm finds processor frequencies for each job which are as close to $\frac{f_b}{r_{\text{LO}}}$ as possible. Suppose job j_{ik} misses its deadline with δ time remaining to be executed. In this case, the algorithm calls the function CheckEmptySpace(). This function reduces $\tilde{C}_l(\text{LO})$ of all the jobs whose finishing time is less than d_{ik}^{Δ} by equal proportion (given in step 3 of the function CheckEmptySpace()). Then it simulates all the jobs whose deadlines are less than d_{ik}^{Δ} using the EDF algorithm, where the finishing time of each job in the table E_{FINAL} is considered to be the deadline. If an empty space is found, then it returns the end time (t^{Δ}) of that empty space. Otherwise, it returns the initial time, i.e., 0. Note that $t^{\Delta} \neq 0$ means the jobs present before t^{Δ} in simulation are the only ready jobs before time t^{Δ} and the arrival time of all the remaining jobs are greater than or equal to t^{Δ}. Then $\tilde{C}_l(\text{LO})$ of all the jobs j_l whose $a_l \geq t^{\Delta}$ and $d_l \leq d_{ik}^{\Delta}$ is updated using the following formula:

$$\tilde{C}_l(\text{LO}) := \tilde{C}_l(\text{LO}) - \frac{\delta}{\sum_{j_l \in J \wedge a_l \geq t^{\Delta} \wedge d_l^{\Delta} \leq d_{ik}^{\Delta}} \tilde{C}_l(\text{LO})} \cdot \tilde{C}_l(\text{LO}) \tag{4.14}$$

Here the extra amount of execution time δ is subtracted in equal proportion from each job j_l whose $a_l \geq t^{\Delta}$ and $d_l \leq d_{ik}^{\Delta}$ to accommodate j_{ik} between its arrival time a_{ik} and finishing time d_{ik}^{Δ}. The process continues until all the jobs whose finishing times lie between t^{Δ} and d_{ik}^{Δ} meet their finishing times. Then the extra amount of workload δ is distributed in equal proportion among all the jobs whose finishing time is greater than d_{ik}^{Δ} using the following formula:

$$\tilde{C}_l(\text{LO}) := \tilde{C}_l(\text{LO}) + \frac{\delta}{\sum_{j_l \in J \wedge d_l^{\Delta} > d_{ik}^{\Delta}} \tilde{C}_l(\text{LO})} \cdot \tilde{C}_l(\text{LO}) \tag{4.15}$$

The above process continues until all the jobs of the table E_{FINAL} meet their finishing times with the computed $\tilde{C}_{ik}(\text{LO})$. Then the processor frequency for each job is calculated using the following formula:

$$f_{ik}^{\text{LO}} = \max\{f_{\min}, \frac{C_{ik}(\text{LO})}{\tilde{C}_{ik}(\text{LO})} \cdot f_b\} \tag{4.16}$$

Finally, $\tilde{C}_{ik}(\text{HI})$ for each HI-criticality job j_{ik} is computed using the following formula:

$$\tilde{C}_{ik}(\text{HI}) := (C_i(\text{HI}) - C_i(\text{LO})) + \tilde{C}_{ik}(\text{LO}) \tag{4.17}$$

Example 4.4.1: Consider the following task set. Here we assume that the base frequency

Task	Period	Criticality	C_i(LO)	C_i(HI)
τ_1	8	HI	2	5
τ_2	12	LO	1	1
τ_3	16	LO	2	2

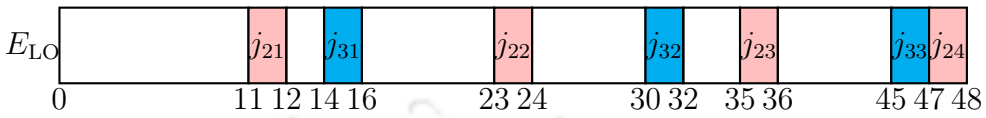


Figure 4.2: Table E_{LO}

f_b of the processor is the same as $f_{\max}$, i.e., 1 and the minimum frequency $f_{\min}$ is 0.2, where $\beta = 1$ and $\alpha = 2.5$. Let us first find tables E_{LO} and E_{HI} in which the LO-criticality and HI-criticality jobs are allocated respectively.

- The lcm of all the periods of the task set is 48, and hence so is the length of the tables E_{LO} and E_{HI}.
- According to Algorithm 9, we choose the LO-criticality jobs and allocate them in E_{LO} in EDF order. Then, each segment of the jobs in EDF order is shifted as close to its deadline as possible according to its C_i(LO) units of execution. The resulting table E_{LO} is given in Fig. 4.2.
- According to Algorithm 10, we choose the HI-criticality jobs in order to allocate them in E_{HI} in EDF order. Then each segment of the jobs in EDF order is shifted as close to its deadline as possible according to its C_i(HI) units of execution. So the first arrival of j_{11} whose deadline is at time instant 8 is allocated in the interval $[3, 8]$. Similarly, the other arrivals are allocated in the intervals $[11, 16]$, $[19, 24]$, $[27, 32]$, $[35, 40]$ and $[43, 48]$ respectively.
- Then we allocate C_i(LO) units of execution of each arrival of j_{ik} and leave the (C_i(HI) − C_i(LO)) units of execution unallocated. Here the first arrival of j_{11} has been allocated its C_i(LO) units of execution time in the interval $[3, 8]$. Then we empty the occurrence of j_{11} in the interval $[5, 8]$ which leaves the interval $[3, 5]$ in table E_{HI}. We repeat the same process for the other arrivals of task j_{11}. The resulting table E_{HI} after this modification is given in Fig. 4.3.
- Finally, we construct the table E_{FINAL} from these two temporary tables.

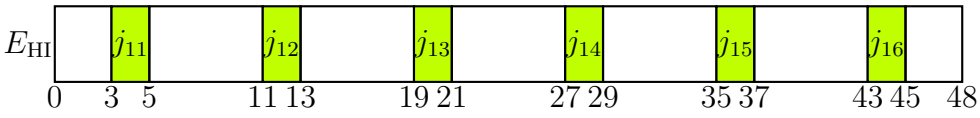


Figure 4.3: Table E_{HI}

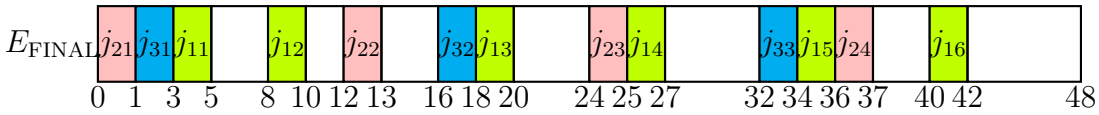


Figure 4.4: Table E_{FINAL}

We construct the table E_{FINAL} according to Algorithm 11.

- We start from time $t = 0$.
- At $t = 0$, both E_{LO} and E_{HI} are empty. Then we search both the tables E_{LO} and E_{HI} starting from time $t = 0$. We find j_{21} in table E_{LO} and j_{11} in table E_{HI} available at time $t = 0$. So we allocate the LO-criticality job from E_{LO}, i.e., j_{21}. We empty the interval $[11, 12]$ in E_{LO} from where the first occurrence of j_{21} was found.

  At $t = 1$, both E_{LO} and E_{HI} are empty. Then we search both the tables E_{LO} and E_{HI} starting from time $t = 1$. We find j_{31} in table E_{LO} and j_{11} in table E_{HI} available at time $t = 1$. So we allocate the LO-criticality job from E_{LO}, i.e., j_{31}. We empty the interval $[14, 15]$ in E_{LO} from where the first occurrence of j_{31} was found. We continue in this way till all the jobs have been allocated in table E_{FINAL}.
- The resulting table E_{FINAL} is given in Fig. 4.4.
- Then the SetFrequency() function is called, where all the jobs, beginning from the right end of E_{FINAL}, are moved as close to their finishing time in table E_χ as possible, where χ is the criticality level of a job. We start with j_{16} present in the interval $[40, 42]$ in E_{FINAL} and move it to the interval $[43, 45]$ which is the finishing time of j_{16} in E_{HI}. We continue until we move all the jobs to their right. Then the resulting table E_{FINAL} is given in Fig. 4.5.
- Here $EMPTY_1^S$ and P are 0 and 48, respectively. The total LO-criticality execution time between $EMPTY_1^S$ and P in table E_{FINAL} is 22.
- $r_{\text{LO}} = \frac{P - EMPTY_1^S}{\sum_{\forall j_{ik} \in J} C_{ik}(\text{LO})} = \frac{48-0}{22} = 2.181$, where $f_{\min} < \frac{f_b}{r_{\text{LO}}}$. This means the lowest possible processor frequency for each job in J is 0.458.

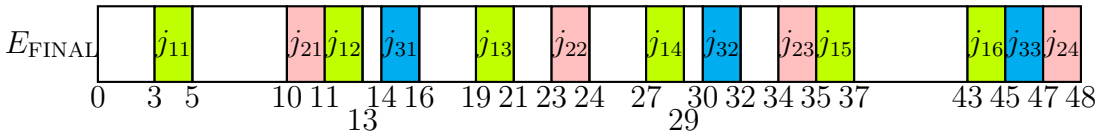


Figure 4.5: Table E_{FINAL} after the moving the job segments to their right

- We find the $\tilde{C}_{ik}(\text{LO})$ for each job of tasks $\langle \tau_1, \tau_2, \tau_3 \rangle$ to be $\langle 4.363, 2.183, 4.363 \rangle$.
- Now we use the relation 4.13 to check the schedulability of each job with the $\tilde{C}_{ik}(\text{LO})$ units of execution time. We show this in the table given below.

Table 4.2: Table showing initial processor frequency allotment

Job	Arrival time	Deadline	Exec Assgn	Rem. exec	Total time elapsed	Frequency alloted
j_{11}	0	5	4.363	0	4.363	0.458
j_{21}	0	11	2.181	0	6.544	0.458
j_{12}	8	13	4.363	0	12.363	0.458
j_{31}	0	16	4.363	0	15.27	0.458
j_{13}	16	21	4.363	0	20.363	0.458
j_{22}	12	24	2.181	0	21.814	0.458
j_{14}	24	29	4.363	0	28.363	0.458
j_{32}	16	32	4.363	0	30.54	0.458
j_{23}	24	35	2.181	0	32.721	0.458
j_{15}	32	37	4.363	0.084	37.084	0.458
j_{16}	40	45	4.363	4.363	0	0.458
j_{33}	32	47	4.363	4.363	0	0.458
j_{24}	36	48	2.181	2.181	0	0.458

- There is a finishing time miss for job j_{15} by 0.084 units in Table 4.2 which is shown in the colored row. Then the 0.084 units of time is reduced from all the jobs whose finishing time lies in $[0, 37]$. As a result, the processor frequencies of those jobs are increased.
- On the other hand, the 0.084 units of extra time is added to the execution times of all those jobs whose finishing time is greater than 37. As a result, the processor frequencies

of such jobs are reduced. This is shown in Table 4.3.

Table 4.3: Table showing final processor frequency allotment

Job	Arrival time	Deadline	Exec Assgn	Rem. exec	Total time elapsed	Frequency alloted
j_{11}	0	5	4.353	0	4.353	0.459
j_{21}	0	11	2.176	0	6.53	0.459
j_{12}	8	13	4.353	0	12.353	0.459
j_{31}	0	16	4.353	0	15.236	0.459
j_{13}	16	21	4.353	0	20.353	0.459
j_{22}	12	24	2.176	0	21.765	0.459
j_{14}	24	29	4.353	0	28.353	0.459
j_{32}	16	32	4.353	0	30.471	0.459
j_{23}	24	35	2.176	0	32.647	0.459
j_{15}	32	37	4.353	0	37	0.459
j_{16}	40	45	4.4	0	44.4	0.45
j_{33}	32	47	4.4	0	45.8	0.45
j_{24}	36	48	2.2	0	48	0.45

- Since we do not find any finishing time miss, the processor frequency for each job is the final one.
- The total normalized energy consumption by the TT-Merge energy-efficient algorithm using the above processor frequencies is 0.09.
- The processor frequencies $f_{\text{LO}}^{\text{LO}}$ and $f_{\text{HI}}^{\text{LO}}$ for the task set of this example calculated by the EDF-VD energy-efficient algorithm are 0.54 and 0.65, respectively.
- The normalized energy consumption computed by the EDF-VD energy-efficient algorithm is 0.21 which is greater than the TT-Merge energy-efficient algorithm.

4.4.1 Energy-efficient EDF-VD versus Energy-efficient TT-Merge

First, we prove that the schedule constructed by the energy-efficient TT-Merge algorithm using $\tilde{C}(\text{LO})$ units of execution time does not have any idle time. We then prove that the

energy-efficient TT-Merge algorithm finds the lowest possible energy consumption among all frequency assignments to jobs that lead to schedulability by the TT-Merge algorithm. Finally, we prove the dominance of Energy-efficient TT-Merge over Energy-efficient EDF-VD. Recall that r_{LO} is $\frac{P-EMPTY_1^S}{\sum_{\forall j_{ik} \in J} C_{ik}(\text{LO})}$, where P is the hyperperiod and $EMPTY_1^S$ is the start time of the first empty interval. We use the following result in our proof.

Lemma 4.4.1: The optimal processor frequency to minimize the total energy consumption while meeting all the finishing times d^{Δ} is constant and equal to $f^{\text{LO}} = \max\{f_{\min}, U_{\text{LO}} \cdot f_b\}$, where $U_{\text{LO}} = \frac{\sum\limits_{\forall j_{ik}} C_{ik}(\text{LO})}{P}$. Moreover, when used along with this speed f^{LO}, any periodic hard real-time policy which can fully utilize the processor can be used to obtain a feasible schedule.

Proof. We know that the function SetFrequency() assigns $\tilde{C}_{ik}(\text{LO})$ units of execution time to each job j_{ik} between its arrival time a_{ik} and its finishing time d_{ik}^{Δ} with processor frequencies $f^{\text{LO}} = U_{\text{LO}} \cdot f_b$. If all the jobs complete execution before their finishing time d_{ik}^{Δ}, then we say that each job is stretched with equal proportion. We need to show that

$$\frac{\sum\limits_{\forall j_{ik}} \tilde{C}_{ik}(\text{LO})}{P} = 1 \tag{4.18}$$

We know that

$$\tilde{C}_{ik}(\text{LO}) = C_{ik}(\text{LO}) \cdot \frac{f_b}{f^{\text{LO}}} = \frac{C_{ik}(\text{LO})}{U_{\text{LO}}} \tag{4.19}$$

Without loss of generality, we assume $f_b = 1$. Now we replace $\tilde{C}_{ik}(\text{LO})$ of the left hand side in Eqn. 4.18 with Eqn. 4.19. So clearly $\frac{\sum \tilde{C}_{ik}(\text{LO})}{P} = 1$. if $f_{\min} > U_{\text{LO}} \cdot f_b$, then we choose the minimum processor frequency available. □

Fact 1: If all jobs in the hyperperiod for a given task can be scheduled using the same processor frequency $\frac{f_b}{r_{\text{LO}}}$ for each job then the function SetFrequency() will assign this frequency to each job. This is because lines 23-35 in the function SetFrequency() are never executed.

Lemma 4.4.2: A task set $\mathscr{T}$ with all tasks scheduled with the same processor frequency $\frac{f_b}{r_{\text{LO}}}$ without violating any finishing time d^{Δ} in the LO-criticality scenario and guaranteeing the execution of $(C(\text{HI}) - C(\text{LO}))$ time units of each HI-criticality job results in the minimum energy consumption among possible frequency assignments under the TT-Merge algorithm.

Proof. We try to assign a uniform processor frequency to each job and utilize the total hyperperiod as much as possible. Without loss of generality, $EMPTY_1^S$ is assumed to be 0.

Then the lowest possible processor frequency $\frac{f_b}{r_{\text{LO}}}$ is the same as the LO-criticality utilization (U_{LO}) of the task set $\mathscr{T}$, where $f_b = f_{\max} = 1$. If $\frac{f_b}{r_{\text{LO}}} > f_{\min}$ and no job misses its finishing time d^{Δ}, then the energy-efficient TT-Merge algorithm generates a schedule with no idle time and each job is expanded with equal proportion using the processor frequency $\frac{f_b}{r_{\text{LO}}}$. In this way, we use the total hyperperiod available to run the task set $\mathscr{T}$, i.e., the total effective task utilization is 1. We can see this from the equation given below:

$$\frac{\sum\limits_{\forall j_{ik}} \frac{C_{ik}(\text{LO})}{\frac{f_b}{r_{\text{LO}}}}}{P} = \frac{U_{\text{LO}}}{\frac{f_b}{r_{\text{LO}}}} = 1 \tag{4.20}$$

From the function SetFrequency(), it is clear that we choose $f_{\min}$ as the lowest possible processor frequency, if $\frac{f_b}{r_{\text{LO}}} \leq f_{\min}$. It can be see that the maximum processor frequency between $f_{\min}$ and $\frac{f_b}{r_{\text{LO}}}$ will always result in a total effective task utilization which is not greater than 1. This follows from Lemma 4.4.1. □

If a job misses its finishing time d_{ik}^{Δ} with processor frequency $\frac{f_b}{r_{\text{LO}}}$, then the lowest possible energy solution does not exist. So our algorithm searches for the processor frequencies which are as close to $\frac{f_b}{r_{\text{LO}}}$ as possible.

Lemma 4.4.3: In the proposed energy-efficient TT-Merge algorithm, i.e., Algorithm 11, the time-triggered schedule generated with $\tilde{C}_{ik}(\text{LO})$ units of execution time does not contain any idle time interval $[t, t']$ where a job with processor frequency $f_{ik}^{\text{LO}} > f_{\min}$ is available.

Proof. This is easy to see from the algorithm. Without loss of generality, we consider $EMPTY_1^S$ to be 0. Initially, the total hyperperiod is distributed among all the jobs. If all the jobs satisfy condition 4.13 with the assigned execution times, then there will not be any idle time in the schedule. Suppose a job j_{ik} misses its finishing time (d_{ik}^{Δ}) by δ amount of times in table E_{FINAL}. Then we call the function CheckEmptySpace() which finds empty spaces before d_{ik}^{Δ} in table E_{FINAL}, if any. First, the function subtracts the extra amount of the assigned execution time δ by which job j_{ik} misses its finishing time d_{ik}^{Δ} is subtracted in equal proportion from each job whose finishing time is less than d_{ik}^{Δ}. Then it simulates to find an empty space from time instant 0 to d_{ik}^{Δ} using EDF algorithm. If an empty space is not found, then we finalize the processor frequency for each job. Otherwise, we subtract the extra amount of the assigned execution time δ in equal proportion from each job j_l whose $a_l \geq t^{\Delta}$ and $d_l \leq d_{ik}^{\Delta}$. Then job j_{ik} meets its finishing time exactly along with all the jobs j_l whose $a_l \geq t^{\Delta}$ and $d_l \leq d_{ik}^{\Delta}$. from the above discussion, it is clear that the jobs which are

allocated before t^{Δ} will not be assigned new processor frequencies, whereas the jobs between t^{Δ} and d_{ik}^{Δ} are. There will not be any idle time in the schedule after d_{ik}^{Δ} as the extra amount of execution time δ is distributed in equal proportion among the jobs whose finishing time is greater than d_{ik}^{Δ}. From the above arguments, it is clear that the schedule generated using our algorithm does not have any idle time. □

Lemma 4.4.4: The frequency assigned by the function SetFrequency() to each job of task set $\mathscr{T}$ when the condition of Lemma 4.4.2 is not satisfied (i.e., the task set is not scheduled by energy-efficient TT-Merge with the frequency assignment of $\frac{f_b}{r_{\text{LO}}}$ for all tasks) results in minimum energy consumption.

Proof. Initially, the function SetFrequency() assigns a single processor frequency f^{LO} to each job, where $f^{\text{LO}} = \frac{f_b}{r_{\text{LO}}}$. If no job misses its finishing time d_{ik}^{Δ}, then each job gets the minimum frequency f^{LO}. This is clear from Lemma 4.4.2. So our energy consumption function becomes as follows:

$$\sum_{\tau_i \in \mathscr{T} \wedge 1 \leq k \leq N_i} C_i(\text{LO}) f_b \cdot \tfrac{1}{f^{\text{LO}}} \cdot (f^{\text{LO}})^{\alpha} \tag{4.21}$$

Then the extra amount of the assigned execution time δ by which job j_{ik} misses its finishing time d_{ik}^{Δ} is subtracted in equal proportion from each job whose finishing time is less than d_{ik}^{Δ}. Then we call the function CheckEmptySpace() which finds empty spaces before d_{ik}^{Δ} in the table E_{FINAL}, if any. If an empty space is not found we finalize the processor frequency for each job. Otherwise, we subtract the extra amount of the assigned execution time δ in equal proportion from each job j_l whose $a_l \geq t^{\Delta}$ and $d_l \leq d_{ik}^{\Delta}$. If the function CheckEmptySpace() finds an empty space at t^{Δ}, then our algorithm does not change the assigned processor frequency to the jobs which are scheduled before t^{Δ}. In other words, the frequency of each job j_l whose $a_l \geq t^{\Delta}$ and $d_l \leq d_{ik}^{\Delta}$ is increased by $x = \frac{K}{\Psi - \delta} - f^{\text{LO}}$ and the frequency of each job whose finishing time lies after d_{ik}^{Δ} is decreased by $y = f^{\text{LO}} - \frac{L}{\Omega + \delta}$, where

$$K = \sum_{j_r \in J \wedge a_r \geq t^{\Delta} \wedge d_r^{\Delta} \leq d_{ik}^{\Delta}} C_r(\text{LO}),$$

$$L = \sum_{j_s \in J \wedge d_s^{\Delta} > d_{ik}^{\Delta}} C_s(\text{LO}),$$

$$\Psi = \sum_{j_r \in J \wedge a_r \geq t^{\Delta} \wedge d_r^{\Delta} \leq d_{ik}^{\Delta}} \tilde{C}_r(\text{LO}) \text{ and}$$

$$\Omega = \sum_{j_s \in J \wedge d_s^{\Delta} > d_{ik}^{\Delta}} \tilde{C}_s(\text{LO}).$$

In other words, K is the sum of execution times of all jobs j_r whose $a_r \geq t^{\Delta}$ and $d_r^{\Delta} \leq d_{ik}^{\Delta}$ and L is the sum of execution times of all jobs whose finishing times lie after d_{ik}^{Δ}. Similarly, Ψ is the sum of execution times after DVFS of all jobs whose $a_r \geq t^{\Delta}$ and $d_r^{\Delta} \leq d_{ik}^{\Delta}$ and Ω is the sum of execution times after DVFS of all jobs whose finishing times lie after d_{ik}^{Δ}. Now our energy consumption function is modified as follows:

$$\sum_{j_q \in J \wedge a_q \leq t^{\Delta} \wedge d_q \leq t^{\Delta}} C_q(\mathrm{LO}) \cdot (f^{\mathrm{LO}})^{\alpha-1} + \sum_{j_r \in J \wedge a_r \geq t^{\Delta} \wedge d_r^{\Delta} \leq d_{ik}^{\Delta}} C_r(\mathrm{LO}) \cdot (f^{\mathrm{LO}} + x)^{\alpha-1} + \sum_{j_s \in J \wedge d_s^{\Delta} > d_{ik}^{\Delta}} C_s(\mathrm{LO}) \cdot (f^{\mathrm{LO}} - y)^{\alpha-1} \tag{4.22}$$

The first part of Eqn. 4.22 always consumes less energy as each job is assigned the same processor frequency and the finishing time of each job is less than t^{Δ}. From Eqn. 4.22, we can verify that $\sum_{j_q \in J \wedge d_q^{\Delta} \leq t^{\Delta}} C_q(\mathrm{LO}) \cdot (f^{\mathrm{LO}})^{\alpha-1}$ will always consume minimum energy as the processor frequency for each job is minimum (f^{LO}), because further reducing the frequency will create empty spaces in the schedule. We need to show that any processor frequency other than $f^{\mathrm{LO}} + x$ and $f^{\mathrm{LO}} - y$ results in increasing the total energy consumption. It is easy to see that decreasing the value of δ decreases the processor frequency $f^{\mathrm{LO}} + x$ which will lead the job j_{ik} to miss its finishing time d_{ik}^{Δ}. Furthermore, we need to show that the energy consumption will increase with the increase of δ. This is because, increasing δ results in increasing $f^{\mathrm{LO}} + x$ and decreasing $f^{\mathrm{LO}} - y$. So we need to show that the sum of the last two terms of Eqn. 4.22 is an increasing function with respect to δ. Now we express the sum of the last two terms of Eqn. 4.22 with respect to δ as follows:

$$\sum_{j_r \in J \wedge a_r \geq t^{\Delta} \wedge d_r^{\Delta} \leq d_{ik}^{\Delta}} C_r(\mathrm{LO}) \cdot \left(f^{\mathrm{LO}} + \tfrac{K}{\Psi - \delta} - f^{\mathrm{LO}}\right)^{\alpha-1} + \sum_{j_s \in J \wedge d_s^{\Delta} > d_{ik}^{\Delta}} C_s(\mathrm{LO}) \cdot \left(f^{\mathrm{LO}} - f^{\mathrm{LO}} + \tfrac{L}{\Omega + \delta}\right)^{\alpha-1} \tag{4.23}$$

We simplify Eqn. 4.23 to get the following equation:

$$\frac{K^{\alpha}}{(\Psi - \delta)^{\alpha-1}} + \frac{L^{\alpha}}{(\Omega + \delta)^{\alpha-1}} \tag{4.24}$$

To show the function in Eqn. 4.24 is an increasing function of δ, we need to differentiate the function with respect to δ. The derivative of the function in Eqn. 4.24 with respect to δ is:

$$\frac{(\alpha - 1) \cdot K^{\alpha}}{(\Psi - \delta)^{\alpha}} - \frac{(\alpha - 1) \cdot L^{\alpha}}{(\Omega + \delta)^{\alpha}} \tag{4.25}$$

Now

$$\begin{aligned} \frac{(\alpha-1)\cdot K^{\alpha}}{(\Psi-\delta)^{\alpha}} - \frac{(\alpha-1)\cdot L^{\alpha}}{(\Omega+\delta)^{\alpha}} &\geq 0 \\ \Leftrightarrow \left(\frac{K}{L}\right)^{\alpha} &\geq \left(\frac{\Psi-\delta}{\Omega+\delta}\right)^{\alpha} \\ \Leftrightarrow \frac{K}{L} &\geq \frac{\Psi-\delta}{\Omega+\delta} \end{aligned} \tag{4.26}$$

If the above condition is true, then the function in Eqn. 4.24 is an increasing function with respect to δ. It is easy to see that decreasing the value of δ leads to a deadline miss. Hence we cannot decrease the value of δ. If we increase δ, then the right-hand side of the inequality 4.26 decreases and is always less than the left-hand side of the inequality 4.26. This is because, increasing δ results in increasing the value of denominator of the right-hand side and decreasing the value of the numerator. So the right-hand side will decrease with increasing δ and the condition is true with increasing δ. Hence, the function in Eqn. 4.24 is an increasing function with respect to δ. As a result, the function in Eqn 4.23 is also an increasing function with respect to δ. □

As a corollary to Lemma 4.4.2 and Lemma 4.4.4, we have our main result.

Theorem 4.4.1: If Algorithm 11 does not declare failure, then the energy consumption in a schedule produced by it is optimal.

Proof. Algorithm 11 achieves the minimum value of δ without missing any deadline. Hence the resulting energy consumption is minimum among all possible feasible schedules according to TT-Merge. □

Theorem 4.4.2: The schedule according to our algorithm consumes no more energy than the one produced by the energy-efficient EDF-VD algorithm.

Proof. From Theorem 4.3.1, we know that EDF-VD schedules fewer task sets than the TT-Merge algorithm. Here we consider the case where both energy-efficient EDF-VD and energy-efficient TT-Merge can schedule a task set. We know that the schedules constructed by the energy-efficient TT-Merge algorithm are without any idle time. Without loss of generality we assume that schedules produced by energy-efficient EDF-VD also have no idle time, as in that case we can always expand a job and decrease the total energy consumption. Since the energy-efficient EDF-VD algorithm finds an optimal solution in two cases, we need to consider both.

Case 1: Energy efficient EDF-VD algorithm assigns the minimum frequency $f_{\min}$ to each job.

According to Lemma 4.4.2, our algorithm assigns the same processor frequency $\frac{f_b}{r_{\text{LO}}}$ to each job, if all job meet their finishing time d^{Δ}. If $\frac{f_b}{r_{\text{LO}}} \leq f_{\min}$, then we set $\frac{f_b}{r_{\text{LO}}}$ to $f_{\min}$. From Lemma 4.4.2, we know that the maximum processor frequency between $f_{\min}$ and $\frac{f_b}{r_{\text{LO}}}$ will always result in the minimum energy solution. Since both the algorithms use the EDF algorithm to check for schedulability, if energy-efficient EDF-VD assigns $f_{\min}$ to each job then so does our algorithm.

Case 2: Energy efficient EDF-VD fails to find the lowest possible energy solution (i.e., a job does not meet its deadline using the processor frequency $f_{\min}$):

Then energy-efficient EDF-VD computes two separate processor frequencies $f_{\text{LO}}^{\text{LO}}$ and $f_{\text{HI}}^{\text{LO}}$ for LO-criticality and HI-criticality jobs, respectively. We assume that energy-efficient EDF-VD successfully schedules all the jobs using processor frequencies $f_{\text{LO}}^{\text{LO}}$ and $f_{\text{HI}}^{\text{LO}}$. Let the processor frequencies for each job assigned by energy-efficient EDF-VD be less than those assigned by our algorithm. We know that our algorithm assigns two different processor frequencies to the job of task set $\mathscr{T}$ if and only if there is a finishing time miss by a job j_{ik} with processor frequency $\frac{f_b}{r_{\text{LO}}}$. Then the processor frequencies of those jobs whose finishing times before d_{ik}^{Δ} are increased using the function CheckEmptySpace() and the rest of the jobs whose finishing times lie after d_{ik}^{Δ} are decreased. In Lemma 4.4.4, we have already proved that increasing or decreasing the processor frequency of any job from those that are assigned by our algorithm will lead to deadline misses or increase in the total energy consumption. If $f_{\text{LO}}^{\text{LO}}$ and $f_{\text{HI}}^{\text{LO}}$ are less than the frequencies assigned by our algorithm, then the jobs will miss their deadline. This is a contradiction. On the other hand, it is easy to see that if $f_{\text{LO}}^{\text{LO}}$ and $f_{\text{HI}}^{\text{LO}}$ are greater than the frequencies assigned by our algorithm, then there will be empty spaces in the schedule. This is a contradiction.

Case 3: One of the processor frequencies $f_{\text{LO}}^{\text{LO}}$ and $f_{\text{HI}}^{\text{LO}}$ computed by energy-efficient EDF-VD is less or the other one is greater than the processor frequency assigned by our algorithm to some job. Here we assume that our algorithm faces only one finishing time miss which results in two different processor frequency assignment for each job. In this case, we get either a finishing time miss when the processor frequency assigned by EDF-VD is less than that assigned by our algorithm or an empty space when the processor frequency assigned by EDF-VD is greater than that assigned by our algorithm. Suppose job j_{ik} misses its finishing time d_{ik}^{Δ}. Then the function SetFrequency() reduced the running time

(by increasing the processor frequency) of all the jobs whose deadline is less than d_{ik}^{Δ} and increasing the running time (by decreasing the processor frequency) of all the jobs whose deadline is greater than d_{ik}^{Δ}. We know that both the algorithms use EDF algorithm to verify schedulability of the jobs. Clearly, EDF-VD has to schedule all those jobs before d_{ik}^{Δ} whose finishing time is less than or equal to d_{ik}^{Δ} as they will miss their deadlines if scheduled later. This is because the finishing time computed by our algorithm for each job is as close to its deadline as possible. From Lemma 4.4.2, we know that the same processor frequency assigned to each job results in minimum energy consumption. Our algorithm assigns two different processor frequency to each job scheduled before and after d_{ik}^{Δ}, respectively. Suppose EDF-VD meets the finishing time d_{ik}^{Δ} and jobs present before and after d_{ik}^{Δ} contains both LO- and HI-criticality. Using Lemma 4.4.2, we know that the same processor frequency assigned to each job results in minimum energy consumption. Hence the schedule generated by the energy-efficient TT-Merge algorithm results in minimum energy consumption. If jobs present before and after d_{ik}^{Δ} contains LO-criticality and HI-criticality, respectively, then the schedule computed by both the algorithm consumes same energy. On the other hand, suppose EDF-VD schedules the jobs present on the left hand side of d_{ik}^{Δ} quickly and the jobs present on the right hand side of d_{ik}^{Δ} slowly as compared to our algorithm. This means EDF-VD increases the value of δ. From Lemma 4.4.4, we know that increasing δ leads to increase in energy consumption. Using Lemma 4.4.2 and 4.4.3, we can easily conclude that the energy-efficient schedule constructed by EDF-VD gets either a deadline miss or has empty spaces in the schedule. This is a contradiction. This case can be easily extended for more than two processor frequencies assigned by our algorithm to a job.

So the processor frequencies assigned by the energy-efficient EDF-VD algorithm are equal or greater than those assigned by our algorithm. Hence the energy consumption for the energy-efficient EDF-VD algorithm is equal or greater than the energy consumed for our algorithm. □

4.4.2 Extension for Discrete Frequency Levels

In previous sections, we discussed the minimization of energy consumption in a mixed-criticality system using continuous processor frequency levels. In practice, continuous processor frequency levels will not be available. We need to extend our algorithm to find a schedule which consumes less energy as compared to the energy-efficient EDF-VD algorithm

while considering only discrete processor frequency levels.

Suppose the set of processor frequencies available in the model is $F = \{f_1, f_2, \ldots, f_{|F|}\}$, where $f_1 = f_{\min}$ and $f_{|F|} = f_{\max}$. We need to find processor frequencies for each job such that the energy consumption can be minimized in the LO-criticality scenario using DVFS method without affecting the schedulability of the system in both the scenarios.

We can modify our algorithm discussed in Section 4.4 to assign processor frequencies from F. For example, if a job j_{ik} is assigned a processor frequency $f_m < f_{ik}^{\text{LO}} < f_n$, then processor frequency f_n can be assigned to job j_{ik}, where f_m and f_n are two consecutive processor frequency in F. Note that the resulting schedule may not be optimal with respect to energy consumption.

4.5 Extension of the Proposed Algorithm for Dependent Task Sets

In previous sections, we have considered task sets with independent tasks. To the best of our knowledge, minimizing the energy consumption of mixed-criticality systems has not been explored for dependent tasks. Here we consider the case of dual-criticality instances with dependent tasks. In Chapter 3, we proposed a time-triggered scheduling algorithm for mixed-criticality systems with dependent tasks which finds the two scheduling tables, i.e., $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$. Here we focus on integrating DVFS with the TT-Merge-Dep algorithm to minimize the energy consumption in mixed-criticality systems with dependency constraints. As in the case of independent tasks, we propose an algorithm which finds a processor frequency f_{ik}^{LO} and energy-efficient LO-criticality WCET $\tilde{C}_{ik}(\text{LO})$ for each job j_{ik} of a task set which minimizes the energy consumption. These $\tilde{C}_{ik}(\text{LO})$ units of execution time are used by the TT-Merge-Dep algorithm to find the scheduling tables which gives minimum energy consumption.

4.5.1 Model

The task model is similar to the one discussed in Section 4.2. The dependencies among the tasks is defined by a *directed acyclic graph* (DAG). A task set is represented in the form of $\mathscr{T}(V, E)$, where V represents the set of tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$ and E represents the dependency between the tasks. If there is an edge $\tau_i \rightarrow \tau_j$, then the execution of τ_i must

precede the execution of τ_j, denoted by $\tau_i \prec \tau_j$. We assume that no HI-criticality job can depend on a LO-criticality job. This means, there will be no task set where an outward edge from a LO-criticality task becomes an inward edge to a HI-criticality task.

Definition 4.5.1: A dual-criticality MC task set with task dependencies is said to be **time-triggered schedulable** if it is possible to construct the two scheduling tables $\mathcal{S}_{\text{LO}}$ and $\mathcal{S}_{\text{HI}}$ for task set $\mathscr{T}$ without violating the dependencies, such that the run-time scheduler schedules $\mathscr{T}$ in a correct manner.

4.5.2 Problem Formulation

Here we formally present our energy optimization problem for mixed-criticality systems with dependent tasks. Our goal is to minimize the LO-criticality scenario energy consumption by expanding the task executions in the schedule while ensuring that they do not miss their deadlines without violating the dependency constraints. Without loss of generality, we calculate the energy consumption minimization up to the hyperperiod P of the task set. As in the earlier case, we find a LO-criticality WCET $\tilde{C}_{ik}(\text{LO})$ for each job of a task set. This is accomplished by expanding the execution time of a task as much as possible so that the minimum expansion of each job is maximized and the processor works on a slower frequency to minimize the energy consumption and each task also meets its deadline without violating the dependency constraints. Then using these LO-criticality WCETs $\tilde{C}_{ik}(\text{LO})$, the TT-Merge-Dep algorithm finds a minimized energy consumption schedule. When the system switches to HI-criticality scenario, the processor frequency is set to f_b. Here the problem is similar to Problem 1 in Section 4.3 with dependency constraints. So our energy objective to be minimized in the LO-criticality scenario by varying the processor frequency is:

$$\frac{1}{P} \cdot \sum_{\tau_i \in \mathscr{T} \wedge 1 \leq k \leq N_i} C_i(\text{LO}) f_b \cdot \frac{1}{f_{ik}^{\text{LO}}} \cdot \beta \cdot (f_{ik}^{\text{LO}})^{\alpha} \tag{4.27}$$

The energy minimization is constrained by:

- Bound for the frequency for each job:

$$f_i \in [f_{min}, f_{max}]. \tag{4.28}$$

where f_i is the frequency of job j_{ik}.

- Construction of table $\tilde{\mathcal{T}}_{\text{LO}}$ and $\tilde{\mathcal{T}}_{\text{HI}}$:

  It should be possible to construct $\tilde{\mathcal{T}}_{\text{LO}}$ and $\tilde{\mathcal{T}}_{\text{HI}}$ using $\tilde{C}_i(\text{LO})$ and $\tilde{C}_i(\text{HI})$ units of execution time of job j_{ik}, respectivley without missing any deadline as TT-Merge-DEP of Chapter 3 (4.29)

  where $\tilde{C}_i(\text{LO})$ and $\tilde{C}_i(\text{HI})$ are the time taken to execute $C_i(\text{LO})$ and $C_i(\text{HI})$ units of execution time, respectively.

- Construction of table $\tilde{\mathcal{S}}_{\text{LO}}$:

  It should be possible to construct $\tilde{\mathcal{S}}_{\text{LO}}$ while ensuring the failure situation (4) of TT-Merge-DEP of Chapter 3 does not occur, at any time t (i.e., $\tilde{\mathcal{T}}_{\text{LO}}[t]$ is not empty and $\tilde{\mathcal{T}}_{\text{HI}}[t]$ is not empty, at time t. (4.30)

  where $\tilde{\mathcal{T}}_{\text{LO}}[t]$ contains a LO-criticality job and $\tilde{\mathcal{T}}_{\text{HI}}[t]$ contains a HI-criticality job at time t, respectively.

- Dependency constraints:

$$\forall i, k, \text{ if } \forall j_{ik} \prec j_{il}, \text{then } e_{ik} < s_{il} \tag{4.31}$$

  where e_{ik} and s_{il} are the completion time and starting time of task j_{ik} and j_{il}, respectively in the scheduling table.

Finally, our energy minimization problem is

$$\begin{aligned} &\textbf{minimize}(4.27) \\ &\textbf{s.t.}(4.28), (4.29), (4.30), (4.31) \end{aligned} \tag{4.32}$$

4.5.3 The Algorithm

In this section, we present an algorithm to find the processor frequency f_{ik}^{LO} for each job which can be used by the TT-Merge-Dep algorithm to schedule a dependent task set successfully. First, we find the tables E_{LO} and E_{HI}. Then the table E_{FINAL} is constructed using these two tables. The length of all these tables are P, i.e., the hyperperiod of all the tasks in the task set. These steps are similar to the TT-Merge-Dep algorithm. But inputs for the TT-Merge-Dep algorithm are non-recurrent jobs. So we need to find all the jobs and the dependencies between them over a hyperperiod P which can be done by the method given in [Bar14].

We then can use the same function SetFrequency() of Section 4.4. But we need to check the dependencies among the jobs. The first step of function SetFrequency(E_{LO},E_{HI},E_{FINAL}) for the dependent jobs moves each segment of a job j_{ik} as close to its finishing time in E_χ as possible without disturbing the dependency constraints, where χ is the criticality of j_{ik}. Then all the steps are similar to the function SetFrequency(E_{LO},E_{HI},E_{FINAL}) of Section 4.4. We explain the algorithm with the help of an example.

Example 4.5.1: Consider the task set given in Table 4.1. The dependencies among the tasks are shown in Fig. 4.6. Here the $f_{\min}$ and $f_{\max}$ are set to 0.5 and 1, respectively. We assume $\alpha = 3$.

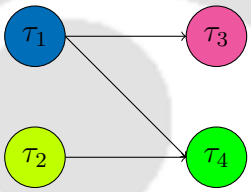


Figure 4.6: Dependencies among the tasks given in Table 4.1

First, we unroll the graph given in Fig. 4.6 to get all the jobs and their dependencies over a hyperperiod using the method given in [Bar14]. The resulting graph is shown in Fig. 4.7.

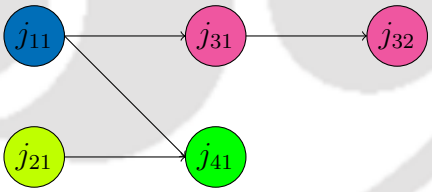


Figure 4.7: Dependencies among the tasks given in Table 4.1 after unroll

Now we find the tables E_{LO} and E_{HI} as shown in Fig. 4.8 using the TT-Merge-Dep algorithm. We then construct table E_{FINAL} by merging the tables E_{LO} and E_{HI} using the

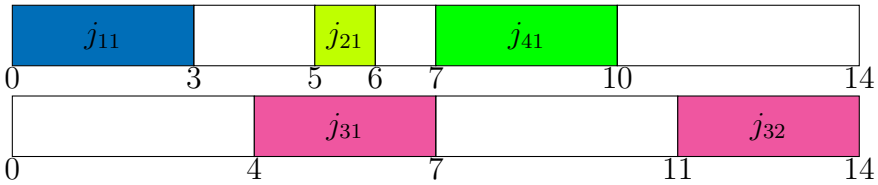


Figure 4.8: Tables E_{LO} and E_{HI}

TT-Merge-Dep algorithm as shown in Fig. 4.9.

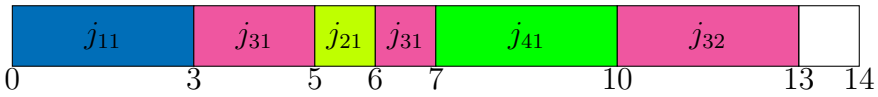


Figure 4.9: Table E_{FINAL}

- Then starting from the right, each job j_{ik} is moved as close to its finishing time in E_χ as possible without disturbing the dependency constraints, where χ is the criticality of job j_{ik}. So job j_{32} is moved to its right from $[10, 13]$ to $[11, 14]$ as its finishing time in table E_{LO} is 14. The final table E_{FINAL} is given in Fig. 4.10.

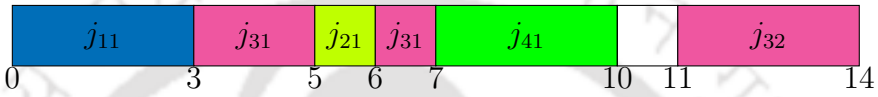


Figure 4.10: Table E_{FINAL} after each job is moved to its finishing time in E_χ

- Here $EMPTY_1^S$ and P are 10 and 14, respectively. The total execution time between $EMPTY_1^S$ and P in table E_{FINAL} is 3.
- $r_{\text{LO}} = \frac{P - EMPTY_1^S}{\sum_{\forall j_{ik} \in J} C_{ik}(\text{LO})} = \frac{14-10}{3} = \frac{4}{3}$ where $f_{\min} < \frac{1}{r_{\text{LO}}}$. This means the lowest possible processor frequency for each job in J is 0.75.
- We find the $\tilde{C}_{ik}(\text{LO})$ for each job $\langle j_{11}, j_{21}, j_{31}, j_{41}, j_{32} \rangle$ to be $\langle 3, 1, 3, 3, 4 \rangle$.
- Now we use formula 4.13 to check the schedulability of each job with the $\tilde{C}_{ik}(\text{LO})$ units of execution time. All the jobs pass this test.
- We then find the processor frequencies $f_{ik}(\text{LO})$ for each job $\langle j_{11}, j_{21}, j_{31}, j_{41}, j_{32} \rangle$ to be $\langle 1, 1, 1, 1, 0.75 \rangle$.
- The total normalized energy consumption by the TT-Merge energy-efficient algorithm using the above processor frequencies is 0.835.

4.6 Results and Discussion

In this section we present the experiments conducted to evaluate our algorithm. The experiments show the impact of the utilization of task sets on the energy-efficient TT-Merge algorithm versus the energy-efficient EDF-VD algorithm. The comparison is done

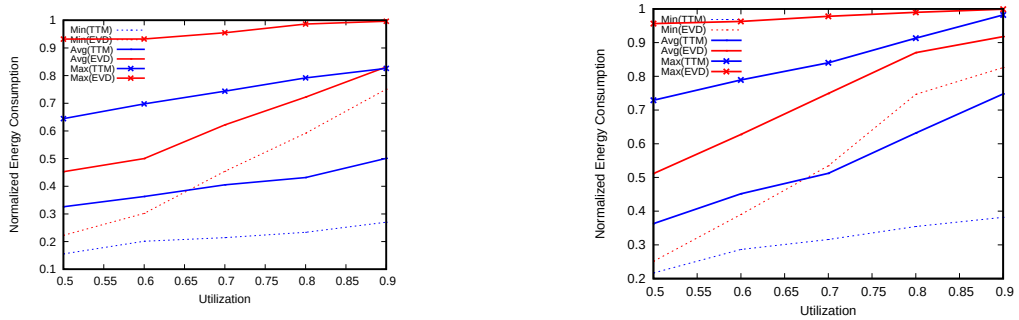
over numerous task sets with randomly generated parameters. The task generation policy may have a significant effect on the experiments. We follow a similar task generation policy as in Chapter 3. The details of the task generation policy are as follows.

- The utilization (u_i) of tasks in a task set are generated according to the UUniFast algorithm [BB05].
- The periods (p_i) of the tasks are generated by using exponential distribution proposed by Davis et al [DZB08].
- The C_i(LO) units of execution time of the tasks are calculated by $u_i \times p_i$.
- The C_i(HI) units of execution time of the tasks are calculated as $C_i(\text{HI}) = \text{CF} \times C_i(\text{HI})$, where CF is the criticality factor which varies between 2 and 6 for each HI-criticality tasks
- Each task set contains at least one HI-criticality job and one LO-criticality job.

In these experiments, we consider only those results where both the energy-efficient TT-Merge algorithm and energy-efficient EDF-VD algorithm successfully find a schedule. For each point on the X-axis, we plot the average results of 200 runs, where a run is a single execution of the algorithm on a task set with a given utilization. The energy consumption plotted in the graphs are normalized energy consumption. We vary the utilization at the LO-criticality scenario of the task sets between 0.5 and 0.9, and we set periods of the tasks between 100 and 1000. The processor frequency f_{min} is set to 0.2, β is set to 1 and α to 2.5. The number of tasks in the task set is set to 10. The names TTM and EVD used in the graphs are shorthand for the energy-efficient TT-Merge and EDF-VD algorithms, respectively.

In the first experiment, we plot the minimum, maximum and average energy consumption of both the algorithms for each utilization. The graph in Fig. 4.11a shows that the energy consumption increases for both the algorithms with the increase in the utilization. It is evident from the graph that the schedule constructed according to the energy-efficient TT-Merge algorithm consumes less energy than the one constructed by the energy-efficient EDF-VD algorithm for all utilizations. For detailed analysis, we use Box-whisker plots.

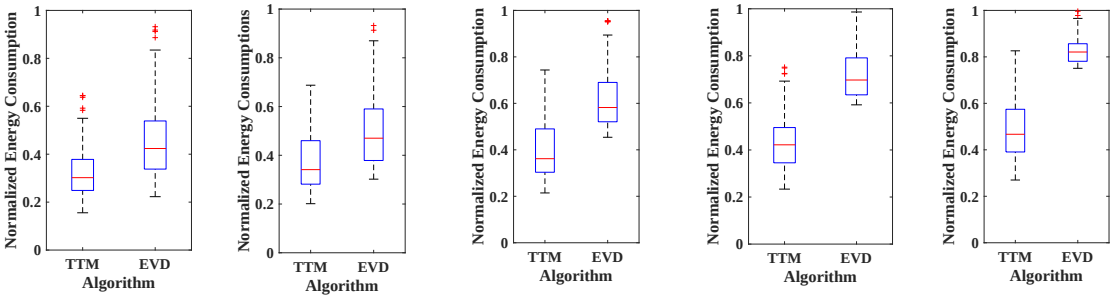
From the graphs in Fig. 4.12, we can see that the maximum energy consumption of task sets by our algorithm lies between 0.2 and 0.6 for all utilizations, where it is between 0.3 and



(a) At least one LO-criticality and one HI-criticality task

(b) More than 50% of HI-criticality tasks

Figure 4.11: Comparison of normalized energy consumption between Energy-Efficient EDF-VD and TT-Merge

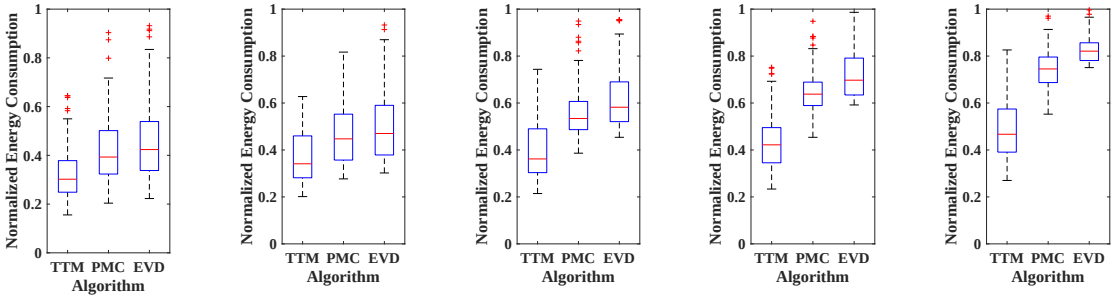


(a) U = 0.5 (b) U = 0.6 (c) U = 0.7 (d) U = 0.8 (e) U = 0.9

Figure 4.12: Comparison between TTM and EVD where LO-criticality scenario utilization ranges from 0.5 to 0.9

0.8 for energy-efficient EDF-VD. For example, when the LO-criticality scenario utilization is 0.7, the energy consumption by the schedule constructed by our algorithm is between 0.3 and 0.5 for maximum number of task sets and the same for the energy-efficient EDF-VD is between 0.5 and 0.7, respectively. Clearly, our algorithm outperforms the energy-efficient EDF-VD algorithm.

The next experiment finds the impact of the number of HI-criticality tasks in a task set. Here each task set contains more than 50% of HI-criticality tasks. The other parameters are the same as the previous experiment. From the graph in Fig. 4.11b, we can observe that the LO-criticality scenario energy consumption increases with the number of HI-criticality tasks increases. This is because of the lack of slack times due to the time reservation for HI-criticality jobs.



(a) U = 0.5 (b) U = 0.6 (c) U = 0.7 (d) U = 0.8 (e) U = 0.9

Figure 4.13: Comparison between TTM, PCM and EVD where LO-criticality scenario utilization ranges from 0.5 to 0.9

Now we compare our algorithm with energy-efficient EDF-VD and the work in [ASK15], which we abbreviated as PMC. Here we assume that the available processor frequencies are $\{0.4, 0.5, 0.6, 0.7, 0.8, 0.9, 1\}$. Now we plot the Box-whisker plots to show the minimum, maximum and average energy consumption for all the three algorithms for each utilization. From Fig. 4.13, it is clear that the schedule constructed according to our algorithm consumes less energy than both the existing algorithms. We also can see that the schedule constructed according to our algorithm consumes 20% and 25% less energy than the one constructed by the energy-efficient EDF-VD and PMC algorithms, respectively at higher LO-criticality utilization.

4.7 Conclusion

In this chapter, we proposed a new algorithm to find energy efficient time-triggered schedules using the TT-Merge algorithm. The proposed algorithm finds a processor frequency and the corresponding LO-criticality WCET for all the jobs of a task set which are then used by the TT-Merge algorithm. We proved that the schedule constructed according to the energy-efficient TT-Merge algorithm consumes less energy than the one constructed by the energy-efficient EDF-VD algorithm. The experiments also validated the fact. We then proved the optimality of our algorithm with respect to energy consumption for schedulability under the TT-Merge algorithm. We also discussed the impact of discrete processor frequency levels on our algorithm. We then extended our algorithm for dependent task sets. As part of future work, we plan to extend this algorithm to the multiprocessor case. The algorithm proposed

in this chapter can be used in a multiprocessor system using the partitioned scheduling technique. We plan to extend our technique for a global multiprocessor scheduling algorithm in the future.

Chapter 5

Time-triggered Scheduling of Multiprocessor Mixed-criticality Systems

5.1 Introduction

In this chapter, we propose an approach to find preemptive, global, time-triggered schedules for mixed-criticality non-recurrent task systems on identical multiprocessor platforms. In Chapter 2 we discussed various types of scheduling policies for traditional multiprocessor systems. Here we investigate a global time-triggered scheduling strategy for a dual-critical mixed-criticality task system to be scheduled on a multiprocessor system. We find two scheduling tables for the LO-criticality and HI-criticality levels (S_{LO} and S_{HI}), which are then used by a scheduler to dispatch the jobs on-line according to the off-line scheduling tables. We show that the scheduling tables can successfully handle the mode changes at any time. We also show that the worst-case time complexity of our proposed algorithm is better than the MCPI algorithm [SPBB15], the only existing time-triggered algorithm for such systems. In addition, our algorithm is much easier to understand and reason about.

The rest of the chapter is organized as follows. Sections 5.2 and 5.3 describes the systems model and related work, respectively. In Section 5.4, we propose our algorithm for independent mixed-criticality jobs. Then Section 5.5 extends the algorithm for dependent jobs. Finally, we discuss the results for our experiment in Section 5.6.

5.2 System Model

The mixed-criticality systems used in this chapter are based on non-recurrent tasks. An instance or job set is a collection of n jobs $\{j_1, j_2, \ldots, j_n\}$, each with a criticality level. Here we focus on dual-criticality jobs, i.e., LO-criticality and HI-criticality. A job j_i is characterized by a 5-tuple of parameters: $j_i = (a_i, d_i, \chi_i, C_i(\text{LO}), C_i(\text{HI}))$, where

- $a_i \in \mathbb{N}$ denotes the *arrival time*, $a_i \geq 0$.
- $d_i \in \mathbb{N}^+$ denotes the *absolute deadline*, $d_i \geq a_i$.
- $\chi_i \in \{LO, HI\}$ denotes the *criticality* level.
- $C_i(\text{LO}) \in \mathbb{N}^+$ denotes the LO-criticality *worst-case execution time*.
- $C_i(\text{HI}) \in \mathbb{N}^+$ denotes the HI-criticality *worst-case execution time*.

We assume that the system is *preemptive* and $C_i(\text{LO}) \leq C_i(\text{HI})$ for $1 \leq i \leq n$. Note that in this chapter, we consider arbitrary arrival times of jobs. Scenarios in the mixed-criticality model discussed above can be of two types, i.e., *LO-criticality scenarios* and *HI-criticality scenarios*. The details of scenario is discussed in Chapter 2. Now we define a schedulability condition for a mixed-criticality instance I.

Definition 5.2.1: A scheduling strategy is *feasible or correct* if and only if the following conditions are true:

1. If all the jobs finish their $C_i(\text{LO})$ units of execution time on or before their deadlines.
2. If any job does not declare its completion after executing its $C_i(\text{LO})$ units of execution time, then all the HI-criticality jobs must finish their $C_i(\text{HI})$ units of execution time on or before their deadlines.

Here we focus on the **time-triggered schedule** [Kop98] of MC instances on a multiprocessor system with identical processors. We will construct two tables S_{HI} and S_{LO} for each processor for a given instance I for use at run-time. The length of the tables is the length of the interval $[\min_{j_i \in I}\{a_i\}, \max_{j_i \in I}\{d_i\}]$. The rules to use the tables S_{HI} and S_{LO} at run-time, (i.e., the *scheduler*) are as follows:

- The criticality level indicator Γ is initialized to LO.

- While ($\Gamma = LO$), at each time instant t the job available at time t in the table S_{LO} for processor P_i will execute on P_i.
- If a job executes for more than its LO-criticality WCET without signaling completion in any processor P_i, then Γ is changed to HI.
- While ($\Gamma = HI$), at each time instant t the job available at time t in the table S_{HI} for processor P_i will execute on P_i.

5.3 Related Work

Most research on mixed-criticality systems focuses on the uniprocessor case (see for example, [BBD+12a, SPBB13]). The increasing functionalities in mixed-criticality systems motivate researchers to turn to multiprocessor systems (see [BCLS14b, GSHT14, GSHT13a, Pat12, SPBB15]). Among the above cited work only [BF11, SPBB13] focus on a time-triggered scheduling algorithm for uniprocessor systems and [SPBB15] introduces a time-triggered scheduling algorithm for multiprocessor systems. To the best of our knowledge, there has not been *any other work* studying time-triggered mixed-criticality scheduling for multiprocessor systems.

Socci et al. [SPBB15] proposed the Mixed-criticality Priority Improvement (MCPI) algorithm to schedule jobs with precedence constraints. In this algorithm, they construct a priority order of jobs from the support algorithm (i.e., a multiprocessor algorithm for non-critical jobs) which is used to find a table for the LO-scenario and the support algorithm is used to schedule the HI-criticality jobs in HI-scenarios. They showed the worst-case time complexity of the algorithm to be $O(mn^3 \log n)$ for independent jobs and $O(|E|n^2 + mn^3 \log n)$ for dependent jobs, where n is the number of jobs in the instance I, m is the number of processors and E depicts the dependency between jobs.

5.4 The Proposed Algorithm: LoCBP

In this section, we propose an algorithm for mixed-criticality jobs on multiprocessor systems which not only schedules the same set of instances as the existing algorithm [SPBB15] but also has a better worst-case time complexity. Hereafter, abbreviated as *LoCBP algorithm* (LO-criticality based Priority).

The time-triggered scheduling approach to mixed-criticality jobs [SPBB15] constructs two scheduling tables S_{LO} and S_{HI} to schedule a dual-criticality instance. Since we consider mixed-criticality jobs for a multiprocessor system, we need two separate scheduling tables for each processor. The schedule constructed by our algorithm is a global one, i.e., a job can be preempted in one processor and resume its execution in another processor. Here we assume that the system is a closely coupled synchronous homogeneous multiprocessor system with shared last level cache and the job context switch time is negligible. We also assume that the cache miss penalty is negligible.

Algorithm 14 LoCBP(I)

Notation:
$I = \{j_1, j_2, ..., j_n\}$, where $j_i =< a_i,\ d_i,\ \chi_i,\ C_i(\text{LO}),\ C_i(\text{HI}) >$.
Input : I
Output : Priority Order (Ψ) of Instance I
Assume the earliest arrival time is 0.

1: Compute the LO-scenario deadline (${d_i}^{\Delta}$) of each job j_i as ${d_i}^{\Delta} = d_i - (C_i(\text{HI}) - C_i(\text{LO}))$;
2: **while** I is not empty **do**
3: Assign a LO-criticality latest deadline[1]job j_i as the lowest priority job if j_i can finish its execution in the interval $[a_i, {d_i}^{\Delta}]$ after all other jobs finish their execution in LO-scenario under the global EDF scheme;
4: If any LO-criticality job cannot be given a lowest priority then a HI-criticality latest deadline job j_i is assigned as the lowest priority job if j_i can finish its execution in the interval $[a_i, {d_i}^{\Delta}]$ after all other jobs finish their execution in LO-scenario under the global EDF scheme;
5: **if** no job is assigned a lowest priority **then**
6: Declare FAIL and EXIT;
7: **else**
8: Add job j_i to the priority order Ψ;
9: Remove job j_i from the instance and continue;
10: **end if**
11: **end while**
12: Construct table S_{LO} for each processor using the priority order;
13: **if** $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ **then**
14: return FAIL and EXIT;
15: **end if**
16: The same order as S_{LO} is followed to allocate the jobs in S_{HI};
17: After a HI-criticality job j_i is allocated its $C_i(\text{LO})$ execution time in S_{HI}, $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time of job j_i is allocated after the rightmost segment of job j_i in S_{LO} without disturbing the priority order Ψ and overwriting LO-criticality jobs in the process, if any;

[1]The original deadline and not the LO-scenario one.

Algorithm 14 determines a priority order, which is used to construct the scheduling tables for all the processors, in steps 1 to 11. First, our algorithm finds the LO-scenario deadline ($d_i{}^{\Delta}$) of each job. For the LO-criticality jobs $d_i{}^{\Delta} = d_i$, but for HI-criticality ones $d_i{}^{\Delta} \leq d_i$. Then the algorithm starts to assign the lowest priority jobs from the instance I. It always selects the latest deadline job to be assigned as the lowest priority job, but LO-criticality jobs are considered before the HI-criticality jobs. A job j_i can be assigned the lowest priority if and only if all other jobs j_k finish their executions when run according to the global EDF algorithm and there remains sufficient time for j_i to complete its $C_i(\text{LO})$ units of execution time before $d_i{}^{\Delta}$. After job j_i is assigned the lowest priority, it is removed from the instance, and the remaining jobs are considered for priority assignment. If at any step a job cannot be assigned a priority, the algorithm declares failure. In step 10, the algorithm constructs table S_{LO}.

In steps 11 to 13, it checks for any possible HI-criticality scenario failure. The subroutine $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ checks if at least one job runs at its $C_i(\text{HI})$ execution time, then all HI-criticality jobs must complete their HI-criticality execution before their deadline. If it does not find a HI-criticality scenario failure from the subroutine $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$, then the priority order constructed by Algorithm 14 can successfully schedule the instance I. Algorithm 14 constructs table S_{LO} for each processor. Then Table S_{HI} is constructed for each processor by allocating the remaining $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time of each HI-criticality job after its $C_i(\text{LO})$ units of execution time in S_{HI} using the same priority order and also a HI-criticality job is given higher priority over LO-criticality jobs. This means a HI-criticality job can overwrite a LO-criticality job in the process of allocating its $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time.

We illustrate the operation of this algorithm by an example.

Example 5.4.1: Consider the mixed-criticality instance given in Table 5.1 to be scheduled on a multiprocessor system having two identical processors P_0 and P_1.

Now we construct a priority order using our algorithm. The LO-scenario deadlines d_i^{Δ} of jobs j_1, j_2, j_3, j_4 are $5, 8, 5, 4$ respectively. Now we start assigning priorities to each job.

- The job j_2 is the latest LO-criticality deadline job. If j_2 is assigned the lowest priority, then j_3 and j_4 can run simultaneously in P_0 and P_1 over $[0, 3]$ and $[0, 2]$ respectively. Then j_1 will run over $[2, 5]$ in P_1. So j_2 can execute its 4 units of execution time in P_0 over $[3, 7]$ to finish by its deadline. Now we can assign job j_2 the lowest priority. We remove job j_2 and consider $\{j_1, j_3, j_4\}$ to find the next lowest priority job.

Table 5.1: An example instance to explain the LoCBP algorithm

Job	Arrival time	Deadline	Criticality	C_i(LO)	C_i(HI)
j_1	1	5	LO	3	3
j_2	0	8	LO	4	4
j_3	0	7	HI	3	5
j_4	0	4	HI	2	2

- If j_1 is assigned the lowest priority, then j_3 and j_4 can run simultaneously on P_0 and P_1 over $[0, 3]$ and $[0, 2]$ respectively. Then j_1 will run over $[2, 5]$ in P_1. So j_1 can execute its 3 units of execution time in P_1 over $[2, 5]$ to finish by its deadline. Now we can assign job j_1 the lowest priority. Next, we remove the job j_1 and consider $\{j_3, j_4\}$ to assign the next lowest priority.

- Since there are two jobs and two processors, any job can be given lower priority among the two. But our algorithm assigns the latest deadline job as the lowest priority job. So job j_3 is given the lowest priority.

Finally, the priority order of the jobs in instance I is $j_4 \triangleright j_3 \triangleright j_1 \triangleright j_2$. Now Algorithm 14 constructs the table S_{LO} for each processor using the above priority order. The table S_{LO} for each processor is given in Fig. 5.1.

Then the $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ subroutine checks for all possible HI-criticality scenarios. We can check that all HI-criticality scenarios are schedulable using the priority order $\{j_4, j_3, j_1, j_2\}$ of I. Finally, table S_{HI} is constructed for each processor by allocating the remaining $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time of each HI-criticality job after its $C_i(\text{LO})$ units of execution time in S_{HI} using the same priority order, where a HI-criticality job is given higher priority over LO-criticality jobs. The table S_{HI} for each processor is given in Fig. 5.2. □

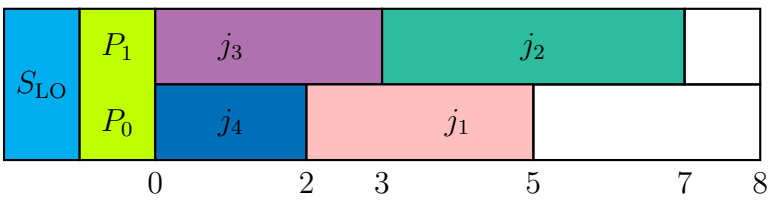


Figure 5.1: Table S_{LO} for processor P_0 and P_1

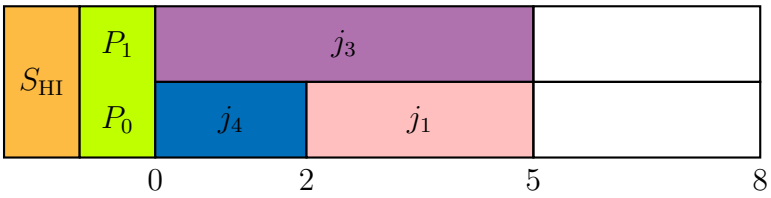


Figure 5.2: Table S_{HI} for processor P_0 and P_1

5.4.1 Correctness Proof

For correctness, we have to show that if our algorithm finds a priority order for instance I and the $anyHIscenarioFailure(S_{\text{LO}}, I)$ subroutine does not fail, then the scheduling tables S_{LO} and S_{HI} will give a correct scheduling strategy. We start with the proof of some properties of the schedule.

Lemma 5.4.1: If the LoCBP algorithm 14 does not declare failure and finds a priority order, then each job j_i receives $C_i(\text{LO})$ units of execution time in S_{LO} and each HI-criticality job j_k receives $C_k(\text{HI})$ units of execution time in S_{HI}.

Proof. First, we show that any job j_i receives $C_i(\text{LO})$ units of execution time in S_{LO}. This follows directly from the algorithm as each job j_i must finish its $C_i(\text{LO})$ units of execution time before ${d_i}^{\Delta} \leq d_i$ to be assigned the lowest priority job.

Next we show that any HI-criticality job j_k receives $C_k(\text{HI})$ units of execution time in S_{HI}. We construct the table S_{HI} according to the same priority order. Since $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ subroutine does not find any HI-criticality scenario failure, so all the HI-criticality jobs have received their $C_i(\text{HI})$ units of execution time. □

Lemma 5.4.2: At any time t, if a job j_i is present in S_{HI} but not in S_{LO}, then the job j_i has finished its execution in S_{LO}.

Proof. We use the same order of jobs in S_{LO} to construct S_{HI}. Whenever a job j_i has executed for time $c_i \leq C_i(\text{LO})$ at time t, then it is present in both the tables S_{LO} and S_{HI}. We know the HI-criticality jobs are allocated their $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time after the allocation of $C_i(\text{LO})$ units of execution time in both S_{HI} and S_{LO}. In S_{HI}, the HI-criticality jobs are higher priority job than LO-criticality jobs. When a job j_i is present in S_{HI} and not in S_{LO} at time t, it means this has already completed its execution in S_{LO}. □

Lemma 5.4.3: At any time t, when a mode change occurs, each HI-criticality job still has $C_i(\text{HI}) - c_i$ units of execution time in S_{HI} after time t to complete its execution, where c_i is the execution time already completed by job j_i before time t in S_{LO}.

Proof. Let a mode change occur at time t. This means that the following statements hold: (i) all the HI-criticality jobs other than the current job, or none of them has completed their $C_i(\text{LO})$ units of execution time at time t, (ii) the current HI-criticality job is the first one to complete its $C_i(\text{LO})$ units of execution time without signaling its completion. We know that all the HI-criticality jobs are allocated their $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time in S_{HI} after the completion of their $C_i(\text{LO})$ units of execution time in both S_{LO} and S_{HI}. If a job j_i has already executed its $C_i(\text{LO})$ units of execution time in S_{LO}, then it requires $C_i(\text{HI}) - C_i(\text{LO})$ units of time to be completed in S_{HI}. When job j_i initiates the mode change, this is the first job which does not signal its completion after completing its $C_i(\text{LO})$ units of execution time. Before time t, the scheduler uses the table S_{LO} to schedule the jobs, while subsequently the scheduler uses table S_{HI} due to the mode change. If a job j_i has already executed its c_i units of execution time in S_{LO}, then it requires $C_i(\text{HI}) - c_i$ units of time to be completed its execution in S_{HI}. We know that the tables S_{HI} and S_{LO} have the same order and according to Lemma 5.4.1 and 5.4.2, each job will get sufficient time to complete its $C_i(\text{HI})$ units of execution time. Hence, each HI-criticality job will get $C_i(\text{HI}) - c_i$ units of time in S_{HI} to complete its execution after the mode change at time t. □

Theorem 5.4.1: If the scheduler dispatches the jobs according to S_{LO} and S_{HI}, then it will be a correct scheduling strategy.

Proof. For the LO-criticality scenarios, all the jobs can be correctly scheduled by the table S_{LO} as proved in Lemma 5.4.1. Now, we need to prove that in a HI-criticality scenario, all the HI-criticality jobs can be correctly scheduled by the table S_{HI}. In Lemma 5.4.1, we have already proved that all the HI-criticality jobs get sufficient units of time to complete their execution in S_{HI}. In Lemma 5.4.3, we have proved that when the mode change occurs at time t, all the HI-criticality jobs can be scheduled without missing their deadline. So from Lemma 5.4.1 and Lemma 5.4.3, it is clear that if the scheduler uses tables S_{LO} and S_{HI} to dispatch the jobs then it will be a correct scheduling strategy. □

5.4.2 Comparison with MCPI Algorithm

Theorem 5.4.2: An instance I is schedulable by the MCPI algorithm [SPBB15] if and only if it is schedulable by our algorithm.

Proof. ($\Rightarrow$) The MCPI algorithm generates a priority order for an instance I which is used to find table S_{LO}. When a mode change occurs, it uses a support algorithm to schedule the HI-criticality jobs of instance I. We need to show that if MCPI generates a priority order for an instance I, then our algorithm will always find a priority order for instance I and the $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ subroutine will not fail.

Suppose the MCPI algorithm finds a priority order for an instance I. Now the least priority job of the priority order (according to the MCPI algorithm) can be either a LO-criticality or HI-criticality job. First, we consider the case where a job is of LO-criticality. Let j_i be the lowest priority job and its criticality be low. So at the time of construction of the table S_{LO}, every higher priority job j_k finishes its $C_k(\text{LO})$ units of execution time and there remains sufficient time for the lowest priority job j_i to finish its $C_i(\text{LO})$ units of execution time in the interval $[a_i, d_i]$. So this condition is the same as the LoCBP algorithm.

Let job j_i be the lowest priority job and its criticality be high. Since MCPI successfully finds the priority order, it must have checked all the scenarios and didn't find any failure. Now after every higher priority job j_k finishes its $C_k(\text{LO})$ units of execution time, there remains sufficient time for the lowest priority job j_i to finish its $C_i(\text{LO})$ units of execution time in the interval $[a_i, d_i^{\Delta}]$. Unlike the LO-criticality job, the HI-criticality jobs need to finish their LO-criticality execution on or before d_i^{Δ}. So this condition is the same as the LoCBP algorithm.

Then j_i is removed from the instance and the next priority can be assigned from the remaining jobs. We can argue in the same way for the remaining jobs. From the above argument, it is proved that the LoCBP algorithm finds the same priority order for instance I as the MCPI algorithm. Since the priority order is the same and the MCPI algorithm does not find any HI-scenario or LO-scenario failure, the $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ subroutine in our algorithm will not fail as well. Thus, for a MCPI schedulable instance, our algorithm can also construct priority tables S_{LO} and S_{HI}.

($\Leftarrow$) Our algorithm generates a priority order for an instance I which is used to find the table S_{LO}. When a mode change occurs, our algorithm uses the table S_{HI} to schedule the HI-criticality jobs which is constructed from the job ordering in S_{LO}. We need to show

that if our algorithm generates a priority order for an instance I, then the MCPI algorithm will always find a priority order and the $anyHIScenarioFailure(PT, T)$ subroutine will not fail.

Suppose our algorithm finds a priority order for an instance I. The least priority job assigned by our algorithm can be either a HI-criticality or a LO-criticality job. First, we consider the case where the lowest priority job is LO-criticality. Let j_i be the lowest priority job and its criticality be LO which means the job j_i finishes its execution between its arrival time and deadline after all other jobs finish their execution. So according to the priority table (SPT) of MCPI, job j_i can be given the lowest priority among the LO-criticality jobs. Since the job can meet its deadline after all other jobs finish their execution, the $PullUp()$ subroutine [SPBB15] will pull up the HI-criticality jobs upward in the priority tree. So according to the MCPI algorithm the job j_i is the lowest priority job among the HI-criticality jobs as well. This shows that the job j_i is the lowest priority job according to the MCPI algorithm.

Now assume j_i is the lowest priority job and its criticality is HI which means the job j_i can finish its execution between its arrival time and deadline after all other jobs finish their execution. Since our algorithm prefers LO-criticality jobs to assign the lowest priority over HI-criticality jobs, there are no LO-criticality jobs available which can be assigned the lower priority. As in the previous case, job j_i is the lowest priority job in the SPT priority table of the MCPI algorithm. Since no LO-criticality job can finish its execution after the execution of job j_i, the $PullUp()$ subroutine will not be able to pull up the HI-criticality job upward in the priority tree. So job j_i is the lowest priority job according to the MCPI algorithm.

So both the algorithms generate the same priority order for instance I. Since our algorithm does not find any HI-scenario failure in the $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ subroutine, the MCPI algorithm also does not find any HI-scenario failure in its $anyHIscenarioFailure()$ subroutine. □

Theorem 5.4.3: The computational complexity of LoBCP (Algorithm 14 on page 107) is $O(mn^3)$, where n is the number of jobs in an instance I and m is the number of processors.

Proof. Line 1 takes $O(n)$ time. In lines 3 and 4, finding the latest deadline job takes $O(n \log n)$ time, simulation of global EDF on m processors takes $O(mn^2)$ times [Hor74]. So the total time taken by lines 3 and 4 is $O(n \log n + mn^2)$. Lines 5 to 10 take $O(1)$ time. Since the while loop in line 2 runs n times, line 3 to 10 require a total of $O(n^2 \log n + mn^3)$

time, i.e., $O(mn^3)$. Lines 12, 13 to 15, 16 and 17 takes $O(mn^2)$ time each. So the overall time complexity of our algorithm is $O(mn^3)$. □

This is in contrast to MCPI [SPBB15], the only existing time-triggered scheduling algorithm for mixed-criticality systems on multiprocessors, whose complexity is $O(mn^3 \log n)$.

5.5 Extension for Dependent Jobs

In previous sections, we have discussed instances with independent jobs. Now, we discuss the case of the dual-criticality instances with dependent jobs. In this section, we modify the algorithm given in Section 5.4 to find the scheduling tables such that if the scheduler discussed in Section 5.2 dispatches the jobs according to these scheduling tables then it will be a correct online scheduling strategy without disturbing the dependencies between them. There exists an algorithm [SPBB15] which can schedule the jobs of an instance I with dependencies with worst-case time complexity $O(|E|n^2 + mn^3 \log n)$, where n is the number of jobs, $|E|$ the number of edges in the DAG and m the number of processors. We claim that our algorithm has a better worst-case time complexity than the existing algorithm.

5.5.1 Model

We use the same model as discussed in Section 5.2. Additionally, an instance of a mixed-criticality system containing dependent jobs can be defined as a *directed acyclic graph* (DAG). An instance I is represented in the form of $I(V, E)$, where V represents the set of jobs, i.e., $\{j_1, j_2, \ldots, j_n\}$ and E represents the edges which depict dependencies between jobs. We assume that a HI-criticality job can depend on a LO-criticality job only if the HI-criticality job depends upon another HI-criticality job. This means, there are some instances where an outward edge from a LO-criticality job j_l becomes an inward edge to a HI-criticality job j_{h1} with another inward edge from a HI-criticality job j_h to job j_{h1}.

Definition 5.5.1: A dual-criticality MC instance I with job dependencies is said to be **time-triggered schedulable** on a multiprocessor system if it is possible to construct the two scheduling tables S_{LO} and S_{HI} for each processor of instance I without violating the dependencies, such that the run-time algorithm described in Section 5.2 schedules I correctly.

5.5.2 The DP-LoCBP Algorithm

Here we propose the DP-LoCBP algorithm which can construct the two scheduling tables S_{LO} and S_{HI} for a dual-criticality instance with dependent jobs. A DAG of mixed-criticality jobs is *MC-schedulable* if there exists a correct online scheduling policy for it. Our algorithm finds a LO-criticality priority order for the jobs of instance I which is used to construct the table S_{LO}. Then the same job allocation order of S_{LO} is used to construct the table S_{HI}, where HI-criticality jobs have greater priority than LO-criticality jobs, and the HI-criticality jobs are allocated their C_iHI units of execution time in S_{HI} without violating the dependency constraints. The priority between two jobs j_i and j_k is denoted by $j_i \triangleright j_k$, where j_i is higher priority than j_k. This priority ordering must satisfy two properties:

- If a node j_i is assigned higher priority than node j_k (i.e., $j_i \triangleright j_k$), then there should not be a path in the DAG from node j_k to node j_i.
- If the DAG is scheduled according to this priority ordering then each job j_i of the DAG must finish its $C_i(\text{LO})$ units of execution time before d_i^{Δ}.

Now we present the algorithm DP-LoCBP which finds a priority order for mixed-criticality dependent jobs.

The DP-LoCBP algorithm finds a priority order which is used to construct the scheduling tables for all the processors in steps 1 to 11. First, DP-LoCBP finds the LO-scenario deadline ($d_i{}^{\Delta}$) of each job. For the LO-criticality jobs $d_i{}^{\Delta} = d_i$, but $d_i{}^{\Delta} \leq d_i$ for the HI-criticality jobs. Then the algorithm starts to assign the lowest priority jobs from the instance I. It always selects the latest deadline job which does not have an outward edge as the lowest priority job, but LO-criticality jobs are considered before the HI-criticality jobs. A job j_i can be assigned the lowest priority if and only if all other jobs j_k finish their execution and there remains sufficient time for j_i to complete its $C_i(\text{LO})$ units of execution time before $d_i{}^{\Delta}$. After a job j_i is assigned the lowest priority, it is removed from the instance and added to the priority order Ψ. Then the remaining jobs are considered for priority assignment. If at any step a job cannot be assigned a priority, the algorithm declares failure. In step 12, the algorithm constructs the table S_{LO}. In steps 13 to 15, it checks for any possible HI-criticality scenario failure. If it does not find a HI-criticality scenario failure, then the priority order constructed by the DP-LoCBP algorithm can successfully schedule the instance I. Then the table S_{HI} is constructed for each processor by allocating $C_i(\text{HI})$ units of execution time of

Algorithm 15 DP-LoCBP(I)

Notation:

$I = \{j_1, j_2, ..., j_n\}$, where $j_i =< a_i,\ d_i,\ \chi_i,\ C_i(\text{LO}),\ C_i(\text{HI}) >$.

Input : I, Dependency relation $E \subseteq I \times I$

Output : Tables S_{LO} and S_{HI}

Assume earliest arrival time is 0.

1: Compute the LO-scenario deadline (${d_i}^{\Delta}$) of each job j_i as ${d_i}^{\Delta} = d_i - (C_i(\text{HI}) - C_i(\text{LO}))$;
2: **while** I is not empty **do**
3: Assign a LO-criticality latest deadline job j_i which does not have an outward edge as the lowest priority job if j_i can finish its execution in the interval $[a_i, {d_i}^{\Delta}]$ after all other jobs finish their execution in LO-scenario under the global EDF scheme;
4: If any LO-criticality job with no outward edge cannot be given the lowest priority then a HI-criticality latest deadline job j_i which does not have an outward edge is assigned as the lowest priority job if j_i can finish its execution in the interval $[a_i, {d_i}^{\Delta}]$ after all other jobs finishes their execution in LO-scenario under the global EDF scheme;
5: **if** no job is assigned a lowest priority **then**
6: Declare FAIL and EXIT;
7: **else**
8: Add the job j_i to the priority order Ψ.
9: Remove job j_i from the instance and continue;
10: **end if**
11: **end while**
12: Construct table S_{LO} for each processor P_i using the priority order Ψ;
13: **if** $anyHIscenarioFailure(S_{\text{LO}},\ I,\ \Psi)$ **then**
14: return FAIL and EXIT;
15: **else**
16: Construct table S_{HI} for each processor P_i using the same order of allocated jobs in S_{LO}.
17: The same order as S_{LO} is followed to allocate the jobs in S_{HI};
18: After a HI-criticality job j_i is allocated its $C_i(\text{LO})$ execution time in S_{HI}, $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time of job j_i is allocated after the rightmost segment of job j_i in S_{LO} without violating the dependency constraints and without disturbing the priority order Ψ;
19: **end if**

each HI-criticality job using the same order of allocated jobs as S_{LO} where a HI-criticality job is given higher priority over LO-criticality jobs. In S_{HI} each HI-criticality job is allocated its $C_i(\text{LO})$ units of execution time without violating the dependency constraints. Once the $C_i(\text{LO})$ units of execution time are allocated for HI-criticality jobs in S_{HI}, the remaining $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time are allocated immediately without disturbing the priority order Ψ and without violating the dependency constraints. At each instant, the allocation is done without violating the dependency constraints.

We illustrate the operation of this algorithm by an example.

Example 5.5.1: Consider the mixed-criticality instance given in Table 5.3 which is going to be scheduled on a multiprocessor system having two homogeneous processors, i.e., P_0 and P_1. The corresponding DAG is given in Fig. 5.4.

Figure 5.3: An example instance to explain the DP-LoCBP algorithm

Job	Arrival time	Deadline	Criticality	$C_i(\text{LO})$	$C_i(\text{HI})$
j_1	0	3	LO	1	1
j_2	0	3	LO	1	1
j_3	0	3	LO	1	1
j_4	0	4	HI	1	3
j_5	1	6	HI	1	3

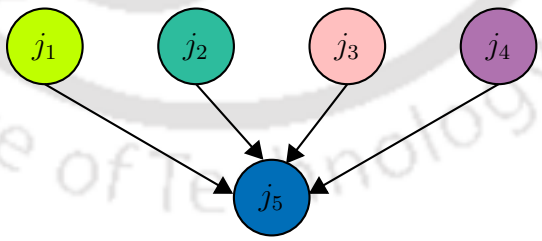


Figure 5.4: A DAG showing job dependencies among the jobs of an instance

Now we construct a priority order using the DP-LoCBP algorithm. The LO-criticality scenario d_i^{Δ} of the jobs $j_1, j_2, j_3, j_4.j_5$ are $3, 3, 3, 2, 4$ respectively. Next we start assigning priorities to each job.

- We start with a node having no outward edges from it. The only such node is job j_5.

So Algorithm 15 assigns job j_5 the lowest priority. If j_5 is assigned the lowest priority, then j_1 and j_2 can run simultaneously in P_0 and P_1 over $[0, 1]$ and $[0, 1]$ respectively. Then j_3 and j_4 can run over $[1, 2]$ in P_0 and P_1 respectively. Then j_5 can easily execute its 1 unit of execution on either P_0 or P_1 over $[2, 3]$ to finish by its LO-scenario deadline (d_i^{Δ}). Now we can assign job j_5 the lowest priority job.

We remove job j_5 and consider $\{j_1, j_2, j_3, j_4\}$ to find the next lowest priority job.

- Since the LO-criticality jobs are given the lowest priority by the proposed algorithm, it is easy to verify that the successive lowest priority jobs will be j_1, j_2 and j_3 respectively. Finally, j_4 is the highest priority job.

So the final priority order of jobs in instance I is $j_4 \triangleright j_3 \triangleright j_2 \triangleright j_1 \triangleright j_5$. The table S_{LO} for each processor is given in Fig. 5.5.

Now the $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ subroutine checks for all possible HI-criticality scenarios. We can check that all HI-criticality scenarios are schedulable using the priority order $j_4 \triangleright j_3 \triangleright j_2 \triangleright j_1 \triangleright j_5$ of the instance I. Finally, the table S_{HI} is constructed for each processor by allocating $C_i(\text{HI})$ units of execution time of each HI-criticality job using the same order of allocated jobs in S_{LO} where a HI-criticality job is given higher priority over a LO-criticality job. On processor P_0, the job order of S_{HI} remains the same as in S_{LO}. Job j_4 is a HI-criticality job and does not depend on any other job, so it is allocated its $C_i(\text{LO})$ units of execution time over $[0, 1]$ and the remaining $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time are allocated in the interval $[1, 3]$. Job j_5 is allocated in the interval $[2, 3]$ in table S_{LO} of P_0. But j_5 is allocated in the interval $[3, 6]$ due to dependency constraints which does not affect the scheduling after a mode change. On processor P_1, job j_3 and j_2 (LO-criticality) which do not depend on any other jobs, are allocated their one unit of execution time in the intervals $[0, 1]$ and $[1, 2]$ respectively. The table S_{HI} for each processor is given in Fig. 5.6. □

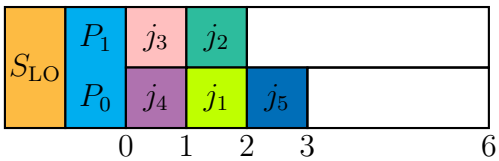


Figure 5.5: Table S_{LO} for processor P_0 and P_1

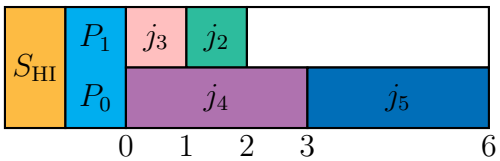


Figure 5.6: Table S_{HI} for processor P_0 and P_1

5.5.3 Comparison with MCPI Algorithm

Theorem 5.5.1: An instance I is schedulable by the MCPI algorithm [SPBB15] if and only if it is schedulable by our algorithm.

Proof. $\Rightarrow$ We need to show that if MCPI generates a priority order for an instance I, then our algorithm will always find a priority order for instance I and the $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ subroutine will not fail.

Suppose the MCPI algorithm finds a priority order for instance I. Now the lowest priority job of the priority order (according to the MCPI algorithm) can be either a LO-criticality or HI-criticality job. First, we prove the case where a job is LO-criticality and then HI-criticality. Let j_i be the lowest priority job and its criticality be LO which means no other job depends on j_i. So at the time of construction of table S_{LO}, every higher priority job j_k finishes its $C_k(\text{LO})$ units of execution time without violating the dependency constraints and there remains sufficient time for the lowest priority job j_i to finish its $C_i(\text{LO})$ units of execution time in the interval $[a_i, (d_i^{\Delta})]$. So this condition is the same as the DP-LoCBP algorithm.

Let job j_i be the lowest priority, and its criticality be HI which means no other job depends on j_i. Since MCPI successfully finds the priority order, it must have checked all the scenarios and does not find any failure in the HI-scenario situations. After every higher priority job j_k finishes its $C_k(\text{LO})$ units of execution time, there remains sufficient time for the lowest priority job j_i to finish its $C_i(\text{LO})$ units of execution time in the interval $[a_i, d_i^{\Delta}]$ without violating the dependency constraints. The HI-criticality jobs need to finish their LO-criticality execution on or before d_i^{Δ} in LO-scenario, so that they have sufficient time to finish their remaining $C_i(\text{HI}) - C_i(\text{LO})$ units of execution time before their deadline d_i. This condition does not violate the dependency constraints as it is the job which does not have an outward edge from it. So this condition is the same as the DP-LoCBP algorithm.

Then j_i is removed from the instance I and the next priority can be assigned from the remaining jobs. We can argue in the same way for the remaining jobs. From the above argument, it is proved that the DP-LoCBP algorithm finds the same priority order, for instance I as the MCPI algorithm. Since the priority order is the same and MCPI does not find any HI-scenario or LO-scenario failure, $anyHIscenarioFailure(S_{\text{LO}}, I, \Psi)$ subroutine in our algorithm will not fail as well. Thus, for a MCPI schedulable instance, our algorithm can also construct priority tables S_{LO} and S_{HI}.

($\Leftarrow$) Our algorithm generates a priority order for instance I which is used to find the table S_{LO}. When a mode change occurs, our algorithm uses the table S_{HI} which is constructed from the job ordering in S_{LO} to schedule the HI-criticality jobs. We need to show that if our algorithm generates a priority order for instance I, then the MCPI algorithm will always find a priority order and the $anyHIScenarioFailure(PT, T)$ subroutine will not fail.

Suppose our algorithm finds a priority order, for instance I. The lowest priority job assigned by our algorithm can be either a HI-criticality or a LO-criticality job. First, we consider the case where a job is LO-criticality. Let j_i be the lowest priority job, and its criticality be LO which means the job j_i can finish its execution between its arrival time and deadline after all other job finishes their execution without violating the dependency constraints. So according to the priority table (SPT) of MCPI, job j_i can be given the lowest priority among the LO-criticality jobs. Since the job can meet its deadline after all other jobs finished their execution, the $PullUp()$ subroutine will pull up the HI-criticality jobs upward in the priority tree. So according to the MCPI algorithm, the job j_i is the lowest priority job among the HI-criticality jobs as well. This shows that the job j_i is the lowest priority job according to the MCPI algorithm.

Let j_i be the lowest priority job, and its criticality be HI which means the job j_i can finish its execution between its arrival time and deadline after all other job finishes their execution without violating the dependency constraints. Since our algorithm prefers LO-criticality jobs to assign the lowest priority over HI-criticality jobs, there are no LO-criticality jobs available which can be assigned lower priority than job j_i. Our algorithm chooses the job with no outward edges which means no job depends on the lowest priority job. So due to the dependency constraints, all the LO-criticality jobs finish before job j_i. Since no LO-criticality job can finish its execution after the execution of job j_i, the $PullUp()$ subroutine will not be able to pull up the HI-criticality jobs upward in the priority tree. So job j_i is the lowest priority job according to the MCPI algorithm.

In the same way, we argue for the next priority assignment of jobs of instance I. □

Theorem 5.5.2: The computational complexity of DP-LoCBP (Algorithm 15 on page 116) is $O(n|E| + mn^3)$, where n is the number of jobs, E the dependency relations among the jobs in the instance I and m the number of processors in the system.

Proof. Line 1 takes $O(n)$ time. In lines 3 - 4, traversing each edges takes $O(|E|)$ time, simulation of global EDF on m processors takes $O(mn^2)$ times [CSB90]. So the total time

taken by lines 3 and 4 is $O(|E|+n\log n+mn^2)$. Lines 5 to 9 take $O(1)$ time in each execution of the loop body. Since the while loop in line 2 runs n times, lines 3 to 9 require a total of $O(n|E| + n^2\log n + mn^3)$ time, i.e, $O(n|E| + mn^3)$ time each. Lines 12, 13 to 14, 16 and 17 to 18 takes $O(mn^2)$ time each. So the overall time complexity of our algorithm is $O(n|E| + mn^3)$. □

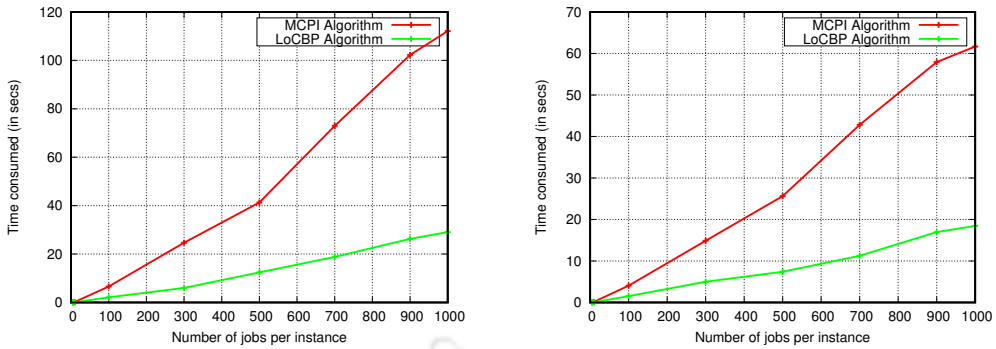
This is in contrast to the MCPI algorithm [SPBB15], the only existing time-triggered scheduling algorithm for the dependent jobs of mixed-criticality systems on multiprocessors is $O(n^2|E| + mn^3\log n)$.

5.6 Results and Discussion

In this section, we present the experiments conducted to evaluate the LoCBP algorithm for the dual-criticality case for non-recurrent jobs. The experiments compare the running times of LoCBP and MCPI. The comparison is done over numerous instances with randomly generated parameters.

The job generation policy may have significant effect on the experiments. The details of the job generation policy [ESD10] are given below.

- The utilization (u_i) of the jobs of instance I are generated according to the Staffords randfixedsum algorithm [Sta06].
- We use the exponential distribution proposed by Davis *et al* [DZB08] to generate the deadline (d_i) of the jobs of instance I.
- The C_i(LO) units of execution of the jobs are calculated by $u_i \times d_i$.
- The C_i(HI) units of execution of the jobs are calculated as $C_i(\text{HI}) = \text{CF} \times C_i(\text{LO})$ where CF is the criticality factor which varies between 2 and 6 for each HI-criticality job j_i in our experiments.
- Each instance I contains at least one HI-criticality job and one LO-criticality job. We have generated random instances for 2, 4, 8 and 16 processors, where each instance has atleast $m + 1$ number of jobs. Each instance is LO-scenario schedulable. We have used an intel core 2 duo processor machine with speed of 2.3 Ghz to conduct the experiments.



(a) Comparison of time consumption of MC-schedulable instances for $m = 2$

(b) Comparison of time consumption of MC-schedulable instances for $m = 4$

Figure 5.7: Comparison of time consumption of MC-schedulable instances with different number of processors

In the first experiment, we fix the number of processors to 2 and let the deadline of the jobs vary between 1 and 2000. The graph in Fig. 5.7a shows the time consumption by each schedulable instances from different numbers of randomly generated instances.

From the graph in Fig. 5.7a, it is clear that our algorithm consumes significantly less time than the MCPI algorithm. As can be seen from Fig. 5.7a, for a multiprocessor with two processors the time consumption by MCPI is much higher than our algorithm. The ratio of time consumed also increases with the increase of number of jobs per instance and is close to five for 1000 jobs. In another experiment, we have shown that the time consumption decreases for $m = 4$, but the ratio of time consumed by our algorithm in comparison to the MCPI algorithm is very much similar to the case $m = 2$, as can be seen in Fig. 5.7b.

5.7 Conclusion

In this chapter, we proposed a new algorithm for time-triggered scheduling of mixed-criticality jobs for multiprocessor systems. We proved that our algorithm has a better worst-case time complexity than the previous algorithm (MCPI). We also proved the correctness of our algorithm. Then we extended our algorithm for dependent jobs and compared the worst-case time complexity with the existing algorithm. We examined the theoretical result by comparing the actual time consumption between LoCBP and MCPI.

Chapter 6

Conclusions and Future Scope of Work

6.1 Summary of the Thesis

The fundamental challenge in scheduling a mixed-criticality (MC) instance is to satisfy both the system designers and certification authorities. We discussed most of the work carried out in the field of schedulability of mixed-criticality systems based on the time-triggered paradigm. We focused on time-triggered scheduling because of the resultant deterministic behavior and easier verification.

In this thesis, we proposed a number of time-triggered scheduling algorithms for various mixed-criticality systems. In the first contribution, we proposed a time-triggered scheduling algorithm (TT-Merge) for non-recurrent task sets. The algorithm constructs two off-line scheduling tables which can be used by a scheduler to dispatch jobs on-line. We proved the proposed algorithm is better than the existing algorithms in terms of the number of schedulable instances. We presented a few experimental results through randomly generated instances (or task sets) to show the extent to which TT-Merge dominates the existing algorithms. Then we extended TT-Merge to schedule dependent jobs, periodic jobs and synchronous reactive systems. We also presented an extended algorithm which constructs time-triggered scheduling tables up to m-criticality levels.

Most researchers have focused on the schedulability of mixed-criticality task sets. Non-functional properties of mixed-criticality systems like *energy consumption* have not been explored widely, and not in the context of time-triggered scheduling. Our time-triggered

energy-efficient scheduling algorithm for mixed-criticality systems is the first such algorithm to the best of our knowledge. Initially, we proposed the algorithm for periodic tasks and then extended it for dependent jobs. We proved that our proposed work consumes less energy than all the existing algorithms based on EDF-VD. Along the way, we also proved that our time-triggered algorithm (TT-Merge) schedules more number of instances (a super set of instances) than the EDF-VD algorithm. Finally, we proved that the proposed energy-efficient time-triggered scheduling algorithm is optimal with respect to the TT-Merge algorithm. Then we extended the proposed algorithm for dependent jobs. We provided a number of experimental results to show the dominance of our algorithm over the existing algorithms in terms of normalized energy consumption.

Finally, we proposed a time-triggered scheduling algorithm for mixed-criticality systems for multiprocessor systems. In this work, we showed that our algorithm schedules the same number of instances as the MCPI algorithm, the only existing time-triggered scheduling algorithm. On the other hand, we showed that the time complexity of our algorithm is better than the MCPI algorithm. We presented experimental results to support the theoretical results.

6.2 Future Scope of Work

This thesis is focused on the time-triggered paradigm of scheduling mixed-criticality task sets. Several schedulability tests for mixed-criticality task sets have been proposed, e.g., periodic tasks, dependent tasks and synchronous reactive systems, etc., while constructing time-triggered scheduling tables. Further, some new design objectives and interesting research challenges have been identified during the course of this work. Some of the potential directions to which the contribution of the thesis can be extended are discussed below.

In Chapter 3, we proposed a time-triggered scheduling algorithm for dependent and independent jobs where we assumed that there is no resource sharing between the jobs. The impact of resource sharing among jobs in the TT-Merge algorithm needs investigation. On the other hand, a speed-up bound for the TT-Merge algorithm has not been computed and should be explored. In Chapter 4, the proposed energy-efficient time-triggered scheduling algorithm is applicable for uniprocessor real-time systems. We plan to extend our technique for a global multiprocessor scheduling algorithm in the future. On the other hand, we minimize the energy consumption in only LO-criticality scenarios. We would like to extend

the work to minimize the energy consumption in both HI-criticality and LO-criticality scenarios. In this thesis, we have not studied fault-tolerance in the context of mixed-criticality systems. Fault-tolerance in the presence of processor faults is a vital functionality of mixed-criticality systems for multiprocessor systems. The impact of time-triggered paradigm in the presence of faults need to be explored in this settings. In particular, we plan to investigate the impact of fault-tolerance on the time-triggered scheduling algorithm proposed in Chapter 5 in the future.

References

[ABW93] NC Audsley, Alan Burns, and AJ Wellings. Deadline monotonic scheduling theory and application. *Control Engineering Practice*, 1(1):71–78, 1993.

[aHCJPH11] Seo-Hyun Jeon andJin Hee Cho, Yangjae Jung, Sachoun Park, and Tae-Man Han. Automotive hardware development according to ISO 26262. In *Advanced Communication Technology (ICACT), 2011 13th International Conference on*, pages 588–592. IEEE, 2011.

[AKTM16a] Sedigheh Asyaban, Mehdi Kargahi, Lothar Thiele, and Morteza Mohaqeqi. Analysis and scheduling of a battery-less mixed-criticality system with energy uncertainty. *ACM Transactions on Embedded Computing Systems (TECS)*, 16(1):23, 2016.

[AKTM16b] Sedigheh Asyaban, Mehdi Kargahi, Lothar Thiele, and Morteza Mohaqeqi. Analysis and scheduling of a battery-less mixed-criticality system with energy uncertainty. *ACM Trans. Embedded Comput. Syst.*, 16(1):23:1–23:26, 2016.

[AMT15a] M Ali Awan, Damien Masson, and Eduardo Tovar. Energy-aware task allocation onto unrelated heterogeneous multicore platform for mixed criticality systems. In *2015 IEEE Real-Time Systems Symposium,*, pages 377–377, 2015.

[AMT15b] Muhammad Ali Awan, Damien Masson, and Eduardo Tovar. Energy-aware task allocation onto unrelated heterogeneous multicore platform for mixed criticality systems. In *IEEE Real-Time Systems Symposium, RTSS 2015, San Antonio, Texas, USA, December 1-4, 2015*, page 377, 2015.

[AMT16] Muhammad Ali Awan, Damien Masson, and Eduardo Tovar. Energy efficient mapping of mixed criticality applications on unrelated heterogeneous multicore

platforms. In *11th IEEE Symposium on Industrial Embedded Systems, SIES 2016, Krakow, Poland, May 23-25, 2016*, pages 63–72, 2016.

[AS00] James H Anderson and Anand Srinivasan. Early-release fair scheduling. In *12th Euromicro Conference on Real-Time Systems, ECRTS.*, pages 35–43. IEEE, 2000.

[ASK15] Ijaz Ali, Jun-Ho Seo, and Kyong Hoon Kim. A dynamic power-aware scheduling of mixed-criticality real-time systems. In *14th IEEE International Conference on Ubiquitous Computing and Communications, IUCC 2015*, pages 438–445, 2015.

[Aud01] Neil C Audsley. On priority assignment in fixed priority scheduling. *Information Processing Letters*, 79(1):39–44, 2001.

[Bar04a] Sanjoy K Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, 53(6):781–784, 2004.

[Bar04b] Sanjoy K Baruah. Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. *IEEE Transactions on Computers*, 53(6):781–784, 2004.

[Bar12] Sanjoy Baruah. Semantics-preserving implementation of multirate mixed-criticality synchronous programs. In *Proceedings of the 20th International Conference on Real-Time and Network Systems*, pages 11–19. ACM, 2012.

[Bar13] Sanjoy Baruah. Partitioned edf scheduling: a closer look. *Real-Time Systems*, 49(6):715–729, 2013.

[Bar14] Sanjoy Baruah. Implementing mixed-criticality synchronous reactive programs upon uniprocessor platforms. *Real-Time Systems*, 50(3):317–341, 2014.

[Bar18] Sanjoy Baruah. Mixed-criticality scheduling theory: Scope, promise, and limitations. *IEEE DESIGN AND TEST*, 35(2):31–37, 2018.

[BB05] Enrico Bini and Giorgio Buttazzo. Measuring the performance of schedulability tests. *Real-Time Systems*, 30(1-2):129–154, 2005.

[BB08] Sanjoy Baruah and Theodore Baker. Schedulability analysis of global edf. *Real-Time Systems*, 38(3):223–235, 2008.

[BB11] Alan Burns and Sanjoy Baruah. *Timing Faults and Mixed Criticality Systems*, volume 6875 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg, 2011.

[BBB+09] James Barhorst, Todd Belote, Pam Binns, Jon Hoffman, James Paunicka, Prakash Sarathy, John Scoredos, Peter Stanfill, Douglas Stuart, and Russel Urzi. A research agenda for mixed-criticality systems. In *Cyber-Physical Systems Week*, APR 2009.

[BBD+10] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'angelo, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. Proceedings of the 35th international symposium on scheduling real-time mixed-criticality jobs. In *Mathematical Foundations of Computer Science 2010*, pages 90–101. Springer, 2010.

[BBD+11] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'angelo, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. Mixed-criticality scheduling of sporadic task systems. In *European Symposium on Algorithms*, pages 555–566. Springer, 2011.

[BBD+12a] S. Baruah, V. Bonifaci, G. D'Angelo, Haohan Li, A. Marchetti-Spaccamela, N. Megow, and L. Stougie. Scheduling real-time mixed-criticality jobs. *IEEE Transactions on Computers*, 61(8):1140–1152, Aug 2012.

[BBD+12b] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo DAngelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. The preemptive uniprocessor scheduling of mixed-criticality implicit-deadline sporadic task systems. In *Real-Time Systems (ECRTS), 2012 24th Euromicro Conference on*, pages 145–154. IEEE, 2012.

[BBD+15] Sanjoy Baruah, Vincenzo Bonifaci, Gianlorenzo D'angelo, Haohan Li, Alberto Marchetti-Spaccamela, Suzanne Van Der Ster, and Leen Stougie. Preemptive uniprocessor scheduling of mixed-criticality sporadic task systems. *Journal of the ACM (JACM)*, 62(2):14, 2015.

[BCA08] Björn B Brandenburg, John M Calandrino, and James H Anderson. On the scalability of real-time scheduling algorithms on multicore platforms: A case study. In *Real-Time Systems Symposium, 2008*, pages 157–169. IEEE, 2008.

[BCLS14a] Sanjoy Baruah, Bipasa Chattopadhyay, Haohan Li, and Insik Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50(1):142–177, 2014.

[BCLS14b] Sanjoy Baruah, Bipasa Chattopadhyay, Haohan Li, and Insik Shin. Mixed-criticality scheduling on multiprocessors. *Real-Time Systems*, 50(1):142–177, 2014.

[BCPV96] Sanjoy K Baruah, Neil K Cohen, C Greg Plaxton, and Donald A Varvel. Proportionate progress: A notion of fairness in resource allocation. *Algorithmica*, 15(6):600–625, 1996.

[BD13] Alan Burns and Rob Davis. Mixed criticality systems: A review. *Department of Computer Science, University of York, Tech. Rep*, 2013.

[BF11] Sanjoy Baruah and Gerhard Fohler. Certification-cognizant time-triggered scheduling of mixed-criticality systems. In *32nd IEEE Real-Time Systems Symposium (RTSS)*, pages 3–12. IEEE, 2011.

[BG03] Sanjoy K Baruah and Joël Goossens. Rate-monotonic scheduling on uniform multiprocessors. *IEEE transactions on computers*, 52(7):966–970, 2003.

[BLS10a] Sanjoy Baruah, Haohan Li, and Leen Stougie. Towards the design of certifiable mixed-criticality systems. In *Proceedings of the 2010 16th IEEE Real-Time and Embedded Technology and Applications Symposium*, RTAS '10, pages 13–22, 2010.

[BLS10b] Sanjoy K Baruah, Haohan Li, and Leen Stougie. Mixed-criticality scheduling: Improved resource-augmentation results. In *Proceedings of the ICSA International Conference on Comput- ers and their Applications (CATA)*, pages 217–223, 2010.

[Bow00] Jonathan Bowen. The ethics of safety-critical systems. *Communications of the ACM*, 43(4):91–97, 2000.

[BV08] Sanjoy Baruah and Steve Vestal. Schedulability analysis of sporadic tasks with multiple criticality specifications. In *Euromicro Conference on Real-Time Systems, 2008. ECRTS'08.*, pages 147–155. IEEE, 2008.

[CK07] Jian-Jia Chen and Chin-Fu Kuo. Energy-efficient scheduling for real-time systems on dynamic voltage scaling (dvs) platforms. In *Embedded and Real-Time Computing Systems and Applications, 2007. RTCSA 2007. 13th IEEE International Conference on*, pages 28–38. IEEE, 2007.

[CLRS09] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition.* The MIT Press, 3rd edition, 2009.

[Com10] International Electrotechnical Commission. Parts 1 – 7 IEC 6150: Functional safety of electrical/electronic/programmable electronic safety-related systems, 2010.

[CRJ06] Hyeonjoong Cho, Binoy Ravindran, and E Douglas Jensen. An optimal real-time scheduling algorithm for multiprocessors. In *27th IEEE International Real-Time Systems Symposium, RTSS'06.*, pages 101–110. IEEE, 2006.

[CSB90] Houssine Chetto, Maryline Silly, and T Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Real-Time Systems*, 2(3):181–194, 1990.

[DB11] Robert I. Davis and Alan Burns. A survey of hard real-time scheduling for multiprocessor systems. *ACM Comput. Surv.*, 43(4):35:1–35:44, October 2011.

[DZB08] Robert I. Davis, Attila Zabos, and Alan Burns. Efficient exact schedulability tests for fixed priority real-time systems. *IEEE Transactions on Computers,*, 57(9):1261–1276, 2008.

[ESD10] Paul Emberson, Roger Stafford, and Robert I. Davis. Techniques for the synthesis of multiprocessor tasksets. In *proceedings 1st International Workshop on Analysis Tools and Methodologies for Embedded and Real-time Systems (WATERS 2010)*, pages 6–11, 2010.

[FBH+06] Helmut Fennel, Stefan Bunzel, Harald Heinecke, Jürgen Bielefeld, Simon Fürst, Klaus-Peter Schnelle, Walter Grote, Nico Maldener, Thomas Weber, Florian

Wohlgemuth, et al. Achievements and exploitation of the autosar development partnership. Technical report, SAE Technical Paper, 2006.

[FMB+09] Simon Fürst, Jürgen Mössinger, Stefan Bunzel, Thomas Weber, Frank Kirschke-Biller, Peter Heitkämper, Gerulf Kinkelin, Kenji Nishikawa, and Klaus Lange. Autosar–a worldwide standard is on the road. In *14th International VDI Congress Electronic Systems for Vehicles, Baden-Baden*, volume 62, page 5, 2009.

[Foh93] Gerhard Fohler. Changing operational modes in the context of pre run-time scheduling. *IEICE transactions on information and systems*, 76(11):1333–1340, 1993.

[Foh95] Gerhard Fohler. Joint scheduling of distributed complex periodic and hard aperiodic tasks in statically scheduled systems. In *16th IEEE Proceedings Real-Time Systems Symposium, 1995.*, pages 152–161. IEEE, 1995.

[fS11] International Organiztion for Standardization. ISO 26262-5:2011 – Road vehicles – Functional safety – Part 5: Product development at the hardware level, November 2011.

[Gal08] Heinz Gall. Functional safety IEC 61508/IEC 61511 the impact to certification and the user. In *Computer Systems and Applications, 2008. AICCSA 2008. IEEE/ACS International Conference on*, pages 1027–1031. IEEE, 2008.

[GFB03] Joël Goossens, Shelby Funk, and Sanjoy Baruah. Priority-driven scheduling of periodic task systems on multiprocessors. *Real-time systems*, 25(2-3):187–205, 2003.

[GSHT13a] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, EMSOFT '13, pages 17:1–17:15. IEEE Press, 2013.

[GSHT13b] Georgia Giannopoulou, Nikolay Stoimenov, Pengcheng Huang, and Lothar Thiele. Scheduling of mixed-criticality applications on resource-sharing multicore systems. In *Proceedings of the Eleventh ACM International Conference on Embedded Software*, page 17, 2013.

[GSHT14] G. Giannopoulou, N. Stoimenov, P. Huang, and L. Thiele. Mapping mixed-criticality applications on multi-core architectures. In *Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6, March 2014.

[HKGT14] Pengcheng Huang, Pratyush Kumar, Georgia Giannopoulou, and Lothar Thiele. Energy efficient dvfs scheduling for mixed-criticality systems. In *Embedded Software (EMSOFT), 2014 International Conference on*, pages 1–10. IEEE, 2014.

[Hor74] WA Horn. Some simple scheduling algorithms. *Naval Research Logistics Quarterly*, 21(1):177–185, 1974.

[HS06] Thomas A Henzinger and Joseph Sifakis. The embedded systems design challenge. In *FM 2006: Formal Methods*, pages 1–15. Springer, 2006.

[HSK+09] Reinhold Hamann, Jürgen Sauler, Stefan Kriso, Walter Grote, and Jürgen Mössinger. Application of ISO 26262 in distributed development ISO 26262 in reality. Technical report, SAE Technical Paper, 2009.

[IF00] Damir Isovic and Gerhard Fohler. Efficient scheduling of sporadic, aperiodic, and periodic tasks with complex constraints. In *The 21st IEEE Proceedings. Real-Time Systems Symposium, 2000.*, pages 207–216. IEEE, 2000.

[Kop98] Hermann Kopetz. The time-triggered model of computation. In *Real-time Systems Symposium*, page 168. IEEE, 1998.

[Kop11] Hermann Kopetz. *Real-Time Systems - Design Principles for Distributed Embedded Applications*. Springer US, Upper Saddle River, NJ, USA, 2nd edition, 2011.

[LB10] Haohan Li and Sanjoy Baruah. Load-based schedulability analysis of certifiable mixed-criticality systems. In *Proceedings of the tenth ACM international conference on Embedded software*, pages 99–108. ACM, 2010.

[LB12] H. Li and S. Baruah. Global mixed-criticality scheduling on multiprocessors. In *24th Euromicro Conference on Real-Time Systems*, pages 166–175, July 2012.

[LDG04] José María López, José Luis Díaz, and Daniel F García. Utilization bounds for edf scheduling on real-time multiprocessor systems. *Real-Time Systems*, 28(1):39–68, 2004.

[LFS+10] Greg Levin, Shelby Funk, Caitlin Sadowski, Ian Pye, and Scott Brandt. Dp-fair: A simple model for understanding optimal multiprocessor scheduling. In *22nd Euromicro Conference on Real-Time Systems (ECRTS),*, pages 3–13. IEEE, 2010.

[Li13] Haohan Li. *Scheduling mixed-criticality real-time systems.* PhD thesis, University of North Carolina at Chapel Hill, 2013.

[Liu00] Jane W. S. W. Liu. *Real-Time Systems.* Prentice Hall PTR, Upper Saddle River, NJ, USA, 1st edition, 2000.

[LJP13a] Vincent Legout, Mathieu Jan, and Laurent Pautet. Mixed-criticality multiprocessor real-time systems: Energy consumption vs deadline misses. In *First Workshop on Real-Time Mixed Criticality Systems (ReTiMiCS)*, pages 1–6, 2013.

[LJP13b] Vincent Legout, Mathieu Jan, and Laurent Pautet. Mixed-Criticality Multiprocessor Real-Time Systems: Energy Consumption vs Deadline Misses. In *First Workshop on Real-Time Mixed Criticality Systems (ReTiMiCS)*, pages 1–6, Taipei, Taiwan, August 2013.

[LL73] Chung Laung Liu and James W. Layland. Scheduling algorithms for multiprogramming in a hard-real-time environment. *Journal of the ACM (JACM)*, 20(1):46–61, 1973.

[NHG+16] Sujay Narayana, Pengcheng Huang, Georgia Giannopoulou, Lothar Thiele, and R Venkatesha Prasad. Exploring energy saving for mixed-criticality systems on multi-cores. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 1–12. IEEE, 2016.

[Pat12] R. M. Pathan. Schedulability analysis of mixed-criticality systems on multiprocessors. In *24th Euromicro Conference on Real-Time Systems*, pages 309–320, July 2012.

[PB00] Peter Puschner and Alan Burns. Guest editorial: A review of worst-case execution-time analysis. *Real-Time Systems*, 18(2-3):115–128, 2000.

[PC14] Santiago Pagani and Jian-Jia Chen. Energy efficiency analysis for the single frequency approximation (sfa) scheme. *ACM Transactions on Embedded Computing Systems (TECS)*, 13(5s):158, 2014.

[PK11a] Taeju Park and Soontae Kim. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *2011 Proceedings of the International Conference on Embedded Software (EMSOFT),*, pages 253–262. IEEE, 2011.

[PK11b] Taeju Park and Soontae Kim. Dynamic scheduling algorithm and its schedulability analysis for certifiable dual-criticality systems. In *Proceedings of the ninth ACM international conference on Embedded software*, pages 253–262. ACM, 2011.

[Pri92] Paul J Prisaznuk. Integrated modular avionics. In *Proceedings of the IEEE National Aerospace and electronics conference, 1992. naecon 1992.,*, pages 39–45. IEEE, 1992.

[PSTW97] Cynthia A Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. In *Proceedings of the twenty-ninth annual ACM symposium on Theory of computing*, pages 140–149. ACM, 1997.

[PSTW02] Cynthia A Phillips, Cliff Stein, Eric Torng, and Joel Wein. Optimal time-critical scheduling via resource augmentation. *Algorithmica*, 32(2):163–200, 2002.

[RB92] RTCA/DO-178B. Software considerations in airborne systems and equipment certification, Dec 1992.

[SB02] Anand Srinivasan and Sanjoy Baruah. Deadline-based scheduling of periodic task systems on multiprocessors. *Information processing letters*, 84(2):93–98, 2002.

[SB08] H. Fennel S. Bunzel. The autosar methodology. In *VDI (ed) FISITA World Automotive Congress. FISITA 2008/F2008-10-023,*. Springer Automotive Media, 2008.

[SPBB13] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Mixed critical earliest deadline first. In *2013 25th Euromicro Conference on Real-Time Systems*, pages 93–102, July 2013.

[SPBB15] D. Socci, P. Poplavko, S. Bensalem, and M. Bozga. Time-triggered mixed-critical scheduler on single and multi-processor platforms. In *HPCC / CSS /ICESS*, pages 684–687, Aug 2015.

[Sta06] Roger Stafford. Random vectors with fixed sum. *See http://www. mathworks. com/matlabcentral/fileexchange/9700*, 2006.

[TFB13] Jens Theis, Gerhard Fohler, and Sanjoy Baruah. Schedule table generation for time-triggered mixed criticality systems. In *Proc. WMC, RTSS*, pages 79–84, 2013.

[The15] Jens Theis. *Certificate Cognizant Mixed-critical Scheduling in Time-triggered Scheduling*. PhD thesis, University of Kaiserslautern, 2015.

[Ves07] S. Vestal. Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *28th IEEE International Real-Time Systems Symposium, 2007. RTSS 2007.*, pages 239–243, Dec 2007.

[ZGYZ16] Qingling Zhao, Zonghua Gu, Min Yao, and Haibo Zeng. HLC-PCP: A resource synchronization protocol for certifiable mixed criticality scheduling. *Journal of Systems Architecture*, 66:84–99, 2016.

[ZGZ14] Qingling Zhao, Zonghua Gu, and Haibo Zeng. HLC-PCP: A resource synchronization protocol for certifiable mixed criticality scheduling. *Embedded Systems Letters*, 6(1):8–11, 2014.

[ZMM04] Dakai Zhu, Rami Melhem, and Daniel Mossé. The effects of energy management on reliability in real-time embedded systems. In *Computer Aided Design, 2004. ICCAD-2004. IEEE/ACM International Conference on*, pages 35–40. IEEE, 2004.

Publications Related to Thesis

Journals

- L. Behera and P. Bhaduri, "*Time-Triggered Scheduling of Mixed-Criticality Systems*", ACM Transactions on Design Automation of Electronic Systems, Volume 22, Issue 4, Number 74, Pages 74:1 - 74:25, 2017. (Published)
- L. Behera and P. Bhaduri, "*An Energy-efficient Time-triggered Scheduling Algorithm for Mixed-criticality Systems*", Design Automation for Embedded Systems, Springer. (Submitted)

Conference

- L. Behera and P. Bhaduri, "*Time-Triggered Scheduling for multiprocessor Mixed-Criticality Systems*", Distributed Computing and Internet Technology. ICDCIT 2018. Lecture Notes in Computer Science (LNCS), vol 10722. Springer, 2018. (Published)

Brief Biography of the Author

Name:	Lalatendu Behera
Father's Name:	Late Achyutananda Behera
Mother's Name:	Premalata Behera
Date of Birth:	15.06.1983
Date of Birth:	Department of Computer Science and Engineering, IIT Guwahati, Assam-781039, India
E-mail:	lalatendu@iitg.ac.in

Lalatendu Behera is a research scholar at the Department of Computer Science and Engineering at IIT Guwahati, India. He completed his M.Tech in Computer Science from the Department of CSE, NIT Rourkela in 2011 and his B.Tech in Computer Science and Engineering from Utkal University in 2006. His research interests include Real-time and embedded systems and Real-time scheduling.

Department of Computer Science and Engineering

Indian Institute of Technology Guwahati

Guwahati 781039, India