Computing Abort Points in Real-Time Systems with Mode Switching

by

Meenaz Devani

(Under the Direction of Shelby Hyatt Funk)

Abstract

In real-time systems, jobs must complete before their deadlines. Oftentimes, real-time systems are required to change their operating modes due to changes in the operating environment. Whenever a system receives a Mode Change Request, the tasks that were running (now called "old mode tasks") should complete their execution, not release any more jobs, and the system should move into the new mode. It has been observed that executing the old mode jobs completely can negatively impact the new mode tasks and the system can become infeasible at some point in the schedule as the Mode Change Deadline cannot be met. In order to meet this Mode Change Deadline, the concept of Abort points has been introduced in this thesis. Abort points are the points in the schedule where the jobs of the old mode tasks can abort whenever there is a Mode Change Request without causing any effect on the new mode tasks. Genetic algorithms were used for computing these abort points. One or two Mode Independent tasks and their overall effect on the system has also been considered.

Index words: Mode Change Request, Mode Change Deadline, Mode Independent Tasks, Abort points

Computing Abort Points in Real-Time Systems with Mode Switching

by

Meenaz Devani

B.E.(CSE), Osmania University, 2014

A Thesis Submitted to the Graduate Faculty
of The University of Georgia in Partial Fulfillment
of the
Requirements for the Degree

Master of Science

Athens, Georgia

2016

©2016

Meenaz Devani

All Rights Reserved

Computing Abort Points in Real-Time Systems with Mode Switching

by

Meenaz Devani

Approved:

Major Professor:	Shelby Hyatt Funk
Committee:	Liming Cai Maria Hybinette

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
December 2016

Computing Abort Points in Real-Time Systems with Mode Switching

Meenaz Devani

November 21, 2016

To Yasmin Devani

for her infinite love and support.

I owe all my achievements to her.

Acknowledgments

I would like to express my sincere gratitude and appreciation to all those who have helped me along the way.

To my major professor, Dr. Shelby Funk, for guiding me throughout my work. She was always there to encourage me and I will forever remain grateful for all her help and support.

To my committee members, Dr. Liming Cai and Dr. Maria Hybinette, for their valuable suggestions on my thesis.

To Gordon Chalmers for helping me with the genetic algorithms.

To my parents, Abdul Malik Devani and Yasmin Devani, for their continuous love and support. This could not have been possible without their motivation and inspiration. Thank you for everything.

Contents

1 Introduction **1**

2 Model and Definitions **4**

2.1 Multi-Mode Scheduling 6

3 Related Work **10**

3.1 Synchronous and Asynchronous systems 10

3.2 Dynamic Priority Scheduling 10

3.3 Tick Scheduling 11

3.4 Fixed Priority Scheduling 12

3.5 UUnifast Algorithm 18

3.6 Dealing with Mode Changes 19

3.7 Introduction to Genetic Algorithms 20

4 Computing Abort points during Mode Switch **23**

4.1 Requirements 23

4.2 Protocol without Periodicity 24

4.3 Protocol with Periodicity 24

4.4 Abort points during Mode Switching 25

4.5 Abort points during Mode Switching without periodicity 26

4.6 Abort points during Mode Switching with periodicity 27

5 Experiments and Results **44**

5.1 Experimental setup 44

5.2 Experiments for 1 Mode Independent task 46

5.3 Experiments for 2 Mode Independent task 52

6 Conclusion and Future Work **58**

REFERENCES **60**

List of Figures

2.1 Example of a Periodic task $T_1 = (5, 3), T_2 = (11, 3)$ 5
2.2 Mode Change 7
2.3 Illustration of Mode Change 8

3.1 Example of Earliest Deadline First Scheduling 11
3.2 Illustration of Fixed Priority Scheduling 13
3.3 Example of Rate Monotonic Scheduling 14
3.4 Example of Deadline Monotonic Scheduling 15
3.5 Tasks meeting their deadlines 18
3.6 Next Generation Children [1] 21
3.7 Steps invoved in a Genetic Algorithm [3] 22

4.1 Old mode tasks and mode independent tasks are being executed . . . 28
4.2 Deadline miss in the new mode 28
4.3 Figure indicating the case where $\Delta = e_I$ 31
4.4 Figure indicating the case where $\Delta < e_I$ 32
4.5 Shifting MI jobs forward 37

5.1 Comparison of Runtimes for 1 MI task 47
5.2 4 Tasks (1 MI) 48
5.3 8 Tasks (1 MI) 49
5.4 16 Tasks (1 MI) 50
5.5 32 Tasks (1 MI) 51

5.6 64 Tasks (1 MI) 52
5.7 Comparison of Runtimes for 2 MI Tasks 53
5.8 8 Tasks (2 MI) 54
5.9 16 Tasks (2 MI) 55
5.10 32 Tasks (2 MI) 56
5.11 64 Tasks (2 MI) 57

List of Tables

2.1 Notations 9

Chapter 1

Introduction

Systems whose behavior depends on the logical correctness of the operations they perform and also on the time at which these operations are performed are called real-time systems [7]. There is fast paced growth in the use of real-time systems today. Some applications requiring real-time systems include aerospace, defence, robotics, medicine, multimedia and several others [18].

Real-time systems are classified as hard real time systems or soft real time systems depending upon their timing requirements. The real-time systems whose timing requirement must always be met or the entire system could fail are referred to as hard real-time systems [20]. Examples include aircraft control systems, missiles, space shuttles, etc. On the other hand, soft real-time systems, such as multimedia applications, are those in which missing a deadline will not lead to any disastrous consequence.

Depending upon the number of processors involved, a real time system can be either a uniprocessor system or a multi-processor system. A uniprocessor system is any system that is executed on only one processor. When more than one processor is involved, it is called a multi-processor real-time system.

Real-time systems contain a series of jobs that are to be executed repeatedly over regular intervals. These repeated jobs, called tasks, can be periodic, sporadic

or aperiodic depending upon their arrival pattern. If a job executes repeatedly at a certain fixed time interval, it is called a periodic task. Every task has a period p_i, worst case execution time e_i and a relative deadline D_i. A periodic task releases every p_i time units. A task is said to be sporadic if p_i is the minimum time between job releases. Aperiodic tasks are the tasks that are released at arbitrary times. All three of these types of tasks have hard deadlines [16].

Scheduling algorithms determine the sequence in which the tasks should be executed. A schedule generated by the scheduling algorithm is said to be valid if the tasks in that schedule can meet their respective deadlines. A task set is feasible if some valid schedule of that task set exists [16].

Depending upon the scheduling algorithm used, tasks are assigned priorities. If a lower priority task is under execution when a higher priority task arrives, the higher priority task can preempt the low priority task. This low priority task will resume its execution when there are no higher priority tasks executing. This is called preemptive scheduling. In non-preemptive scheduling, when a particular task is allocated the processor, the other tasks cannot get access to the processor until the executing task completes. This thesis considers preemptive scheduling on a uniprocessor platform.

Some real-time systems have to operate in various modes. Systems have to change their operating mode whenever they detect a change in their environment or when a Mode Change Request is encountered [11]. A mode change is the transition of the system from the old mode into the new mode. A system can have various modes depending upon its requirements. Each mode in the system in turn has its own set of tasks and functionality. These mode changes are necessary in dealing with hardware faults or environmental changes. If some component in the system fails, the system could remove one functionality and replace it with another [9]. Mode changes can add flexibility to real-time systems by helping in adapting the changing environment, or providing robustness in the presence of faults.

Mode changes in the system can sometimes lead to failure because the workload on the processor is changed dynamically. Also, some of the activities that were guaranteed to complete successfully in the steady state may fail due to the mode change [9]. Some real-time systems still lead to major accidents due to confusion regarding the behavior of modes as discussed in [17].

Any mode changing system can have 3 types of tasks: old mode tasks, mode independent tasks and new mode tasks [12]. Old mode tasks execute before a mode change request occurs in the system. New mode tasks execute after the system moves into the new mode i.e., after mode change deadline. Mode independent tasks execute in both old and new modes.

Timing requirements must be met if the mode changes are to be implemented successfully. To deal with these mode changes, we worked on the concept of *abort points*. Abort points are the points in the schedule where the old mode tasks can abort in the system whenever a mode change request is encountered in the system. By doing so, new mode tasks will not be impacted by these old mode tasks and the system can complete the mode transition successfully. This thesis uses Genetic Algorithms to find the abort points.

Chapter 2

Model and Definitions

Real-time systems contain task sets which generate sequences of deadline constrained jobs. A task set is defined by $\tau = \{T_1, T_2, T_3, ..., T_n\}$. Every task in the task set has its own arrival time, period, worst case execution time and a deadline. The 4-tuple convention for each task can be given as $T_i = (a_i, p_i, e_i, D_i)$

where a_i : arrival time of task i

p_i : period of task i

e_i : worst case execution time of task i

D_i : deadline of task i

T_i generates a sequence of jobs which are denoted by $J_{i,1}$, $J_{i,2}$, ... The arrival time of $J_{i,j}$ will be denoted by $a_{i,j}$, the deadline will be denoted by $D_{i,j}$ and the execution times will be denoted by e_i. These tasks can be periodic or sporadic. In a periodic task set, each job of the task is released every p_i time units after its arrival. The first arrival will be at time $a_{i,1} = a_i$. Subsequent arrivals will be at times $a_{i,j+1} = a_{i,j} + p_i$. In a sporadic task set, jobs are released at least p_i time units apart: $a_{i,1} >= a_i$ and $a_{i,j+1} >= a_{i,j} + p_i$.

In this thesis, we are considering that all tasks arrive at time 0 and deadlines are equal to periods for all the task sets. The convention for this will be $T_i = (p_i, e_i)$.
Example of preemptive periodic task set: $T_1 = (5, 3)$, $T_2 = (11, 3)$.

Assume T_1 has higher priority than T_2. A schedule of this task system can be given as shown in Figure 2.1. The up arrow indicates arrival of a job and deadline of a previous job, rectangle indicates the execution of the task during that given interval. Observe, a preemption occurs at time 5 when the second job of high priority task T_1 arrives. As T_2 has a lower priority it gets preempted until the completion of T_1 and then it resumes its execution.

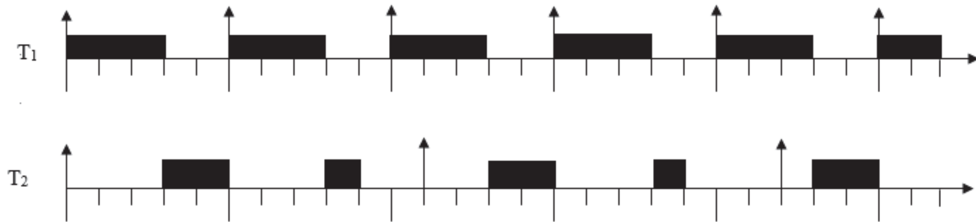


Figure 2.1: Example of a Periodic task $T_1 = (5, 3), T_2 = (11, 3)$

A scheduling algorithm will decide which job of a task will execute at a given point in the schedule. Examples of scheduling algorithms include EDF (Earliest Deadline First), RM (Rate Montonic), etc [7]. Schedulability analysis is used to guarantee that all deadlines will be met while using a specific scheduling algorithm. The task utilization is the ratio of execution time of that task and its period. The task utilization of task T_i can be given by the equation

$$u_i = e_i/p_i \tag{2.1}$$

The total utilization of the task set can be given by the equation

$$U = \sum_{i=1}^{n} e_i/p_i \tag{2.2}$$

Worst case response time of a task is defined as the maximum amount of time a task may take to complete its execution from the time it was released [23]. It is

the sum of execution time of that task and the interference caused due to the high priority tasks in the schedule. Time Demand Analysis(TDA) is used to compute the worst case response time of the both synchronous and asynchronous task sets. Worst case response time of task T_i is denoted bt R_i.

2.1 Multi-Mode Scheduling

Real time systems may be required to operate in various modes and adapt the mode changes for successful running of the system. Example of a multi-mode real time system is an aircraft control system [12]. An aircraft has to operate in several different modes such as take-off mode, taxi mode and landing mode. It has to execute different tasks in different modes. They are required to change their operating modes whenever there is a change in the system's environment. A Mode Change Request (MCR) is initiated in the system. A MCR is defined as an event after which the mode transition should begin. The mode before MCR is called the old mode. After the mode change is complete, the system should move into the new mode [12]. Other examples of multi-mode real-time systems include Meeadaptive control systems in the automotive domain, new mobile phone applications, etc. [22].

A MCR can occur at any time in the system except, in some cases, during a mode transition. A mode change is said to be successful if it meets the Mode change Deadline (MCD) i.e., if the new mode tasks are able to begin executing by MCD [10]. A relative mode change deadline, denoted by X, is the amount of time the system has to execute the old mode tasks and change the mode in order to meet the MCD.

$$X = MCD - MCR \tag{2.3}$$

The new mode tasks can release after the mode change has been done successfully. There are 3 types of tasks during a mode change [12]:

1. Old Mode Tasks

2. Mode Independent Tasks

3. New Mode Tasks

The basic illustration of Mode Change is shown in Figure 2.2.

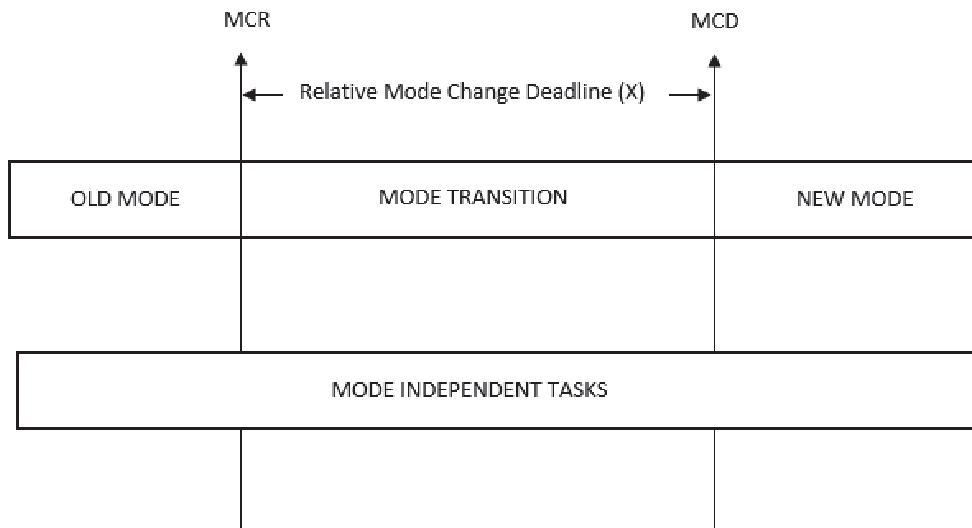


Figure 2.2: Mode Change

1. Old Mode Tasks: These are the tasks that cannot execute after the MCD. Whenever a MCR is encountered in the system, these tasks can either be executed completely or be aborted.

2. Mode Independent Tasks: These tasks have to execute both in old and new modes and also during the mode transition.

3. New Mode Tasks: These tasks can execute only in the new mode after MCD.

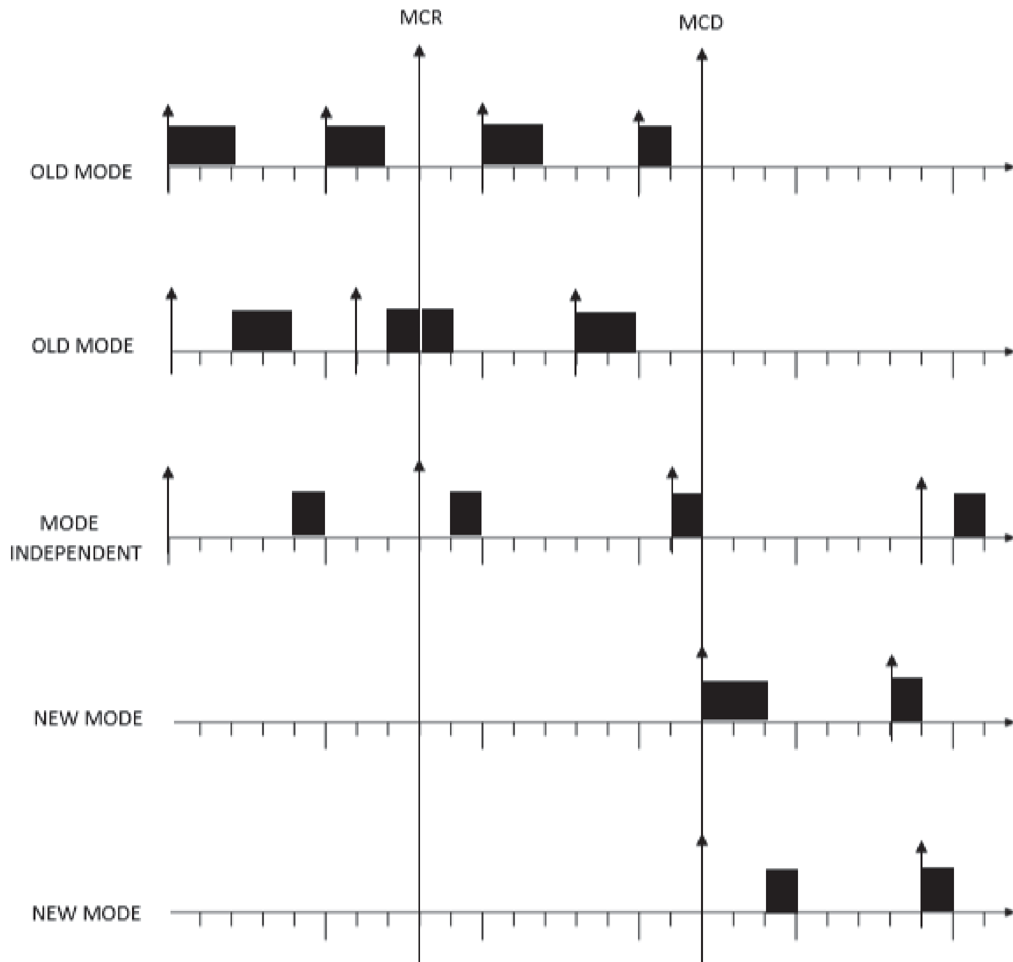


Figure 2.3: Illustration of Mode Change

Figure 2.3 depicts the mode change scenario. After MCR, no more old mode jobs will be released and the jobs that were previously released, either have to complete their execution or abort.

When mode independent tasks are present in the schedule, that system is said to be with periodicity. These mode independent tasks have to execute irrespective of the mode change.

Table 2.1 summarizes the notations introduced in this chapter.

Table 2.1: Notations

Notation	Description
T_i	Task i
p_i	Period of task i
e_i	Worst case execution time of task T_i
D_i	Deadline of task i
a_i	Arrival time of task T_i
$J_{i,j}$	j^{th} job of task T_i
$a_{i,j}$	Arrival time of $J_{i,j}$
MCR	Mode Change Request Time
MCD	Mode Change Deadline
X	Relative Mode Change Deadline
MIT	Mode Independent Task
n	Number of tasks
u_i	Utilization of task T_i
U	Total utilization
R_i	Worst Case Response Time of task T_i

Chapter 3

Related Work

This chapter is divided into 10 sections. Section 3.1 defines synchronous and asynchronous systems. Sections 3.2 through 3.4 describes different types of scheduling algorithms. Section 3.4 describes the Critical Instant Theorem. Section 3.4 covers Time Demand Analysis that is used to find the worst case response times of the tasks. Section 3.5 provides UUnifast algorithm that is used for generating the task-sets. Section 3.6 describes the options available to deal with mode changes in the system. Section 3.7 is a brief introduction to Genetic Algorithms.

3.1 Synchronous and Asynchronous systems

The release time of the first job of task T_i is called the *phase* of T_i. In synchronous real-time systems, all the tasks will have phase 0 - i.e., all the tasks will be released at the same time. In asynchronous real-time systems, phases can be arbitrary - i.e., the jobs can be released at arbitrary times in the schedule.

3.2 Dynamic Priority Scheduling

In this type of scheduling, different jobs of the same task may have different priorities. Examples of Dynamic Priority scheduling algorithms include *Earliest Deadline First (EDF)*,

Least-Laxity First (LLF).

<u>*EDF*</u>: In EDF scheduling, jobs with earliest deadlines will be given higher priority.

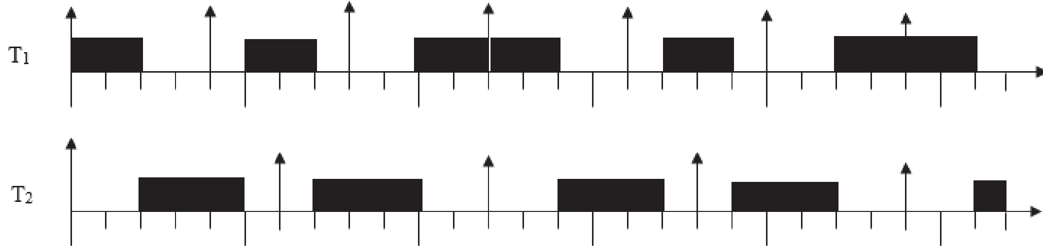


Figure 3.1: Example of Earliest Deadline First Scheduling

Example: Consider a task set $T_1 = (4, 2)$ and $T_2 = (6, 3)$. The EDF schedule for this task set is as shown in figure 3.1. EDF is job level dynamic - i.e., different jobs of the same task can have different priorities.

<u>*LLF*</u>: *Laxity* is defined as the amount of time a job can be in the waiting queue without missing its deadline. In LLF, jobs which have least laxity will be given higher priority.

3.3 Tick Scheduling

In Tick Scheduling, it is assumed that the scheduler only executes at specific points in time. The time between scheduler execution is called the scheduler's quantum. If a job arrives or completes within a quantum boundary, the scheduler waits until the next quantum boundary before checking if a different job should execute. Also, all the jobs of the tasks following tick scheduling execute for at least 1 time unit.

3.4 Fixed Priority Scheduling

This is also called Static priority scheduling. In fixed priority scheduling, all the jobs of a task have same priority and these priorities will not change at any point in the system [7]. It is assumed that no two tasks have same priority. The scheduler will choose the tasks to be executed depending upon their priorities - i.e., the task with highest priority will be executed as soon as it arrives. If there is a lower priority job executing when a higher priority job arrives, the lower priority job will get preempted and the higher priority jobs will execute.

Example: Consider a system with 4 tasks and all of them arrive at time 0. Let $T_1 = (5,2), T_2 = (7,1), T_3 = (10,2), T_4 = (18,3)$. The total system utilization is 0.9. Assuming that our priority ordering is $T_1 > T_2 > T_3 > T_4$, the schedule can be as shown in Figure 3.2.

In figure 3.2 as T_1 has the highest priority, it will execute as soon as it arrives. After T_1 completes its execution, if T_2 has arrived, it will be executed next as it has the next highest priority. The lowest priority task T_4 starts executing at time 8. It has to execute for 3 time units to complete its execution. But as the higher priority job of task T_1 arrives at time 10, T_4 will be preempted and T_1's second job will complete its execution. When no other higher priority jobs are executing, T_4 will be given the processor and it will complete its execution and it meets its deadline.

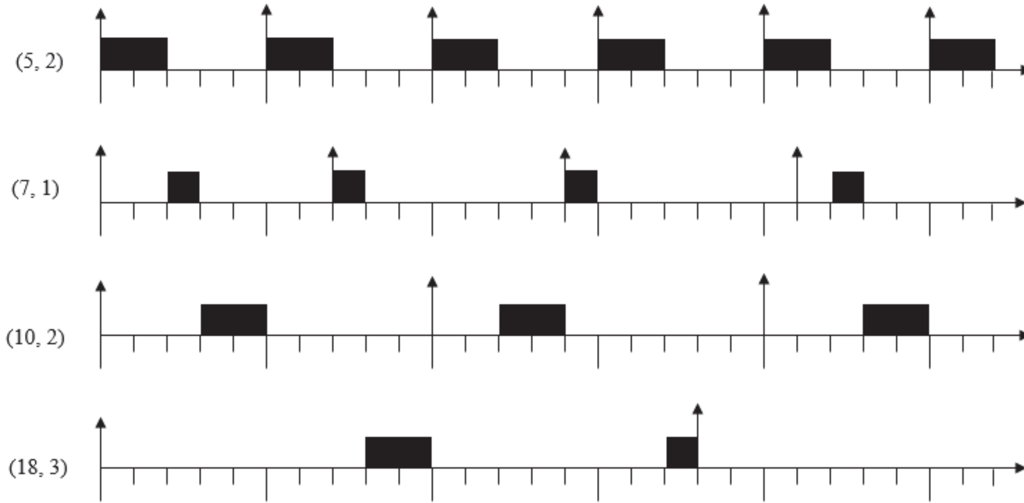


Figure 3.2: Illustration of Fixed Priority Scheduling

Rate Monotonic (RM) Scheduling

In RM scheduling, tasks with smaller periods have higher priority.
Let π_i and p_i denote the priority and period of T_i respectively. For any two periodic tasks T_i and T_j, if $\pi_i < \pi_j$, then $P_i > P_j$ [7]. Typically, we index tasks by their priority i.e., if $i < j$, then $p_i <= p_j$.

Example: Consider a task set with 3 tasks. The tasks are of the form $T_i = (P_i, e_i)$. Let $T_1 = (7, 4)$, $T_2 = (10, 1)$, $T_3 = (12, 2)$. As the tasks are indexed from smallest to largest period, T_1 is given highest priority. The next highest priority is given to T_2. The lowest priority task is T_3 because it has the largest period.
The schedule is as shown in Figure 3.3.

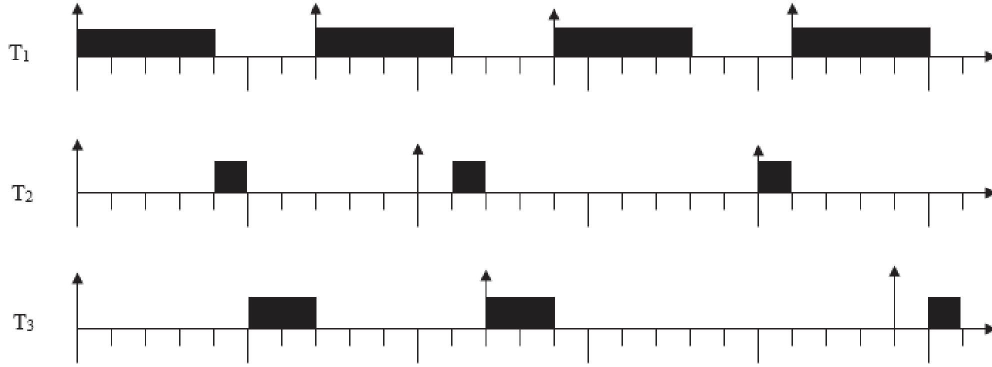


Figure 3.3: Example of Rate Monotonic Scheduling

Deadline Monotonic (DM) scheduling

DM scheduling was proposed by Leung and Whitehead in [13]. In DM scheduling, tasks with smaller relative deadlines have higher priority.

Let D_i and pi_i denote the deadline and priority of T_i respectively. For any two periodic tasks T_i and T_j, if $D_i < D_j$, then $\pi_i > \pi_j$ [13].

Example: Consider a task set with 3 tasks. The tasks are of the form $T_i = (p_i, e_i, D_i)$. When $p_i = D_i$, we use (p_i, e_i).

Let $T_1 = (7, 4)$, $T_2 = (10, 1, 2)$, $T_3 = (12, 2)$. T_2 is given highest priority as it has a smaller relative deadline. The next highest priority is given to T_1. The lowest priority task is T_3 because it has the largest deadline.

As we are considering deadline equals period in this thesis, both RM and DM will generate the same schedule. Hence, we can say that tasks are assigned priorities based on RM or DM scheduling algorithms. Also, Rate Monotonoc scheduling is optimal for fixed-priority scheduling when deadlines are equal to periods on uniprocessor platforms [7].

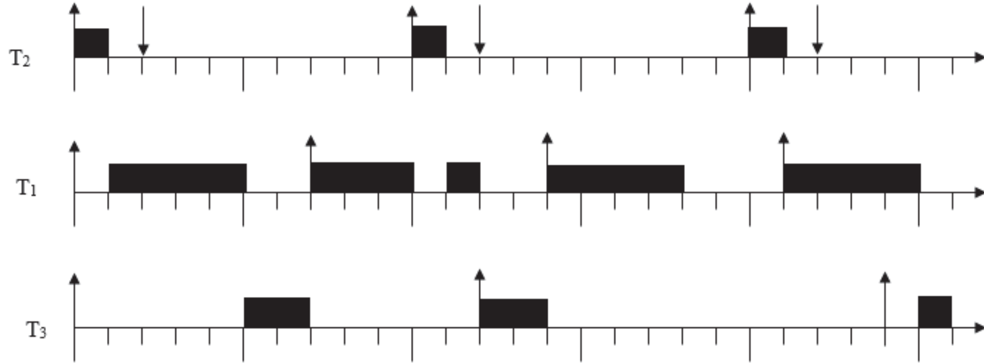


Figure 3.4: Example of Deadline Monotonic Scheduling

Critical Instant Theorem

Critical Instant Theorem applies only to fixed priority scheduling. A *critical instant* for a fixed priority task is defined to be an instant at which a request for that task will have its worst-case behavior [7]. A critical instant of a task T_i is the release time of its any job $J_{i,k}$ such that

1. If the task meets all its deadlines, then $J_{i,k}$ has the maximum response time of all jobs released by T_i.
2. If the task misses some deadlines, the response time of $J_{i,k}$ is greater than D_i.

The critical instant theorem was proposed by Liu and Layland in [7]. According to the critical instant theorem, in a fixed priority system, the worst case response time of any task T_i occurs when the jobs belonging to all the high priority tasks are released at the same instant. The preemptions caused due to the high priority tasks are responsible for this worst case response time. The proof of this theorem is discussed in [7].

Time Demand Analysis (TDA)

TDA applies to synchronous systems when *deadlines* $\leq$ *periods*, which is the worst case scenario by the critical instant theorem. TDA is used to find the Worst Case Response Time (WCRT) of the tasks when using fixed priority scheduling. It was proposed by Lehoczky, Sha, and Ding in [14]. TDA is used to check the schedulability of the task set - i.e., to verify whether all the tasks meet their respective deadlines. For synchronous task sets, TDA computes the exact WCRT. For asynchronous task sets, TDA computes an upper bound on WCRT. In either case, TDA can be used as a schedulability test for RM and DM scheduling.

Let $w_i(t)$ denote the worst case demand in an interval of length t, then the TDA function for any task T_i is defined as the minimum value t such that

$$w_i(t) = e_i + \sum_{k=1}^{i-1} \lceil t/p_k \rceil \cdot e_k \tag{3.1}$$

Note that $\sum_{k=1}^{i-1} \lceil t/p_k \rceil \cdot e_k$ is the worst-case interference due to the higher priority tasks that task T_i may suffer.

TDA performs iterative computation inorder to find the WCRT of a task. The iterative computation is performed as follows: [16]

For any given task number i,

let $t^{(0)} = e_i$

when $k = 0$, $t^{(1)} = w_i(t^{(0)})$

$t^{(k+1)} = w_i(t^{(k)})$ for all $k = 1, 2, 3, ...$

The value of $w_i(t^{(0)})$ can be calculated from the TDA equation.

The iterations are repeated until either of the following occur

1. $t^{(k)} > D_i$ which indicates that the system is infeasible or
2. $t^{(k+1)} = t^{(k)} \leq D_i$ which indicates that the task T_i will meet its deadline

TDA is computed for all tasks T_i where $i = 1, 2, 3, ..., n$.

Example 1: Tasks that meet their deadlines

Let $\tau = \{T_1, T_2, T_3\}$ where $T_1 = (6, 4)$, $T_2 = (11, 1)$, $T_3 = (12, 2)$.

The tasks are of the form $T_i = (p_i, e_i)$.

The RM schedule of this task set is shown in figure 3.5.

Say, we want to check if task T_3 meets its deadline or not. The WCRT computation of task T_3 is as follows:

$t^{(0)} = e_3 = 2$

$t^{(1)} = w_3(t^{(0)}) = e_3 + \lceil t^{(0)}/P_1 \rceil \cdot e_1 + \lceil t^{(0)}/P_2 \rceil \cdot e_2$

$t^{(1)} = 2 + \lceil 2/6 \rceil \cdot 4 + \lceil 2/11 \rceil \cdot 1$

$t^{(1)} = 2 + 4 + 1$

$t^{(1)} = w_3(2) = 7$

Now, $t = 7$

$w_3(7) = 2 + \lceil 7/6 \rceil \cdot 4 + \lceil 7/11 \rceil \cdot 1$

$w_3(7) = 2 + 8 + 1$

$w_3(7) = 11$

Now, $t = 11$

$w_3(11) = 2 + \lceil 11/6 \rceil \cdot 4 + \lceil 11/11 \rceil \cdot 1$

$w_3(11) = 2 + 8 + 1$

$w_3(11) = 11$

Therefore, $WCRT(T_3) = 11$.

As $11 < 12$, we can say that task T_3 will meet its deadline. The computation will be similar for finding the WCRTs of T_1 and T_2. If P_3 were 10, however, T_3 would miss its deadline. Hence, by using Time Demand Analysis, we can find whether or not a synchronous task set meets its deadline.

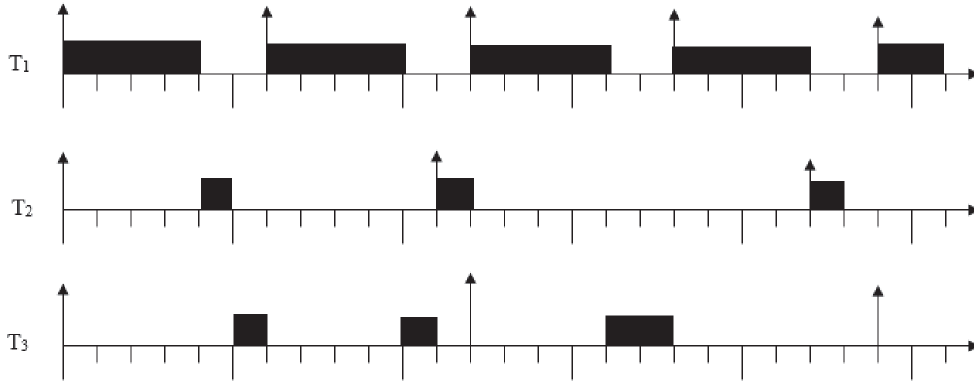


Figure 3.5: Tasks meeting their deadlines

3.5 UUnifast Algorithm

A algorithm called UUnifast algorithm was proposed by Bini and Buttazzo in [8]. This algorithm is used in this thesis for randomly generating the task sets for experimental purposes.

The input to this algorithm is a utilization value $\bar{U}$ and the number of tasks (n). The algorithm will generate n tasks whose total utilization is approximately equal to $\bar{U}$.

The Matlab code in [8] for this algorithm is as follows:

$function\ vectU = UUniFast(n, \bar{U})$

$sumU = \bar{U}$

$for\ i = 1 : n - 1$

$\quad nextSumU = sumU. * rand^{(1/(n-i))};$

$\quad vectU(i) = sumU - nextSumU;$

$\quad sumU = nextSumU;$

end

From the above code, depending upon the n and $\bar{U}$ input values, *vectU* values are computed. Deadlines are generated randlomly using *rand* function of MATLAB.

Execution times are generated using previously computed $VectU$ values and the deadlines.

3.6 Dealing with Mode Changes

As discussed in the previous chapters, real-time systems may have to execute in different modes due to changes in the environment. Scheduling during the mode transition must be done carefully to ensure jobs meet their deadlines.
There are different ways to treat old mode tasks as described in [10] and [15]:

1. Immediately aborting the old-mode tasks: When the old mode tasks are aborted at MCR, the new mode tasks can be released and they can start executing immediately after MCR occurs. When the system has to change its mode to an emergency mode, this will work fine (provided there are no MI tasks). In some cases, however data consistency issues can arise if the old mode tasks are immediately aborted. For example, if the execution of a new mode task was dependent on the value generated after executing an old mode task, aborting this required old mode task will create a consistency issue. The data consistency problem is discussed in [6].

2. Allowing old mode tasks to execute completely: If the old mode tasks are allowed to execute completely, MCD can be missed. Moreover, this can affect the schedule of the new mode tasks by increasing the latency of transition as the new mode tasks have to wait until all the old mode tasks have completed their execution [12].

3. Aborting only some of the old mode tasks: This is more flexible than the above mentioned options. Old mode tasks that were released can execute until there is no feasibility issue. When the system detects that executing a particular old mode task can cause a missed MCD, then the old mode tasks are aborted at

the next safe point in the task's execution. We considered this option in the thesis.

3.7 Introduction to Genetic Algorithms

Genetic algorithms (GAs) are computer programs that mimic the processes of biological evolution in order to solve problems and to model evolutionary systems [19]. Some of the applications of GAs are: [19]

1. Optimization
2. Automatic Programming
3. Machine learning
4. Immune system models
5. Ecological models etc.

A Genetic Algorithm starts by generating a population of candidate solutions which are called *individuals or phenotypes*. Every candidate solution contains some properties called *chromosomes*. New candidates are constructed by a process called reproduction. The solutions of GA are evaluated using function called the *fitness function*. The evoultion from current population to another set of new population is an iterative process where each iteration is called a *generation* [21].

GA's employ three main operations [19] [2]:

1. Selection: This is used for selecting chromosomes, which are candidate solutions, in the population for reproduction. The next generation chromosomes are selected based upon their fitness value. Only the fittest chrosomes are selected as candidate solutions for next generation.

2. Crossover: This is used to create a new chromosome (called offspring) by exchanging subsequences of parent chromosomes.

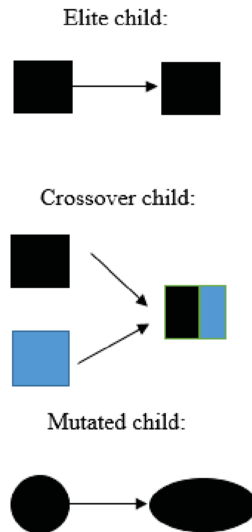


Figure 3.6: Next Generation Children [1]

3. Mutation: GA makes some changes to the chromosomes using mutation function. It is used by the GA to broaden the search space.

Three types of individuals are selected to be in the next generation. They are [1]:

1. Elite children: The individuals with best fitness value are called elite children. These are automatically included in the next generation.

2. Crossover children: These are created by crossover operation of the GA.

3. Mutated children: These are created by the mutation function i.e., by making some random changes to a single parent.

Formation of next generation children is as shown in Figure 3.6.

The basic structure of the genetic algorithm is shown in Figure 3.7.

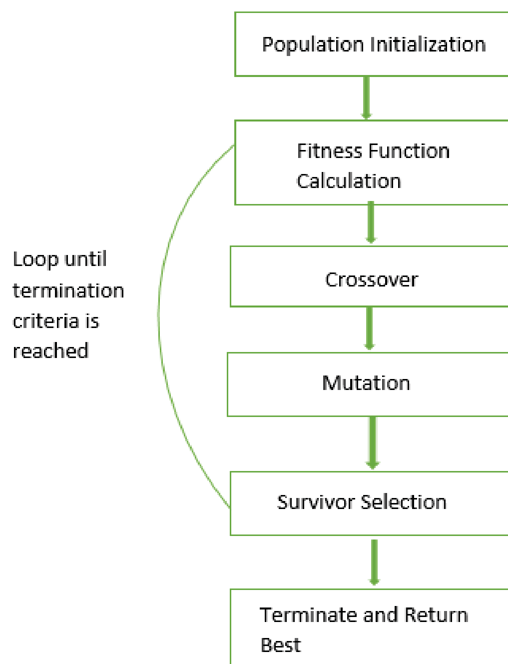


Figure 3.7: Steps invoved in a Genetic Algorithm [3]

Chapter 4

Computing Abort points during Mode Switch

This chapter has been divided into 6 sections. Section 4.1 specifies the requirements for a mode switch. Section 4.2 and 4.3 contains protocols without and with periodicity respectively. Section 4.4 introduces the concept of abort points. Section 4.5 and 4.6 contains protocols for calculating abort points during mode switch without and with periodicity respectively.

4.1 Requirements

The requirements from [12] that we considered for mode-switching in this thesis are:

1. Schedulability: All the old mode and new mode tasks should meet their deadlines. No deadline should be missed because of MCR.

2. Periodicity: We consider Mode Independent Tasks in our schedule. These tasks are supposed to execute irrespective of mode change. The execution of these tasks should not have a negative impact on the new mode tasks.

4.2 Protocol without Periodicity

This thesis uses the protocol described in [12] with some modifications. We are assuming that a MCR can not occur during any mode transition.

The protocol without periodicity is as follows:

- Old mode tasks execute normally till MCR is encountered in the schedule.
- Some of the old mode tasks can be aborted immediately if executing them completely can cause MCD miss.
- Some or all old mode tasks will execute during the mode transition if they do not cause any infeasibility to the schedule.
- If the old mode tasks are to be aborted, the Matlab GA is used to find the maximum amount of time between consecutive abort points for each old mode task.
- New mode tasks will be released at MCD. The system moves into the new mode when there are no old mode jobs running.

This repeats till another MCR is encountered in the system. The protocol will remain same for all the MCRs.

4.3 Protocol with Periodicity

The protocol with periodicity is as follows:

- Old mode tasks execute normally along with Mode Independent tasks until MCR is encountered in the schedule.
- Some of the old mode tasks can be aborted immediately when the MCR occurs.
- Some old mode tasks will execute during the mode transition.

- Mode Independent tasks will execute for Δ time units before MCD.
- Old mode jobs are allowed to execute depending upon their priorities (highest priority old mode job is allowed to execute first and so on) till MCD.
- If the old mode tasks are to be aborted, the GA is used to find the abort point for each old mode task.
- New mode tasks will be released at MCD when the system moves into the new mode. At this point there should be no old mode jobs running.
- Mode Independent tasks will execute normally with these new mode tasks in the new mode.

This repeats till another MCR is encountered in the system. The protocol will be the same for all the MCRs.

Computing the abort points and meeting the Mode Change Deadline is our goal.

4.4 Abort points during Mode Switching

As mentioned above, if the old mode tasks are to be aborted, the GA is used to find the abort points. We need to specify a fitness function and constraint function. The final solution is decided depending upon the fitness values and constraints specified. The GA will generate the values that meet the constraints specified in the constraint function.

1. Fitness Function: The fitness function is a function that we want to maximize or minimize. It is alo known as objective function [4].
2. Constraint function: The constraints that are to be met by the GA solver are specified in the constraint function. At least one constraint is required to run the GA solver. The constraints in our GA are used to ensure the MCD is met.

4.5 Abort points during Mode Switching without periodicity

The system without periodicity contains only old mode and new mode tasks. No Mode Independent tasks are considered in this case.

Whenever a MCR occurs in the schedule, all the old mode jobs are executed if they do not cause any Mode Change Deadline miss. If they cause a MCD miss, which is predetermined by using TDA function, some tasks need to be aborted. We assume that the jobs have already executed for at least 1 time unit before a MCR occurs as per the definition of Tick Scheduling.

- Let X =Mode Change Relative Deadline
- Sum_{old} = Sum of remaining execution times of the old mode jobs after they have executed for 1 time unit

$$Sum_{old} = \sum_{T_i \in M_{old}} (e_i - 1)$$

- A = Amount of time old mode execution can take place in X

Case 1: If $Sum_{old} \leq X$, no deadline is missed. Hence we execute all the old mode jobs till MCD and then the new mode jobs are releseased and executed.

Case 2: If $Sum_{old} > X$, the mode change deadline can be missed and hence we need to abort some old mode jobs and allow only some old mode tasks to execute. The jobs that are to be aborted depends upon the fitness function used.

Fitness Functions: This thesis considers several fitness functions.

Constraint Function: The constraint that we placed on GA solver is sum of abort points should be $\leq X$.

Old mode tasks can execute for A amount of time in X and then all the remaining old mode jobs are to be aborted inorder to meet MCD.

Example: Consider $T_1 = (4, 1)$, $T_2 = (10, 3)$, $T_3 = (11, 2)$ and $T_4 = (20, 4)$. The tasks have already executed for 1 time unit. Let e_{rem} be the remaining amount of execution time. $e_{rem} = [0, 2, 1, 3]$.

$Sum_{old} = 0 + 2 + 1 + 3 = 6$

If $X = 6$, then $Sum_{old} \leq X$, so no deadline is missed and there is no need to abort any of the old mode jobs.

If $X = 5$, then $Sum_{old} > X$, MCD can be missed and hence we need to abort some of the old mode jobs.

The input to the GA is $[0, 2, 1, 3]$

The output from the GA is $[0, 2, 1, 2]$.

In this case, 1 time unit of task T_4 is aborted as executing it completely can cause MCD miss and this can further have a negative impact on the new mode tasks.

4.6 Abort points during Mode Switching with periodicity

Impact of Mode Independent Tasks on new mode tasks

If the mode independent tasks are not able to execute completely during the mode transition phase, they can cause the new mode tasks to become infeasible at some point in the schedule.

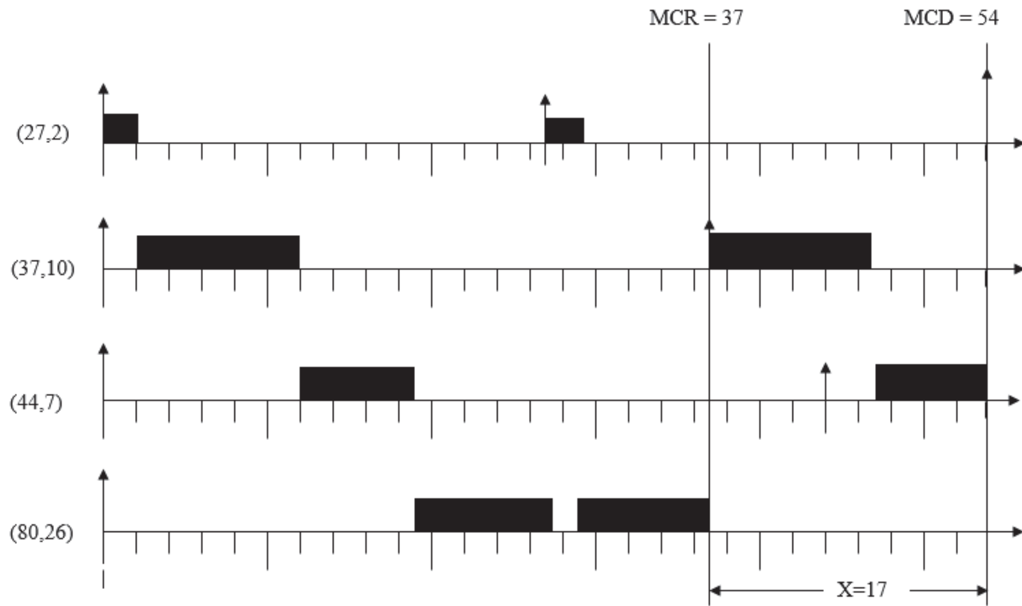


Figure 4.1: Old mode tasks and mode independent tasks are being executed

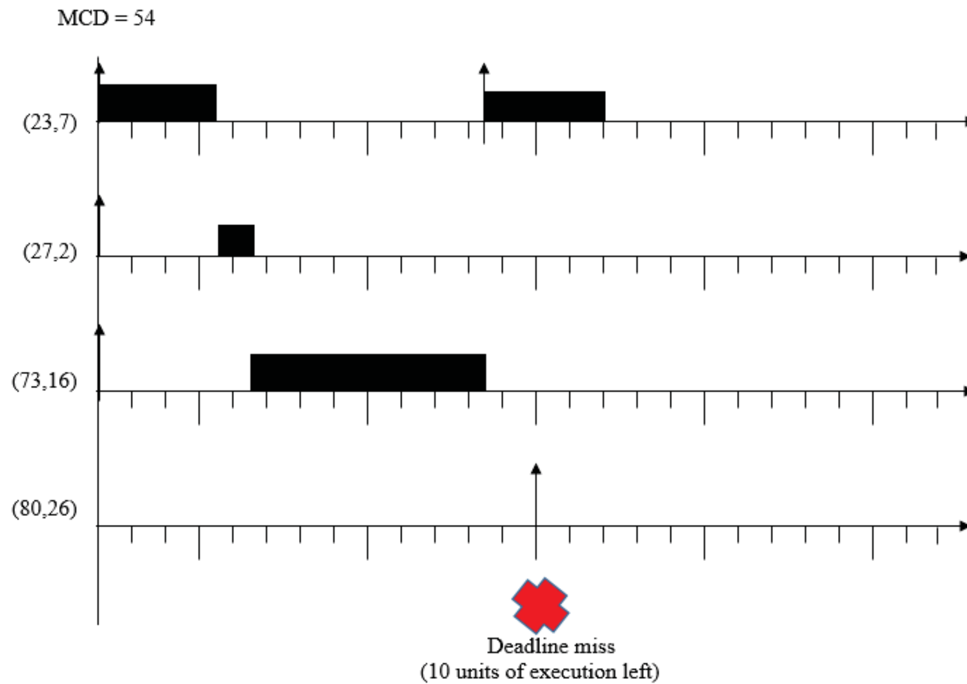


Figure 4.2: Deadline miss in the new mode

Consider a schedule where $T_1 = (27, 2)$, $T_2 = (37, 10)$, $T_3 = (44, 7)$ and $T_4 =$

$(80, 26)$, $T_5 = (23, 7)$, $T_6 = (73, 16)$.

T_2, T_3 are old mode tasks, T_1, T_4 are mode independent tasks and T_5, T_6 are new mode tasks.

The impact of mode independent tasks on new mode when they are not allowed to execute before the MCD is as shown in figures 4.1 and 4.2.

The task priorities are assigned based on RM scheduling algorithm. The tasks were executing normally until a MCR is encountered at time 37. At that time T_4 has executed for 16 time units and 10 time units of execution was left. During the mode transition phase, the high priority old mode tasks - i.e., T_2 and T_3 executes leaving no time for T_4 to execute during that interval. MCD occurs at time 54 and at this point of time, the system moves into the new mode. As mode independent tasks should also execute in the new mode, the schedule of T_1 and T_4 will remain the same even in the new mode but their priorities will differ based on the periods of new mode tasks. In the new mode, high priority tasks will execute first and T_4 never gets a chance to execute until time 80 which is its deadline. At this point, a deadline is missed making the system infeasible. If this MI task T_4 was allowed to execute before the MCD aborting some or all of the old mode jobs, the working of the system would have remained unaffected.

Therefore, we ensure the following to guarantee no new mode deadline misses:

1. MI tasks execute to completion before MCD or
2. Last released MI job should execute from its release to MCD.

Abort points when there is 1 Mode Independent task in the schedule

In this type of schedule, apart from the old and new mode tasks, we have 1 Mode Independent Task. All the occurrences of the Mode Independent task should be executed in the interval [MCR, MCD] as they must execute irrespective of the Mode

Change. In the remaining time, the old mode jobs can execute. To know how much time old mode tasks have in the interval [MCR, MCD], we need to calculate the demand of the Mode Independent task in the same interval.

Below is the notation we will use in this section (See figure 4.3)

e_I = Execution time of Mode Independent Task

p_I = Period of the Mode Independent Task

MCR = Mode Change Request Time

MCD = Mode Change Deadline

X = Relative Mode Change Deadline - i.e., $X = MCD - MCR$

a_I = Last arrival time of Mode Independent Task before MCR

Y = Amount of time between a_I and MCR

k = Number of times Mode Independent Task releases a new job in [MCR, MCD].

L = Length of the interval from a_I to MCD i.e., $L = X + Y$

Δ = Amount of time last released job of MI Task has to execute before MCD

ξ_I = Remaining Mode Independent Task execution time at MCR

R_I = Worst Case Response Time of Mode Independent Task

We want to deal with the worst case behavior of the Mode Independent task. To know the minimum amount of time available for old mode tasks, the demand of the Mode Independent task should be made as high as possible in the interval [MCR, MCD].

Lemma 4.6.1. *The demand of the Mode Independent Task in the interval [MCR, MCD] is high when* $\Delta = e_I$.

Proof. To ensure schedulability of the new mode tasks i.e., to make sure that new mode tasks are not negatively impacted by the mode independent jobs, we ensure that:

1. last released MI job should complete its execution at or before MCD, as discussed

in the previous section

2. last released MI job should execute from its release until MCD.

The Δ value can either be $= e_I$ or $< e_I$.

Case 1: $\Delta = e_I$:

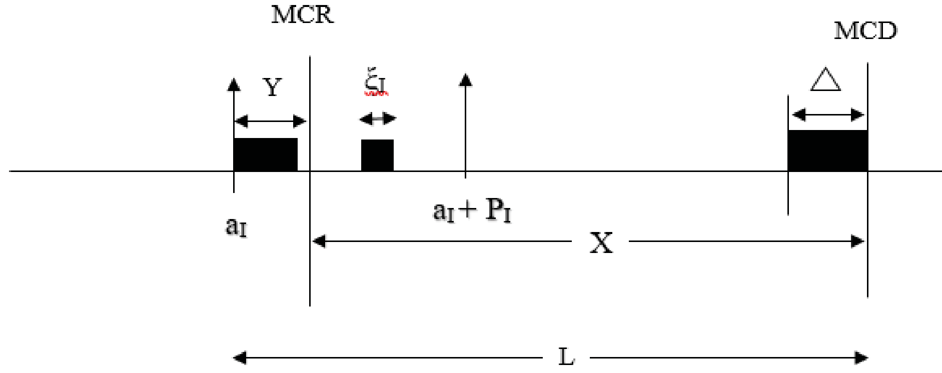


Figure 4.3: Figure indicating the case where $\Delta = e_I$

The old mode tasks should stop executing before the arrival of this last Mode Independent job before MCD and the MI task can execute until MCD.

Case 2: $\Delta < e_I$:

The Mode Independent Task cannot complete its execution and in the remaining time, the old mode tasks can execute until $MCD - \Delta$.
We need to know which of the above 2 cases will be the worst case i.e., in which case the demand of the Mode Independent Task is higher when compared to the other. We know that the demand in the interval [MCR, MCD] will be high when $\xi_I + \Delta$ is as large as possible $\implies a_I$ should be made to arrive as late as possible.

Since $X + Y - \Delta$ is a multiple of p_I, we can say that $X + Y - \Delta$ is divisible by

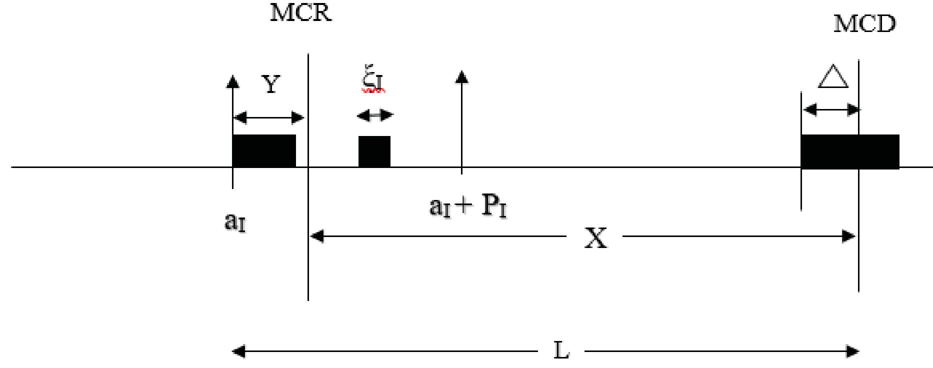


Figure 4.4: Figure indicating the case where $\Delta < e_I$

p_I. Therefore, the number of releases of the MI jobs in the interval [MCR, MCD] excluding the last MI job can be given as:

$$k = \begin{cases} ((X + Y - \Delta)/P_I) - 1, & \text{if } X > 0 \\ (X - \Delta)/P_I, & \text{if } Y = 0 \end{cases}$$

In the above equation, we subtract -1 as we know that the job released at $MCR - Y$ is released before MCR.

We now compare the case where $\Delta = e_I$ to the case where $\Delta < e_I$. Let $w = e_I - \Delta$. If T_I's release time is decreased by w time units i.e., T_I is shifted back w time units, then the demand of the last job of T_I will increase by w. Also, the demand of the first job(s) of T_I will decrease by *at most* w.
Therefore, the total MI demand is maximized when $\Delta = e_I$.

So, Mode Independent Task should execute exactly e_I time units before the MCD.

When $\Delta = e_I$, k can given by

$$k = \left\lfloor \frac{(X - e_I)}{p_I} \right\rfloor + 1 \tag{4.1}$$

The remaining MI execution at MCR can be calculated as:

$$\xi_I \leq R_I - Y \tag{4.2}$$

where R_I is the $WCRT$ of that task and $Y = (k \cdot P_I) + e_I - X$.

Demand of Mi task in [MCR, MCD] is computed using:

$$Demand = k \cdot e_I + \xi_I \tag{4.3}$$

The amount of time old mode tasks can execute in the mode transition is:

$$Amount_{oldmode} = X - Demand \tag{4.4}$$

□

Steps to find available time for old mode tasks during mode transition phase

1. The utilization of the task set is computed using the formula 2.2.
2. The interval [MCR, MCD] is choosen randomly i.e., a MCR can occur at any time in the schedule.
3. The task which will behave as MI task is also selected randomly.
4. The number of releases of MI task in [MCR, MCD] i.e., k value is computed using equation 4.1.

5. Y value is computed from the k value.

6. R_I value is calculated to know if there is any remaining execution of previously released MI task at MCR.

7. Overall demand of MI task in [MCR, MCD] is computed using 4.3.

8. From the demand, we get the amount of time remaining for the old mode tasks.

9. This $Amount_{oldmode}$ is given as a constraint to the GA for calculating the abort points.

10. The GA uses the constraint function and the previously specified fitness functions to generate the result.

Example:

Consider $T_1 = (37, 7)$, $T_2 = (46, 8)$, $T_3 = (77, 14)$, $T_4 = (92, 19)$.

Using the steps mentioned in 4.6, we get:

Utilization = 0.76

$[MCR, MCD] = 49$

T_1 is selected as MI task

$k = 2$

$Y = 32$

$R_I = 7$ implies there is no MI execution left at MCR

$Demand = 14$

$Amount_{oldmode} = 35$

Remaining executions of old mode task = [0, 7, 13, 18]

Abort points using different fitness functions are as follows:

Fitness Function 1: [0 5 13 17]

Fitness Function 2: [0 5 12 18]

Fitness Function 3: [0 4 13 18]

Abort points when there are 2 Mode Independent tasks in the schedule

In this type of schedule, apart from the old and new mode tasks, we have 2 Mode Independent Tasks. All the occurrences of the Mode Independent tasks should be executed in the interval [MCR, MCD] as they must execute irrespective of the Mode Change. In the remaining time, the old mode jobs can execute. To know how much amount of time old mode tasks have in the interval [MCR, MCD], we need to calculate the demand of these Mode Independent tasks in the same interval.

Let T_{I_1} = Task with latest arrival before MCR.

T_{I_2} = Second Mode Independent Task

$T_{I_j} = (e_{I_j}, p_{I_j})$ where $j = 1, 2$

a_{I_1} = Arrival time of T_{I_1}

a_{I_2} = Arrival time of T_{I_2}

Y = Amount of time between a_{I_1} and MCR

Z = Amount of time between a_{I_2} and MCR

V = Amount of time between the first release of the MIT_1 after a_{I_2} and MCR

k_1 = Number of occurrences of the MIT_1 in the interval [MCR, MCD]

k_2 = Number of occurrences of MIT_2 in the interval [MCR, MCD]

KMCR = Number of occurrences of MIT_1 in the interval Z-Y

ξ_{I_1} = Remaining T_{I_1} execution time at MCR

ξ_{I_2} = Remaining T_{I_2} execution time at MCR

R_{I_2} = WCRT of T_{I_2} in Δ

MCR, MCD, X, Δ defined as in section 4.6

We want to deal with the worst case behavior of Mode Independent tasks. The worst case behavior of mode independent tasks occurs when demand of these tasks is as high as possible in the interval [MCR, MCD].

Lemma 4.6.2. *The demand of two Mode Independent Tasks in the interval [MCR, MCD] is high when* $\Delta = e_I + R_{I_2}$.

Proof. To ensure schedulability of the new mode tasks, i.e., to make sure that new mode tasks are not negatively impacted by the mode independent jobs after MCD, we want either of the following cases to occur:
1. Last released MI jobs should complete its execution at or before MCD
2. Last released MI jobs should execute from its release until MCD.

We want the demand of the Mode Independent Tasks to be as high as possible in the interval [MCR, MCD]. For the demand to be as high as possible, the sum of ξ_{I_1} and ξ_{I_2} should be made as high as possible which implies that the arrival of both the Mode Independent Tasks should be made as late as possible in the schedule.

As we are considering two Mode Independent Tasks in the interval Δ, the release and execution of high priority mode independent task can cause some interference to the execution of low priority mode independent task. So, we need to consider the R_{I_2} in the interval Δ.

WCRT of T_{I_2} in Δ is given by:

$$R_{I_2}^k = e_{I_2} + \left(\left\lceil \frac{R_{I_2}^{k-1} + e_{I_1}}{p_{I_1}} \right\rceil - 1 \right) \cdot e_{I_1} \tag{4.5}$$

where $\left\lceil \frac{R_{I_2}^{k-1} + e_{I_1}}{p_{I_1}} \right\rceil - 1$ is the number of occurences of T_{I_1} in Δ.

Assume we have a schedule as shown in figure 4.5 with worst case MI demand and $\Delta < R_{I_2}$. Let $T_{I_1,\delta}$ be the last job of T_{I_1} released before time δ and let $T_{I_1,D}$ be the last job of T_{I_1} released before MCD.
We shift T_{I_1} forward until one of the following occurs:

1. $T_{I_1,\delta}$ completes at time δ

2. $T_{I_1,D}$ completes at time MCD

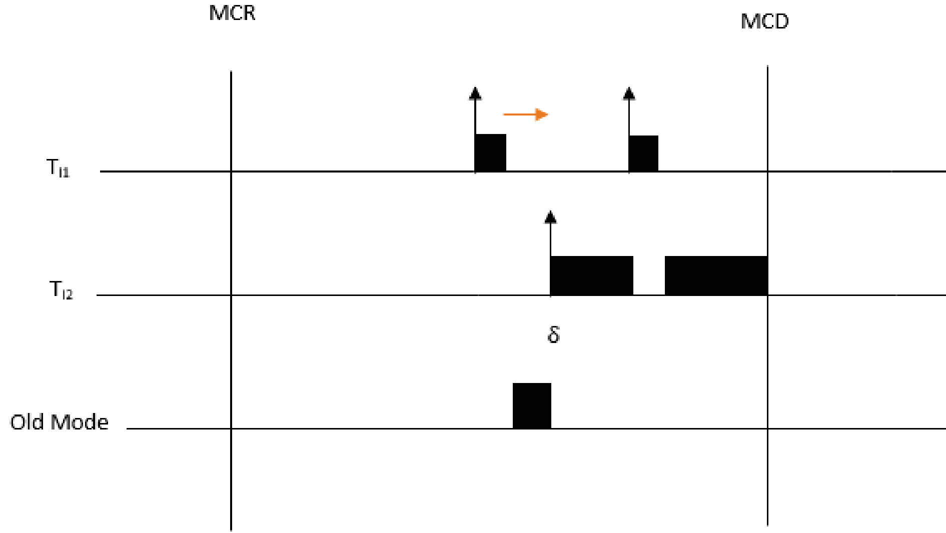


Figure 4.5: Shifting MI jobs forward

The total demand in [MCR, MCD] does not decrease during this shift as all jobs released during the interval are still being released and are able to complete during this interval. By shifting T_{I_1} or T_{I_2} forward, their previous arrivals a_{I_1} and a_{I_2} also shift forward. Shifting a_{I_1} and a_{I_2} forward does not decrease the overall demand of the Mode Independent Tasks in the interval [MCR, MCD]. Infact, having them arrive as late as possible can increase the remaining execution times (ξ_{I_1} and ξ_{I_2}), thus increasing the overall demand (as Δ is also increased). Also, having old mode jobs execute before the mode independent jobs does not change the demand in [MCR, MCD].

Therefore, $\Delta = e_{I_1} + R_{I_2}$. □

Claim: Δ starts with the release of one or both MI tasks

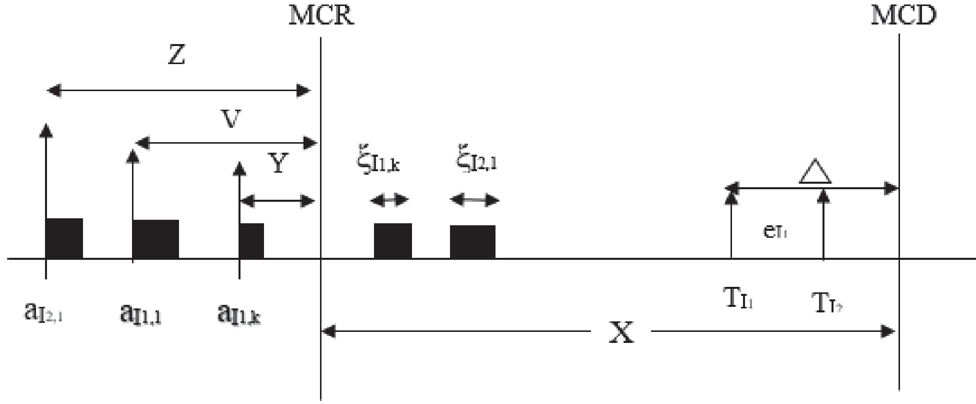
T_{I_1} and/or T_{I_2} are pushed forward such that we will have Δ start with the release of one or both MI jobs. Either of the following cases occur:

Case A: If T_{I_1} is released at $MCD - \Delta$

Case B: If T_{I_2} is released at $MCD - \Delta$

We have 2 tasks to execute during Δ: $T_{I_1} = (P_{I_1}, e_{I_1})$, $T_{I_2} = (P_{I_2}, e_{I_2})$

Case A:



When T_{I_1} is released before T_{I_2} and T_{I_1} has higher priority than T_{I_2}, it will complete its execution in e_{I_1} time units. Even if T_{I_2} has released, it cannot execute until T_{I_1} completes. We can push the release of T_{I_2} at a time when T_{I_2} completes so that it will not get preempted until new job of T_{I_1} arrives. T_{I_2} can be made to release at time $MCD - \Delta + e_{I_1}$ and it executes untill T_{I_1} releases again.

The number of occurrences of T_{I_1} in [MCR, MCD] can be found using

$$k_{1_A} = \lfloor (X - \Delta)/P_{I_1} \rfloor + \lceil \Delta/P_{I_1} \rceil \tag{4.6}$$

When T_{I_2} is released at $MCD - \Delta + e_{I_1}$ and as its previous releases are multiples of P_{I_2}, the number of occurrences of T_{I_2} in the interval [MCR, MCD] can be given as

$$k_{2_A} = \lfloor (X - \Delta + e_{I_1})/P_{I_2} \rfloor + \lceil (\Delta - e_{I_1})/P_{I_2} \rceil \tag{4.7}$$

The arrival of T_{I_1} before MCR can be given by

$$Y = k_{1_A} \cdot P_{I_1} - (X - \Delta) \tag{4.8}$$

The arrival of T_{I_2} before MCR can be given by

$$Z = k_{2_A} \cdot P_{I_2} - (X - R_{I_2}) \tag{4.9}$$

Subtracting R_{I_2} from X will give T_{I_2}'s arrival in Δ when T_{I_1} is released at time $X - \Delta$.

Here we need to find the first occurrence of T_{I_1} after the arrival of T_{I_2} before MCR. So, we will need number of occurrences of T_{I_1} in the interval $Z - Y$ and it can be given as

$$KMCR = \lfloor (Z - Y)/P_{I_1} \rfloor \tag{4.10}$$

The total number of occurrences of T_{I_1} in the interval MCR-Z

$$V = Y + (KMCR \cdot P_{I_1}) \tag{4.11}$$

Using 4.10 and 4.11, the first occurrence of T_{I_1} after the arrival of T_{I_2} before MCR is given as

$$V_1 = MCD - V \tag{4.12}$$

We want to find how much can the last job of T_{I_1} execute before MCR and what would be its remaining execution at MCR.

$$X_{I_1} = Y - Intf_{old} \tag{4.13}$$

where $Intf_{old}$ is the higher priority old mode task interference in the interval Y (if any) which is given by

$$Intf_{old} = \Sigma\lceil Y/P_i\rceil \cdot e_i \tag{4.14}$$

Amount T_{I_1} can execute in the interval Y is:

$$X_{I_1} = Y - \Sigma\lceil Y/P_i\rceil \cdot e_i \tag{4.15}$$

Remaining amount of execution of T_{I_1} at MCR is

$$\xi_{I_1} = e_{I_1} - X_{I_1} \tag{4.16}$$

Now, we want to find how much T_{I_2} can execute in the interval of length Z and what will be its remaining execution time at MCR. Maximum amount of execution of T_{I_2} in the interval Z is

$$X_{I_1} = Z - Intf \tag{4.17}$$

where Intf is the sum of the interferences caused due to old mode tasks and due to the Mode Independent Task T_{I_1} before MCR.

Interference caused due to T_{I_1} in the interval of length V:

$$Intf_{T_{I_1}} = \lceil V/P_{I_1}\rceil \cdot e_{I_1} \tag{4.18}$$

Interference caused due to old mode tasks in the interval of length Z:

$$Intf_{oldmode} = \Sigma\lceil Z/P_i\rceil \cdot e_i \tag{4.19}$$

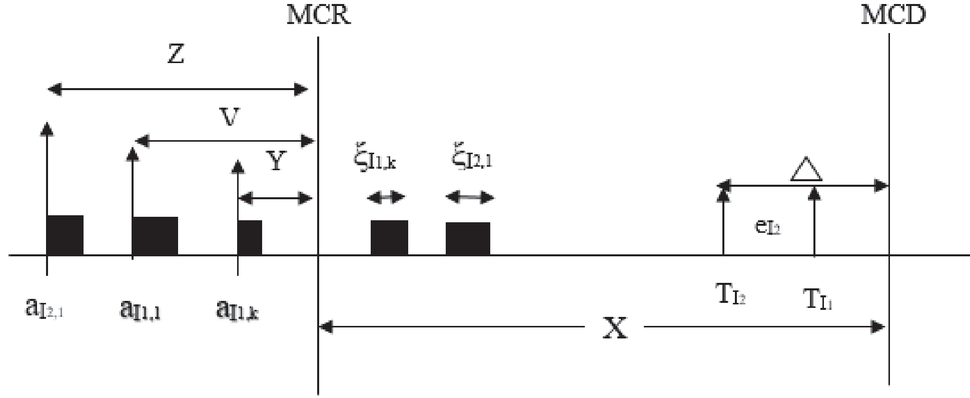
Amount of time T_{I_2} executes before MCR:

$$X_{I_2} = Z - \Sigma\lceil Z/P_i\rceil \cdot e_i + \lceil V/P_{I_1}\rceil \cdot e_{I_1} \tag{4.20}$$

Remaining execution of T_{I_2} at MCR

$$\xi_{I_2} = e_{I_2} - X_{I_2} \tag{4.21}$$

Case B:



When T_{I_2} is released before T_{I_1}, it can execute until T_{I_1} releases. As T_{I_1} has higher priority than T_{I_2}, everytime T_{I_1} is released, T_{I_2} will be preempted.

If T_{I_2} takes its R_{I_2} to complete its execution during Δ and as we want to shift the new release of T_{I_1} such that T_{I_1} is released after T_{I_2} completes to avoid preemptions, we can say that the last job of T_{I_1} will be released at $MCD - e_{I_1}$ which will execute till MCD.

We need to find number of occurrences of T_{I_2} in $X - \Delta$. As the previous releases are multiples of P_{I_2}, we can get number of occurrences in this interval by $\lfloor (X - \Delta)/P_{I_2} \rfloor$. Total number of occurrences of T_{I_2} in the interval [MCR, MCD] can be given as

$$k_{1_B} = \lfloor (X - \Delta)/P_{I_2} \rfloor + \lceil \Delta/P_{I_2} \rceil \tag{4.22}$$

When T_{I_1} is released at $MCD - e_{I_1}$ and as its previous releases are multiples of P_{I_1}, number of occurrences of T_{I_1} in the interval [MCR, MCD] can be given as

$$k_{2_B} = \lfloor (X - \Delta + e_{I_1})/P_{I_1} \rfloor + \lceil (\Delta - e_{I_1})/P_{I_1} \rceil \tag{4.23}$$

The arrival of T_{I_1} before MCR can be given by

$$Y_B = k_{2_B} \cdot P_{I_1} - (X - R_{I_1}) \tag{4.24}$$

where $R_{I_1} = \Delta - e_{I_1}$.

The arrival of T_{I_2} before MCR can be given by

$$Z_B = k1 \cdot P_{I_2} - (X - \Delta) \tag{4.25}$$

Number of occurrences of T_{I_2} in the interval $Z - Y$ can be given as

$$KMCR_B = \lfloor (Z_B - Y_B)/P_{I_1} \rfloor \tag{4.26}$$

Total number of occurrences of T_{I_2} in the interval MCR-Z

$$V_B = Y_B + (KMCR \cdot P_{I_1}) \tag{4.27}$$

The amount of time T_{I_1} executes before MCR:

$$X_{I1_B} = Y_B - \Sigma \lceil Y_B/P_i \rceil \cdot e_i \tag{4.28}$$

where $\Sigma \lceil Y/P_i \rceil \cdot e_i$ is the interference caused due to high priority old mode tasks

The remaining amount of execution at MCR:

$$\xi_{I_1} = e_{I_1} - X_{I_1} \tag{4.29}$$

The amount of time T_{I_2} executes before MCR:

$$X_{I_2} = Z_B - \Sigma\lceil Z_B/P_i\rceil \cdot e_i + \lceil V_B/P_{I_1}\rceil \cdot e_{I_1} \tag{4.30}$$

where $\Sigma\lceil Z/P_i\rceil \cdot e_i$ is the interference caused due to old mode tasks and $\lceil V/P_{I_1}\rceil \cdot e_{I_1}$ is the interference caused due to first mode independent task

$$\xi_{I_2} = e_{I_2} - X_{I_2} \tag{4.31}$$

Total remaining mode independent execution at MCR is:

$$\xi_I = \xi_{I_1} + \xi_{I_2} \tag{4.32}$$

Calculate the above equations for both Case A and Case B. Select the case which has highest ξ_I value as it will give us the highest mode independent demand in X.

The old mode tasks can execute for $X - Demand_{MI}$.

Chapter 5

Experiments and Results

This chapter contains three sections: Section 5.1 contains the values used for performing the experiments. Sections 5.2 and 5.3 contains the experiments and results for 1 and 2 Mode Independent tasks respectively.

5.1 Experimental setup

The following values were used in the experiment setup:

- Processor: All the experiments were performed on uniprocessor platform. 5th generation Intel core i7 processor was used. Core i7 processor was used to get faster CPU performance when compared to other intel processors like core i5 or core i3.
- Task sets: Task sets were generated by the UUnifast algorithm as described in section 3.5. 100 task sets were used for all the experiments.
- Tasks per task set: Number of tasks used were 2^n where n=2, 3, 4, 5 and 6.
- Utilization: Utilization values were dependent upon the task set generated. It was always less than 1 (as anything greater than 1 can cause deadline miss).
- Population: Population sizes used were 500, 750 and 1000.

- Generations: 350, 425 and 500 generations were used for poplulation sizes 500, 750 and 1000 respectively.

- Elite count: Elite count of 0.05*population was used.

- Crossover fraction: Crossover fraction of 0.9 was used as an input to the genetic algorithm. Crossover fraction is used to determine the number of next generation crossover children. Pairs of parents from the initial population are combined to generate crossover children as shown in 3.6. Crossover enables the genetic algorithm to extract the best genes from different individuals and recombine them into potentially superior children [1]. Crossover fraction of 0 means that all children are generated through mutation. Crossover fraction of 1 means that all children other than elite children are generated by crossover [5].

- Calculating the crossover children: For population size = 500 and crossover fraction = 0.9,
  Elite children = 25
  Crossover children = 0.9*475 = 427
  Remaining 48 children are generated by mutation.

- Stopping criteria: Stall generation limit and function tolerance specifies the stopping criteria. Stall generation limit of 250 was used for all the experiments. GA runs until the average relative change in the fitness function value over stall generations is less than function tolerance [1]. It was set to 10^{-6}.

- Bounds to the GA: The bounds used were:

  1. Lower bound: 0

  2. Upper bound: $e_i - 1$ (Remaining execution time of old mode tasks as we are assuming tick scheduling)

- Constraint function: One constraint function was used through out. The

constraint specified was sum of abort points should always be less than or equal to the relative mode change deadline.

- Fitness functions: The fitness functions used are:

  1. $\sum \frac{\delta_i}{e_i}$, where i ranges from 1 to the number of tasks used
  2. $\sum \frac{\delta_i}{p_i}$, where i ranges from 1 to the number of tasks used
  3. $\sum \delta_i \cdot U$, where i ranges from 1 to the number of tasks used

5.2 Experiments for 1 Mode Independent task

Comparison of Runtimes for different Fitness Functions

In this section, the runtime of GA for finding abort points using different fitness functions is analyzed for various population sizes and generations. The graphs in the figure 5.1 show differences in runtimes for different fitness functions as well as for different population and generation values.

- When population = 500 and generations = 350

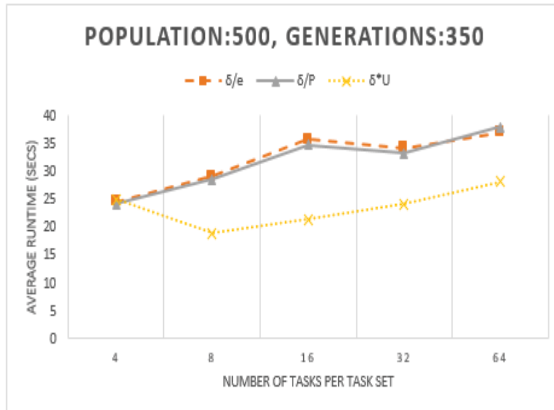
  For 4 tasks per task set, GA took almost the same amount of time to compute abort points using all the 3 fitness functions. As the number of tasks per task set is less, lesser computations are needed and so it took almost the same time. When the number of tasks per task set was increased i.e., when 8, 16, 32 and 64 tasks per task set were used, the runtime of GA was nearly the same for fitness functions $\sum \frac{\delta}{e}$ and $\sum \frac{\delta}{p}$. But for the fitness function $\sum \delta \cdot U$, the average runtime was very low when compared to other fitness functions. It may be due to the fact that, the utilization value (U) was constant for each task set. Multiplication of δ with a constant value may take lesser time in comparison to the divison performed using variable execution and period values per task.

- When population = 750 and generations = 425

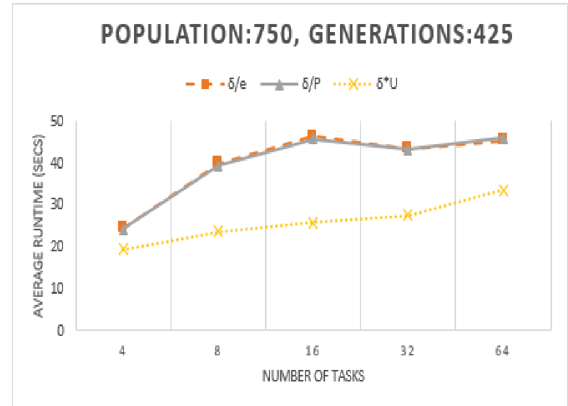
  When the population size and number of generations were increased to 750 and 425 respectively, the average runtime was also increased. Like previous case, the average runtime for fitness functions $\sum \frac{\delta}{e}$ and $\sum \frac{\delta}{p}$ remained approximately same and the runtime using $\sum \delta \cdot U$ was very less compared to other fitness functions.

- When population = 1000 and generations = 500

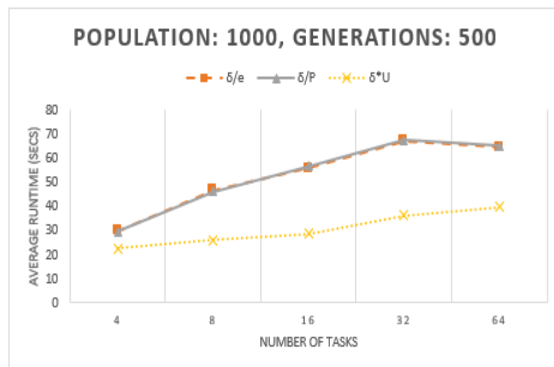
  The results of this case were also similar to the above 2 cases.



(a)



(b)



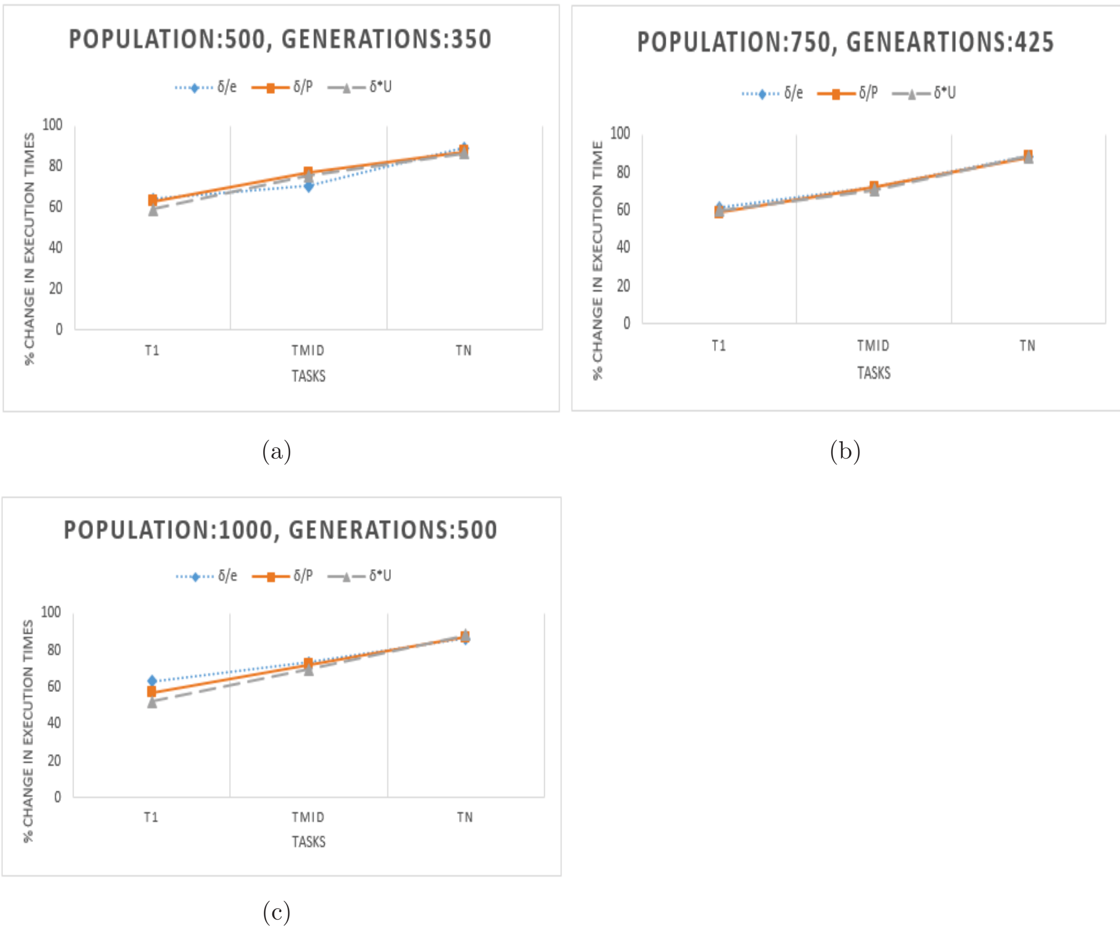
(c)

Figure 5.1: Comparison of Runtimes for 1 MI task

Comparison of Abort points for different Fitness Functions

4 Tasks per Task Set

In this case when we had 4 tasks per task set. For all the fitness functions as shown in figure 5.2, it was observed that percentage change in the execution times of high priority tasks was less when compared to the mid priority tasks. Percentage change in the execution time of low priority tasks was the highest using all the fitness functions. This means that GA aborted low priority tasks much more than high and mid priority tasks.



(a)

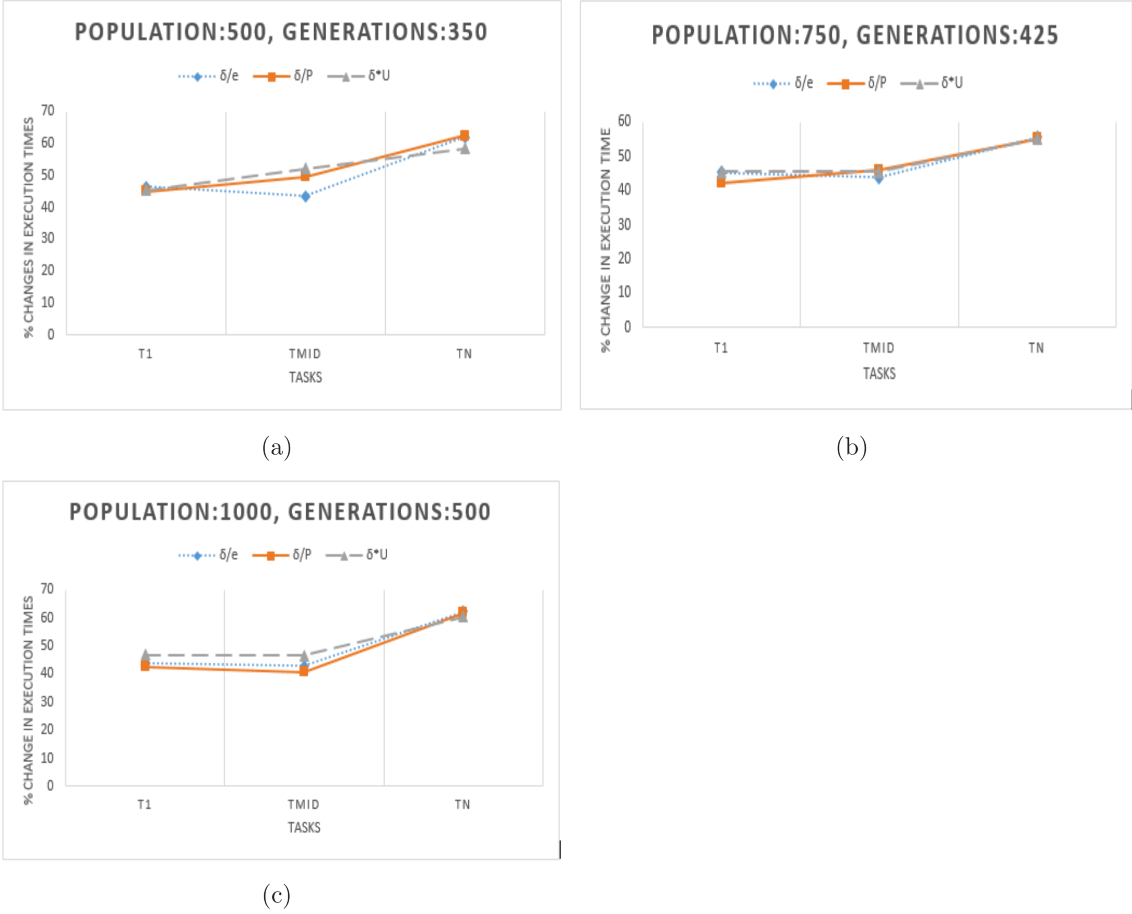
(b)

(c)

Figure 5.2: 4 Tasks (1 MI)

8 Tasks per Task Set

All the fitness functions in this case and the next case (where we have 16 tasks per task set) behaved in the same fashion. The results generated by the GA were close to each other as shown in 5.3 and 5.4. The low priority tasks were aborted the most.

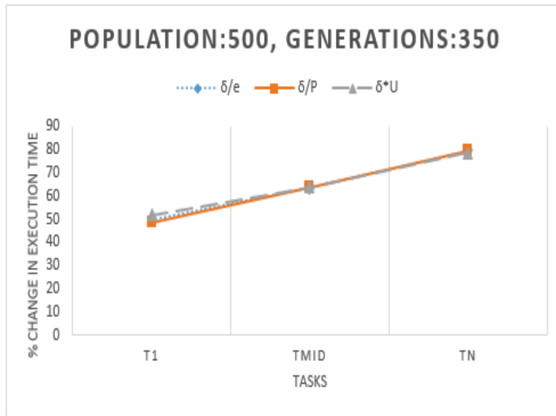


(a) (b) (c)

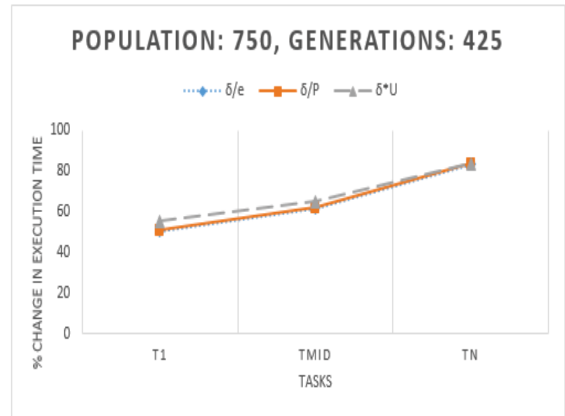
Figure 5.3: 8 Tasks (1 MI)

16 Tasks per Task Set

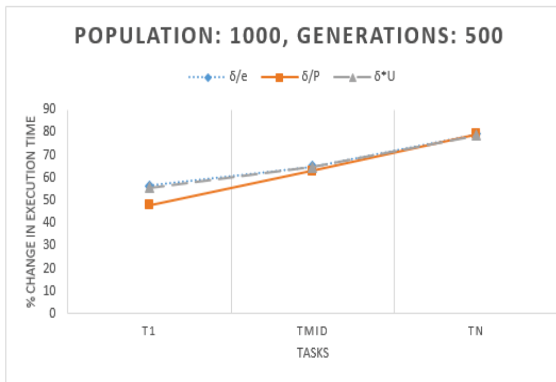
The percentage changes in T_{mid} and T_n were similar and very less difference in the percentage changes for T_1 was observed.



(a)



(b)

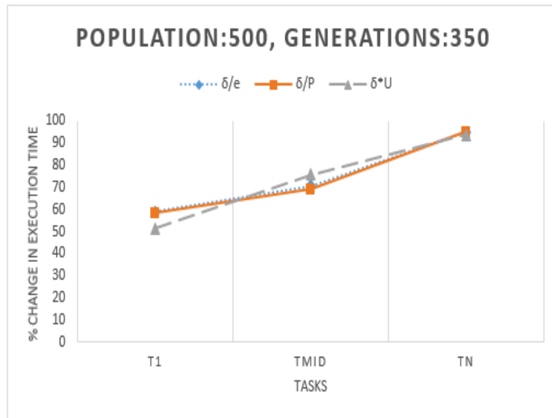


(c)

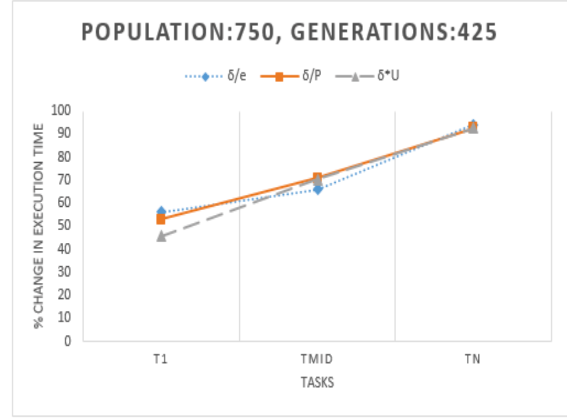
Figure 5.4: 16 Tasks (1 MI)

32 Tasks per Task Set

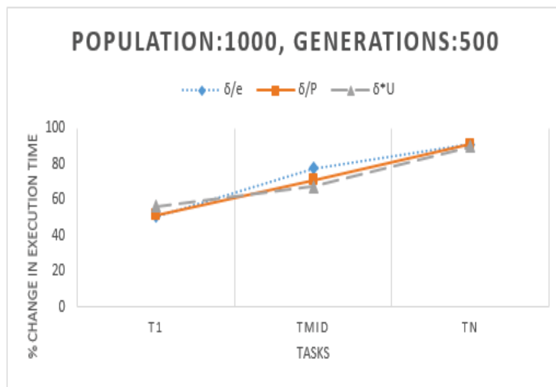
In this case and in the next case, the solution generated by the GA using $\sum \delta \cdot U$ was slightly better than using the other fitness functions. High priority tasks were favored the most using $\sum \delta \cdot U$ and low priority tasks were favored the least. Like the other cases, $\sum \frac{\delta}{e}$ and $\sum \frac{\delta}{p}$ gave the same result. The graphs are as shown in figures 5.5 and 5.5.



(a)



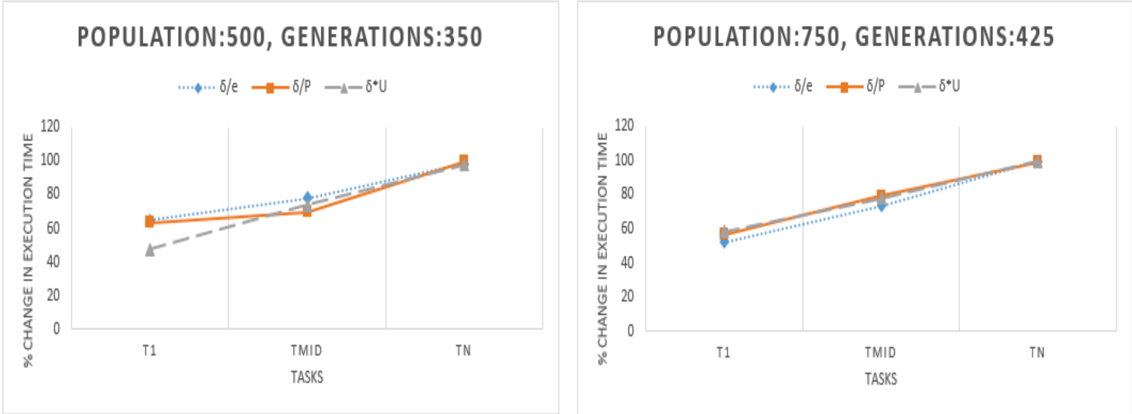
(b)



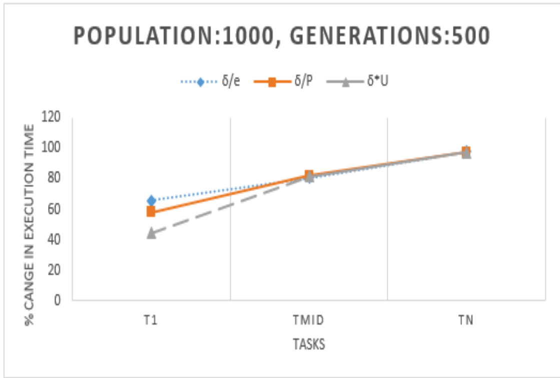
(c)

Figure 5.5: 32 Tasks (1 MI)

64 Tasks per Task Set



(a) (b)



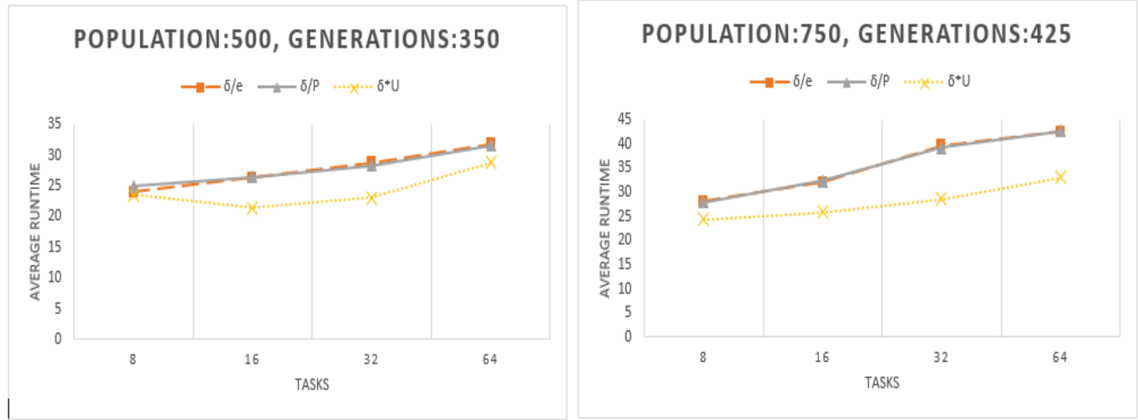
(c)

Figure 5.6: 64 Tasks (1 MI)

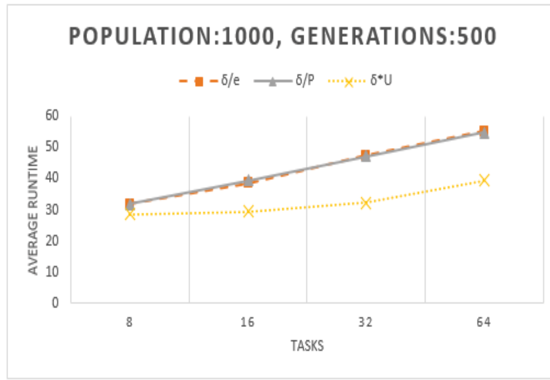
5.3 Experiments for 2 Mode Independent task

Comparison of Runtimes for different Fitness Functions

The runtimes for 2 MI tasks behaved similar to that of 1 MI task exepriments. $\sum \frac{\delta}{e}$ and $\sum \frac{\delta}{p}$ took a little more time when compared to $\sum \delta \cdot U$ as shown in 5.7. In case of runtimes, $\sum \delta \cdot U$ was the best out of three fitness functions used.



(a) (b)



(c)

Figure 5.7: Comparison of Runtimes for 2 MI Tasks

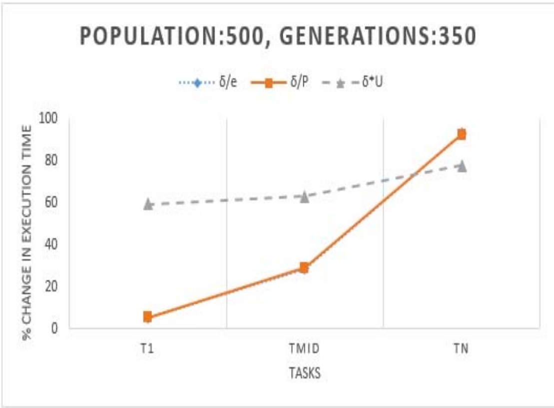
Comparison of Abort points for different Fitness Functions

A trend was observed in this case: the fitness functions $\sum \frac{\delta}{e}$ and $\sum \frac{\delta}{p}$ seemed to favor the execution of high priority tasks and abort the low priority tasks. On the other hand, the fitness function $\sum \delta \cdot U$ did not favor any tasks and the abort points were uniformly distributed. The graphs following this trend are shown in figures 5.8, 5.9, 5.10 and 5.11.

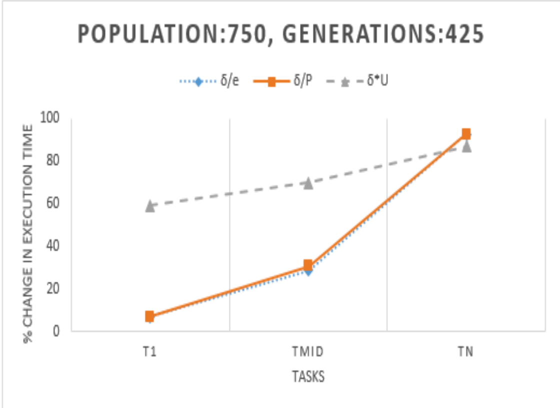
If we want to have the changes in the old mode tasks to be uniformly distributed, $\sum \delta \cdot U$ would be a better choice but if we want to give more importance to the execution of any high priority tasks, we can choose from $\sum \frac{\delta}{e}$ or $\sum \frac{\delta}{p}$ fitness functions

as they behaved almost in the same way. The fitness functions used here are relevant to the generic real-time applications. However, for specific applications, the fitness functions can be altered to suit the needs of the user.

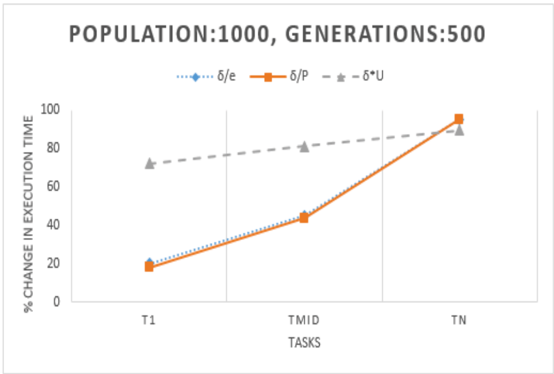
8 Tasks



(a)



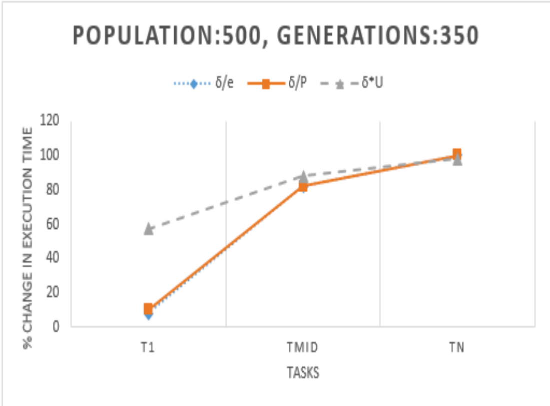
(b)



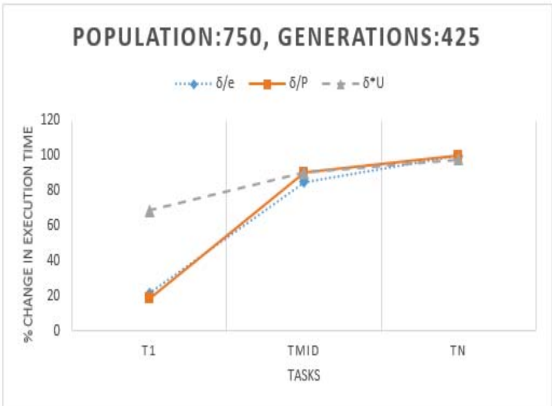
(c)

Figure 5.8: 8 Tasks (2 MI)

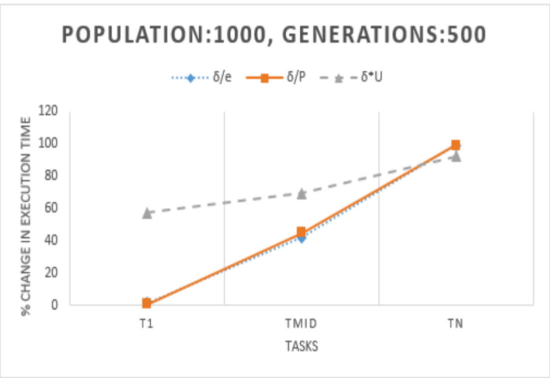
16 Tasks



(a)



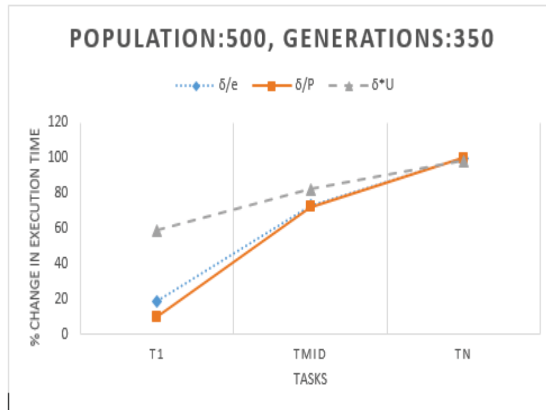
(b)



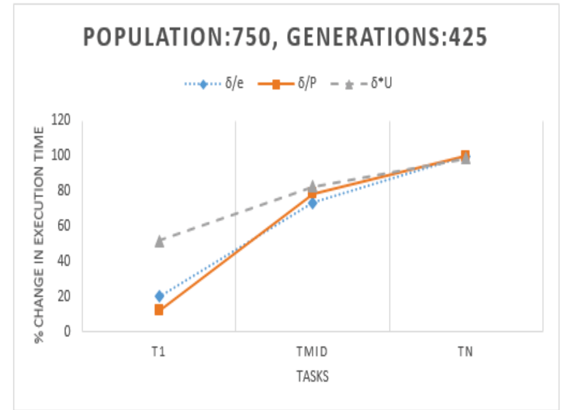
(c)

Figure 5.9: 16 Tasks (2 MI)

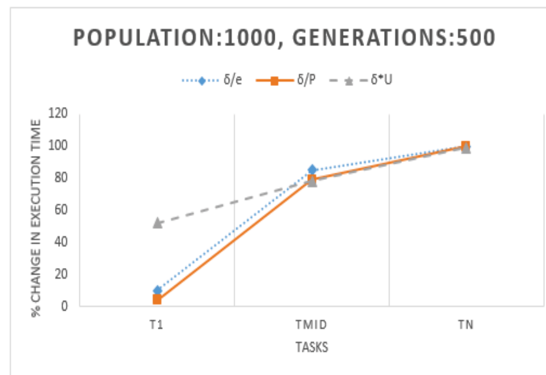
32 Tasks



(a)



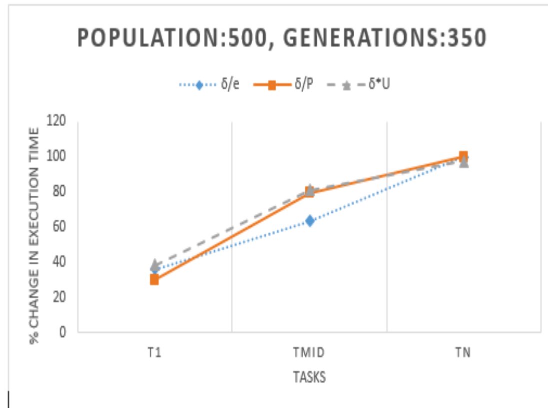
(b)



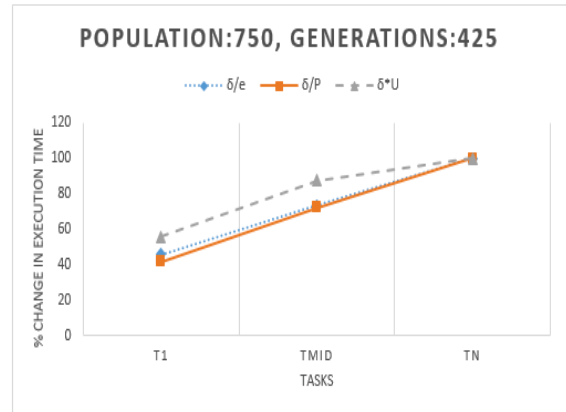
(c)

Figure 5.10: 32 Tasks (2 MI)

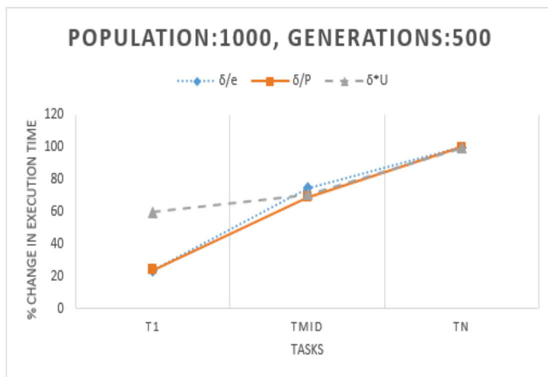
64 Tasks



(a)



(b)



(c)

Figure 5.11: 64 Tasks (2 MI)

Chapter 6

Conclusion and Future Work

Many real-time applications exhibit a multi-mode behavior [12]. They are required to switch between various modes while still meeting all the deadlines. This mode switching, in which old, new and mode independent tasks may execute, can cause a temporal overload on the system. The overload may cause the real-time application to miss its deadlines thereby affecting the functionality of the system. To deal with this issue, the mode change behavior was simulated and the results were analyzed using 2 MI tasks in the schedule to ensure that all deadlines will be met during the mode switch. A genetic algorithm was used to find the abort points where the old mode tasks can abort without causing any negative impact on the new mode tasks. Different fitness functions were used to analyze the behavior of GA. The use of these fitness functions is for generic applications. For specific applications, the fitness functions can be tailored as per the requirements of the system.

There are many avenues to explore along these lines. We have used and analyzed the behavior of upto 2 MI tasks in the system. Having just 2 MI tasks showed huge increase in the complexity of the system. This work can be extended to 3 or more MI tasks. Additionally, other platforms such as multiprocessors could be examined. Crossover rate in the GA was kept constant as 0.9. This rate can be varied to study the different types of solution generated by the genetic algorithm as well as exploring

other GA parameters such as mutation, etc. This work can also be extended by using different fitness functions apart from those used here.

REFERENCES

[1] http://www.mathworks.com/help/gads/how-the-genetic-algorithm-works.html. See Section: Creating the Next Generation, Link accessed on 10/14/2016.

[2] http://in.mathworks.com/help/gads/genetic-algorithm-options.html. Link accessed on 10/14/2016.

[3] http://www.tutorialspoint.com/genetic_algorithms/genetic_algorithms_quick_guide.htm. See Section: Basic Structure, Link accessed on 10/14/2016.

[4] http://www.mathworks.com/help/gads/some-genetic-algorithm-terminology.html. See Section: Fitness Functions, Link accessed on 09/15/2016.

[5] https://www.mathworks.com/help/gads/vary-mutation-and-crossover.html. See Section: Setting the Crsossover Fraction, Link accessed on 10/14/2016.

[6] Alonso A. *Extensiones a los métodos de planificación de sistemas de tiempo real críticos basados en prioridades.* PhD thesis, Universidad Politécnica de Madrid, Spain, 1994.

[7] Liu C.L., and James.W. Layland. Scheduling algorithms for multiprogramming in a hard real-time environment. *Journal of the ACM*, 1973.

[8] Bini E. and Buttazzo G.C. Measuring the performance of schedulability tests. In *Real-Time Systems, 30(1-2)*, pages 129–154, 2005.

[9] Farcas E. *Scheduling Multi-Mode Real Time Distributed Components*. PhD thesis, University of Salzburg, Dept. of Computer Science, 2006.

[10] Fohler G. J. *Flexibility in statically scheduled hard real-time systems*. PhD thesis, Technische Universitat Wien, 1994.

[11] Nelis V., Goosens J., and Andersson B. Two protocols for scheduling multi-mode real-time systems upon indentical multiprocessors. In *Proceedings of the 21st Euromicro Conference on Real-Time Systems*, pages 151–160, 2009.

[12] Real J., and Crespo A. Mode change protocols for real-time systems: A survey and a new proposal. *Journal of Real-Time Systems, 26*, pages 161–197, 2004.

[13] Leung J.Y.T. and Whitehead J. On the complexity of fixed-priority scheduling of periodic real-time tasks. *Performance Evaluation, Vol 2, Issue 4*, pages 237–250, 1982.

[14] Lehoczky C., Sha L., and Ding Y. The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In *IEEE Real-Time Systems Symposium*, 1989.

[15] Jahanian, Lee and Mok. Semantics of modechart in real time logic. In *Proceedings of the 21st Hawaii International Conference on Systems Sciences*, pages 479–489. IEEE, 1988.

[16] Jane W.S. Liu. *Real-Time Systems*. Prentice Hall, 2000.

[17] Degani A., Shafto M., and Kirlk A. Modes in automated cockpits: Problems, data analysis and a modeling framework. In *Proceedings of the 36th Israel Annual Conference on Aerospace Sciences*, 1996.

[18] Rajab M. *Real-Time Systems: Theory and Practice*. Prentice Hall, 2010.

[19] M. Mitchell. Genetic algorithms: An overview. In *Complexity, Vol 1*, pages 31–39, 1995.

[20] Brinkley S. *Aperiodic Task Scheduling for Real-Time Systems*. PhD thesis, Carnegie Mellon University, 1990.

[21] Russell S., and Norvig P. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[22] Simon P., Nikolay S., and Lothar T. Reliable mode changes in real-time systems with fixed priority or edf scheduling. In *Design, Automation and Test in Europe Conference and Exhibition (DATE)*, 2009.

[23] Yu, Yang, Sun and Jiang. Improving the efficiency of schedulability tests for fixed priority preemptive systems. *Software Engineering, Artificial Intelligence, Networking and Parallel/Distributed Computing*, pages 31–44.