GENERALIZED MULTI-FRAME SCHEDULING WITH FAULT TOLERANCE

by

Likhita Kode

(Under the Direction of Shelby Funk)

Abstract

Computer systems are prone to faults. Faults are of two types: transient and permanent. The performance of events becomes unpredictable upon the occurrence of such faults. This behavior introduced by transient faults might cause deadlines to be missed making hard real-time systems fail. Check–pointing is a cost-effective way of detecting faults. A generalized multi–frame (GMF) task is a sporadic task model where the execution times, deadlines and minimum separation times are all N-array vectors. Upon an occurrence of a fault, the set of GMF tasks re—execute causing an increase in the overall execution time of the job causing other jobs to miss their deadlines. To make the system schedulable, this thesis introduces checkpointing mechanism called Generalized Multi–frame with Fault Tolerance (GMF–FT) for the GMF tasks. We tackle faults by checkpointing the set of GMF tasks and find the worst-case recovery time sequence of the given task set.

Index words: Fault, Checkpoint, Frame, GMF, Recovery time

Generalized Multi-frame scheduling with Fault Tolerance

by

Likhita Kode

B.Tech., Jawaharlal Nehru Technological University, 2015

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

2019

©2019

Likhita Kode

All Rights Reserved

Generalized Multi-Frame Scheduling with Fault Tolerance

by

Likhita Kode

Approved:

Major Professors: Shelby Hyatt Funk

Committee: Liming Cai
Maria Hybinette

Electronic Version Approved:

Suzanne Barbour
Dean of the Graduate School
The University of Georgia
May 2019

.

Generalized Multi–frame Scheduling with Fault Tolerance

Likhita Kode

2019

Acknowledgments

I would like to thank my major Professor, Dr. Shelby Funk for the constant support, guidance, and patience. She has been a great mentor and helped me throughout my research and my defense.

I would also like to thank my committee members, Dr. Maria Hybinette and Dr. Liming Cai for their valuable suggestions and input.

I owe a huge thanks to my parents who have been my backbone and helped me to strive and achieve my goals. Thank you for your love and support.

Last but not the least, my family and friends who have constantly been there with me on this journey.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

A real-time system is one which includes both logical and temporal correctness requirements– i.e., a system that produces correct outputs at the right time. Examples of real time systems are power plant controllers and airplane autopilot systems. Real–time systems can be classified as hard real–time systems or soft real-time systems. Hard real-time systems are systems where, if the response does not occur within a specified time interval catastrophic consequences can occur. Examples: flight control systems, automotive systems. On the other hand, soft real systems are those whose response times are not critical for the system to operate, and failure of meeting a deadline would not hurt the system. Examples: banking system and multimedia.

The basic units of work in real-time systems are *jobs*. A job is defined by its execution time, relative deadline and arrival time. A hard real–time job must execute for its given execution time within its arrival time and deadline.

In real–time systems, jobs often execute repeatedly. We call these repeated jobs a *task*. These tasks can be periodic, sporadic or aperiodic depending upon their arrival pattern. Periodic tasks are those where the separation between release times is the

same for all jobs. Sporadic tasks have a minimum separation between the release of its jobs. Aperiodic tasks are the tasks that are released at arbitrary times.

The periodic task model has been generalized to the multi–frame model [1] and the generalized multi–frame model. In these models, a sequence of jobs executes repeatedly [2].

Computer systems are prone to faults. Faults can be of two types, transient and permanent faults. Transient faults are more common and are major sources of system errors. They are usually resultant of temporary environmental conditions [3]. The performance of events becomes unpredictable upon the occurrence of such faults. Permanent faults are the ones that do not go away with time and are due to irreparable damages to hardware.

There needs to be a mechanism that checks for any fault occurrences. This problem has been extended and considered for periodic and sporadic task systems. This thesis considers the problem of fault tolerance for GMF task systems. A method called Generalized Multi-frame with Fault Tolerance (GMF–FT) is presented that ensures the jobs are scheduled and do not miss their deadlines. In particular, this thesis provides analysis for ensuring GMF tasks meet their deadlines in the presence of transient faults. For small systems, experiments demonstrate that, while this system is pessimistic, it only rejects a small number of task sets that could actually meet their deadlines.

In the following chapters, we discuss the various models and definitions used in real–time systems as seen in Chapter 2. Chapter 3 talks about the different kinds of processors, schedulers, scheduling algorithms, scheduling analysis strategies etc. This research is discussed in detail in Chapter 4 where, we define the framework and how the algorithm works, following up with experiments and results in Chapter 5.

Chapter 2

Model and Definitions

In real–time systems, many operations are performed repeatedly. These repeated operations are called tasks. Each task set is defined as $\tau = \{T_1, T_2, \ldots, T_n\}$ and each task T_i denoted by a 4–tuple representation (ϕ_i, E_i, D_i, P_i),where

- ϕ_i = release time of first job (i.e., task's phase),
- E_i = worst case execution time of each job,
- D_i = task relative deadline (amount of time between each job's release time and deadline), and
- P_i = task period

If $\phi_i = 0$, we use a 3–tuple (E_i, D_i, P_i) and if $\phi_i = 0$ and $P_i = D_i$, we use a pair (E_i, P_i).

Worst Case Execution Time (WCET) is the maximum length of time a task could take to execute without missing its deadline on a processor. The response time of a task is defined as the time when the task starts to execute to the time it finishes its execution. Worst Case Response Time (WCRT) is the maximum response time a task can have.

Each task in the system is a collection of number of jobs. Each job J_i is designated with its execution times E_i, deadline $D_{i,j}$ and arrival time $a_{i,j}$. These tasks can either be periodic or sporadic in nature. If the tasks are periodic, every job arrives exactly P_i time units, after its predecessor. Meaning, i^{th} job of task has an arrival time $a_{i,j}$, a deadline $a_{i,j} + D_{i,j}$, and an execution requirement of E_i, where:

- $a_{i,0} = \phi_i$
- $a_{i,j+1} = \phi_i + j \times P_i$

For example, let us assume that the arrival time for the task is 0, the notation being used is $T_i = (E_i, D_i, P_i)$. Example: Let us consider two tasks $T_1 = (3, 5, 8)$ and $T_2 = (4, 10, 10)$. We assume that T_1 is of a higher priority than T_2. Figure 2.1 illustrates the schedule of the task set. The arrows pointing upward represent the arrival of the jobs in the task and the downward arrow represent the deadlines.

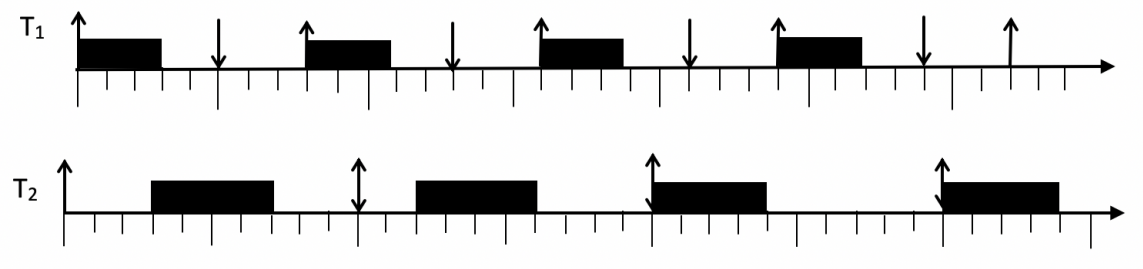


Figure 2.1: Periodic Task Model

Since task T_1 is of higher priority, it starts to execute from time 0 and releases a job every 8–time units. Each job has a deadline of 5–time units after it arrives. Task T_2 is released at time 0, and starts its execution at 4–time units as it must wait for the higher priority task, T_1. T_2 releases its jobs at every 10–time units.

The task utilization is the proportion of time the processor spends executing the tasks over long intervals. The utilization for task T_i is denoted as follows:

$$U_i = \frac{E_i}{P_i}$$

The total utilization of task set τ is given as:

$$U = \sum_{i=1}^{n} \frac{E_i}{P_i}$$

Example: The utilization for T_1 is

$$U_1 = \frac{3}{8}$$

The utilization for T_2 is

$$U_2 = \frac{3}{10}$$

Total Utilization

$$U = \frac{3}{8} + \frac{3}{10} = \frac{27}{40} = 0.67$$

.

2.1 Synchronous and Asynchronous Systems

Synchronous systems are systems where all the jobs of the tasks are released at the same time. Asynchronous systems, on the other hand, are those where jobs are released at arbitrary times instead of the same time.

2.2 Generalized Multi-Frame Model

Multi–frame tasks were introduced by Mok and Chen in 1996 as a form of representation of the generalized form of a periodic task model defined by Lui and

Layland in 1973. A multi–frame task T is represented by a tuple $(\vec{E}, \vec{P})$ where $\vec{E} = \{E_0, E_1, E_2, \ldots, E_{N-1}\}$ are the execution times and $\vec{P} = \{P_0, P_1, P_2, \ldots, P_{N-1}\}$ are the minimum separations respectively. The task generates infinite number of frames in succession where the arrival times of the frames are P time units apart from each other.

A generalized form of multi–frame model [1] leads to a new model called the generalized multi–frame (GMF) model [2] where, the deadlines differ from the periods, the minimum separations are not identical [4]. A GMF task T is represented using a 3–tuple notation $(\vec{E}, \vec{D}, \vec{P})$ where $\vec{E}$, $\vec{D}$ and $\vec{P}$ are N-array vectors where $\vec{E} = \{E_0, E_1, E_2, \ldots, E_{N-1}\}$ are the execution times; $\vec{D} = \{D_0, D_1, D_2, \ldots, D_{N-1}\}$ are the relative deadlines, and $\vec{P} = \{P_0, P_1, P_2, \ldots, P_{N-1}\}$ are the minimum separations respectively.

Meaning, i^{th} frame of task T has an arrival time a_i, a deadline $a_i + d_i$, and an execution requirement of e_i, where:

- $a_0 \geq 0$, and $a_{i+1} \geq a_i + P_{i \mod N}$,
- $d_i = D_{i \mod N}$ and
- $e_i = E_{i \mod N}$

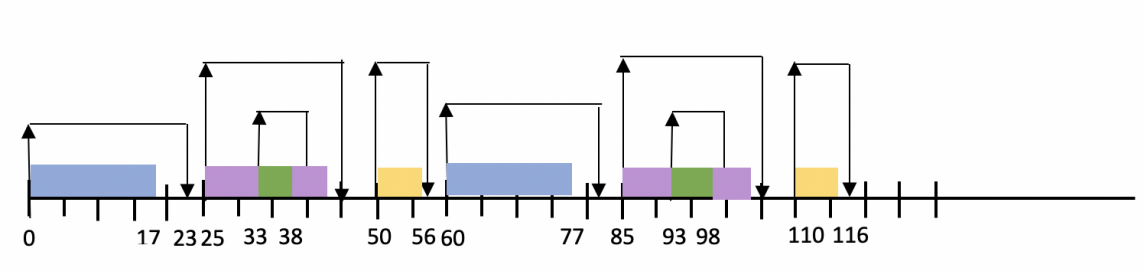


Figure 2.2: Illustration of a GMF Task

For example: Let us consider the GMF task $T = ([17, 12, 5, 6]; [23, 20, 7, 7]; [25, 8, 17, 10])$ with $N = 4$. A schedule of this GMF task set is shown in Figure 2.2. For task T_i, R_i denotes the frame number of the task. The colored blocks represent the execution

times of each frame. R_i^j represents the j^{th} occurrence of the i^{th} frame of task T_i. We see that frame 1 is released (indicated by the up arrow) at time=25 ($P_0 = 25$) and has a relative deadline of 20 time units (indicated by the down arrow at time = 45), i.e., $d_1 = 25 + 20 = 45$. Before frame 1 finishes execution, frame 2 is released at time $= 25 + 8 = 33$ (i.e., $a_2 = a_1 + P_1 = 33$) and has a relative deadline of 7 time units. Therefore, its deadline is $d_2 = 33 + 7 = 40$. The next frame is released at 50 time units ($a_3 = a_2 + P_2 = 33 + 17 = 50$) and has a relative deadline of 7 time units. The next set of the frames, i.e., R_i^2 where $i = \{0, 1, 2, 3\}$ start to repeat from 60 time units $(25 + 8 + 17 + 10)$.

2.3 Task Model for Checkpointing

The unpredictable behavior introduced in the system by a transient fault might cause deadlines to be missed making the hard real-time system fail. Check pointing is a cost-effective way through which faults can be detected in real-time systems [5] when compared to different techniques like message logging [6], retry [7], and replication [7]. The saved state of the task that consists of data variables and system register contents, is called a *checkpoint*. Checkpointing can also be used for improving reliability in databases where an audit trail keeps a record of the last transactions after the last successful checkpoint. This avoids re-execution of the whole process improving the processing time. To ensure the correctness of the computation, an acceptance test is done at each checkpoint.

Let O_i be the computation time overhead necessary to establish one checkpoint in task T_i. The overhead consists of two parts, O_i^A and O_i^S, where O_i^A is the overhead for performing acceptance test and O_i^S is the overhead for saving the system state at each checkpoint. In this study, due to reasons of simplicity, we use a single factor

for overheads (O_i). Therefore,

$$O_i = O_i^A + O_i^S$$

We are primarily concerned with transient faults which are resolved by re–execution. We assume that the operating system kernel has adequate fault containment by using redundancy as well as robust data structures. We also assume a fault-free scheduler – i.e., the scheduler has no adverse effects on the occurrence of faults as it operates at the kernel level, which provides isolation of faults and prevents system crashing [8]. We assume that a fault can adversely affect only one task at a time [9], and it is detected at some intermediate checkpoints or at the end of that task's execution and the minimum inter arrival time between faults is denoted as T_F. Although somewhat pessimistic, this assumption is realistic since, in many implementations, task errors are detected by acceptance tests.

The Table 2.1 below shows the change in execution times with respect to the checkpointing mechanism.

Table 2.1: Execution times with Checkpoints

ExecutionTime	RelativeDeadline	Min.Interval	No.of.CheckPoints	Overhead	NewExecutionTime
2	4	5	2	1	3
4	6	5	2	1	5
6	8	4	4	1	8

The table below gives a brief jist of the definitions used.

Table 2.2: Notations

Notation	Description
T_i	Task i
E_i	Execution time of task T_i
D_i	Relative deadline of task T_i
P_i	Minimum Inter-arrival time or Period of task T_i
J_i	Job of a task T_i
a_i	Arrival time of task T_i
$a_{i,j}$	Arrival of job j in task T_i
u_i	Utilization of T_i
U	Total Utilization of task set τ
O_i^A	Overhead to perform Acceptance Test
O_i^S	Overhead to perform state save
O_i	Overhead for Performing CheckPointing
m_i	Number of Checkpoints for task T_i
R_i	Frame number of the task T_i
R_i^j	Represents the j^{th} occurrence of f^{th} frame of task T_i
T_F	Minimum inter arrival time between faults
$WCRT$	Worst Case Recovery Time
$WCET$	Worst Case Execution Time

Chapter 3

Related Work

This chapter is divided into 7 sections. Section 3.1 defines what a uniprocessor and multiprocessor systems are. Section 3.2 talks about schedulers used for scheduling tasks. Section 3.3 defines the preemptive and non–preemptive scheduling. Section 3.4 defines what a static schedule is and also shows various static scheduling algorithms. Section 3.5 defines dynamic scheduling and algorithms that follow dynamic scheduling. Section 3.6 illustrates how analysis of scheduling algorithms is done by various methods.

3.1 UniProcessor and MultiProcessor

Real-time systems usually contain several tasks that are to be executed. Depending on the number of processors being involved, real-time systems can be classified as uniprocessor and multiprocessor systems. A uniprocessor system is the one where all jobs share and execute on a single processor. A multiprocessor system is one where there are several processors available for the jobs to share and execute on.

3.2 Preemptive and Non–preemptive Schedulers

Real-time tasks can be scheduled using various scheduling algorithms. A schedule is said to be valid if it meets all time constraints and a task set is said to be A–schedulable if the scheduling algorithm A produces a valid schedule for that task set. Scheduling algorithms may either be preemptive or non-preemptive. A preemptive scheduler is the one which allows jobs to be interrupted midway through their execution and then resume from where they left off. A non– preemptive scheduler is one which executes jobs from start to finish making even urgent tasks wait.

3.3 Preemptive and Non–Preemptive Scheduling

Preemptive Scheduling: Is a priority driven scheduling where the tasks with highest priority tasks will currently be utilizing the processor. If a task is already executing and a higher priority task comes into the system, then the task with the lower priority is removed and is returned only after the higher priority tasks finish their execution.

For example, let us consider a task set described in Table 3.1. The schedule is an EDF schedule, which is a preemptive schedule and, deadlines = periods as shown in Figure 3.1.

Table 3.1: Preemptive Scheduling

Task Number(T)	Arrival Time(a)	Execution Time(E in ms)	Deadlines(D)
1	0	4	10
2	4	3	5

T_1 arrives at time 0. So, the CPU is allocated to the task T_1, as there are no other tasks in the system. T_2 arrives at time 4. At time=4, the job of task T_1 is

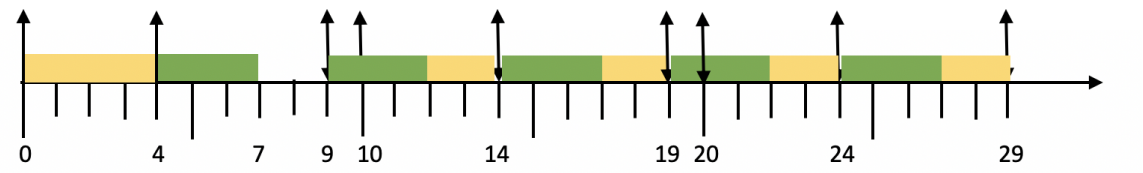


Figure 3.1: Illustration of Preemptive Scheduling

preempted by higher priority task T_2. Task T_2 starts to execute as it has the lowest deadline of $4 + 5 = 9$ time units. At 9 time units, another job of task T_2 is released and starts to execute until 12 time units. Even though task T_1 is released at time =10, it is preempted by the higher priority task T_2 and needs to wait until task T_2 has finished its execution.

Non-Preemptive Scheduling: In this kind of schedule, the task executing on the processor will execute completely without being interrupted. Since the processor does not switch from one task to another, there is no switching overhead. For example, let us consider a task set described in Table 3.1 and the schedule is as shown in Figure 3.2.

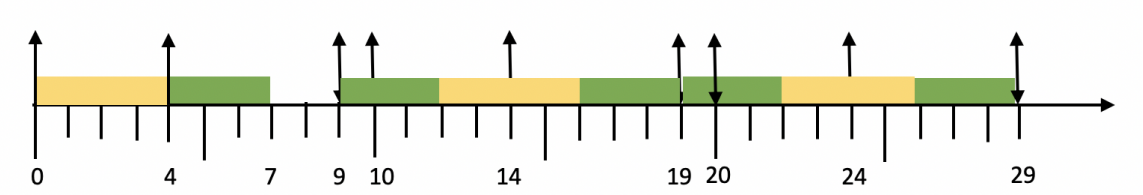


Figure 3.2: Illustration of Non–Preemptive Scheduling

T_1 arrives at time 0. So, the CPU is allocated to the task T_1, as there are no other tasks in the system. T_2 arrives at time 4. At time=4, the job of task T_1 finished executing and task T_2 starts to execute and finishes by time = 7. At time = 9, task T_2 is released and continues to execute until time = 13. Even though task T_1 released at time =10 it has to wait until task T_2 finishes its execution. Task T_1 starts

its execution at time =12 and continues to execute until time=16 and task T_2 keeps waiting until task T_1 finishes to execute even though it has been released at time = 14.

3.4 Static Scheduling

Static Scheduling or Priority Driven scheduling or Fixed Priority Scheduling [10] is a way of scheduling real time systems where enough information about deadlines, periods, maximum delay etc., are to be known beforehand to design the system for scheduling process. In such scheduling process, the priorities are assigned to all tasks and these priorities do not change with time. It is assumed that no two tasks have the same priority [11].

Example: Let us consider three tasks as in Table 3.2 which are denoted using the notation: $T_i = (E_i, P_i)$. We assume that deadlines are equal to periods and the following tasks priorities assigned are fixed. Figure 3.3 illustrates the preemptive schedule of these tasks.

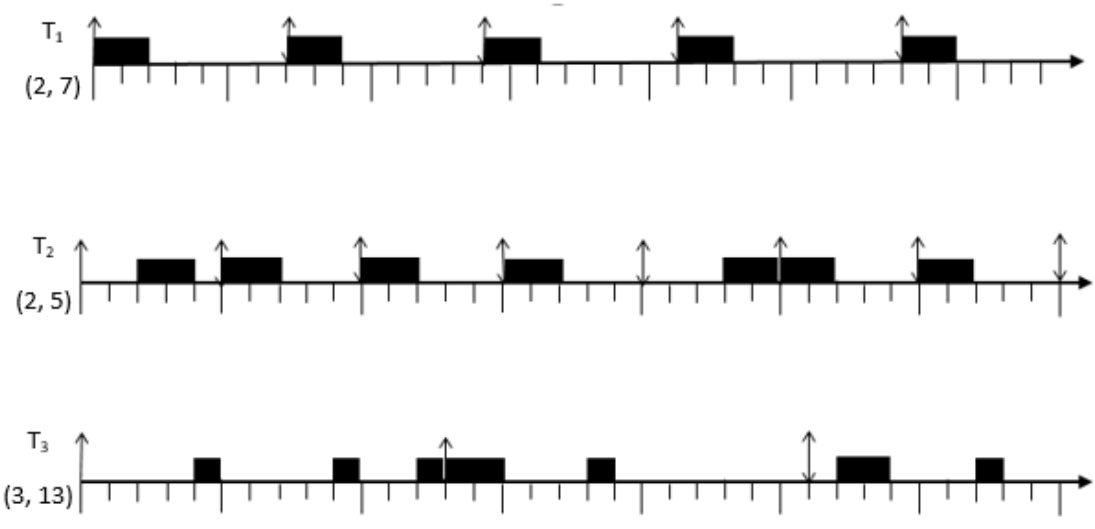


Figure 3.3: Illustration of Static Scheduling

Task T_1 has the highest priority with its execution time being 2 milliseconds. Next in priority is task T_2 with priority=2 and execution time 2 milliseconds. Last in line is task T_3 with priority=3 and execution time 3 milliseconds. Since task T_1 is the highest priority, it starts to execute as soon as it arrives in the system. Tasks T_2 and T_3 need to wait for task T_1 to finish before they can execute. Notice that the task T_3 is interrupted at time 5, when task T_2 releases its second job.

Table 3.2: Static Scheduling

Task Number(T)	Priority	ExecutionTime(E in ms)	Period(P)
1	1	2	7
2	2	2	5
3	3	3	13

Rate Monotonic (RM) Scheduling

This is an optimal static-priority scheduling algorithm [10] where priorities are assigned according to their periods. A task with a shorter period is of a higher priority than the task that has a larger period i.e., $T_i > T_j$ iff $P_i < P_j$ [10]. The task model assumed is set of periodic tasks (E_i, P_i) and $D_i = P_i$.

Example: Let us consider two tasks T_1 and T_2 as shown in Table 3.3 which have 2 milliseconds and 5 milliseconds respectively as their execution times. The periods for each are 5 and 13 respectively. Since task T_1 has a shorter period, it is of the highest priority among the two and the starts to execute as soon as it arrives into the system. Task T_2 must wait even if it enters the system until T_1 completes its execution. The schedule is shown in the Figure 3.4 below.

Table 3.3: Rate Monotonic Scheduling

Task Number(T)	ExecutionTime(E in ms)	Deadline(D)
1	1	5
2	5	13

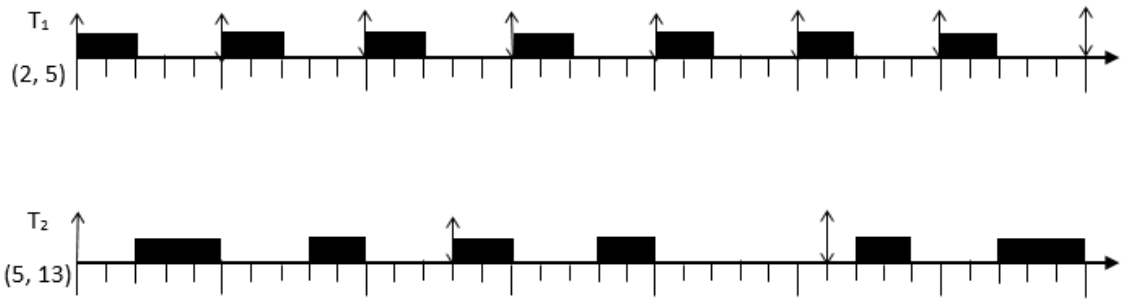


Figure 3.4: Illustration of Rate Monotonic Scheduling

Deadline Monotonic (DM) Scheduling

DM Scheduling is another kind of priority driven schedule where priority is given to tasks that have shorter deadlines [12]. The tasks with shorter deadlines are of a higher priority than the ones with longer deadlines. The task model assumed is set of periodic tasks (E_i, D_i, P_i) and $D_i \leq P_i$ [13].

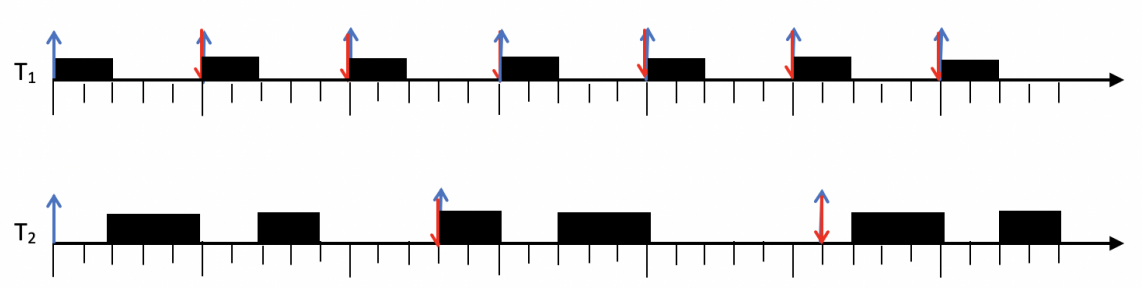


Figure 3.5: Illustration of Deadline Monotonic Scheduling

Example: Let us consider a task set as shown in the Table 3.3. Task T_1 has a deadline of 5 time units and task T_2 has a deadline of 13 time units. Since the

deadline for task T_1 is the lowest, it is of the highest priority among the two and starts to execute as soon as it arrives into the system. Task T_2 must wait even if it enters the system until T_1 completes its execution. The schedule is illustrated in Figure 3.5.

3.5 Dynamic Scheduling

Dynamic scheduling is a schedule where job arrivals are not necessarily known before execution. In these schedules, jobs' priorities are determined during execution.

Earliest Deadline First (EDF)

EDF [10] is an optimal scheduling algorithm on preemptive uniprocessors. This is a dynamic schedule where priorities are determined by which task has the earliest absolute deadline. The task with the earliest deadline is the highest priority hence the name earliest deadline first.

Example: Let us consider 3 tasks with the parameters as shown in Table 3.4. The tasks are scheduled as shown in Figure 3.6. Initially, task T_1 has the highest priority as its absolute deadline is the shortest when compared to the rest of the tasks in the system, therefore, it starts to execute right away. The next in queue is task T_2 as it has a deadline of 5-time units. It starts to execute as soon as T_1 completes its execution. The last to enter the system is task T_3 with a deadline of 7-time units and starts to execute after the higher priority tasks finish their execution. Notice that when task T_1 releases its second job at $t = 4$, it has to wait for task T_3 to complete executing because task T_3's job has an earlier deadline that task T_1's second job.

Table 3.4: EDF Scheduling

Task Number(T)	ExecutionTime(E in ms)	Deadline (D)
1	1	4
2	2	5
3	1	7

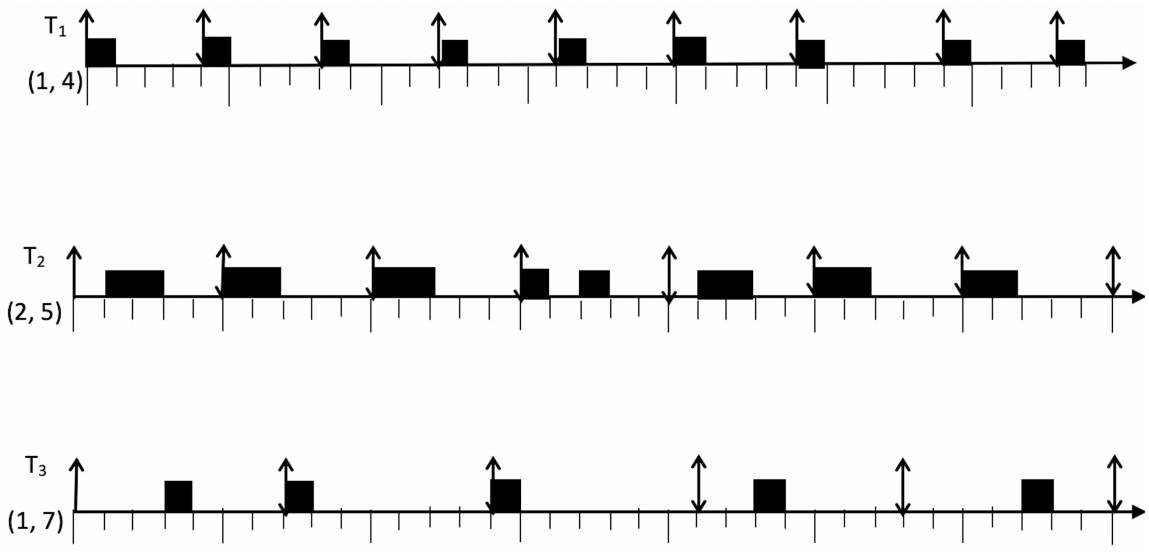


Figure 3.6: Illustration of Earliest Deadline First

Least Laxity First (LLF)

Laxity is the amount of time a job can be in the wait queue without missing its deadline [10]. The laxity of a real–time task T_i at time t, $L_i(t)$, is defined as follows: $L_i(t) = D_i(t) - E_i(t)$ where $D_i(t)$ is the absolute deadline of the task at time t and $E_i(t)$ is the amount of computation remaining to be complete [14]. The LLF scheduling algorithm assigns higher priority to the tasks with the least laxity.

Example: Let us consider a task set as show in Table 3.5 and the LLF schedule illustrated in 3.7.

At time instant 0 i.e., t=0 for T_1: $L_1(0) = 8 - 2 = 6$ At $t = 0$ for T_2: $L_2(0) =$

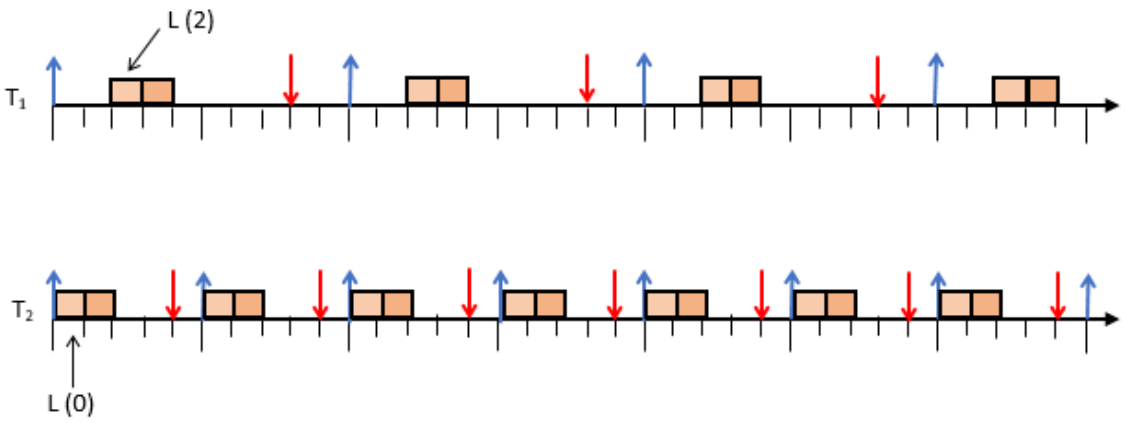


Figure 3.7: Illustration of Least Laxity First

Table 3.5: LLF Scheduling

Task Number(T)	ExecutionTime(E in ms)	Deadline (D)	Period(P)
1	2	8	10
2	2	4	5

$4 - 2 = 2$ Since laxity for task T_2 is the least out of the two, T_2 starts to execute for 1 time units. At time instant 1 i.e., t=1 for T_1: $L_1(1) = 7 - 2 = 5$ At $t = 0$ for T_2: $L_2(1) = 3 - 1 = 2$ Again, laxity for task T_2 is the least out of the two, T_2 executes for 1 time units. At time instant 2 i.e., $t = 2$ for T_1: $L_1(2) = 6 - 2 = 4$ Since T_2 finished executing within its period, T_1 can start to execute.

3.6 Scheduling Analysis

Scheduling algorithms are used by real–time systems to determine the order in which the tasks are executed such that no task or minimum number of tasks do not miss their deadlines. A *schedulability test* decides whether a task can be scheduled such that no other tasks in the system miss their deadlines [15]. In real–time scheduling, a *scheduling analysis* is performed to ensure the system can be schedulable and tested

by schedulability tests.

Time demand Analysis (TDA)

The TDA determines the worst–case response times of task sets scheduled by a fixed priority algorithm [16] assuming tasks $T_1, \ldots, T_n$ are indexed according to priority and all tasks have $D_i \leq P_i$. TDA acts as a check for the tasks schedule and make sure the tasks meet their deadlines.

Let $w_i(t)$ denote the worst case $level - i$ 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 $w_{i(t)} = \sum_{j=0}^{n}(E_j.\lceil \frac{t}{T_j} \rceil)$ gives the cumulative demands on the processor made $T_1, T_2, \ldots, T_n$ by tasks over $[0, t]$ [16].

Time demand Analysis can be found by iteratively computing the function. The result will be T_i's worst case response time. The computation is as follows:
For any task T_i,

1. $t^{(k+1)} = w_i(t^{(k)})$, where $t^{(0)}$ is equal E_i
2. Find $t^{(k)}$ iteratively until either
   - $t^{(k)} > D_i$ (system is infeasible), or
   - $t^{(k+1)} = t^{(k)} \leq D_i$ (T_i can meet its deadline)
3. Repeat for each task $T_i, i = 1, 2, \ldots, n$.

Let $\tau_i = T_1, T_2, T_3$ where $T_i = (P_i, E_i)$. $T_1 = (3, 1)$, $T_2 = (4, 2)$, $T_3 = (9, 1)$. The Rate Monotonic schedule of the following tasks are as shown below in Figure 3.8. Say, we want to check if task T_3 meets its deadline or not. The WCRT computation of task

T_3 is as follows:

$$t^{(0)} = E_3 = 1$$

$$t^{(1)} = w_3(t^{(0)}) = E_3 + \lceil \frac{t^{(0)}}{P_1} \rceil . E_1 + \lceil \frac{t^{(0)}}{P_2} \rceil . E_2$$

$$t^{(1)} = 1 + \lceil \frac{1}{3} \rceil .1 + \lceil \frac{1}{4} \rceil .2$$

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

$$t^{(1)} = w_3(2) = 4$$

Now, $t = 4$

$$w_3(4) = 1 + \lceil \frac{4}{3} \rceil .1 + \lceil \frac{4}{4} \rceil .2$$

$$w_3(4) = 1 + 2 + 2$$

$$w_3(4) = 5$$

Now, $t = 5$

$$w_3(5) = 1 + \lceil \frac{5}{3} \rceil .1 + \lceil \frac{5}{4} \rceil .2$$

$$w_3(5) = 1 + 2 + 4$$

$$w_3(5) = 7$$

Now, $t = 7$

$$w_3(7) = 1 + \lceil \frac{7}{3} \rceil .1 + \lceil \frac{7}{4} \rceil .2$$

$$w_3(7) = 1 + 3 + 4$$

$$w_3(7) = 8$$

Now, $t = 8$

$$w_3(8) = 1 + \lceil \frac{8}{3} \rceil .1 + \lceil \frac{8}{4} \rceil .2$$

$$w_3(8) = 1 + 3 + 4$$

$$w_3(8) = 8$$

Therefore, $WCRT(T_3) = 8$ Since $8 < 9$, we can say that task T_3 meets its deadline. The same computation is to be followed to find out the WCRT for tasks T_1 and T_2. If P_3 was any less than 8 then the task would miss its deadline. Therefore, by using TDA, we can make sure all the tasks in the system can meet their deadlines.

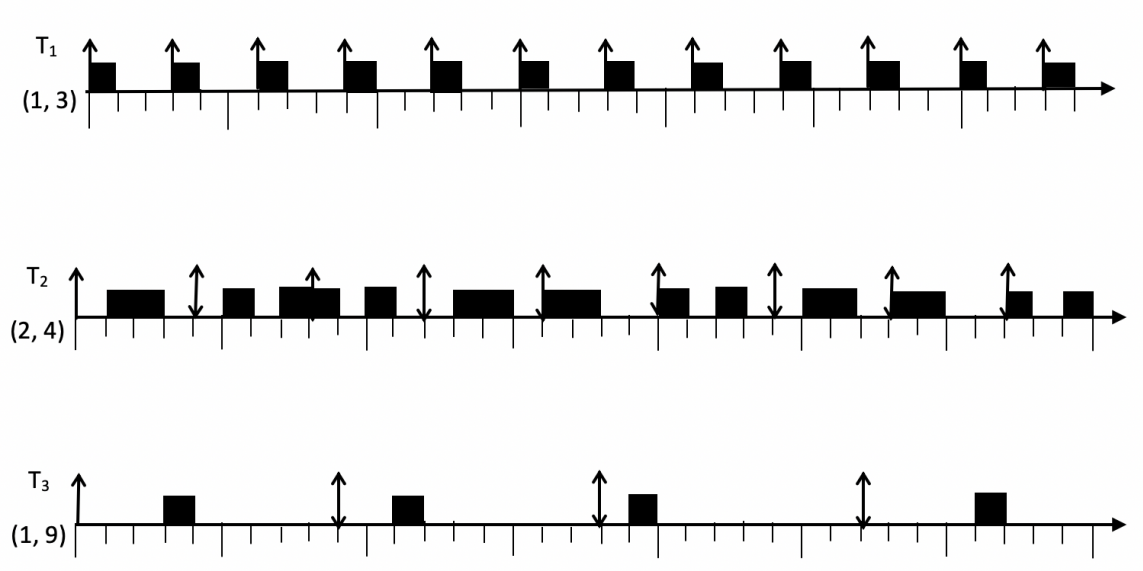


Figure 3.8: Illustration of Time-Demand Analysis

Buildlist Algorithm

Buildlist algorithm is used to find the demand function for GMF tasks. It is a lookup list for the demand bound function over a period of interval, that is, $dbf(T, t)$ where t is small interval. The buildlist algorithm runs in time $O(N^2 \log N)$ and produces a list of sorted pairs of (workload, interval) for a given interval t as stated in [2]. The

$dbf(T,t)$ can be determined from this list in $O(logN)$ time, using binary search.

For example: Let us consider two task sets as show in the Table 3.6. The buildlist algorithm is shown in 3.7.

Table 3.6: Buildlist Algorithm

Task 1 $T_1 = [13,5],[20,27],[53,42]$
Task 2 $T_2 = [11,14],[35,34],[62,77]$

Algorithm build-list

1. Generate ordered pairs ("workload", "interval size") as follows:
   For $i \leftarrow 0$ to $N-1$, do
   For $j \leftarrow 0$ to $N-1$ do
   generate the ordered pair $(e(R_i)+e(R_{i+1})+\cdots+e(R_{i+j}),\quad d(R_{i+j})-a(R_i))$.
2. Sort the ordered pairs into an array in increasing order of interval size (within interval size, in *decreasing* order of workload).
3. Delete all those ordered pairs whose workloads are not strictly larger than the workloads of all ordered pairs occurring prior to them in the sorted array.

Figure 3.9: Buildlist Algorithm

The algorithm when run for Task 1:

Table 3.7: Task 1 Buildlist Algorithm

i/j	j=0	j=1
i=0	$e(R_0), d(R_1)-a(R_0)=(13,80-0)=(13,80)$	$(R_1), d(R_1)-a(R_1)=(5,80-53)=(5,27)$
i=1	$e(R_0)+e(R_1), d(R_1)-a(R_0)=(13+5,80-0)=(18,80)$	$e(R_1)+e(R_2), d(R_2)-a(R_1)=(5+13,115-53)=(18,62)$

Here, N = total number of frames of each task. Therefore, $i=2$ and $j=2$ and $e(Ri)$ denotes the execution time of the frame and $d(R_{i+j})$ is the deadline of frame$(i+j)$ and $a(R_i)$ is the arrival time of frame i. The algorithm when run for Task 2:

Table 3.8: Task 2 Buildlist Algorithm

i / j	j=0	j=1
i=0	$e(R_0), d(R_1) - a(R_0) = (11, 96 - 0) = (11, 96)$	$(R_1), d(R_1) - a(R_1) = (14, 96 - 62) = (14, 34))$
i=1	$e(R_0) + e(R_1), d(R_1) - a(R_0) = (11 + 14, 96 - 0) = (25, 96)$	$e(R_1) + e(R_2), d(R_2) - a(R_1) = (14 + 11, 174 - 62) = (25, 112)$

1. Sort pairs in increasing order of interval. Within interval, decreasing order of workloads.

   Task 1: (5,27), (18,62), (18,80), (13,80)

   Task 2: (14,34), (25,96), (11,96), (25,112)

2. Delete pairs where workload > workloads prior to them

   Task 1: (5,27), ~~(18,62)~~, (18,80), ~~(13,80)~~

   Task 2: (14,34), (25,96), ~~(11,96), (25,112)~~

The final pairs are:

Task 1: (5,27), (18,62)

Task 2: (14,34), (25,96).

The total demand of the task set is show in Figure 3.10.

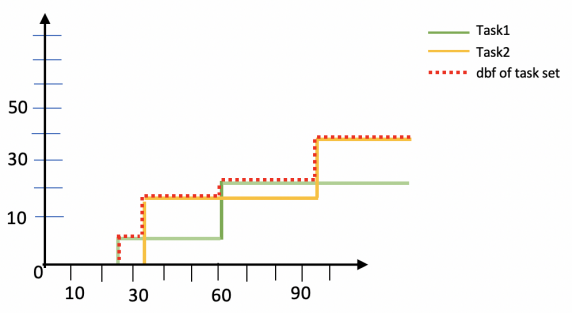


Figure 3.10: Total dbf of the task set

Demand Bound Function (dbf)

Demand Bound Function (dbf) was first introduced for periodic tasks for EDF scheduling. A demand bound function is defined as the maximum processor demand by workload for any interval t. Real–time system is schedulable under EDF if and only if $dbf(t) \leq t$ for all interval t [17]. The demand function for a task T_i is a function of an interval $[t_1, t_2]$ that gives the amount of computation time that must be completed in $[t_1, t_2]$ for T_i to be schedulable. *dbf* for a task is defined as,

$$dbf(t_1, t_2) = \sum_{t_1 \geq a_i, t_2 \leq D_i} E_i$$

where, i is the task number.

dbf is used to check the feasibility of the GMF tasks without any resources. A demand bound function $dbf(T, t)$ represents the total workload of jobs generated by task T, that are both released and have deadlines within the time interval of length t where t is a positive real number [18]. This $dbf(T, t)$ is a non-decreasing step function.

For example: Consider the following task sets $T_1 = (1, 2), T_2 = (2, 4), T_3 = (1, 7)$.

Let the time interval $t = 6$, dbf is illustrated as in Figure 3.12. Then the $dbf(T, 6) = 1 * (E_1) + 1 * (E_2)$

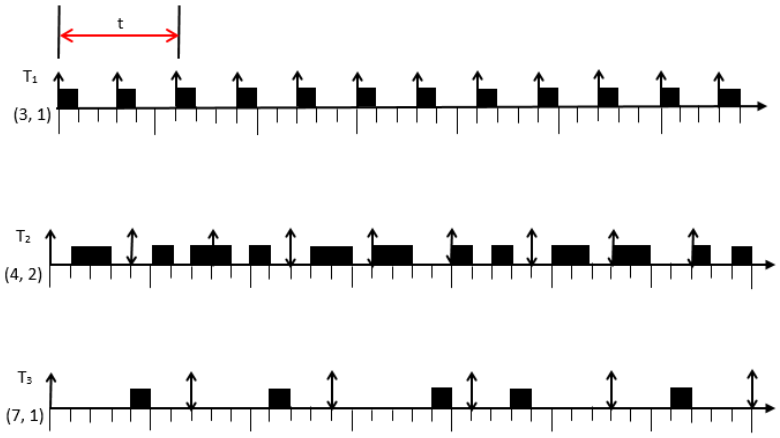


Figure 3.11: Demand Bound Function(dbf)

$dbf(T, 6) = 1 + 2$

$dbf(T, 6) = 3$

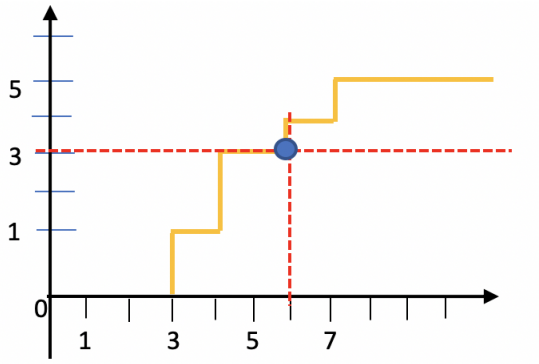


Figure 3.12: Demand Bound Function(dbf)

3.7 Analysis of Checkpointing

Checkpointing is an important mechanism to increase the reliability of real–time systems. Whenever a transient fault occurs, such as crash, power failure or a software

fault, systems information may be lost or corrupted. Checkpointing mechanism saves the system state and, in case of any faults, the mechanism can retrieve and restore the most recent saved state of the system and execution can be resumed normally.

This is how we deal with tasks with faults:

1. Let us consider a set of n tasks, $\{T_1, T_2, \ldots, T_n\}$ in which tasks are ordered according to the assigned priorities with minimum inter-arrival time P_i, worst case execution time (WCET) E_i and deadline D_i and $D_i \leq P_i$.

2. Each task T may be blocked by lower priority tasks for at most B_i time units.

3. Let O_i be the computation time overhead necessary to establish one checkpoint in task T_i. In fact the overhead consists of two parts, i.e, $O_i = O_i^A + O_i^S$, where O_i^A is the overhead for performing acceptance test and O_i^S is the overhead for saving the system state at each checkpoint.

4. Let us consider a task set with only one task, say T_1. Suppose, the task T_1 is divided into m_1 equal segments where E_1 is divisible by m_1 and perform a checkpoint at the end of each segment. If we assume that there can be at most one fault during T_1's execution then the worst-case execution time inclusive of checkpoint overhead and recovery, denoted by $E_1^{m_1}$, is given by:

$$E_1^{m_1} = E_1 + (m_1 - 1)O_1 + \frac{E_1}{m_1} + O_1$$

$$= E_1 + m_1 O_1 + \frac{E_1}{m_1}$$

Note that at the beginning of the task execution we need to perform a checkpoint (without an acceptance test) and at the end of the task execution we need to perform an acceptance test (without checkpoint). Hence, by combining this

two and represent it as the last O_1 in previous equation, the execution time in effect becomes,

$$E_1 + (m_1 - 1)O_1$$

and the fault recovery time becomes $\frac{E_1}{m_1} + O_1$.

Let us assume a task $T = (E, D) = (6, 10)$ where, execution time = 6 time units, deadline = 10 time units as seen in Figure 3.13(a) and *deadline* = *period*. Let us assume that number of checkpoint $m = 4$ and overhead for each checkpoint $O_i = 1$ as shown in Figure 3.13(b) .

Let us assume that task T faults at time $t = 4$ time units seen in Figure 3.13(c). This fault is discovered when the task reaches the checkpoint is at $t = 6$.

The worst–case execution time inclusive of checkpoint overhead and recovery, denoted by $E_1^{m_1}$, is given by,

$$E_1^{m_1} = E_1^{m_1} + (m-1)O_1 + \frac{E_1^{m_1}}{m_1} + O_1$$

$$= E_1^{m_1} + (m-1)O_1 + \frac{E_1^{m_1}}{m_1}$$

Therefore, the execution time in effect for the task changes to

$$E = E + (m-1) * O_i$$

$$E = 6 + (2-1) * 1$$

$$E = 7$$

Fault recovery time = $\frac{E}{m} + O_i$, where $\frac{E}{m} \in \mathbb{Z}$ is the interval size = $(\frac{6}{3}) + 1 = 2 + 1 = 3$ Therefore, it rolls back to the latest checkpoint which is at time $t = 3$ time units and

re–starts its execution with execution time $7 + 3 = 10$ time units shown in Figure 3.13(d).

Since each checkpoint has an overhead, checkpointing becomes a bounding problem as the mechanism can only be applied for task sets whose computation times (i.e., execution times) are greater than the checkpoint overheads. If the execution times are below the overhead, then simply re-execution of the whole task makes it more feasible than insertion of checkpoints.

In the above scenario, we have considered a single task. But, if the computation is to be done for the whole task set, then the response times would be:

$$R_i = C_i + (m_i - 1)O_i + B_i + \sum_{(j \in hp(i)} \lceil \frac{R_i}{T_j} \rceil (C_j + (m_j - 1)O_j) + \lceil \frac{R_j}{T_F} \rceil \max_{hp(i) \cup (i)} (\lceil \frac{E_j}{m_j} + O_j \rceil) \tag{3.1}$$

where: $\lceil \frac{R_i}{T_F} \rceil$ is the number of faults and T_F is the minimum inter arrival time between faults and $\max_{j \in hpi \cup i}(\lceil \frac{E_j}{m_j}) + O_j \rceil)$ is the worst case computation requirement upon a fault.

3.8 Bruteforce Algorithm

The brute force algorithm, also known as exhaustive search or generate and test, is a basic problem– solving methodology where in all possible combinations are checked to see whether they satisfy for the solution. Logically, brute force search is the simplest way to check whether all the possibilities lead to a solution. But, the cost of this methodology is proportional to the number of inputs used for the solution. Therefore, for a practical problem, using this ideology could require combinational runtime. Because of this nature, the brute force algorithms are used when the problem size is limited or when the set of candidate solutions are a manageable size. The main

disadvantage for bruteforce is while tackling real-world problems where the number of inputs are very large.

A classic example for bruteforce algorithm is the problem of a traveling salesman problem (TSP) where a salesman needs to determine the order in which he visits the cities to minimize the total distance traversed. This is a NP-hard problem whose solution size increases exponentially with the increase in the number of cities.

Example: Let us consider that the salesman needs to travel all cities, starting and ending with his home city. The distance between each city is as illustrated in Figure 3.14 . The salesman needs to find the shortest route of travel. The four nodes of the graph namely A, B, C and D represent the four cities and the edges between then represent the distance between each city. The salesman needs to start at the node or vertex A and end at the same node.

There are three different paths that can be traversed.

$A \to B \to C \to D \to A = 16$

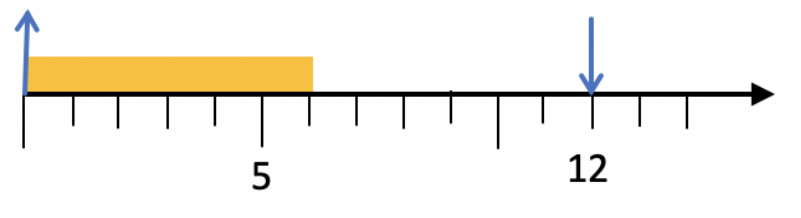
$A \to B \to D \to C \to A = 22$

$A \to C \to B \to D \to A = 24$

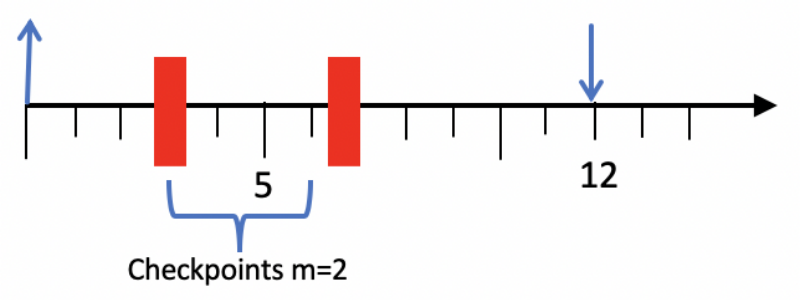
Therefore, the best distance path with the minimum distance to travel is $A \to B \to C \to D \to E \to A$, of value 16.

Increasing to just 10 cities would result in having to consider 181440 paths, and 20 cities would require more than 6×10^{16} paths.

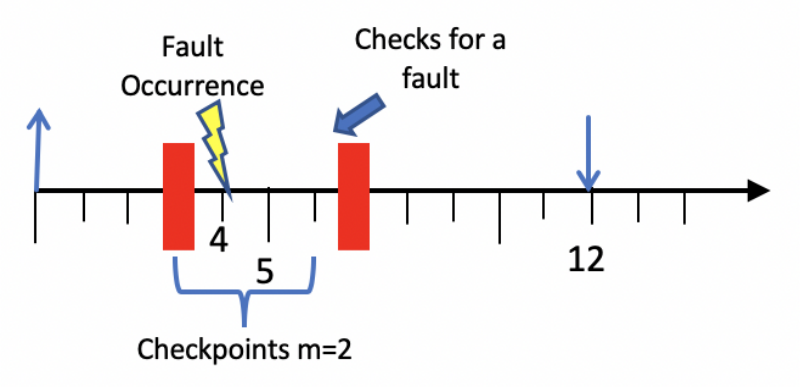
This research uses brute force method in finding schedules for a comparison to the algorithm presented in Chapter 4.



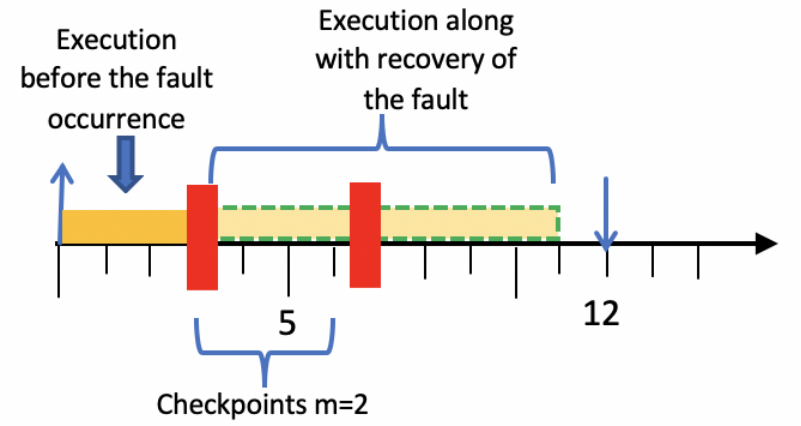
(a) Normal Execution of the Task.



(b) Checkpoints.



(c) Fault Occurrence and Fault Discovery



(d) Re-execution of the task with fault tolerance

Figure 3.13: Checkpointing and Recovery of a task.

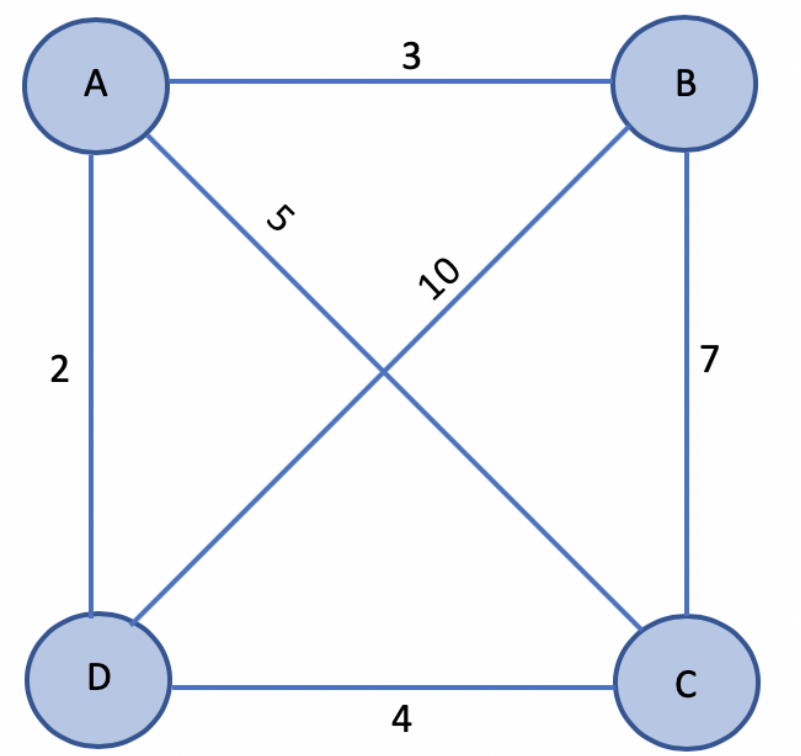


Figure 3.14: Bruteforce Algorithm for TSP

Chapter 4

Generalized Multi–frame Model with Fault tolerance

While fault tolerance algorithms have been developed for periodic and sporadic tasks [17], there have been no fault tolerance algorithms developed for GMF tasks. For this work, we need to allocate a recovery time to each frame of the task.

This work determines a worst–case recovery sequence for GMF tasks. We use this recovery sequence to compute a modified demand bound function that in co-operates recovery demand. The new GMF with checkpointing task model will be represented by a 4–tuple notation $(\vec{E}, \vec{D}, \vec{P}, \vec{R})$ where $\vec{E}, \vec{D}, \vec{P}$ and r_i are N–array vectors $[E_0; E_1 \ldots, E_{N-1}]$ of execution requirements, $[D_0; D_1 \ldots, D_{N-1}]$ of (relative) deadlines, $[P_0; P_1 \ldots, P_{N-1}]$ of minimum separations and $[R_0; R_1 \ldots, R_{N-1}]$ of recovery times, respectively. The interpretation is as follows: The i^{th} frame of task T has an arrival time a_i, a deadline d_i , an execution requirement of e_i, and a recovery time of r_i where

- $a_0 \geq 0$ and $a_{i+1} \geq a_i + P_{i \mod N}$
- $d_i = D_{i \mod N}$,

- $e_i = E_{i \mod N}$ and
- $r_i = R_{i \mod N}$

where r_i is the maximum recovery time for the task frame.

4.1 This Research

Our focus in this paper is to determine if a generalized multiframe (GMF) tasks are EDF–schedulable on a uniprocessor in the presence of faults. That is, given a system of GMF tasks, we can determine if the tasks can be scheduled in a way that all of them meet their deadlines even upon occurrence of faults during their execution on a uniprocessor using the EDF scheduling algorithm. The schedulability of the tasks are determined by the *buildlist algorithm* and in combination with *demand bound function.*

4.2 General Framework

In this section, we define the abstract task model used in the research and determine that the task mode is schedulable with fault tolerance built-in.

Task Independence Assumptions

- Assumption 1: The run-time of one task does not depend on other tasks executing in the system. Each task is independent and driven by different external factors. It is prohibited for a task to generate a job in response to another job.
- Assumption 2: There are no references of workload constraints to absolute time. That is, there is no compulsion that task T generates a job at time–instance 5.

- Assumption 3: The frame with a higher recovery time will also have a higher deadline. This is to in–cooperate the fault tolerance mechanism.

This research does not hold for systems where all jobs are generated at the same time or where some jobs are guaranteed to release before other jobs.

Whereas, the research does hold in distributed systems where each task executes jobs on different resources. We can also see that this assumption is violated by periodic tasks systems where a task T with parameters execution time E, deadline D and period P are scheduled to release each job at exactly P time units apart over an interval of time. However, the research holds for sporadic task systems, where the period P is a minimum separation of time between the jobs generated by task T.

4.3 GMF with Fault Tolerance Algorithm

Let us consider the GMF tasks along with the recovery times scheduled with the EDF scheduling algorithm. We assume that T_F is the minimum inter–arrival time between two faults and ℓ is the length of the schedule. The recovery time of the frame is the extra time that is added to its execution time upon the occurrence of a fault. Unlike any set scheduling algorithm, we have built this algorithm to understand and produce the worst case recovery demand of the recovery pattern for a given GMF task set.

This algorithm accounts for recovery times by creating a new GMF task whose execution times are the worst–case pattern of recovery times. This task is added to the GMF task system and the buildlist analysis is performed as described in Chapter 4.

The new GMF task T_R is generated as follows:

Step 1: Calculate the number of fault intervals for the entirety of the schedule. This is

done by $\lceil \frac{\ell}{T_F} \rceil$.

Step 2: Identify the earliest T_F interval with no scheduled recovery time, $[k.T_F, (k+1)T_F]$

Step 3: Choose the unscheduled frame with the highest recovery time ($T_{i,j}$).

Step 4: Calculate the number of intervals that the frame can have a possibility of failure. Allocate the execution time of T_R to be r in the given frames. $m = \lceil \frac{(D_i - 1)}{T_F} \rceil + 1$ and allocate $r_{i,j}$ recovery demand to T_F intervals starting at time $k.T_F, \ldots, (K+m).T_F$.

Keep in mind, a frame can have only one fault in a single T_F interval.

Step 5: Check for next arrival of the frame. If the frame is from the same task T_i, then it can repeat only after it reaches the total period P_i of the task and $P_i \leq \ell$, assuming the frame begins at time $t = 0$. If the frame does not start at time 0, then the earliest time a frame can repeat is $k.T_F + P_i$ such that $(k.T_F + P_i) \leq \ell$ and go to Step 3.

Step 6: Find the next largest recovery time and go to Step 2.

Creation of new GMF task

After filling up all the T_F intervals, a new GMF taskset is created with change in few of the frame parameters.

- The execution time of the frame will be its recovery time.
- The relative deadline will change to $relativedeadline - (x.TF + recoverytime)$ where x is the number of intervals between the first filled interval of the frame and the last filled interval by the frame. If the difference is less than 0 or less than the $recoverytime$, then the $relativedeadline = \max(relativedeadline - (x.TF + recoverytime), recoverytime)$ of the frame.

- The period would be the T_F interval value.
- The recovery time would remain the same value as it was for the original frame.

For Example: Let us consider a GMF task set of 2 tasks–each containing 3 frames.

$T_1 = \{[8, 11, 5], [17, 19, 10], [40, 38, 42], [2, 3, 2]\}$

$T_2 = \{[2, 5, 5], [20, 19, 15], [20, 35, 20], [3, 2, 1]\}$

The schedule for GMF with fault tolerance is shown in Figure 4.1. The total period for task T_1 is 120, for task T_2 is 75.

The length of the schedule, ℓ is the highest period among all the task sets. Length of schedule, $\ell = 120$.

The minimum inter arrival time between faults, $T_F = 30$.

Number of Intervals = $\lceil \frac{120}{30} \rceil = 4$.

As there are two frames with the same recovery times, let us take the frames from T_1 which has the shortest deadline. Since, T_{12} has the highest recovery (of 3 time units) and deadline = 19, it can be placed in $\lceil \frac{(19-1)}{30} \rceil + 1 = 2$ intervals.

Frame T_{12} has a recovery of 3 in the second interval and a deadline of $19 - 3 = 16$ in the first interval. At time = 75, task T_2's frames are repeated since the total period for task T_2 is 75

The earliest time that T_2 can start is the end of the second interval since T_{12} is executing. This is because, even when the task's total period is 75 time units, we do not know when frame T_{12} actually starts to execute. The frame with the highest recovery from T_2 is T_{21} with recovery time of 3 time units

The number of time frame T_{21} can be repeated is $\lceil \frac{(20-1)}{30} \rceil + 1 = 2$ intervals. Frames T_{21} has a deadline of 3 in the last interval and a deadline of $20 - 3 = 17$ in its third interval. The ending intervals of the frames possible failure have a deadline which is

equal to their recovery time is because the worst case for a frame to have a fault is right at the beginning of the interval. When such situation occurs, the frame is to be executed right away.

The new GMF–FT task is $\{[3, 3, 3, 3], [16, 3, 17, 3], [30, 30, 30, 30], [3, 3, 3, 3]\}$.

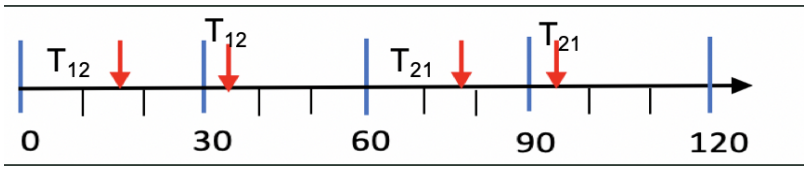


Figure 4.1: GMF with Fault Tolerance

4.4 Correctness of Generalized Multi-frame with Fault Tolerance (GMF–FT)

The correctness of the GMF–FT is proved by the theorem stating that the sequence of recovery times generated by GMF–FT algorithm causes a worst case recovery demand for the task set when compared with any other series.

First, we define a couple of terms.

Definition 1: Assume we are scheduling a GMF task set on a uniprocessor with fault tolerance. Let $S = (\rho_0, \rho_1, \ldots, \rho_k)$ be a sequence of recovery times. For each recovery ρ_i, let d_{ρ_i} be the absolute deadline of the frame that failed in the T_F interval $[i.kT_F, (i+1)T_F]$. Then the demand from recovery sequence S during the interval $[0, t]$ is $r_{dmd}(S, [0, t]) = \sum_{d_{\rho_i} \leq t} \rho_i$.

We now show that the task T_R, generated by the GMF–FT algorithm, provides

an upper bound of recovery demand.

Theorem: Let us consider two series of recovery demand S and $S^{'}$ for a GMF task scheduled with fault interval of T_F. $S = \{\rho_0, \rho_1, \ldots, \rho_{l-1}\}$ is a series of recovery times that is constructed from our algorithm GMF–FT.

Let $S^{'} = \{\rho_0^{'}, \rho_1^{'}, \ldots, \rho_\eta^{'}\}$ be any recovery possible series. Then, $\forall t \geq 0, r_{dmd}(S, [0, t]) \geq r_{dmd}(S^{'}, [0, t]))$.

Proof. We wish to show that the recovery sequence generated by our algorithm GMF–FT is the worst case recovery sequence and the demand of the sequence is either greater than or equal to any other recovery sequence for the given fault interval. We show this by a "swapping argument". We repeatedly swap the demand of the series $S^{'}$ each time making it closer to the demand of S. After each swap, we show that the recovery demand cannot decrease.

For each recovery ρ_k, let $Q(\rho_k) = T_{i,k}$ the frame that failed, causing the recovery demand of ρ_i. Initially, $S^{''} = S^{'}$. Repeat the following until $S^{''} = S$.

If $\rho_k = \rho_k^{''}$ for all $k = 0, \ldots, l-1$ we have shown $r_{dmd}(S, [0, t]) \geq r_{dmd}(S^{'}, [0, t]) \forall t$, which completed the proof.

If $Q(\rho_i) = Q(\rho_i^{''})$ for $i = 0, 1, \ldots, \eta$ then $S = S^{''}$ and the theorem is proved.
Otherwise, let $k = \min_i\{Q(\rho_i) \neq Q(\rho_i^{''})\}$.
Let r_i and $r_i^{''}$ be the recovery times of the frames $Q(\rho_i)$ and $Q(\rho_i^{''})$, respectively. Then, $r_i \geq r_i^{''}$.

This follows directly from the way recovery times are allocated according to the GMF–FT algorithm, as GMF–FT allocates recovery times from the longest to shortest.

We modify $S^{''}$ where initially $S^{''} = S^{'}$, as follows:

- If $Q(\rho_i)$ is in the series $S^{''}$, shift it forward in the series to the k^{th} spot and shift $\rho_k^{''}, \ldots, \rho_\eta$ backwards. This effectively means the associated tasks are arriving

later which is permitted by the sporadic task model.

- If $Q(\rho_i)$ is not in the series $S^{''}$, replace $Q_i^{''}$ with ρ_i.

In both cases, the demand of $S^{''}$ which is initially equal to $S^{'}$, cannot decrease because the demand of $\rho_i^{''}$ is replaced with the larger demand of ρ_i.

Note: On the i^{th} step, $r_0, \ldots, r_{l-1}$ and $r_0^{''}, \ldots, r_{l-1}^{''}$ are unchanged. Therefore, the theorem holds for all $i = 0, \ldots, l-1$ after completing all steps of the iteration. $\square$

In conclusion, this chapter introduces the new Generalized Multi–Frame with Fault Tolerance (GMF–FT) algorithm and the creation of a new GMF task set. The task set created by the algorithm is a pessimistic task set as we are greedily choosing the highest recovery frames to be executed at the beginning (i.e., with artificially early deadlines) so as to have a higher demand at the beginning of time rather than having it later causing the remaining tasks to miss their deadlines. Furthermore, when the pessimistic recovery task is combined with the buildlist demand, the recovery demand might not correspond to the tasks that are contributing to the worst–case demand without recovery.

Chapter 5

Experiments and Results

This chapter is divided into 6 sections. Section 5.1 deals with the experimental setup of the obtained results, section 5.2 shows the comparison of the average total demand for bruteforce and GMF–FT algorithms for 2 tasks and 2 frames and 3 tasks and 2 frames. Section 5.3 represents the results comparing the average recovery time over fault interval ranges for both the algorithms for 2 tasks and 2 frames and 3 tasks and 2 frames. Sections 5.4 shows the comparison of number of task sets that are feasible and infeasible for the two algorithms. Section 5.5 compares the runtimes for bruteforce and GMF–FT algorithms. Lastly, Section 5.6 deals with experimental results only for GMF–FT algorithm for larger task sets.

5.1 Experimental Setup

The following values are used in experimental setup:

- Processor: All the experiments were performed on uniprocessor platform. 7^{th} generation Intel core $i7$ processor was used.
- Tasks per task set: Number of GMF tasks used were 2, 3, 4 and 5, with 2 and

3 frames.

- Periods: The frame periods are randomly generated between 100 –500 time units.
- Execution times: The execution times are randomly generated to be an integer between $\frac{1}{10}th$ of the frame's period and $\frac{1}{3}rd$ of the frame's period.
- Deadlines: The frames' deadlines are randomly generated to be an integer between $\frac{1}{2}$ of the frame's period and the frame's period.
- Checkpoints: The number of checkpoints are generated randomly between 2 and 4.
- Recovery Times: The recovery times are calculated to be $\lceil \frac{executiontime}{num.of.checkpoints} \rceil$.
- feasibility: Task sets that fail Baruah's feasibility test [2] are discarded.

After the task sets are randomly generated, to ensure the feasibility of the task set, the buildlist algorithm is simulated as discussed in Section 3.4. The task sets that are determined infeasible that is, if their workload > interval until we find enough feasible tasks sets before adding recovery. From the feasible task sets we determine the length of the schedule for which the bruteforce and GMF–FT algorithms.

- Length of the schedule: This is the entire time interval for which the algorithms run for and is calculated to be highest total period of the task T_i from the task set τ.
- Fault Interval T_F: This is the minimum inter arrival time between two faults and is calculated to be $\lceil \frac{P}{f} \rceil$, where $P = \max_i P_i$ is the maximum total task period and f is the total number of frames of all tasks.

Bruteforce Algorithm Simulation: The bruteforce algorithm as mentioned in Section 3.6 is an exhaustive search. In this research, the bruteforce algorithm is used for computing all possible combinations of sporadic task arrivals within the system using the EDF schedule as the scheduling algorithm and the worst case recovery time scenarios for these arrivals because bruteforce takes so long to run, we can only run this algorithm on small task sets.

GMF–FT Simulation: The same task sets used for bruteforce, are used for simulating the GMF–FT algorithm mentioned in Section 4.3.

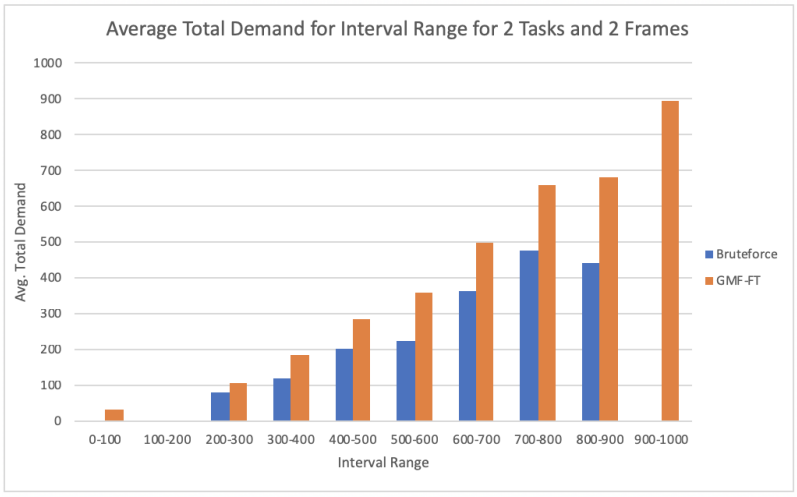
5.2 Average Total Demand for Bruteforce and GMF–FT Algorithms

In this simulation scenario, the task system is then run for various schedule lengths and different T_F intervals by both bruteforce and GMF–FT algorithms. The comparisons are illustrated in Figures 5.1(a) through 5.1(b) are used to compare the average of the total demand of feasible task sets consisting of 2 tasks and 2 frames and 3 tasks and 2 frames respectively.

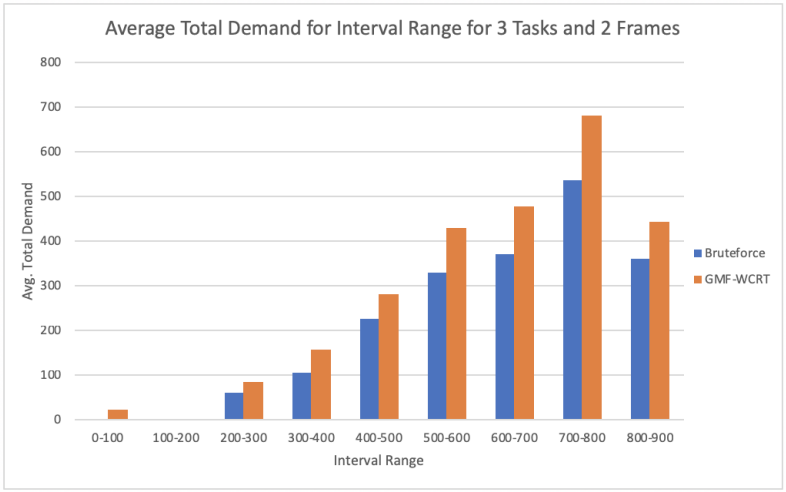
Experiments with 2 Tasks and 2 Frames

The total demand per interval are the resultant pairs of the buildlist algorithm as discussed in Section 3.4., is run for both bruteforce and GMF–FT algorithms. The $x - axis$ of the graph illustrated in Figure 5.1 represent the interval lengths and the $y - axis$ represent the average total demand.

The total demand of the GMF–FT is computed using Baruah's buildlist algorithm [2] with the additional recovery task computed as described in Chapter 4. The bruteforce algorithm computes demand directly using exhaustive search.



(a) Average Total Demand for 2 Tasks and 2 Frames



(b) Average Total Demand for 3 Tasks and 2 Frames

Figure 5.1: Average Total Demand for an Interval Range for Bruteforce and GMF–FT Algorithms.

We see that the demand computed for bruteforce is always less than the demand produced GMF–FT algorithm (as expected). The GMF–FT has a much higher demand as the the recovery demand is pessimistic, as discussed in Chapter 4. Only feasible task sets are shown in these figures. Therefore, both the algorithms produce demands that are less than the $x = y$ reference line as seen in Figure A 1.

In the Figure 5.1, we see that the two figures, Figure **??** and Figure 5.1(b) represent the average total demand values for intervals ranging from 0-1300. This is the demand from task execution and recover times.In Figure **??** we see a gradual increment of the average demand for GMF–FT as the interval ranges increase.

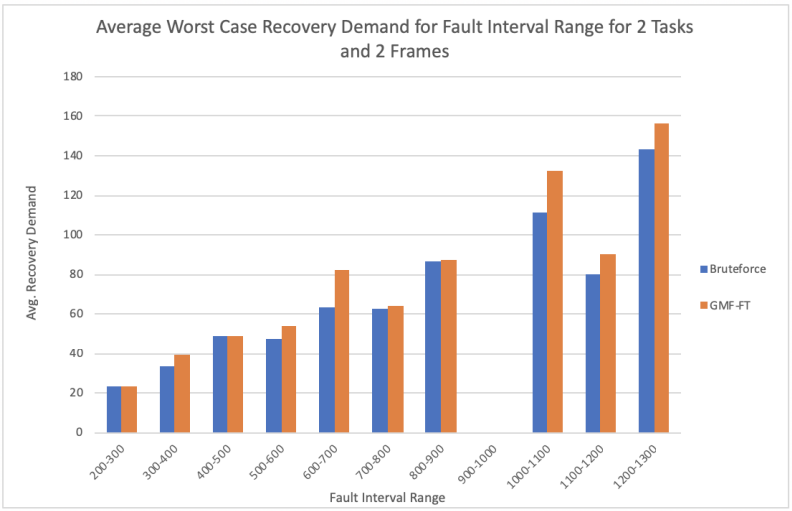
Experiments with 3 Tasks and 2 Frames

In this simulation scenario, the task system τ comprises of 3 tasks each with 2 frames as seen in Figure 5.1(b). The task system is then run for various schedule lengths and different T_F intervals by both bruteforce and GMF–FT algorithms as in Figure 5.1.

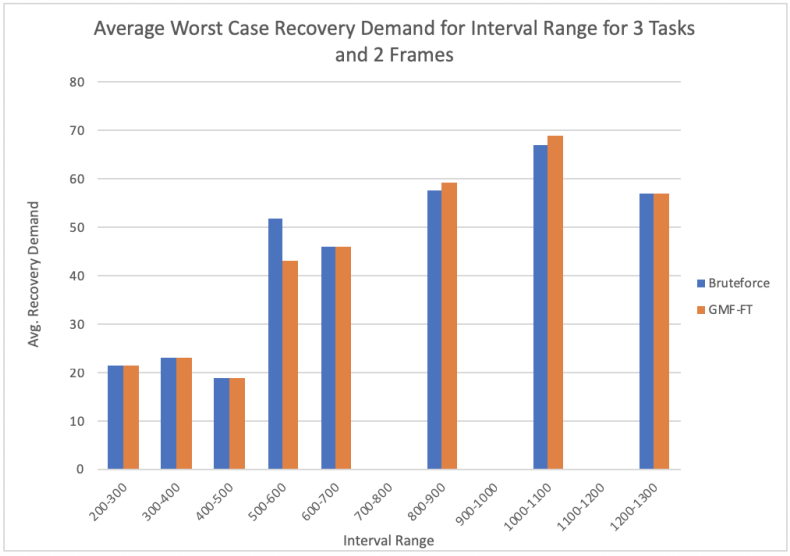
We see that in the interval range 800–900, the demand instead of increasing in fashion, goes down. This is because the values for the tasks are generated in random causing the demand to fluctuate. The values of the task set might not follow a certain fashion to keep up, with he increase in demand trend going on over all interval ranges.

5.3 Average Worst Case Recovery Demand for Bruteforce and GMF–FT algorithm

In this section we see graphs related to the averaged worst case recovery demand for an fault interval range. The task sets considered are the same as in Figure 5.1.
In this Figure 5.2, we see that the two figures, Figure 5.2(a) and Figure 5.2(b) representing the average worst case recovery demand values for intervals ranging from 200-1300. In Figure 5.2(a) we see a gradual increment of the average worst case recovery demand for GMF–FT as the interval ranges increase and the average worst case recovery demand does not fluctuate much for bruteforce for 2 tasks and 2 frames. We see that the worst case recovery demand computed for bruteforce and



(a) Average Worst case Recovery Demand for 2 Tasks and 2 Frames



(b) Average Worst case Recovery Demand for 3 Tasks and 2 Frames

Figure 5.2: Average Worst case Recovery Demand for an Interval Range for Bruteforce and GMF–FT Algorithms.

GMF–FT algorithms is almost equal for the fault intervals. As seen before, infeasible task sets are not included in these graphs.

Experiments with 2 Tasks and 2 Frames

In this simulation scenario, the task system τ comprises of 2 tasks each with 2 frames as seen in Figure 5.2(a). The task system is then run for various schedule lengths and different T_F intervals by both bruteforce and GMF–FT algorithms as in Figure 5.2.

Experiments with 3 Tasks and 2 Frames

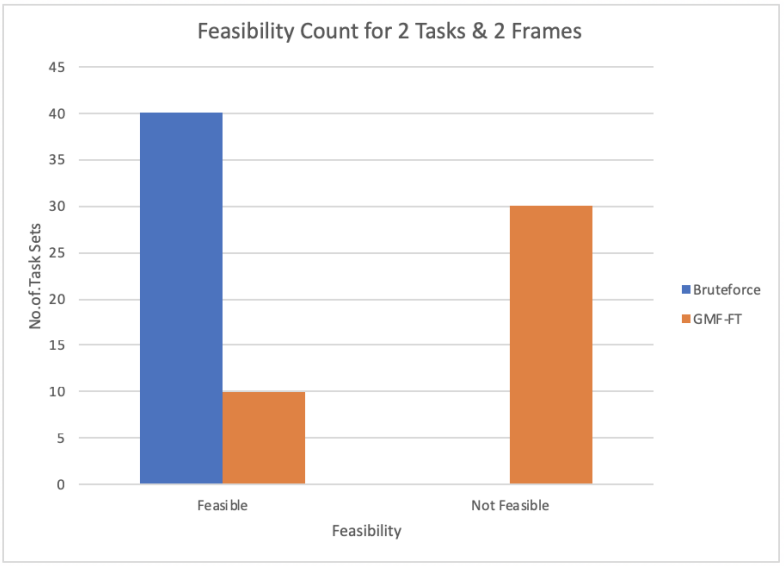
In this simulation scenario, the task system τ comprises of 3 tasks each with 2 frames as seen in Figure 5.2(b). The task system is then run for various schedule lengths and different T_F intervals by both bruteforce and GMF–FT algorithms as in Figure 5.1.

5.4 Comparison of Number of Task sets for Bruteforce and GMF–FT Algorithm

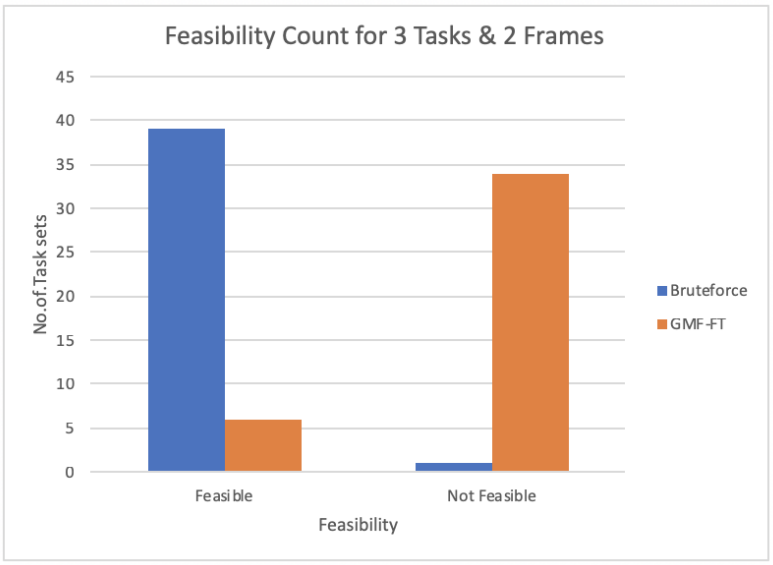
The $x - axis$ on the graphs illustrated in 5.3 represents the number of feasible and non–feasible task sets for both bruteforce and GMF–FT algorithms.

Count for 2 Tasks and 2 Frames

As we see in the Figure 5.3(a), out of 40 total task sets, all 40 are feasible for bruteforce and only 10 for GMF–FT. This is because, in GMF–FT, when the fault tolerance is added onto the total demand of the task set, there are tasks that might miss their deadlines. This results in a non–feasible taskset. Hence, there are 30 task sets that are not–feasible for GMF–FT.



(a) Task Sets count for 2 Tasks and 2 Frames



(b) Task Sets count for 3 Tasks and 2 Frames

Figure 5.3: Comparison of Task sets or Bruteforce and GMF–FT Algorithms.

Count for 3 Tasks and 2 Frames

As we see in the Figure 5.3(b), out of 40 total task sets, 39 are feasible for bruteforce and only 6 for GMF–FT. This is because, in GMF–FT, when the fault tolerance is

added onto the total demand of the interval, the tasks might miss their deadlines and this results in a non–feasible taskset. Hence, there are 34 task sets that are not–feasible for GMF–FT. Out of these 34 non–feasible tasks sets by GMF–FT, there is 1 such task set that is neither feasible by bruteforce nor by GMF–FT.

5.5 Comparison of Runtime Bruteforce and GMF–FT Algorithm

In this section, we compare the run times of the two algorithms. This comparison gives an idea of how the two algorithms run with increasing number of tasks and frames. We see that the bruteforce algorithm takes longer to run with increase in number of tasks and frames whereas, the GMF–FT algorithm takes much less time to run with the same number of tasks and frames.

Table 5.1: Comparison of Runtime Bruteforce and GMF–FT Algorithms

Number of Tasks, Number of Frames	Bruteforce Runtime	GMF–FT Runtime
2,2	45 seconds	30 second
3,2	3 hours	45 seconds
2,3	No heap space	42 seconds

As we can see from Table 5.1, the bruteforce algorithm's runtime keeps exponentially increasing with the increase in number of tasks and frames. For higher number of frames and tasks, the algorithm cannot even finish running without giving an"out of memory" or "out of heap space" error. On the other hand, the GMF–FT algorithm runs in within a minute for higher number of tasks and frames. Due to this reason solely, we could not run the bruteforce algorithms for more number of tasks and frames as it takes a long time ranging from hours to days to run. Hence, the next

section of the chapter shows experiments that are done only for GMF–FT algorithm since the runtime of the algorithm is much faster when compared to the bruteforce.

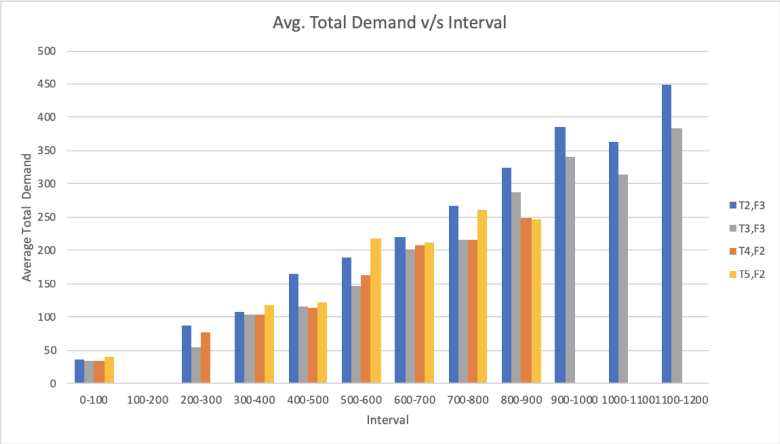
5.6 Experiments only for GMF-FT Algorithm

This section deals with experiments done only for GMF–FT Algorithm. There are different combinations of tasks and frames in this sections. In these experiments we see how the pessimistic GMF–FT behaves with increasing number of tasks and frames and analyse the total demand of the system as well the worst case recovery demand caused due to fault tolerance built in. The task sets considered in this section are too large for bruteforce algorithm.

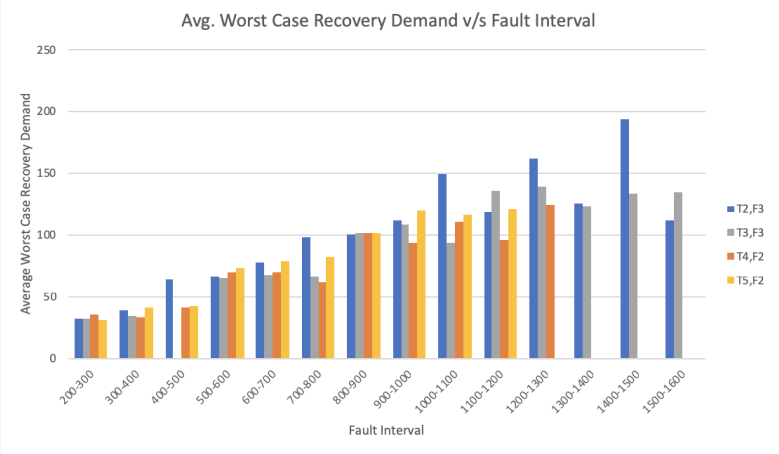
Average Total Demand and Average Worst case Recovery Demand for GMF–FT Algorithm

We also see that number of tasks is more significant than number of frames i.e., as number of tasks increase, total demand increases. From the figures Figure 5.4(a) and Figure 5.4(b) the task sets taken are: 2 tasks with 3 frames, 3 tasks with 3 frames, 4 tasks with 2 frames and 5 tasks with 2 frames. We can see that as the number of frames increase, their demands are present across increasing intervals and as we increase the number of tasks in the system, the demands only remain in the first few intervals and do not spread across the schedule length. But, in all cases, the demand gradually increase with increase in interval ranges.

The last scenario, with 5 tasks and 2 frames appears to have lower average demand , but that is due to the fact that almost all the task sets have failed the GMF–FT test as shown in the next section.



(a) Average Average Total Demand



(b) Average Worst Case Recovery Demand

Figure 5.4: Average Total Demand and Average Worst case Recovery Demand for GMF–FT Algorithms.

Comparison of Number of Feasible Tasks for GMF–FT

From the figures Figure 5.5 the task sets taken are: 2 tasks with 3 frames, 3 tasks with 3 frames, 4 tasks with 2 frames and 5 tasks with 2 frames and 30 task sets of each are considered. 5 tasks sets are feasible for 2 tasks with 3 and 4 tasks with 2 frames. 3 tasks sets are feasible for 3 tasks with 3 frames and 1 task set for 5 tasks with 2 frames.

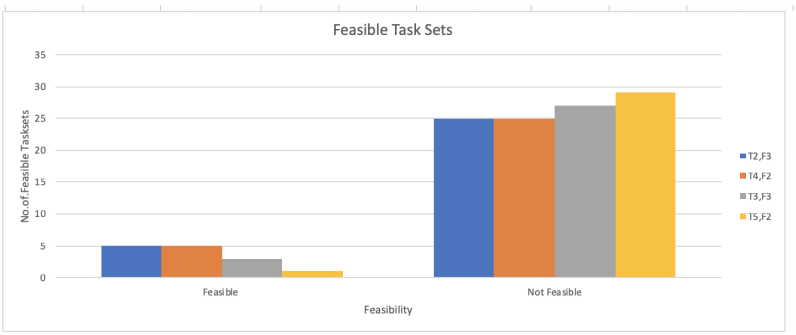


Figure 5.5: Comparison of Number of Feasible Tasks for GMF–FT.

We see that the pessimism of the GMF–FT algorithm increases as both number of tasks and number of frames increases.

Chapter 6

Conclusion and Future Work

This thesis unfolds a new task model that combines Generalized Multi–frame model and checkpointing fault tolerance analysis into one. This new model is called Generalized Multi–frame Model with Fault Tolerance. This model was achieved by converting the GMF task set into a 4–tuple task set with the addition of recovery times.

Experimental results demonstrate that this pessimistic approach when compared to the bruteforce algorithm does not under perform. The expectation of this new algorithm was to have the total demand of the task set inclusive of recovery to be higher than the demand of the task set without the recovery added to it and it did so. The worst case recovery demand was also greater when compare to the demand produced by bruteforce. Also, the number of infeasible task sets were comparatively less. Out of 80 task sets combined were used for comparisons between bruteforce and GMF–FT, 106 of them turned out to be infeasible by GMF–FT. Experiments that were run solely to analyze GMF–FT did not give establish a certain pattern to determine the behavior of the algorithm.

This thesis analytically proves that the GMF–FT algorithm computes a pessimistic approximation of the worst–case recovery demand. The experimental results

demonstrate that the level of pessimism is too extreme to be useful as the number of tasks and frames increase.

As for runtime, while the GMF–FT algorithm is a pessimistic approach for finding the worst case recovery sequence of fault patterns in a task set, it runs in polynomial time where as bruteforce runs in exponential time depending on the number of frames in the task set. The larger the task set, the time complexity exponentially increases for bruteforce whereas, for GMF–FT the time complexity doesn't depend on the number of frames but it depends on the number of fault intervals considered.

In future work, we hope to improve the accuracy of the GMF–FT algorithm by allowing it to find results less pessimistic for longer task sets while always finding an upper bound for the recovery demand. The creation of a recovery task approach has required to add a number of pessimistic assumptions causing large number of task sets to be infeasible. We suspect that there might be significantly another approach that can be taken in order to have more feasible task sets.

Incorporating fault tolerance into real–time systems is essential for ensuring proper operation of these systems. This research is a first step towards exploring this topic for GMF task sets.

REFERENCES

[1] A.K.Mok and D.Chen, "A multiframe model for real-time tasks," in *Real-Time Systems Symposium*, pp. 635 – 645, 1996.

[2] S. D. S.Gorinsky and A.K.Mok, "Generalized multiframe tasks," *The International Journal of Time-Critical Computing Systems*, vol. 17, no. 1, pp. 5 — 22, 1999.

[3] A. R.I.Davis and S.Punnekkat, "Feasibility analysis of fault-tolerant real-time task sets," in *Euromicro Real-Time Systems Workshop*, pp. 29 – 33, 1996.

[4] S. A.K.Mok and L.Rosier, "The preemptive scheduling of sporadic, real-time tasks on one processor," in *Real-Time Systems Symposium*, pp. 247–273, 1990.

[5] S. A.Burns and R.Davis, "Analysis of checkpointing for real-time systems," *Journal of Real Time Systems*, vol. 20, no. 1, pp. 83 — 102, 2001.

[6] A. J.L.Welch, "Efficient distributed recovery using message logging," in *Proceedings of the Eighth Annual ACM Symposium on Principles of Distributed Computing*, pp. 223 – 238, 1989.

[7] S.Hwang and C.Kesselman, "A flexible framework for fault tolerance in the grid," *Journal of Grid Computing*, vol. 1, pp. 251 —- 272, 2003.

[8] F. M. P. J. T. T. S.Oberthuer and K.Stahl, *Hardware-dependent Software*. Springer Netherlands, 2009.

[9] G.Miremadi and J.Torin, "Evaluating processor-behaviour and three error-detection mechanisms using physical fault injection," *IEEE Transactions on Reliability*, vol. 44, no. 3, pp. 441 – 454, 1995.

[10] C.L.Liu and J.Layland, "Scheduling algorithms for multiprogramming in a hard real-time environment," *Journal of ACM*, vol. 20, no. 1, pp. 46 – 61, 1973.

[11] B. A. S. Baruah and J. Jonsson, "Static–priority scheduling on multiprocessors," in *Proceedings 22nd IEEE Real-Time Systems Symposium*, pp. 193 – 202, 2001.

[12] N. Audsley, A. Burns, M. F. Richardson, and A. J. Wellings, "Hard real-time scheduling: The deadline-monotonic approach," in *Proceedings in IEEE Workshop on Real-Time Operating Systems and Software*, pp. 133–137, 1991.

[13] J. L. J.Whitehead, "On complexity of fixed-priority scheduling of periodic real-time tasks," vol. 2, no. 4, pp. 237 – 250, 1982.

[14] S.H.Oh and S. Yang, "A modified least-laxity-first scheduling algorithm for real-time tasks," in *Proceedings of the 5th International Conference on Real-Time Computing Systems and Applications*, pp. 31 – 36, 1998.

[15] J. Singh and S. P. Singh, "Schedulability test for soft real-time systems under multiprocessor environment by using an earliest deadline first scheduling algorithm," vol. arXiv:1205.0124 [cs.OS], 2012.

[16] J. L.Sha and Y.Ding, "Rate monotonic scheduling algorithm: Exact characterization and average case behavior," in *Real-Time Systems Symposium*, pp. 166 – 171, 1989.

[17] S. Baruah, "Dynamic- and static-priority scheduling of recurring real-time tasks," *Real-Time Systems*, vol. 24, no. 1, pp. 93–128, 2003.

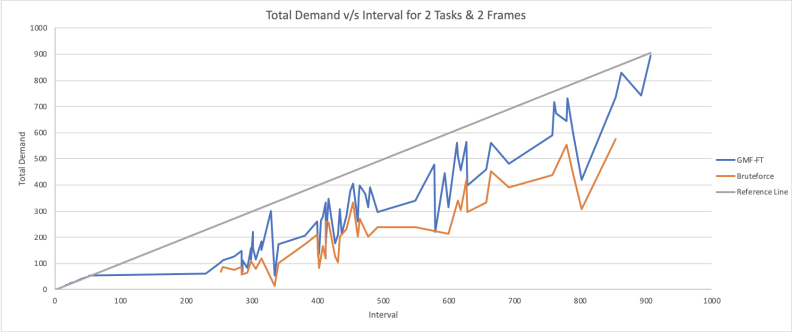
[18] P. N. M.Stigge and W.Yi, "An optimal resource sharing protocol for generalized multiframe tasks," *Journal of Logical and Algebraic Methods in Programming*, vol. 84, no. 1, pp. 92 – 105, 2015.

APPENDIX

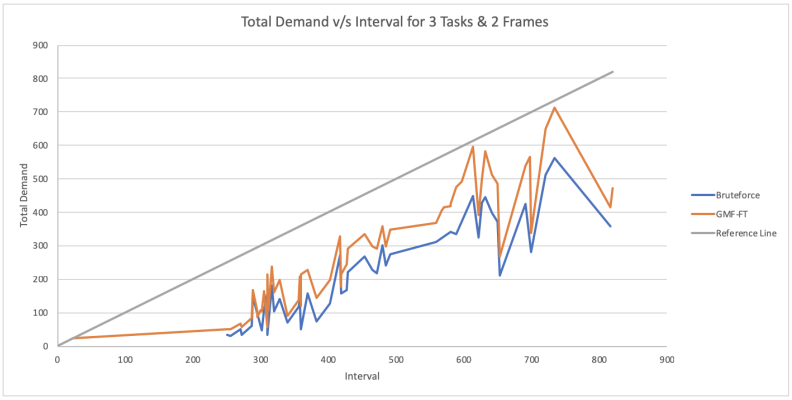
The following figures in this section illustrate the total demand over intervals and total worst case recovery demand over fault intervals. These images are the basis from which the values have been averaged to achieve the average total demand and average worst case demands as seen in Sections 5.2 and 5.3.

Figure A 1 illustrates the total demands over the intervals for 2 tasks and 2 frames in Figure 1(a) and for 3 tasks and 2 frames in Figure 1(b) for both bruteforce and GMF–FT algorithms. The reference $x = y$ line indicates the maximum demand level where the maximum workload of the task is equal to its interval size.

Figure A 2 illustrates the total worst case recovery demands over the intervals for 2 tasks and 2 frames in Figure 2(a) and for 3 tasks and 2 frames in Figure 2(b) for both bruteforce and GMF–FT algorithms. The reference $x = y$ line indicates the maximum recovery level where the maximum worst case recovery demand of the task is equal to its interval size.

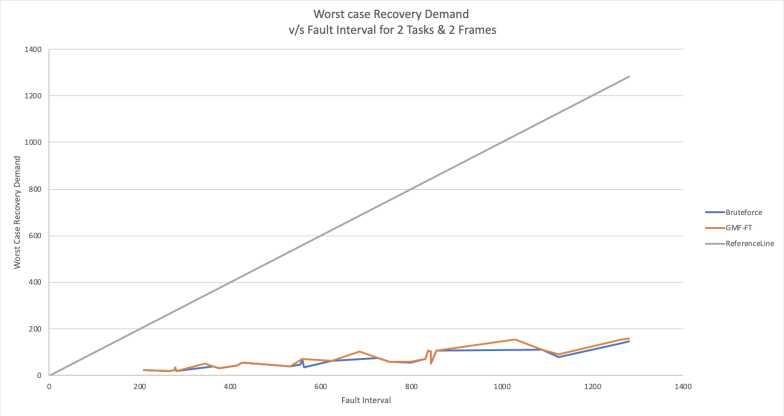


(a) Total Demand for 2 Tasks and 2 Frames

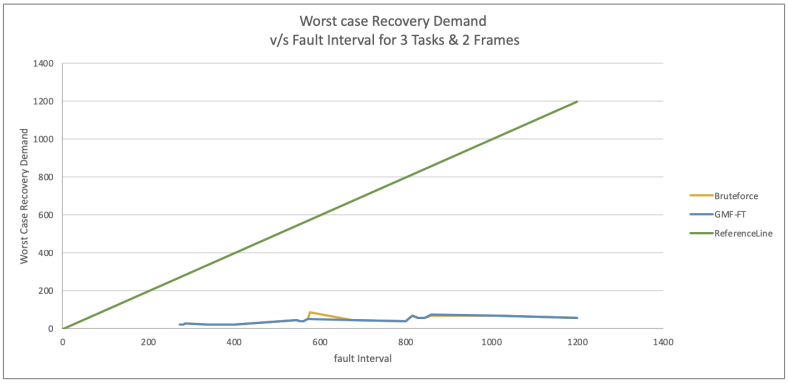


(b) Total Demand for 3 Tasks and 2 Frames

Figure A 1: Total Demand for an Interval for Bruteforce and GMF–FT Algorithms.



(a) Total Worst Case Recovery Demand for 2 Tasks and 2 Frames



(b) Total Worst Case Recovery Demand for 3 Tasks and 2 Frames

Figure A 2: Total Worst Case Recovery Demand for a Fault Interval for Bruteforce and GMF–FT Algorithms.