THESIS FOR THE DEGREE OF LICENTIATE OF ENGINEERING

Scheduling Algorithms For Fault-Tolerant Real-Time Systems

RISAT MAHMUD PATHAN

Division of Networks and Systems
Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
Göteborg, Sweden 2010

Scheduling Algorithms For Fault-Tolerant Real-Time Systems
Risat Mahmud Pathan

Technical Report No. 65L
ISSN 1652-876X
Department of Computer Science and Engineering
Dependable Real-Time Systems Group

Contact Information:

Department of Computer Science and Engineering
Chalmers University of Technology
SE-412 96 Göteborg, Sweden
Phone: +46 (0)31-772 52 16
Fax: +46 (0)31-772 36 63
E-mail: `risat@chalmers.se`

Printed by Chalmers Reproservice
Göteborg, Sweden 2010

Scheduling Algorithms For Fault-Tolerant Real-Time Systems

Risat Mahmud Pathan
Department of Computer Science and Engineering
Chalmers University of Technology

Abstract
This thesis deals with the problem of designing efficient fault-tolerant real-time scheduling algorithms for independent periodic tasks on uni- and multiprocessor platforms. The well-known Rate-Monotonic (RM) scheduling algorithm is assumed as it is widely used in many commercial systems due to its simplicity and ease of implementation. First, a uniprocessor RM scheduling algorithm is analyzed to derive an efficient and exact feasibility condition considering fault-tolerance. Second, a multiprocessor scheduling algorithm is designed to achieve efficient utilization of the processors while meeting the task deadlines. The goal of the former algorithm is to achieve reliability while the goal of the latter algorithm is to achieve a high performance. In this thesis, it is also discussed how to blend these two metrics into the same scheduling framework.

The uniprocessor RM scheduling algorithm is analyzed using a novel composability technique considering occurrences of multiple faults. Based on this analysis, the exact feasibility of the fault-tolerant schedule of a task set can be determined efficiently in terms of time complexity. This algorithm exploits time redundancy as a cost-efficient means to tolerate faults. The fault model considered is very general in the sense that faults can occur in any task and at any time (even during recovery), and covers a variety of hardware and software faults.

The multiprocessor RM scheduling algorithm is designed to achieve a high average utilization of the processors while meeting all task deadlines. The algorithm uses a task-splitting technique, in which a bounded number of tasks are allowed to migrate their execution from one processor to another. It is proved that, using the algorithm, all tasks can meet their deadlines if at most 55.2% of the processor capacity is requested. The load on the processors are regulated to enable the design of an efficient admission controller for online scheduling that scales very well with an increasing number of processors.

Keywords: Real-Time Systems, Periodic Task Scheduling, Rate-Monotonic Scheduling, Uniprocessor, Multiprocessors, Fault-Tolerant Scheduling, Partitioned Scheduling, Task-Splitting Algorithms, Online Scheduling

List of Publications

This thesis is based on and extends the results in the following works:

- ▷ Risat Mahmud Pathan and Jan Jonsson, "Load Regulating Algorithm for Static-Priority Task Scheduling on Multiprocessors," *to appear in Proceedings of the 24th IEEE International Parallel and Distributed Processing Symposium (IPDPS 2010)*, Atlanta, USA, 19-23 Apr, 2010.
- ▷ Risat Mahmud Pathan, Jan Jonsson and Johan Karlsson, "Schedulability of Real-Time Fault-Tolerant Systems Using Multi-core," *presented at the 6th HiPEAC Industrial Workshop*, Massy, France, 26 Nov, 2008.
- ▷ Risat Mahmud Pathan, "Fault-Tolerant Real-Time Scheduling using Chip Multiprocessors," in *Proceedings Supplemental volume of the 7th European Dependable Computing Conference (EDCC 2008)*, Kaunas, Lithuania, 7-9 May, 2008.
- ▷ Risat Mahmud Pathan, "Recovery of Fault-Tolerant Real-Time Scheduling Algorithm for Tolerating Multiple Transient Faults," in *Proceedings of the 10th International Conference of Computer and Information Technology (ICCIT 2007)*, Dhaka, Bangladesh, 27-29 Dec, 2007.
- ▷ Risat Mahmud Pathan, "Probabilistic Analysis of Real-Time Scheduling of Systems Tolerating Multiple Transient Faults," in *Proceedings of the 9th International Conference of Computer and Information Technology (ICCIT 2006)*, Dhaka, Bangladesh, 21-23 Dec, 2006.
- ▷ Risat Mahmud Pathan, "Fault-tolerant Real-Time Scheduling Algorithm for Tolerating Multiple Transient Faults," in *Proceedings of the 4th International Conference on Electrical and Computer Engineering (ICECE 2006)*, Dhaka, Bangladesh, 19-21 Dec, 2006.

Acknowledgments

First of all, I would like to thank my supervisor Docent Jan Jonsson for his excellent comments, invaluable ideas, and most importantly his confidence in me to carry out this research. His knowledge, feedback and guidance have provided me the inspiration to work as a PhD student in real-time systems.

Special thanks and gratitude to Professor Johan Karlsson for sharing with me his knowledge about fault-tolerant computer systems.

I would also like to take the opportunity to thank Professor Sanjoy Baruah who gave me some insights regarding my work on multiprocessor scheduling when I met him last year at the ARTIST Summer School in France.

I also thank Professor Per Stenstörm and Professor Philippas Tsigas for their helpful discussion about the direction of my research at the advisory committee meeting.

Many thanks to my colleagues at the Department of Computer Science and Engineering for creating such a friendly and stimulating working environment. I especially thank Dr. Raul Barbosa and Daniel Skarin who offered me valuable discussion, comments, and also helped me with issues related to LaTeX.

I want to express my deepest gratitude and thanks to my parents who have been encouraging me in pursuing my study. Finally, I thank my wife Nashita Moona and our son Mahir Samran Pathan for their patience and love.

Risat Mahmud Pathan

This research has been funded by the Swedish Agency for Innovation Systems (VINNOVA) under the funding IDs P30619-2 (TIMMO) and S4207 (NFFP-4).

Contents

1 Introduction

Computer systems are ubiquitous. Examples of versatile uses of computers range from playing computer games to diagnosing a shuttle in space. Many of these computer systems are known as *real-time systems* that have timing constraints. One of the most important timing constraints of real-time systems is *meeting the deadlines* of the application tasks. Consequently, the correctness of the real-time systems depend not only on their logical and functional output, but also depend on the time *when* the output is generated. One class of real-time systems that have stringent timing constraints are called *hard* real-time systems. If the timing constraints of hard real-time systems are not satisfied, then the consequences may be catastrophic, like threat to human lives or significant economic loss. So, it is of utmost importance for hard real-time systems designers to ensure that all timing constraints will be met once the system is in mission.

In addition to satisfying the timing constraints of hard real-time system, the functional correctness of the application ought to be guaranteed. Having timely output of the real-time application is of no use if the system deviates from its specified output. The cause of such deviated behavior of computer system is the occurrences of *faults* in the system. For example, after the computer system failed in the London Stock Exchange on September 8, 2008, the stock trading halted for several hours. Hard real-time systems must thus satisfy all the timing constraints of the applications and have to be fault-tolerant in the presence of faults. The timing constraints of the real-time applications can be satisfied using appropriate *task scheduling* and the required level of reliability can be achieved by means of *fault-tolerance*.

Achieving fault-tolerance in computer systems requires employing redundancy either in space or in time. Space redundancy is provided by additional hardware, for example, using extra processors. However, due to cost, volume and weight considerations providing space redundancy may not be always viable, for example, in space, automotive or avionics applications. To achieve fault-tolerance in such systems, time redundancy is used in the form of task *recovery* (re-execution of the original task or execution of a different version of the task) when faults occur. Fault-tolerance using time redundancy cannot be addressed independently of task scheduling issues. This is because time redundant execution as a means for tolerating faults may have a negative impact on the schedule of the tasks in the sense that it might lead to missed deadlines for one or more of the tasks. Consequently, *there is a need for the design of fault-tolerant scheduling algorithms that minimize such intrusive impact resulting from the recovery operations to tolerate faults.*

The recent trend in the processor industry for developing Chip Multiprocessors (CMPs) enables many embedded applications to be deployed on CMPs. Whereas the uniprocessor real-time task scheduling theory is considered very mature, a comprehensive multiprocessor scheduling theory has yet to be developed. Since many of the well-understood uniprocessor scheduling algorithms perform very poorly (in terms of hard real-time schedulability) on multiprocessors, developing new multiprocessor scheduling algorithms has recently received considerable attention. Due to the trend of an increased use of CMPs in embedded systems, *there is a need for the design of efficient multiprocessor scheduling algorithms and their extension toward fault-tolerance.*

Many real-time systems are dynamic in nature, that is, application tasks arrive and leave the system dynamically (for example, in online multimedia, video conferencing or distance learning applications). In such rapidly changing workloads, whether or not to accept a task into the system when it arrives is decided by an *admission controller*. If a task is accepted by the admission controller, then the scheduling algorithm has to decide *when* to execute the task. If there are multiple processors, *where* (tha is, on which processor) to execute the new task adds one more dimension to the scheduling decision. However, all such online decisions have to be made very fast so that the timeliness of other already-admitted tasks into the system is not jeopardized. Considering the trend in developing CMPs with many cores and the diverse ranges of online applications, *there is a need for the design of efficient online admission controllers and online scheduling algorithms for multiprocessors.*

This thesis addresses the problem of designing efficient real-time scheduling algorithms for independent periodic tasks on uni- and multi-processor platforms. The RM scheduling algorithm is assumed because it is widely used in many commercial and safety-critical systems due to its simplicity and ease of implementation. The first main contribution of the thesis deals with real-time task-scheduling analysis on uniprocessor systems under the consideration of multiple occurrences of faults. A uniprocessor fault-tolerant scheduling algorithm, called `FTRM`, is proposed. The algorithm `FTRM` is based on a necessary and sufficient (exact) feasibility condition that ensures that all task deadlines are met if, and only if, this exact condition is satisfied. `FTRM` can tolerate transient faults in hardware as well as a variety of software faults. The fault model considered in this thesis is very general in the sense that faults can occur in any task and at any time, even during the execution of a recovery operation. `FTRM` can precisely determine whether or not a total of f faults can be tolerated within any time interval equal to the maximum period of any task in the periodic task set. As will be discussed in the thesis, the exact feasibility condition used by `FTRM` is directly applicable to partitioned multiprocessor scheduling, that is, where tasks are pre-assigned to processors and never migrate.

The second main contribution of this thesis deals with the design of an efficient online real-time task-scheduling algorithm for multiprocessor systems. The basis of the design is an offline partitioned multiprocessor scheduling algorithm, called Interval-Based-Partitioned-Scheduling (`IBPS`), that pre-assigns tasks to processors. To achieve higher performance than normally achievable with partitioned scheduling, a bounded number of tasks, called *split tasks*, are allowed to migrate exactly once from one processor to another. It is proved that `IBPS` can meet the deadlines of all task if at most 55.2% capacity of the multiprocessor platform is requested. Based on this sufficient feasibility condition, an online version of the algorithm, called Online Interval-Based-Partitioned-Scheduling (`O-IBPS`), is derived. The algorithm maintains a load regulation policy that guarantees that `O-IBPS` scales well with higher number of processors (for example, CMPs with many cores). Due to the simplicity of the feasibility condition, the admission controller of `O-IBPS` can also determine very fast (in linear time) whether or not to accept a newly-arriving task. If admission controller accepts the new task, then no task misses its deadline. If the sufficient condition is not satisfied and if the new task is a critical task, then the admission controller of `O-IBPS` can evict a less critical task and can admit the new task into the system. As will be discussed in the thesis, the scheduling algorithm `O-IBPS` can be used to make a trade-off between performance and reliability by modeling a recovery operation as a newly arriving task when fault occurs in an already admitted task.

The rest of this thesis is organized as follows: Chapter 2 describes the related background of real-time and fault-tolerant systems. The major goals and contributions of this thesis are discussed in Chapter 3. Then, Chapter 4 presents the necessary models (that is, task, system and fault models) used in this work. The analysis of the uniprocessor fault-tolerant scheduling algorithm `FTRM` is presented in Chapter 5 and its applicability to multiprocessor scheduling is discussed. In Chapter 6, the sufficient feasibility condition of the offline multiprocessor scheduling algorithm `IBPS` is analyzed and then its extension to the corresponding online algorithm `O-IBPS` is presented. Finally, Chapter 7 concludes this thesis with a discussion on the applicability and extendability of the proposed algorithms.

2 Preliminaries

In this chapter, the related background of this work is presented by discussing the basic concepts of real-time scheduling followed by a discussion of related concepts regarding fault-tolerant systems.

2.1 Real-Time Systems

Real-time systems are computerized systems with timing constraints. Real-time systems can be classified as *hard real-time systems* and *soft real-time systems*. In hard real-time systems, the consequences of missing a task deadline may be catastrophic. In soft real-time systems, the consequences of missing a deadline are relatively milder. Examples of hard real-time systems are space applications, fly-by-wire aircraft, radar for tracking missiles, etc. Examples of soft real-time systems are on-line transaction used in airline reservation systems, multimedia systems, etc. This thesis deals with scheduling of periodic tasks in hard real-time systems. Applications of many hard real-time systems are often modeled using recurrent tasks. For example, real-time tasks in many control and monitoring applications are implemented as recurrent periodic tasks. This is because periodic execution of recurrent tasks is well understood and predictable [LL73, PM98, ABD$^+$95, SAA$^+$04, BF97]. The most relevant real-time task scheduling concepts in this thesis are: periodic task system, ready (active) task, task priority, preemptive scheduling algorithm, feasibility condition of a scheduling algorithm, offline and online scheduling.

2.1.1 Periodic Task Systems

The basic component of scheduling is a *task*. A task is unit of work such as a program or code block that when executed provides some service of an application. Examples of task are reading sensor data, a unit of data processing and transmission, etc. A *periodic task system* is a set of tasks in which each task is characterized by a *period*, *deadline* and *worst-case execution time (WCET)*.

Period: Each task in periodic task system has an inter-arrival time of occurrence, called the *period* of the task. In each period, a *job* of the task is released. A job is ready to execute at the beginning of each period, called the *released time*, of the job.

Deadline: Each job of a task has a *relative deadline* that is the time by which the job must finish its execution relative to its released time. The relative deadlines of all the jobs of a particular periodic task are same. The *absolute deadline* of a job is the time instant equal to released time plus the relative deadline.

WCET: Each periodic task has a WCET that is the maximum execution time that that each job of the task requires between its released time and absolute deadline.

If the relative deadline of each task in a task set is less than or equal to its period, then the task set is called a *constrained* deadline periodic task system. If the relative deadline of each task in a constrained deadline task set is exactly equal to its period, then the task set is called an *implicit* deadline periodic task system. If a periodic task system is neither constrained nor implicit, then it is called an *arbitrary* deadline periodic task system. In this thesis, scheduling of implicit deadline periodic task system is considered.

2.1.2 Task Independence

The tasks of a real-time application may be dependent on one another, for example, due to resource or precedence constraints. If a resource is shared among multiple tasks, then some tasks may be blocked from being executed until the shared resource is free. Similarly, if tasks have precedence constraints, then one task may need to wait until another task finishes its execution. In this thesis, all tasks are assumed to be independent, that is, there exists no dependency of one tasks on another. The only resource the tasks share is the processor platform.

2.1.3 Ready Tasks

For a periodic task system, a job of a task is released in each period of the task. All jobs that are released but have not completed their individual execution by a time instant t are in the set of *ready (active) tasks* at time t. Note that, there may be no job in the set of ready tasks at one time instant or there may be a job of all the tasks in the set of ready tasks at another time instant.

2.1.4 Task Priority

When two or more ready tasks compete for the use of the processor, some rules must be applied to allocate the use of processor(s). This set of rules is governed by the priority discipline. The selection (by the runtime dispatcher) of the ready task for execution is determined by the priorities of the tasks. The priority of a task can be *static* or *dynamic*.

Static Priority: In static (fixed) priority discipline, each task has a priority that never changes during run time. The different jobs of the same task have the same priority relative to any other tasks. For example, according to Liu and Layland, the well known RM scheduling algorithm assigns static priorities to tasks such that *the shorter the period of the task, the higher the priority* [LL73].

Dynamic Priority: In dynamic priority discipline, different jobs of a task may have different priorities relative to other tasks in the system. In other words, if the priority of jobs of the task change from one execution to another, then the priority discipline is dynamic. For example, the well known Earliest-Deadline-First (EDF) scheduling algorithm assigns dynamic priorities to tasks such that *a ready task whose absolute deadline is the nearest has the highest priority* [LL73].

2.1.5 Preemptive Scheduling

A scheduling algorithm is *preemptive* if the release of a new job of a higher priority task can preempt the job of a currently running lower priority task. During runtime, task scheduling is essentially determining the highest priority active tasks and executing them in the free processor(s). For example, RM and EDF are examples of preemptive scheduling algorithm.

Under non-preemptive scheme, a currently executing task always completes its execution before another ready task starts execution. Therefore, in non-preemptive scheduling a higher priority ready task may need to wait in the ready queue until the currently executing task (may be of lower priority) completes its execution. This will result in worse schedulability performance than for the preemptive case. In this thesis, preemptive scheduling on uni- and multiprocessor platforms is considered.

2.1.6 Work-Conserving Scheduling

A scheduling algorithm is work conserving if it never idles a processor whenever there is a ready task awaiting execution on that processor. A work conserving scheduler guarantees that whenever a job is ready and the processor for executing the job is available, the job will be dispatched for execution. For example, scheduling algorithms RM and EDF are work-conserving by definition.

A non work-conserving algorithm may decide not to execute any task even if there is a ready task awaiting execution. If the processor should be idled when there is a ready task awaiting execution, then the non work-conserving scheduling algorithm requires information about all tasks parameters in order to make the decision when to idle the processor. Online scheduling algorithms typically do not have clairvoyant information about all the parameters of all future tasks, which means such algorithms are generally work-conserving. In this thesis, the work-conserving RM scheduling algorithm is considered.

2.1.7 Feasibility and Optimality of Scheduling

To predict the temporal behavior and to determine whether the timing constraints of an application tasks will be met during runtime, *feasibility analysis* of scheduling algorithm is conducted. If a scheduling algorithm can generate a schedule for a given set of tasks such that all tasks meet deadlines, then the schedule of the task set is *feasible*. If the schedule of a task set is feasible using a scheduling algorithm A, we say that the task set is *A-schedulable*.

A scheduling algorithm is said to be *optimal*, if it can feasibly schedule a task set whenever some other algorithm can schedule the same task set under the same scheduling policy (with respect to for example, priority assignment, preemptivity, migration, etc.). For example, Liu and Layland [LL73] showed that the RM and EDF are optimal uniprocessor scheduling algorithm for static and dynamic priority, respectively.

Feasibility Condition (FC)

For a given a task set, it is computationally impractical to simulate the execution of tasks at all time instants to see in offline whether the task set will be schedulable during runtime. To address this problem, feasibility conditions for scheduling algorithms are derived. A feasibility condition is a (set of) condition(s) that are used to determine whether a task set is feasible for a given scheduling algorithm. The feasibility condition can be *necessary and sufficient (exact)* or it can be *sufficient* only.

Necessary and Sufficient FC (Exact test): A task set will meet all its deadlines if, and only if, it passes the exact test. If the exact FC of a scheduling algorithm A is satisfied, then the task set is A-schedulable. Conversely, if the task set is A-schedulable, then the exact FC of algorithm A is satisfied. Therefore, if the exact FC of a task set is not satisfied, then it is also true that the scheduling algorithm can not feasibly schedule the task set.

Sufficient FC: A task set will meet all its deadlines if it passes the sufficient test. If the sufficient FC of a scheduling algorithm A is satisfied, then the task set is A-schedulable. However, the converse is not necessarily true. Therefore, if the sufficient FC of a task set is not satisfied, then the task set may or may not be schedulable using the scheduling algorithm.

In this thesis, an exact and a sufficient feasibility conditions are derived for RM scheduling on uniprocessor and multiprocessors, respectively.

2.1.8 Minimum Achievable Utilization

A processor platform is said to be fully utilized when an increase in the computation time of any of the tasks in a task set will make the task set unschedulable on the platform. The least upper bound of the total utilization[1] is the minimum of all total utilizations over all sets of tasks that fully utilize the processor platform. This least upper bound of a scheduling algorithm is called the *minimum achievable utilization* or *utilization bound* of the scheduling algorithm. A scheduling algorithm can feasibly schedule any set of tasks on a processor platform if the total utilization of the tasks is less than or equal to the minimum achievable utilization of the scheduling algorithm. In this thesis, minimum achievable utilization bound for the proposed multiprocessor scheduling algorithm is derived.

[1]Utilization of one task is the ratio between WCET and its period. Total utilization is the sum of all tasks' utilization (formally defined later) of a task set.

2.1.9 Scheduling Algorithms

Scheduling algorithms execute tasks on a particular processor platform which can be classified as either uniprocessor or multiprocessors. All scheduling algorithms in this thesis are based on the RM scheduling paradigm for implicit deadline periodic task system.

Uniprocessor Scheduling

Uniprocessor scheduling algorithm executes tasks on a single processor. The schedulability of a given set of tasks on uniprocessor platform can be determined using feasibility condition of the algorithm.

Liu and Layland in [LL73] derived a sufficient feasibility condition for RM scheduling on uniprocessor based on minimum achievable utilization. Necessary and sufficient (exact) feasibility conditions for uniprocessor RM scheduling have been derived in [LSD89, JP86, ABR+93].

It is worth mentioning at this point that the RM algorithm is widely used in industry because of its simplicity, flexibility and its ease of implementation [SLR86, LSD89]. It can be used to satisfy the stringent timing constraints of tasks while at the same time can also support execution of aperiodic tasks to meet the deadlines of the periodic tasks. RM can be modified easily, for example, to implement priority inheritance protocol for synchronization purpose [SRL90]. The conclusion of the study in [SLR86] is that " ... *the rate monotonic algorithm is a simple algorithm which is not only easy to implement but also very versatile*".

Multiprocessor Scheduling

In multiprocessor scheduling, tasks can be scheduled using one of the two basic multiprocessor scheduling principles: *global* scheduling and *partitioned* scheduling. In global scheduling, a task is allowed to execute on any processor even when it is resumed after preemption. This is done by keeping all tasks in a global queue from which tasks are dispatched to the processors based on priority (possibly by preempting some lower priority tasks). In partitioned scheduling, the task set is grouped in different task partitions and each partition has a fixed processor onto which all the tasks of that partition are assigned. A *task assignment algorithm* partitions the task set and assigns the tasks in the local queues of each processor. In partitioned scheduling, ready tasks assigned in one processor are not allowed to execute in another processor even if the other processor is idle. Evidently, tasks can migrate in global scheduling while no migration is allowed in partitioned scheduling. The advantage of partitioned scheduling is that once tasks are assigned to processors, each processor can execute tasks based on mature uniprocessor scheduling algorithms. Many static-priority scheduling policies for both global [ABJ01, Lun02, Bak06, BG03, LL08, BCL05, And08] and partitioned [DL78, DD86, AJ03, FBB06, LMM98, LBOS95, LDG04, LGDG03, OB98, OS95] approaches have been well studied.

It has already been proved that there exists some task set with load slightly greater than 50% of the capacity of a multiprocessor platform on which a deadline miss must occur for both global and partitioned static-priority scheduling [ABJ01, OB98]. Therefore, the minimum achievable utilization bound for both global and partitioned multiprocessor scheduling cannot be greater than 50%. Moreover, it is also well-known that applying the uniprocessor RM scheme to multiprocessor global scheduling can lead to missed deadlines of tasks even when the workload of a task set is close to 0% of the capacity of the multiprocessor platform. This effect is known as *Dhall's effect* [DL78, Dha77]. Technique to avoid Dhall's effect for static-priotity is proposed in [ABJ01] and is further improved in [Lun02, And08]. Luckily, *Dhall's effect* is absent in partitioned scheduling. The main challenge for partitioned scheduling is instead to develop an efficient task assignment algorithm for partitioning a task set. However, since the problem of determining whether a schedulable partition exists is an NP-complete problem [GJ79], different heuristic have been proposed for assigning tasks to multiprocessors using partitioned scheduling. The majority of the heuristics for partitioned scheduling are based on different bin-packing algorithms (such as First-Fit or Next-Fit [LDG04, GJ79]). One bin-packing heuristic relevant for this thesis is the First-Fit (FF) heuristic.

First-Fit (FF) Heuristic: With the FF heuristic, all processors (e.g. processor one, processor two, and so on) and tasks (task one, task two and so on) are indexed. Starting with the task with lowest index (task one), tasks are feasibly assigned to the lowest-indexed processor, always starting with the first processor (processor one). To determine if a non-assigned task will be schedulable on a particular processor with the already-assigned tasks, a uniprocessor feasibility condition is used. If a task cannot be assigned to the first processor, then this task is considered to assign in the second processor and so on. If all the tasks are assigned to the processors, then the partitioning of the task set is successful. If some task cannot be assigned to any processor, then the task set can not be partitioned using FF.

Tasks may be indexed based on some ordering of the task parameters (for example, sort the task set based on period or utilization) or can simply follow any arbitrary ordering for indexing. For example, Dhall and Liu in [DL78] proposed partitioned scheduling for RM based on FF heuristic where tasks are sorted based on increasing period. Baker in [OB98] analyzed RM using FF in which tasks are assigned to processors according to the given input order of the tasks (without any sorting). In this thesis, FF heuristic refers to task assignment to processors without any sorting.

Task-Splitting Algorithms: The different degrees of migration freedom for tasks in the global and partitioned scheduling can be considered as two extremes of multiprocessor scheduling. While in global scheduling no restriction is placed for task migration from one processor to another, partitioned scheduling disallows migration completely. This strict non-migratory characteristic of partitioned multiprocessor scheduling is relaxed using a promising concept called *task-splitting* in which some tasks, called *split-tasks*, are allowed to migrate to a different processor. Task splitting does not mean dividing the code of the tasks; rather it is migration of execution of the split tasks from one processor to another. Recent research has shown that task splitting can provide better performance in terms of schedulability and can overcome the limitations of minimum achievable utilization for pure partitioned scheduling [AT06, AB08, ABB08, KY08a, KY08b, KY07, KY09b, ABD05, LRL09, KLL09]. This thesis presents a multiprocessor scheduling algorithm, called `IBPS`, based on a task-splitting technique.

2.1.10 Offline and Online Scheduling

When the complete schedulability analysis of a task system can be done before the system is put in mission, the scheduling is considered as *offline (static)* scheduling. In order to predict feasibility of a task set, offline scheduling analysis requires the availability of all static task parameters, like periods, execution time, and deadlines. If all task parameters are *not* known before the system is put in mission, then complete schedulability analysis is not possible to predict the feasibility of the newly arriving tasks and such system considers *online (dynamic)* scheduling. Since a newly arriving task can interfere with the execution of already existing tasks in the system, an *admission controller* is needed to determine whether to accept a new task that arrives online.

The feasibility condition of a scheduling algorithm can be used as the basis for designing an admission controller for dynamic systems. However, evaluating the feasibility condition when a new task arrives must not take too long time. This is because using processing capacity to check the feasibility condition could detrimentally affect the timing constraints of the existing tasks in the system. Moreover, after accepting a task using the admission controller, the task assignment algorithm for partitioned multiprocessor scheduling must neither take too long time nor disturb the existing schedules in a large number of processors to assign the newly accepted task (for example, task assignment algorithms that require sorting).

Evaluating the exact feasibility test for uniprocessor RM scheduling derived in [LSD89, ABR$^+$93, JP86] usually takes long time and thus may not be adequate for online admission controller if the system has a large number of tasks. Moreover none of the partitioned scheduling algorithms that require sorting of the tasks before assignment to processors are suitable for online scheduling. This is because, whenever a new task arrived online, sorting of the tasks would require reassignment of all tasks to the processors (for example, RM first-fit in [DL78] requires sorting).

In contrast, a sufficient feasibility condition (closed form) that can be derived offline provides an efficient way to reason about the effect of changing workload in online systems. Interesting approaches for developing efficient uniprocessor RM feasibility conditions for online scheduling can be found in [BBB03, LLWS08, SH98]. In this thesis, the design of an efficient admission controller for online multiprocessor systems is proposed.

2.2 Fault-Tolerant Systems

A system is something that provides some service. A system can be standalone or can be part of a bigger system. The service a system provides may be used by another system to provide another service. For example, the memory of a computer system provides service to the applications running on the computers and the applications in turn provides service to the user of the computer system.

A fault-tolerant system is one that continues to perform its specified service in the presence of hardware and/or software faults. In designing fault-tolerant systems, mechanisms must be provided to ensure the correctness of the expected service even in the presence of faults. Due to the real-time nature of many fault-tolerant systems, it is essential that the fault-tolerance mechanisms provided in such systems do not compromise the timing constraints of the real-time applications. In this section, the basic concepts of fault-tolerant systems under the umbrella of real-time systems are discussed.

2.2.1 Failure, Error, and Faults

Avižienis and others define the terms *failure*, *error* and *faults* in [ALRL04].

Failure A system *failure* occurs when the service provided by the system deviates from the specified service. For example, when a user can not read his stored file from computer memory, then the expected service is not provided by the system.

Error An *error* is a perturbation of internal state of the system that may lead to failure. A failure occurs when the erroneous state causes an incorrect service to be delivered, for example, when certain portion of the computer memory is corrupted or broken and stored files therefore cannot be read by the user.

Fault The cause of the error is called a *fault*. An active fault leads to an error; otherwise the fault is dormant. For example, impurities in the semiconductor devices may cause computer memory in the long run to behave unpredictably.

If a fault remains dormant during system operation, then there is no error. If the fault leads to an error, then the fault must be tolerated so that the error does not lead to system failure. To tolerate faults, errors must be detected in any fault-tolerant system. Identifying the characteristics of the faults that are manifested as errors is an important issue to design effective fault-tolerant systems. Faults in systems may be introduced during development (for example, design and production faults) or due to the interaction with the external environment (for example, faults entering via user interface or due to natural process such as radiation). To that end, faults are grouped as: development, physical and interaction faults [ALRL04]. Based on persistence faults can further be classified as permanent, intermittent, and transient [Joh88]. Faults can occur in hardware or/and software.

Hardware Faults: A permanent or hard fault in hardware is an erroneous state that is continuous and stable. Permanent faults in hardware are caused by the failure of the computing unit. Transient faults are temporary malfunctioning of the computing unit or any other associated components which causes incorrect results to be computed. Intermittent faults are repeated occurrences of transient faults.

Transient faults and intermittent faults manifest themselves in a similar manner. They happen for a short time and then disappear without causing a permanent damage. As will be evident later, the proposed fault-tolerance techniques to tolerate transient faults are also equally applicable for tolerating intermittent faults.

Software Faults: All software faults, known as software bugs, are permanent. However, the way software faults are manifested as errors leads to categorize the effect as: permanent and transient errors. We characterize the effect of software faults that are always manifested as permanent errors. For example, initializing some global variable with incorrect value that is always used during any execution and producing the output of the software is an example of a permanent error. We characterize the effects of the software faults that are not always manifested as transient errors. Such transient errors may be manifested in one particular execution of the software and may not manifest themselves at all in another execution. For example, when the execution path of a software varies based on the input (for example, sensor values) or the environment, a fault that is present in one particular execution path may manifest itself as an transient error only when certain input values are used. This fault may remain dormant when a different execution path is taken, for example, due to a change in the input values or environment.

Sources of Hardware Transient Faults: The main sources of transient faults in hardware are environmental disturbances like power fluctuations, electromagnetic interference and ionization particles. Transient faults are the most common, and their number is continuously increasing due to high complexity, smaller transistor sizes and low operating voltage for computer electronics [Bau05].

Rate of Transient Faults: It has been shown that transient faults are significantly more frequent than permanent faults [SKM$^+$78, CMS82, IRH86, CMR92, Bau05, SABR04]. Siewiorek and others in [SKM$^+$78] observed that transient faults are 30 times more frequent than permanent faults. Similar result is also observed by Castillo, McConnel and Siewiorek in [CMS82]. In an experiment, Iyer and others found that 83% of all faults were determined to be transient or intermittent [IRH86]. The results of these studies show the need to design fault-tolerant system to tolerate transient faults.

Experiments by Campbell, McDonald, and Ray using an orbiting satellite containing a microelectronics test system found that, within a small time interval ($\sim$ 15 minutes), the number of errors due to transient faults is quite high [CMR92]. The result of this study shows that in space applications, the rate of transient faults could be quite high and a mechanism is needed to tolerate multiple transient faults within a particular time interval.

The fault-tolerant scheduling algorithm proposed in this thesis considers tolerating multiple faults within a time interval equal to the maximum period of the tasks in a periodic task set.

2.2.2 Error Detection Techniques

An active fault leads to an error. To tolerate a fault that leads to an error, fault-tolerant systems rely on effective error detection mechanisms. Similarly, the design of many fault-tolerant scheduling algorithm relies on effective mechanisms to detect errors. Error detection mechanisms and their coverage determine the effectiveness of the fault-tolerant scheduling algorithms.

Error detection can be implemented in hardware or software. Hardware implemented error detection can be achieved by executing the same task on two processors and compare their outputs for discrepancies (*duplication and comparison technique using hardware redundancy*). Another cost-efficient approach based on hardware is to use a watchdog processor that monitors the control flow or performs reasonableness checks on the output of the main processor [MCS91]. Control flow checks are done by verifying the stored signature of the program control flow with the actual program control flow during runtime. In addition, today's modern microprocessors have many built-in error detection capabilities like, error detection in memory, cache, registers, illegal op-code detection, and so on [MBS07, WEMR04, SKK^{+}08].

There are many software-implemented error-detection mechanisms: for example, executable assertions, time or information redundancy-based checks, timing and control flow checks, and etc. Executable assertions are small code in the program that checks the reasonableness of the output or value of the variables during program execution based on the system specification [JHCS02]. In time redundancy, an instruction, a function or a task is executed twice and the results are compared to allow errors to be detected (*duplication and comparison technique used in software*) [AFK05]. Additional data (for example, error-detecting codes or duplicated variables) are used to detect occurrences of an error using information redundancy [Pra96].

In summary, there are numerous ways to detect the errors and a complete discussion is beyond the scope of this thesis. The fault-tolerant scheduling algorithms proposed in this thesis rely on effective error-detection mechanisms.

3 Goals and Contributions

The complexity of hardware and software in computerized system is increasing due to the "push-pull" effect between development of new software for existing hardware and advancement in hardware technology for forthcoming software. On the one hand, high-speed processors pull the development of new software with more functionalities (possibly with added complexities) and on the other hand, new software push the advancement of new hardware (with added complexities). The increasing frequency of occurrences of transient faults in increasingly-complex hardware and the increasing likelihood of having more bugs in increasingly-complex software require effective and cost-efficient fault-tolerant mechanisms in today's computerized systems.

The overall goal of this thesis is to design efficient fault-tolerant real-time scheduling algorithms for both uniprocessor and multiprocessors. First, the uniprocessor scheduling algorithm `FTRM` is developed using an exact feasibility condition considering occurrences of multiple faults (Chapter 5). This thesis also suggests how this uniprocessor scheduling algorithm can be extended towards multiprocessor platforms. Second, based on a task-splitting paradigm, a multiprocessor scheduling algorithm `IBPS` is designed considering some important practical features, such as ease of debugging, low overhead of splitting and scalability (Chapter 6). The schedulability of the offline algorithm `IBPS` is analyzed and an efficient online scheduling version, called `O-IBPS`, is proposed. In addition, this thesis suggests how this online scheduling algorithm can be extended towards fault-tolerance.

Traditionally, redundant hardware is used as a means for tolerating faults. However, due to cost, weight, and space considerations, some systems may instead require that faults are tolerated using time redundancy. Simple re-execution or execution of recovery block (ta is, different implementation of the task) are two viable approaches to exploit time redundancy to achieve fault-tolerance. However, exploitation of time redundancy consumes processor capacity in the schedule which may cause task deadlines to be missed. To address this problem, this thesis proposes an efficient uniprocessor scheduling algorithm that can tolerate multiple transient faults using time redundant execution of application tasks. In addition, the recent trend in processor architecture design to have many cores in one chip [KAO05] motivates the design of an efficient multiprocessor scheduling algorithm. The major contributions of this work are as follows:

C1 *Uniprocessor scheduling (Chapter 5)*– A necessary and sufficient feasibility condition is derived for RM uniprocessor fault-tolerant scheduling to tolerate multiple faults. The exact RM feasibility condition of each task is derived based on the maximum total workload requested within the released time and deadline of the task. To calculate this maximum total workload considering the occurrences of faults, a novel technique to compose the execution time of the higher priority jobs is used. The proposed method considers a very general fault model such that multiple faults can occur in any task and at any time (even during recovery). The analysis considers a maximum of f faults that can occur within a time interval equal to the maximum period, denoted as T_{max}, of the tasks in a periodic task set. The feasibility condition and the composability operations are implemented in an algorithm, called `FTRM`.

The proposed composition technique is very efficient because it can exploit knowledge about the critical instant (identification of the worst-case workload scenario). To that end, the run time complexity of `FTRM` is $O(n \cdot N \cdot f^2)$, where N is the maximum number of task jobs (of the n periodic tasks) released within any time interval of length T_{max}. To the best of my knowledge, no other previous work has derived an exact fault-tolerant uniprocessor feasibility condition that has a lower time complexity than that is presented in this thesis for the assumed general fault model. The proposed fault-tolerant uniprocessor scheduling analysis can easily be extended to partitioned multiprocessor RM scheduling.

C2 *Multiprocessor scheduling (Chapter 6)*– A partitioned multiprocessor scheduling algorithm, called `IBPS`, is proposed based on a task-splitting technique. It is proved that the offline scheduling algorithm `IBPS` has a minimum achievable utilization bound of 55.2%. This algorithm is one of the first works[1] to overcome the fundamental limitation of a 50% minimum achievable utiliation bound of the traditional, non-task-splitting partitioned multiprocessor scheduling for static-priority tasks.

In `IBPS`, the load in each processor is regulated in such a way that at most four processors in the system may have an individual load less than 55.2%. This regulation enables an efficient design of a corresponding online scheduling algorithm, called `O-IBPS`. More specifically, finding the best processor to which an accepted online task should be assigned requires searching at most four (underloaded) processors. Similarly, when a task leaves the system, some of the remaining tasks may need to be reassigned on at most five processors to regulate the load for future admittance of new tasks. The task assignment algorithm `IBPS` runs in linear time, which means that reassignment of tasks on a bounded number of processors for the purpose of load regulation will be very efficient. Since tasks only need to be (re-)assigned on a bounded number of processors, the algorithm `O-IBPS` will be very efficient and scale well for systems with a large number of processors (for example, CMPs having many cores). The only other work solving the same problem [LRL09] is not suitable as an online scheduler because reassignment of tasks may in the worst case involve all processors.

One useful application of the `O-IBPS` algorithm is in fault-tolerant real time systems. When a fault occurs in such a system, the required recovery operation to tolerate faults can be considered as a request for a new online task. Based on the available capacity of the processor and the criticality of the existing tasks in the system, it is thus possible to design a fault-tolerant online scheduler. To that end, this thesis proposes three different approaches to achieve fault-tolerance for the online multiprocessor scheduling.

[1]The development of our algorithm took place in parallel with the development of a competing task-splitting technique [LRL09], but the latter work was published first and also had a higher utilization bound.

4 Models

The design of fault-tolerant scheduling algorithms is generally addressed based on the models of the target systems. The *task, system* and *fault* models used in this thesis are presented in this chapter.

4.1 Task Model

To model the recurrent tasks of real-time applications, the popular *periodic task model* is used in this thesis [LL73]. The basic notations and some important concepts used for a periodic task set is presented next.

Periodic Task Set: In this thesis, scheduling of n implicit-deadline periodic tasks in set Γ= $\{\tau_1, \tau_2, \ldots, \tau_n\}$ is considered. Each of the tasks τ_i in set $\{\tau_1, \tau_2, \ldots, \tau_n\}$ is characterized by a pair (C_i, T_i), where C_i represents the WCET and T_i is the period of task τ_i. Each task τ_i is released and ready for execution at the beginning of each period T_i and requires at most C_i units of execution time before next period. The relative deadline of a task τ_i is equal to its period T_i, that is, Γ is an implicit deadline task system.

RM priority: The priority of task τ_i is greater than the priority of task τ_j if $T_i < T_j$. This is the priority assignment governed by the RM scheduling policy, in which a task with smaller period has higher priority. In case, two tasks have equal periods they have the same priority and the run-time dispatcher can arbitrarily break the tie.

Utilization: The *load* or *utilization* of a task τ_i is denoted by $u_i = C_i/T_i$. The *total load* or *total utilization* of any task set A is $U(A) = \sum_{\tau_i \in A} u_i$. For example, the total utilization of the task set Γ is $U(\Gamma)$, which is the total load of the task set Γ.

Jobs of Tasks: A *job* of a task τ_i is released in each period T_i. All the tasks in set Γ are released at the same time and it is assumed[1] that this time is zero. The j^{th} job of task τ_i is denoted by $\tau_{i,j}$ for $j = 1, 2, \ldots \infty$. Job $\tau_{i,j}$ is released at time $(j-1) \cdot T_i$ and has an absolute deadline[2] at time $j \cdot T_i$. Formally, the released time $r_{i,j}$ and deadline $d_{i,j}$ of job $\tau_{i,j}$ for $i = 1, 2, \ldots n$ and $j = 1, 2, \ldots \infty$ are defined as follows:

$$r_{i,j} = (j-1) \cdot T_i \tag{4.1}$$

$$d_{i,j} = j \cdot T_i \tag{4.2}$$

Critical Instant: The critical instant of a task is the released time at which the interference on the task from the higher priority tasks is maximized. Liu and Layland (without considering faults) have proved that the critical instant of uniprocessor RM scheduling for any task occurs when the task is released simultaneously with the release of all of its higher priority tasks [LL73].

The uniprocessor schedulability analysis of algorithm `FTRM` in this thesis must consider the critical instant of each task. Moreover, since the multiprocessor scheduling algorithm `IBPS` proposed in this thesis is based on partitioned scheduling, the RM schedulability analysis on each processor in the multiprocessor system must also consider the critical instant for each task assigned to that particular processor.

Under fault-tolerant scheduling, there is one job of each task in which the occurrence of faults have the greatest impact. In such case, the faults may occur in that particular job of the task and/or in any job of its higher priority tasks. Ghosh *et al.* showed that, when faults occur and time redundancy is used to tolerate faults in uniprocessor RM scheduling, the critical instant is when all tasks are released simultaneously [GMMS98a]. The reasoning is as follows: if the completion of job J of a task is delayed by Δ time units due to

[1]This latter assumption is only for convenience in our worst-case schedulability analysis.

[2]In the rest of this document ‘deadline’ refers to the absolute deadline of a job if not mentioned otherwise.

the occurrence of some faults in J or its higher-priority jobs, then some other lower priority job J' of some other task will be delayed by at most Δ time unit if both J and J' are released simultaneously.

4.2 System Model

The system model considered in this thesis is either a uniprocessor or a multiprocessor platform[3]. The multiprocessor platform consists of a number of uniprocessors connected by some interconnection network. The expected service in each uniprocessor is delivered by executing a number of tasks assigned to the processor.

In partitioned multiprocessor scheduling, the tasks to execute on a processor are assigned by some task assignment algorithm. The run time system in each processor is a RM uniprocessor scheduler. Each processor executes a number of real-time tasks using RM prioritization. Tasks are assumed to be independent, that is, there is no resource sharing except for the processor. The cost of a preemption and context-switch is assumed to be negligible.

4.3 Fault Model

Designing fault-tolerant scheduling algorithm needs to guarantee that all tasks deadlines are met when faults occur even under the worst-case load condition. No fault-tolerant system can, however, tolerate an arbitrary number of faults within a particular time interval. The scheduling guarantee in fault-tolerant system is thus given under the assumption of a certain fault model.

In this thesis, the fault model mainly assumes tolerating the faults due to which the error is transient either in hardware or software. In addition, permanent software errors[4] are also considered in the fault model when diverse implementation of the software is available. It is assumed that transient faults are short lived and would not reappear when re-executing the same task. This is a reasonable assumption since it can be implemented simply by resetting the processor before re-execution. Our fault-tolerant mechanism can also tolerate certain class of software faults. If the effect of faults in software is manifested as transient error that would not re-appear upon re-execution, then such faults can be tolerated using simple re-execution of the task. For example, due to changes in the environment or changes in the input parameters, the execution

[3]By ‘processor’ we also mean an individual processing core in CMPs.

[4]permanent software faults means bugs that are always present or permanent in nature.

path a software takes could be different from one execution to another. In such case, it is expected that the same error would not occur (since a different execution path is taken) if the software is simply re-executed. If the effect of faults in a software is manifested as a permanent error, then re-execution of the same software can not mitigate such faulty behavior. In such case a different version of the software (that is called, a recovery block) can be executed when error is detected.

Time redundancy is considered in this thesis for tolerating multiple faults. Faults are assumed to be detected at the end of execution of a task. This assumption is must for the worst-case schedulability analysis as is pointed in [PM98]. When fault occurs during execution of a task and error is detected, either the faulty task is simply *re-executed* or a *recovery block* corresponding to the faulty task is executed. The recovery block of a task is a different implementation of the same task to achieve diversity as is used in N-version programming [Avi85]. The recovery block of a task has the same period as the original task but may have a different WCET than that of the original task. When a task is executed for the first time, it is called the *primary copy* of the task. After an error is detected, the re-execution or execution of the recovery block is called the *recovery copy* of the task.

The re-execution of the task or execution of the recovery block is activated when an error is detected. We assume that a combination of software and hardware error-detection mechanisms are available to detect the occurrences of faults. There are many software and hardware based error-detection mechanisms as is discussed in Section 2.2.2. It is also assumed that the error-detection and fault-tolerance mechanisms are themselves fault-tolerant.

Perfect error detection coverage is assumed for simplicity of the schedulability analysis. However, a probabilistic analysis of fault-tolerant schedulability with imperfect error detection coverage can be addressed similar to [BPSW99, AH06, Pat06] and such an analysis is not the addressed in this thesis. The error detection overhead is considered as part of the WCET of the task. There is no fault propagation, that is, faults affect only the results produced by the executing task. This no-fault-propagation assumption is reasonable and is a requirement in the design of many safety-critical applications, for example, in Integrated Modular Avionics (IMA) systems, as is discussed in [Bar08]. Permanent processor failure is assumed to be tolerated using system level fault tolerance [Bar08] and not considered in this thesis.

In summary, the fault model considered in this thesis has reasonable representativity and very general to tolerate a variety of faults in hardware/software.

5 Uniprocessor Scheduling

This chapter presents the analysis of RM scheduling on uniprocessor for tolerating multiple faults. The outcome of the analysis is the derivation of a necessary and sufficient feasibility condition for fault-tolerant RM scheduling of periodic tasks on a uniprocessor. An algorithm, called `FTRM`, is presented based on this necessary and sufficient feasibility condition. Using algorithm `FTRM`, the feasibility of a set of periodic tasks on uniprocessor can be determined efficiently considering multiple occurrences of faults.

5.1 Introduction

The importance of dependability is increasing as computers are taking a more active role in everyday control applications. Fault-tolerance in such systems is an important aspect to guarantee the correctness of the application even in the event of faults. In many safety-critical systems, use of time redundancy is considered as a cost-efficient means to achieve fault-tolerance. In such systems, when an error is detected tasks are simply re-executed or a different version, called recovery block, of the task is executed. Due to the additional real-time requirements of such systems, it is essential that exploitation of time redundancy as a means for tolerating faults must not compromise the timeliness guarantee of the other tasks in the system.

To guarantee both correctness and timeliness behavior of safety-critical real-time systems it is necessary to design a fault-tolerant scheduling algorithm. To that end, an offline uniprocessor schedulability analysis is presented in this chapter. The objective of the analysis is to find the condition needed to verify that a set of n static-priority periodic tasks will meet the deadlines even when faults occur in the system. The outcome of the analysis is the derivation of a necessary and sufficient (exact) feasibility condition for RM scheduling considering occurrences of multiple faults on a uniprocessor. Based on this exact feasibility condition, a fault-tolerant uniprocessor scheduling algorithm `FTRM` is proposed for scheduling n periodic tasks in task set Γ.

Th exact feasibility condition of the task set Γ is derived in terms of the the exact RM feasibility condition of each task based on the maximum total workload requested within the released time and deadline of the task. To calculate this maximum total workload considering the occurrences of faults, a novel technique to compose the execution time of the higher priority jobs is used. The main important characteristic of the proposed composability technique is that it is not only applicable for tasks with implicit deadline and RM priority, but also for tasks with constrained deadlines and any fixed-priority policy. Therefore, the proposed feasibility analysis technique in this chapter would enable the derivation of an exact feasibility condition for any fixed-priority scheduling of constrained or implicit deadline task systems (for example, in fault-tolerant deadline-monotonic). However, in this thesis, the composability mechanism is described for uniprocessor RM scheduling.

The scheduling analysis of `FTRM` considers occurrences of a maximum of f faults within any time interval of length T_{max} where T_{max} is the largest period of any task in the periodic task set Γ. The run-time complexity of `FTRM` is shown to be $O(n \cdot N \cdot f^2)$ where n is the number of tasks in a periodic task set, N is the maximum number of jobs released within any interval of length T_{max}. To the best of my knowledge, this the first fault-tolerant RM scheduling algorithm that considers such a general fault model as described in Section 4.3. A similar work exists for EDF considering the same fault model [Ayd07]. Compared to that work our proposed algorithm is more efficient (in terms of time complexity) due to a new way to calculate the impact of higher-priority tasks on the schedulability of a particular task.

The uniprocessor analysis presented in this chapter is applicable to partitioned multiprocessor scheduling, where each processor executes tasks using uniprocessor scheduling algorithm. A task assignment algorithm assigns the tasks from set Γ to the processors of a multiprocessor platform. To determine whether an unassigned task can be feasibly assigned to a processor, the proposed exact fault-tolerant RM feasibility condition for a uniprocessor can be used, which guarantees that each processor can tolerate up to f number of faults within any time interval equal to the maximum length of the periods of the tasks assigned to that particular processor.

The rest of the chapter is organized as follows: the necessary models and theories used for uniprocessor fault-tolerant schedulability analysis is presented in Section 5.2. Then, related work in fault-tolerant scheduling is presented in Section 5.3. The problem statement is formally given in Section 5.4. The fault-tolerant RM schedulability analysis for one individual task is presented in Section 5.5. Then, in Section 5.6, the necessary and sufficient fault-tolerant RM feasibility condition for a complete task set is derived. The algorithm `FTRM` is presented in Section 5.7 and its applicability to the multiprocessor setting is discussed. Section 5.8 concludes this chapter.

5.2 Background

5.2.1 Task Model

The task model used for the uniprocessor schedulability analysis is same as the one presented in Section 4.1. The task model we consider assumes a set of n implicit deadline periodic tasks Γ =$\{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task $\tau_i \in \Gamma$ is characterized by WCET C_i and period T_i. The j^{th} job of task τ_i is denoted by $\tau_{i,j}$. The job $\tau_{i,j}$ is released at time $r_{i,j}$ and has an absolute deadline by time $d_{i,j}$ as is defined in Eq. (4.1). We define T_{max} to be the maximum period of any task in the task set Γ =$\{\tau_1, \tau_2, \ldots, \tau_n\}$, that is,

$$T_{max} = \max_{i=1}^{n} \{T_i\}$$

Without loss of generality we assume that tasks are sorted in set Γ in order of decreasing priority, that is, all tasks in set $\{\tau_1, \tau_2, \ldots \tau_{i-1}\}$ are of higher-priority than the priority of task τ_i. Moreover, we denote the maximum number of jobs released within any time interval of length T_{max} by N.

5.2.2 Fault Model

The fault model presented in Section 4.3 is extended here for exact RM feasibility analysis on a uniprocessor. A number of f faults that may occur within any time interval of length equal to T_{max} is considered. The faults can occur during the execution of any primary and/or recovery copy of a task. A new recovery copy is activated when an error is detected at the end of execution of a primary or recovery copy of a task. The recovery copy could simply be the re-execution of the primary copy or it could be a different implementation of the same task, called recovery block, in which case the WCET may not be same as the WCET of the primary copy. Within any time interval of length T_{max}, the f faults may occur in the same task's primary and recovery copies or may occur in different tasks. The recovery copy of a task τ_i is executed with the same priority as that of task τ_i.

For each task τ_i, the primary copy executes first. If an error is detected at the end of execution of the primary copy, the first recovery copy of the task is ready to execute. Again an error may be detected at the end of execution of the recovery copy which in turn would trigger the execution of next recovery copy and so on. Therefore, each task must have f recovery copies in case if all the f faults occur in the same job of a task. If we say k faults occur in a job of task τ_i, then we mean that the first fault occurs in the primary copy of the job of task τ_i and each of the subsequent $(k-1)$ faults occurs in each subsequent recovery copy of the same job of task τ_i. Note that, when the cumulative execution demand within an interval of length T_{max} due to f faults is at its maximum, then it is necessary that all the f faults occur within that interval.

Remember that an error is assumed to be detected at the end of execution of a task's primary or recovery copy. We assume that, during execution of a particular primary or recovery copy of a task, at most one fault could occur. This assumption is essential for the worst-case schedulability analysis. Because when an error is detected at the end of execution of a task's primary or recovery copy, the overhead for executing the recovery operation does not depend on the number of faults affecting that particular faulty copy.

The f faults may occur in one job or might occur in different jobs of different tasks released within a time interval of length T_{max}. The fault-tolerant scheduling algorithm must guarantee that, for any combination of the occurrences of the f faults in the jobs released within any interval of length T_{max}, the schedule has to be fault-tolerant. Remember that we assume that there are maximum N jobs released within any interval of length T_{max}. There are different possibilities of the occurrences of the f faults in the N jobs. One possibility is that all the f faults occur in one of the N jobs. Another possibility is that a different number of faults occur in different jobs. Each such possibility of fault occurrence is called a *fault pattern* [Ayd07, LMM00]. Given a set of jobs in A we denote any possible combination of k faults that can occur in the jobs in set A by k-fault-pattern, $k = 0, 1, 2 \dots f$. For example, if $k = 0$, no fault occurs within the jobs in set A.

To achieve fault-tolerance, it has to be ensured that all the jobs released within any interval of length T_{max} will meet the deadlines for any possible f-fault pattern. The question that arises at this point is: *what are the different possible fault patterns that one must consider for RM feasibility analysis of N jobs released within a time interval of length T_{max}?* In other words, in how many ways the f faults could occur in N jobs that are released within any time interval of length T_{max}. According to combinatorial theory, there are a total of $\binom{N+f-1}{f}$ different ways the f faults could occur in N jobs. As is already pointed out in [Ayd07], number of different fault patterns given by the binomial coefficient $\binom{N+f-1}{f}$ is equal to

$$\binom{N+f\text{-}1}{f} = \Omega\left(\left(\frac{N}{f}\right)^f\right) = O(N^f) \tag{5.1}$$

according to [CLRS01].

The RM feasibility analysis on uniprocessor considering this exponential number of fault patterns may not be computationally practical if N and f are large. To overcome this problem, a dynamic programing technique is used in this thesis to find an exact RM feasibility condition for a task set considering f faults that could occur in any interval of length T_{max}. The time complexity of this technique for evaluating the exact feasibility condition is $O(n \cdot N \cdot f^2)$.

5.2.3 RM Schedulability

RM scheduling is widely used in many real-time systems because of its simplicity and ease of implementation. The analysis of uniprocessor RM scheduling for an implicit deadline periodic task system is addressed by Liu and Layland in [LL73]. Liu and Layland proved that RM is an optimal fixed-priority scheduling algorithm on a uniprocessor. They derived a sufficient feasibility condition (without considering fault-tolerance) that can check RM feasibility of a set of n periodic tasks in $O(n)$ time. Necessary and sufficient (exact) feasibility conditions for uniprocessor RM scheduling have been derived in [LSD89, JP86, ABR$^+$93].

Liu and Layland in [LL73] proved that if the first job of each task can meet its deadline when all tasks are simultaneously released (known as a critical instant), then all the jobs of a task set are RM schedulable. The exact RM feasibility condition proposed by Lehoczky, Sha and Ding in [LSD89] is derived by assuming that all tasks are released at time 0. In [LSD89], the cumulative execution demand by the tasks in set $\{\tau_1, \tau_2, \ldots \tau_i\}$ over an interval $[0, t)$ is given as follows:

$$W_i(t) = \sum_{j=1}^{i} C_j \cdot \lceil \frac{t}{T_j} \rceil$$

The necessary and sufficient condition for RM feasibility of a periodic task set Γ =$\{\tau_1, \tau_2, \ldots, \tau_n\}$ according to [LSD89] is given in Eq. (5.2) and Eq. (5.3):

1. Task τ_i can be scheduled using RM if and only if:

$$L_i = \min_{t \in Q_i} \frac{W_i(t)}{t} \leq 1 \tag{5.2}$$

   where $Q_i = \{k \cdot T_j \mid j = 1 \ldots i, k = 1 \ldots \lfloor \frac{T_i}{T_j} \rfloor\}$

2. The task set Γ is RM schedulable if and only if:

$$\max_{i=1\ldots n} L_i \leq 1 \tag{5.3}$$

The exact feasibility condition for a task τ_i in the task set $\{\tau_1, \tau_2, \ldots, \tau_n\}$ is given in Eq. (5.2). Based on the exact feasibility condition for each one of the tasks $\tau_i \in \{\tau_1, \tau_2, \ldots, \tau_n\}$, the exact feasibility condition of the entire task set $\{\tau_1, \tau_2, \ldots, \tau_n\}$ is derived in Eq. (5.3).

In this thesis, the fault model considers different number of faults in different jobs of the same task due to various fault patterns. Therefore, the execution time of different jobs of the same task could be different. Consequently, because the worst-case fault pattern is not known in advance, the exact analysis as given in Eq (5.3) is not applicable for the exact fault-tolerant schedulability analysis. However, in this thesis an approach similar to Eq. (5.2) and Eq. (5.3) is used—in the sense that the exact fault-tolerant RM feasibility condition for the task set Γ is derived in terms of the exact fault-tolerant feasibility condition of individual task.

5.3 Related Work

Many approaches exist in the literature for tolerating faults in task scheduling algorithms. Traditionally, processor failures (permanent faults) are tolerated using Primary and Backup (PB) approaches in which the primary and recovery copies of each task are scheduled on two different processors [BT83, KS86, SWG92, Gho, OS94, GMM97, MM98, BMR99, AOSM01, KLLS05b, KLLS05a]. Since this thesis does not deal with permanent faults, these works will not be further discussed.

Ghosh, Melhem and Mossé proposed fault-tolerant uniprocessor scheduling of aperiodic tasks considering transient faults by inserting enough slack in the schedule to allow for the re-execution of tasks when an error is detected [GMM95]. They assumed that the occurrences of two faults are separated by a minimum distance. Pandya and Malek analyzed fault-tolerant RM scheduling on a uniprocessor for tolerating one fault and proved that the minimum achievable utilization bound is 50% [PM98]. The authors also demonstrated the applicability of their scheme for tolerating multiple faults if two faults are separated by a minimum time distance equal to T_{max}. In this thesis, the proposed algorithm FTRM can tolerate f faults within any time interval equal to T_{max} and no restriction is placed between the time distance between two consecutive faults within T_{max}.

Ghosh *et al.* derived a utilization bound for RM uniprocessor scheduling for tolerating single and multiple transient faults using a concept of backup utilization [GMMS98b]. To tolerate f transient faults, the utilization of the backup is set to f times the maximum utilization of any task given that a fault model similar to the one in this thesis is used. Whereas the recovery scheme in [GMMS98b] allows recovery tasks to execute at a priority higher than that of the faulty task, the recovery scheme in this thesis executes recovery copies at the same priority as the faulty task.

Liberto, Melhem and Mossé derived both exact and sufficient feasibility conditions for tolerating f transient faults for a set of aperiodic tasks using EDF scheduling [LMM00]. They showed that for a set of n aperiodic tasks in which a maximum of f faults could occur, the exact test can be evaluated in $O(n^2 \cdot f)$ time using a dynamic programming technique. However, the authors of [LMM00] consider recovery copy of a faulty task simply as a re-execution of the primary copy and do not consider the execution of a recovery block (that is, diverse implementation of a task with a possibly different execution time) when an error is detected.

Burns, Davis, and Punnekkat derived an exact fault-tolerant feasibility test for any fixed priority system using recovery blocks or re-execution [BDP96]. This work is extended in [PBD01] to provide the exact schedulability tests employing check-pointing for fault recovery. In [MdALB03], de A Lima and Burns proposed an optimal fixed priority assignment to tasks for fault-tolerant scheduling based on re-execution. The fixed priorities of the tasks can be determined in $O(n^2)$ time for a set of n periodic tasks. The schedulability analysis in [BDP96, MdALB03] require the information about the minimum time distance between any two consecutive occurrences of transient faults within the schedule, and only considers simple re-execution or execution of a recovery block when error is detected. In the latter case, the execution time of the recovery block is the same regardless of the number of execution of a particular job. This is in contrast to our proposed method where each recovery block for a particular job may have different execution time.

Based on the *last chance strategy* of Chetto and Chetto [CC89] (in which recovery tasks execute at late as possible), software faults are tolerated by considering two versions of each periodic tasks: a primary task and a recovery block [HSW03]. Recovery blocks are scheduled as late as possible using a backward RM algorithm (schedule from backward of time). Similar to the work in [MdALB03], the work in [HSW03] considers that there is only one recovery block of each task and therefore do not have the provision for considering different recovery blocks if more than one fault affect the same task.

Santos *et al.* in [SSO05] derived a feasibility condition for determining the combinations of faults in jobs that can be tolerated using fault-tolerant RM scheduling of periodic tasks. In order to guarantee that the system can tolerate multiple transient faults for any combination of faults, all possible fault patterns has to be considered in their derived condition which gives an intractable time complexity. Moreover, the authors assumed that a fault can occur only in the primary copy of a job. The work in [SSO05] is based a notion, called k-RM schedulable (originally proposed in [SUSO04]). By k-RM schedulable, the authors mean that there are at least k free time slots available between the release time and deadline of each task. The time complexity to determine whether a task set is k-RM schedule is $O(n \cdot T_{max})$ which can be significant for large T_{max} [SUSO04].

Aydin in [Ayd07] proposed aperiodic and periodic task scheduling based on an exact EDF feasibility analysis in which a recovery copy of a task can be different from the primary copy. Aydin considers a fault model in which a maximum of f transient faults could occur in any task of the aperiodic task set. The schedulability analysis in [Ayd07] is based on processor demand analysis [BRH90]. For periodic task systems, the proposed exact feasibility test in [Ayd07] is evaluated in $O(A^2 \cdot B^2)$ time, where A is the number of jobs released within the first hyper-period (i.e. least common multiple of all the tasks periods) and B is the number of faults that can occur within the first hyper-period. In this thesis, we derive an exact RM feasibility condition for which the run-time complexity is $O(n \cdot N \cdot f^2)$ where N is the maximum number of jobs of the n periodic tasks released within a time interval of length T_{max}, and f is the maximum number of faults that can occur within any time interval of length

T_{max}. To see the difference between the terms A^2 and ($n \cdot N$) exist in the time complexity figures $O(A^2 \cdot B^2)$ and $O(n \cdot N \cdot f^2)$, consider $n = 4$ tasks such that the periods of the four tasks are $T_1 = 3$, $T_2 = 7$, $T_3 = 11$, and $T_4 = 13$. The length of the hyper-period is $lcm\{3 \cdot 7 \cdot 11 \cdot 23\} = 3003$. The number of jobs released within the hyper-period is $A = 1934$. On the other hand, the number of jobs released within any interval of length T_{max} =13 is $N = 10$. The value of A^2 and ($n \cdot N$) are $1934^2 = 3740356$ and $(4 \cdot 10) = 40$ in the the time complexity $O(A^2 \cdot B^2)$ and $O(n \cdot N \cdot f^2)$ of the EDF algorithm in [Ayd07] and our proposed FTRM algorithm, respectively. Therefore, for a given failure rate, the time complexity of our proposed exact RM test for periodic task set is more efficient than that of the exact EDF test proposed in [Ayd07].

In conclusion, most of the work related to developing fault-tolerant scheduling algorithms using time redundancy consider a fault model that is not as general as the fault model considered in this thesis. Only the work in [Ayd07] for EDF scheduling considers a fault model similar to the one addressed in this thesis. However, the exact EDF test in [Ayd07] is less efficient in terms of time complexity than that of ours for RM. Therefore, if both efficiency and preciseness is required, our FTRM algorithm can provide better performance than that of EDF scheduling in [Ayd07]. However, it should be mentioned that, in [Ayd07], a sufficient test (lacking preciseness in the feasibility testing) for EDF scheduling of periodic tasks is proposed that is more efficient than our proposed exact RM feasibility condition.

5.4 Problem Formulation

The uniprocessor fault-tolerant scheduling algorithm FTRM proposed in this thesis is based on an exact RM feasibility condition. We consider occurrences of a maximum of f faults within any time interval of length T_{max}. The f faults could be distributed over any jobs that are eligible to execute within any time interval of length T_{max}. Note that a job is eligible to execute between its released time and its deadline. The problem addressed in this thesis is:

Is the task set Γ RM-schedulable if a maximum of f faults occur within any time interval of length equal to T_{max}?

The exact RM feasibility condition of the task set Γ for the fault-tolerant scheduling algorithm FTRM can be derived based on exact feasibility condition of each task $\tau_i \in \Gamma$, for $i = 1, 2, \ldots n$. If a maximum of f faults can occur within a time interval of length T_{max}, then the maximum number of faults that can occur within any time interval of length T_i, for $i = 1, 2, 3, \ldots n$, can be at most f. Following this, the last problem statement can be re-written as:

Is task τ_i RM-schedulable if a maximum of f faults occur within any time interval of length equal to T_i, for $i = 1, 2, \ldots n$?

It is clear that, if the exact feasibility condition for each task $\tau_i \in \Gamma$ can be determined, then the exact feasibility condition for the entire task set Γ follows immediately (similar to the approach in [LSD89]).

To ensure that task τ_i is RM-schedulable on a uniprocessor, the critical instant for which the load imposed by the higher-priority tasks on task τ_i is maximized needs to be considered in the fault-tolerant schedule. Under our assumed fault model, the critical instant in the uniprocessor fault-tolerant schedule is when all the tasks are released at the same time (as discussed in Section 4.1). In this thesis, without loss of generality, we assume that all the tasks are released simultaneously at time zero. Similar to [LSD89], in order to derive the exact feasibility condition of task τ_i, it is sufficient to derive the exact feasibility condition for the first job of each task $\tau_i \in \Gamma$. The first job of task τ_i become eligible for execution at time 0 and must finish its execution (including any possible recovery execution due to faults) before time T_i. Consequently, the problem addressed can finally be re-written as:

Is the first job of task τ_i RM-schedulable if a maximum of f faults occur within the time interval [0, T_i), for $i = 1, 2, \ldots n$?

In the rest of this chapter, the exact feasibility condition of task τ_i refers to the exact feasibility condition of the first job of τ_i unless otherwise noted.

In next Section 5.5, the exact feasibility analysis of task τ_i within $[0, T_i)$ is presented. In order to find the exact feasibility condition, the maximum total work completed within $[0, T_i)$ by the jobs of the tasks $\{\tau_1, \tau_2 \ldots \tau_i\}$ is calculated based on two *load factors* within $[0, T_i)$. In subsection 5.5.1, the first load factor that is equal to the maximum work needs to be completed by a job of task τ_i is calculated. Then in subsection 5.5.2, the second load factor that is equal to the maximum work completed within $[0, T_i)$ by the higher priority jobs of the tasks $\{\tau_1, \tau_2 \ldots \tau_{i-1}\}$ is calculated. This second load factor is calculated by determining, for selected time points within $[0, T_i)$, the different sets of higher priority jobs for which all jobs in each such set are released at the same time. Using these identified sets (note that, all the higher-priority jobs in one such set are released at the same time), the execution of all higher-priority jobs is then abstracted by means of two *composition* techniques, called *vertical composition* and *horizontal composition*, to find the maximum work completed by the higher priority jobs within $[0, T_i)$ in subsection 5.5.2.

It is worth mentioning at this point that the proposed composability technique is not only applicable for tasks with RM priority, but also for tasks with any fixed-priority policy (for example, deadline-monotonic scheduling). The novelty of the following composability technique enable to do schedulability analysis that requires to find the worst-case workload within a given time interval considering occurrences of faults. In this thesis, the composability technique applied to a set of periodic tasks having RM priority is demonstrated in next section.

5.5 Load Factors and Composability

In this section, we derive the fundamental theoretical building blocks for the schedulability analysis of task τ_i within the time interval $[0, T_i)$ in terms of load factors and compositions. The task τ_i may *not* have exclusive access to the processor within the entire time interval $[0, T_i)$ because jobs of higher-priority tasks are eligible for execution within this interval. To determine whether the first job of task τ_i is feasible using RM, the amount of execution completed by higher-priority jobs within $[0, T_i)$ needs to be calculated. Note that the maximum amount of execution completed by the higher-priority jobs depends on different fault patterns affecting these higher-priority jobs. By subtracting the maximum amount of execution completed by the higher- priority jobs within $[0, T_i)$ from T_i, the available time for execution of task τ_i within $[0, T_i)$ can be derived. To determine whether the available execution time for task τ_i is enough for its complete execution within $[0, T_i)$, we need to know the maximum amount of execution required to be completed by the first job of task τ_i. This amount of execution depends on the number of faults exclusively affecting task τ_i within $[0, T_i)$.

The *worst-case workload* within $[0, T_i)$ is the maximum amount of execution completed by the jobs of the tasks in set $\{\tau_1, \tau_2 \ldots \tau_i\}$ that are released within $[0, T_i)$. Remember that at most f faults could occur within $[0, T_i)$. To find this worst-case workload required to be completed within $[0, T_i)$ by the jobs of the tasks in set $\{\tau_1, \tau_2 \ldots \tau_i\}$, we have to consider (i) the occurrences of k faults affecting the jobs of higher-priority tasks (including their recovery copies), and (ii) the occurrences of $(f - k)$ faults exclusively affecting the first job of task τ_i and its recovery copies, for $k = 0, 1, 2, \ldots f$. In summary, to find the worst-case workload within $[0, T_i)$, we need to determine the following two workload factors, for $k = 0, 1, 2 \ldots f$.

1. `Load-Factor-i`: Execution time required by task τ_i when $(f - k)$ faults exclusively affect the first job of task τ_i, for $k = 0, 1, 2 \ldots f$.

2. `Load-Factor-HPi`: Execution time required by the higher-priority jobs (that is, jobs of tasks $\{\tau_1, \tau_2 \ldots \tau_{i-1}\}$) within $[0, T_i)$ when k faults affect these higher-priority jobs in this interval, for $k = 0, 1, 2 \ldots f$.

The worst-case workload within $[0, T_i)$ can now be defined as the sum of these two load factors such that this sum is maximized for some k, $0 \leq k \leq f$. To meet the deadline of task τ_i, the complete execution of task τ_i (including the execution of its recovery copies) must take place within the interval $[0, T_i)$. However, parts of the execution of jobs having higher priority than the priority of task τ_i may take place outside the interval $[0, T_i)$. If the execution of any higher-priority job takes place outside the interval $[0, T_i)$, the execution time beyond time instant T_i must not be accounted for in the calculation of `Load-Factor-HPi`. This is to avoid overestimating the amount of worst-case workload within the interval $[0, T_i)$.

If the maximum sum of `Load-Factor-i` and `Load-Factor-HPi` (for some k) is less than T_i, then task τ_i has enough time to finish its complete execution within $[0, T_i)$. In such case, task τ_i is RM-schedulable. Thus, based on the values of the two workload factors, the exact feasibility condition for task τ_i can be derived. The calculation of the two workload factors (that is, value of `Load-Factor-i` and `Load-Factor-HPi`) are presented in subsection 5.5.1 and subsection 5.5.2, respectively.

A necessary condition for the theories that follows is that the total workload of the primary copy and the maximum number of activated recovery blocks for a particular job does not exceed the period of the task of the job. Following the task model, the WCETs of the primary copy of all the jobs of task τ_i are equal, and is denoted by C_i. Similarly, the WCETs of the b^{th} recovery copy of all the jobs of task τ_i are also equal, and is denoted by R_i^b, for $b = 1, 2, \ldots f$. If a total of x faults occur in task τ_i (one fault occurs in the primary copy of a job of τ_i and $(x-1)$ faults occur in the $(x-1)$ recovery copies of the job of τ_i), then the total execution requirement for tolerating x faults for the job of τ_i is $[C_i + \sum_{b=1}^{x} R_i^b\ ]$. Consequently, the necessary condition for schedulability of task τ_i, for all $i = 1, 2, \ldots n$, is that

$$[C_i + \sum_{b=1}^{f} R_i^b\ ] \leq T_i \tag{5.4}$$

We assume that, for each task $\tau_i \in \Gamma$, the inequality in Eq. (5.4) holds.

5.5.1 Calculation of `Load-Factor-i`

The value of `Load-Factor-i` is the execution time required by task τ_i when $(f - k)$ faults exclusively affect task τ_i, for $k = 0, 1, 2 \ldots f$. If an error is detected after the primary copy of task τ_i finishes execution, then the first recovery copy of task τ_i is ready for execution. If an error is detected at the end of execution of a recovery copy of task τ_i, then the next recovery copy of task τ_i is ready for execution. Remember that the WCET of the b^{th} recovery copy of task τ_i is denoted by R_i^b, for $b = 1, 2 \ldots f$. We denote the total execution time required due to the $(f - k)$ faults affecting the primary and recovery copies of a particular job of task τ_i by

$$C_i^{(f\,-\,k)} = [C_i + \sum_{b=1}^{f-k} R_i^b]$$

The value of `Load-Factor-i` is equal to $C_i^{(f\,-\,k)}$ and has to be calculated for all $k = 0, 1, 2, \ldots f$. The value of $C_i^{(f\,-\,k)}$ can be calculated recursively using Eq. (5.5) as follows:

$$C_i^{(f\,-\,k)} = \begin{cases} C_i & \text{if } (f-k) = 0 \\ \\ R_i^{(f-k)} + C_i^{(f\,-\,k\,-\,1)} & \text{if } (f-k) > 0 \end{cases} \tag{5.5}$$

The value of $C_i^{(f\,-\,k)}$ is set equal to C_i when $(f - k)$ is equal to 0. When $(f - k)$ is equal to 0, only the execution time of the primary copy of task τ_i is considered in Eq. (5.5). In the recursive part of Eq. (5.5), the execution time of the $(f - k)^{th}$ recovery copy of task τ_i and the execution time due to a total of $(f - k - 1)$ faults affecting task τ_i are added to find the value of $C_i^{(f\,-\,k)}$.

To find the worst-case workload within the interval $[0, T_i)$, the value of $C_i^{(f\,-\,k)}$ need to be known for all $k = 0, 1, 2, \ldots f$. Using Eq. (5.5), starting from $k = f, (f-1), \ldots 0$, the value $C_i^{(f\,-\,k)}$ can be calculated for all $(f-k) = 0, 1, 2 \ldots f$ using a total of $O(f)$ addition operations. The task τ_i must complete $C_i^{(f\,-\,k)}$ units of execution within the interval $[0, T_i)$ to tolerate $(f - k)$ faults that exclusively affect task τ_i. The calculation of `Load-Factor-i` is now demonstrated using an example.

Example: *Consider the example task set {τ_1, τ_2, τ_3} given in Table 5.1 for f=2. The first column in Table 5.1 represents the name of each task. The second column represent the period of each task. The WCET of the primary copy of* *each task is given in the third column. The fourth and fifth columns represent the WCET of the first and second (since $f = 2$, at most two faults can occur in one task) recovery copies of each task, respectively. Note that the WCET of a recovery copy of a task may be greater or smaller than the WCET of the primary copy of the corresponding task. Using Eq.* (5.5)*, the amount of execution time*

τ_i	T_i	C_i	R_i^1	R_i^2
τ_1	10	3	2	3
τ_2	15	3	4	2
τ_3	40	9	8	6

Table 5.1: *Example task set with f=2 recovery copies for each task*

required for each task τ_i due to ($f - k$) faults exclusively affecting task τ_i is calculated in Eq. (5.6) *for $k = 0, 1, 2$ as follows:*

$$\begin{aligned}
&\textit{For task } \tau_1, \\
&\quad C_1^0 = C_1 = 3 \\
&\quad C_1^1 = R_1^1 + C_1^0 = 2 + C_1^0 = 2 + 3 = 5 \\
&\quad C_1^2 = R_1^2 + C_1^1 = 3 + C_1^1 = 3 + 5 = 8 \\
&\textit{For task } \tau_2, \\
&\quad C_2^0 = C_2 = 3 \\
&\quad C_2^1 = R_2^1 + C_2^0 = 4 + C_2^0 = 4 + 3 = 7 \\
&\quad C_2^2 = R_2^2 + C_2^1 = 5 + C_2^1 = 2 + 7 = 9 \\
&\textit{For task } \tau_3, \\
&\quad C_3^0 = C_3 = 9 \\
&\quad C_3^1 = R_3^1 + C_3^0 = 8 + C_3^0 = 8 + 9 = 17 \\
&\quad C_3^2 = R_3^2 + C_3^1 = 6 + C_3^1 = 6 + 17 = 23
\end{aligned} \tag{5.6}$$

∎

We will use the example task set in Table 5.1 in the rest of this chapter as our running example. The calculation of the value of `Load-Factor-HPi` is presented in next subsection.

5.5.2 Calculation of `Load-Factor-HPi`

The value of `Load-Factor-HPi` is the maximum execution time completed within $[0, T_i)$ by jobs having higher priority than the priority of task τ_i, when k

faults affect these higher-priority jobs within $[0, T_i)$. If the execution of some of these higher-priority jobs takes place outside $[0, T_i)$, then only the execution that takes place within $[0, T_i)$ must be considered in the calculation of `Load-Factor-HPi`. This is a very crucial issue in determining the value of `Load-Factor-HPi`, as can be seen in the following example.

Example: *Consider the first job of task τ_2 in Table 5.1 that is to be scheduled within the interval $(0, 15]$, since $T_2 = 15$. The jobs $\tau_{1,1}$ and $\tau_{1,2}$ are released within the interval $[0, 15)$ and have higher priority than the priority of task τ_2. The primary copies of the jobs $\tau_{1,1}$ and $\tau_{1,2}$ execute within the interval $[0, 3)$ and $[10, 13)$, respectively. Now, consider a 2-fault pattern in which the first and the second faults occur in the primary and the first recovery copy of job $\tau_{1,2}$, respectively. The detection of the second error in the first recovery copy of job $\tau_{1,2}$ triggers the execution of the second recovery copy of job $\tau_{1,2}$. The first and second recovery copies of job $\tau_{1,2}$ executes within the interval $[13, 15)$ and $[15, 18)$, respectively. The schedule of the jobs $\tau_{1,1}$ and $\tau_{1,2}$ including the execution of the recovery copies for the considered 2-fault pattern is shown in Figure 5.1. The total execution time required by the higher-priority jobs $\tau_{1,1}$ and $\tau_{1,2}$ is $(3 + 3 + 2 + 3) = 11$ time unit (including time for recovery). Notice that, the second recovery copy of job $\tau_{1,2}$ executes outside the interval $[0, T_2)$. The value of maximum execution time by the jobs $\tau_{1,1}$ and $\tau_{1,2}$ within the interval $[0, T_2)$ is equal to $(3 + 3 + 2) = 8$, not 11.* ▮

When calculating the worst-case workload within $[0, T_i)$ to derive the exact RM feasibility condition task task τ_i, the value of `Load-Factor-HPi` must not be overestimated. To calculate the value of `Load-Factor-HPi`, we need to identify the jobs that are released within interval $[0, T_i)$ and have higher priority than the priority of task τ_i. The set of jobs having higher-priority than the priority of task τ_i is denoted by a set HP_i such that each job in set HP_i is released within the interval $[0, T_i)$. That is, the set HP_i is defined in Eq. (5.7) as follows:

$$\mathsf{HP}_i = \{\tau_{p,q} | \; p < i \text{ and } r_{p,q} < T_i\} \tag{5.7}$$

According to Eq. (5.7), if job $\tau_{p,q} \in \mathsf{HP}_i$, then task τ_p has shorter period (that is, higher priority) than task τ_i and the released time of job $\tau_{p,q}$ (that is, value of $r_{p,q}$ defined in Eq. (4.1)) is less than T_i. Each of the higher-priority jobs in set HP_i is eligible for execution at or after its released time within $[0, T_i)$. In the case of our running example, the sets HP_i for $i = 1, 2, 3$ are determined for the three tasks in Table 5.1.

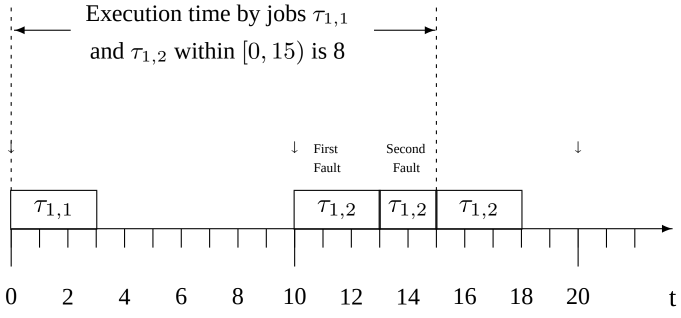

Figure 5.1: *Schedule of jobs $\tau_{1,1}$ and $\tau_{1,2}$. The downward vertical arrows denotes the arrival time of the jobs of τ_1. The two faults occur in the primary copy and the first recovery copy of job $\tau_{1,2}$. The maximum amount of total execution by the jobs $\tau_{1,1}$ and $\tau_{1,2}$ due to the two faults is equal to 11. However, the amount of maximum total execution by the jobs $\tau_{1,1}$ and $\tau_{1,2}$ within the interval $[0, 15)$ is 8, not 11.*

Example: *Using Eq.* (5.7) *for the task set in Table 5.1 we have,*

$$\begin{aligned} \mathsf{HP}_1 &= \emptyset \\ \mathsf{HP}_2 &= \{\ \tau_{1,1}\ ,\tau_{1,2}\ \} \qquad (5.8) \\ \mathsf{HP}_3 &= \{\tau_{1,1}, \tau_{1,2}, \tau_{1,3}, \tau_{1,4}, \tau_{2,1}, \tau_{2,2}, \tau_{2,3}\} \end{aligned}$$

∎

Remember that N is the maximum number of jobs that are released within the time interval $[0, T_{max})$. Therefore, the number of jobs having higher priority than the priority of task τ_i that are released within $[0, T_i)$ is at most N. If the released time of a higher-priority job $\tau_{p,q}$ is earlier than T_i, then job $\tau_{p,q}$ is included in set HP_i. Therefore, the time complexity to find the set HP_i is $O(N)$.

When considering the feasibility of task τ_i, we need to calculate the value of `Load-Factor-HPi` for a k-fault pattern such that the k faults affect the jobs in set HP_i, for $k = 0, 1, \ldots f$. The value of `Load-Factor-HPi` is a measure of how much computation is completed within the interval $[0, T_i)$ by the higher-priority jobs in set HP_i due to the k-fault pattern. The amount of computation completed by the jobs in set HP_i within $[0, T_i)$ depends on how much workload is requested by the jobs in HP_i due to the k-fault pattern. Aydin in [Ayd07] used a dynamic programming technique to compute the maximum workload requested by a set of aperiodic tasks due to a k-fault pattern. Using an approach similar to that in [Ayd07], we determine the maximum workload requested by a set of higher-priority jobs that are all released at a particular time instant t within the time interval $[0, T_i)$.

We denote the maximum workload requested by a set of jobs in set A, all released at a particular time instant t, by function $L_k(A)$ for a k-fault pattern[1]. Note that the value of $L_k(A)$ is the maximum workload requested by the jobs in set A, not the actual amount of execution by the jobs in set A within $[0, T_i)$. the The function $L_k(A)$ is defined recursively (similar to [Ayd07], but the difference being that all the jobs in set A have the same released time) in Eq. (5.9) and Eq. (5.10). The basis of the recursion is defined in Eq. (5.9) considering exactly one job $\tau_{x,y}$ in set A, for $k = 0, 1, 2, \ldots f$, as follows

$$L_k(\{\tau_{x,y}\}) = C_x^k \tag{5.9}$$

The value of $L_k(\{\tau_{x,y}\})$ represents the amount of execution time requested by job $\tau_{x,y}$ when k faults exclusively affect the primary and recovery copies of job $\tau_{x,y}$. Remember that the value of C_x^k is defined in Eq. (5.5) as the maximum amount of execution time required by the task τ_x when k faults exclusively affect a particular job of this task. The value of C_x^k in the right hand side of Eq. (5.9) can be calculated using Eq. (5.5) in $O(f)$ time, for all $k = 0, 1, 2 \ldots f$.

By assuming that the value of $L_k(A)$ is known, we compute the value of $L_k(\text{A} \cup \{\tau_{x,y}\})$ recursively, for $k = 0, 1, 2 \ldots f$, as follows:

$$L_k(\text{A} \cup \{\tau_{x,y}\}) = \overset{k}{\underset{q=0}{max}} \left\{ L_q(A) + L_{k-q}(\{\tau_{x,y}\}) \right\} \tag{5.10}$$

In Eq. (5.10), the value of $L_k(\text{A} \cup \{\tau_{x,y}\})$ is maximum for one of the $(k+1)$ possible values of q, for $0 \leq q \leq k$. The value of q is selected such that, if q faults occur in the jobs in set A and $(k-q)$ faults occur exclusively in job $\tau_{x,y}$, then $L_k(\text{A} \cup \{\tau_{x,y}\})$ is at its maximum for some q, $0 \leq q \leq k$. The working of Eq. (5.10) is now demonstrated using an example.

Example: *Consider the task τ_3 given in Table 5.1. The jobs, having higher priority than the priority of task τ_3, that are released at time $t = 0$ are in the set A={$\tau_{1,1}$, $\tau_{2,1}$}. If we want to determine the maximum workload requested by the higher-priority jobs in set A={$\tau_{1,1}$, $\tau_{2,1}$} due to a k-fault pattern, then we need to calculate the value of $L_k(A)$. To calculate $L_k(A)$, we have to evaluate the base in Eq.* (5.9) *for each of the jobs in set A considering occurrences of k faults*

[1]The jobs in set A are released at time t. The time instant t is not included in function $L_k(A)$ and can be understood from the context. Although the value of $L_k(A)$ can be calculated independent of t, the context t is important for the schedulability analysis.

exclusively affecting that job. Since f is equal to 2, the possible values of k are 0, 1 and 2. According to Eq. (5.6)*, the maximum execution time required for job $\tau_{1,1}$ is C_1^0 =3, C_1^1 =5 and C_1^2 =8 for $k = 0$, $k = 1$ and $k = 2$ faults exclusively affecting job $\tau_{1,1}$, respectively. The maximum execution time required for job $\tau_{2,1}$ is C_2^0 =3, C_2^1 =7 and C_2^2 =9 for $k = 0$, $k = 1$ and $k = 2$ faults exclusively affecting job $\tau_{2,1}$, respectively, according to Eq.* (5.6)*. Using the base of the recursion in Eq.* (5.9) *we have,*

$$L_0(\{\tau_{1,1}\}) = C_1^0 = 3 \quad L_1(\{\tau_{1,1}\}) = C_1^1 = 5 \quad L_2(\{\tau_{1,1}\}) = C_1^2 = 8$$

$$L_0(\{\tau_{2,1}\}) = C_2^0 = 3 \quad L_1(\{\tau_{2,1}\}) = C_2^1 = 7 \quad L_2(\{\tau_{2,1}\}) = C_2^2 = 9$$

Using Eq. (5.10)*, the value of $L_k(A)$ for $k = 0, 1, 2$ and A={$\tau_{1,1}$,$\tau_{2,1}$} can be calculated as follows:*

$$\begin{aligned} L_0(\{\tau_{1,1}, \tau_{2,1}\}) &= \max_{q=0}^{0} \{L_q(\{\tau_{1,1}\}) + L_{0-q}(\{\tau_{2,1}\})\} \\ &= L_0(\{\tau_{1,1}\}) + L_0(\{\tau_{2,1}\}) \\ &= 3 + 3 = 6 \end{aligned}$$

$$\begin{aligned} L_1(\{\tau_{1,1}, \tau_{2,1}\}) &= \max_{q=0}^{1} \{L_q(\{\tau_{1,1}\}) + L_{1-q}(\{\tau_{2,1}\})\} \\ &= max\{\ L_0(\{\tau_{1,1}\}) + L_1(\{\tau_{2,1}\})\ , \\ &\qquad L_1(\{\tau_{1,1}\}) + L_0(\{\tau_{2,1}\})\} \\ &= max\ \{3 + 7, 5 + 3\} = 10 \end{aligned}$$

$$\begin{aligned} L_2(\{\tau_{1,1}, \tau_{2,1}\}) &= \max_{q=0}^{2} \{L_q(\{\tau_{1,1}\}) + L_{2-q}(\{\tau_{2,1}\})\} \\ &= max\{\ L_0(\{\tau_{1,1}\}) +\ L_2(\{\tau_{2,1}\})\ , \\ &\qquad L_1(\{\tau_{1,1}\}) + L_1(\{\tau_{2,1}\})\ , \\ &\qquad L_2(\{\tau_{1,1}\}) + L_0(\{\tau_{2,1}\})\} \\ &= max\ \{3 + 9, 5 + 7, 8 + 3\} = 12 \end{aligned}$$

The maximum amount of workload requested by the jobs in set A={$\tau_{1,1}$,$\tau_{2,1}$} is $L_0(A)$=6, $L_1(A)$=10, and $L_2(A)$=12 for $k = 0, 1$ and 2 fault patterns, respectively. ∎

Time complexity to calculate L_k(A $\cup$ {$\tau_{x,y}$}): There are total $(|A|+1)$ jobs in set ($A\cup$ $\{\tau_{x,y}\}$). For each one of the $(|A|+1)$ jobs, evaluating the base case using Eq. (5.9) can be done using Eq. (5.5) in O(f) steps for all $k = 0, 1, 2, \ldots f$. Therefore, evaluating the base for all the jobs in set ($A\cup$ $\{\tau_{x,y}\}$) requires [$(|A|+1) \cdot O(f)$]= $O(|A| \cdot f)$ operations.

For the recursive step, if the value of $L_k(A)$ is known, then there are $(k+1)$ possibilities for the selection of q in Eq. (5.10) to compute $L_k(\text{A} \cup \{\tau_{x,y}\})$ for a given k, $0 \leq k \leq f$. Therefore, computing $L_k(\text{A} \cup \{\tau_{x,y}\})$ requires $O(k)$ operations ($k+1$ additions and k comparisons) for a particular k. Given that the value of $L_k(A)$ is known for all $k = 0, 1, 2, \ldots f$, then computing $L_k(\text{A} \cup \{\tau_{x,y}\})$ requires total $O(0+1+2\ldots f)$=$O(f^2)$ operations, for all $k = 0, 1, \ldots f$.

Starting with one job in set A, a new job $\tau_{x,y}$ is considered when computing the value of $L_k(\text{A} \cup \{\tau_{x,y}\})$. By including one job $\tau_{x,y}$ in the set A at each step, the set ($A\cup\{\tau_{x,y}\}$) is formed. Therefore, for all the jobs in the set ($A\cup\{\tau_{x,y}\}$), the total time complexity to recursively compute the value of $L_k(\text{A} \cup \{\tau_{x,y}\})$ is equal to $[(|A|+1) \cdot O(f^2)] = O(|A| \cdot f^2)$.

Therefore, the total time complexity for the base and recursive steps to compute $L_k(\text{A} \cup \{\tau_{x,y}\})$ is $O(|A| \cdot f + |A| \cdot f^2)$= $O(|A| \cdot f^2)$. ▮

As mentioned before, the value of `Load-Factor-HPi` is the maximum execution completed within the interval $[0, T_i)$ by the jobs having higher priority than the priority of task τ_i for a k-fault pattern. The maximum execution completed by the set of higher-priority jobs within $[0, T_i)$ may not be same as the maximum workload requested by this set of higher-priority jobs for a k-fault pattern. The value of $L_k(A)$ is calculated considering that all the jobs in set A are released at the same time, say at time t. Consider that the set A contains the jobs having higher priority than the priority of task τ_i and all the jobs in set A are released at time t. If the value of $L_k(A)$ is greater than ($T_i - t$), then the maximum amount of work completed by the higher-priority jobs in set A within the interval $[0, T_i)$ is at most ($T_i - t$) using the work-conserving algorithm RM. If $L_k(A)$ is less than or equal to ($T_i - t$), then the maximum amount of work that can be completed by the jobs in set A within the interval $[0, T_i)$ is at most $L_k(A)$.

In order to find the amount of execution completed by the jobs of the higher-priority tasks within the time interval $[0, T_i)$, the higher-priority jobs released at different time instants within the time interval $[0, T_i)$ are *composed*. A composed task is not an actual task in the system rather a way to represent the execution of a collection of higher-priority jobs in a compact (composed) way. The execution time of a composed task (formally defined later) represents the maximum amount of execution within the interval $[0, T_i)$ if the jobs represented by the composed tasks have exclusive access to the processor within the interval $[0, T_i)$. In other words, the execution time of a composed task is the amount of maximum execution within the interval $[0, T_i)$ if only the jobs represented by the composed task are allowed to execute within the interval $[0, T_i)$.

The composition of the higher-priority tasks are done in two steps: first by *vertical composition* and then by *horizontal composition*. Each vertically-composed task abstracts[2] the higher-priority jobs that are all released at a particular time instant within $[0, T_i)$. Each horizontally-composed task abstracts the higher-priority jobs that are abstracted by more than one vertically-composed task.

Vertical Composition

Consider a set of all jobs that are released at time instant t, $t < T_i$ and have higher priority than the priority of task τ_i. To compactly represent these higher-priority jobs, we define a vertically-composed task $V_{\{t\}}$ for time instant t such that the composed task $V_{\{t\}}$ abstracts the set of higher-priority jobs that are all released at time t, such that $0 \leq t < T_i$.

The execution time of the composed task $V_{\{t\}}$ (formally calculated later) denotes the maximum amount of execution that can be completed within $[0, T_i)$ by the higher-priority jobs that are released at time t such that only the jobs represented by $V_{\{t\}}$ are allowed to execute within $[0, T_i)$.

One vertically-composed task is formed for each time instant within $[0, T_i)$ at which new higher-priority jobs are released.

Example: *Consider the feasibility of task τ_3 in Table 5.1. The first job of task τ_3 is released at time 0 and has its deadline by time $T_3 = 40$. The tasks τ_1 and τ_2 are the higher-priority tasks of task τ_3. The release of the higher-priority jobs at different time instants within the interval $[0, 40)$ is shown in Figure 5.2 using downward arrows. New jobs of the higher-priority tasks are released at time instants 0, 10, 15, 20 and 30. At each of these five time instants, a vertically-composed task is formed (that abstracts the released jobs shown in each oval in Figure 5.2). The five composed tasks are denoted by $V_{\{0\}}$,$V_{\{10\}}$,$V_{\{15\}}$,$V_{\{20\}}$ and $V_{\{30\}}$ in Figure 5.2.* ∎

To form the vertically-composed tasks, we need to determine the different time points within $[0, T_i)$ where new jobs of the higher-priority tasks are released. The set of time points, denoted by S_i, where jobs having higher priority

[2]We use this as a short form for ”represents an abstraction of”.

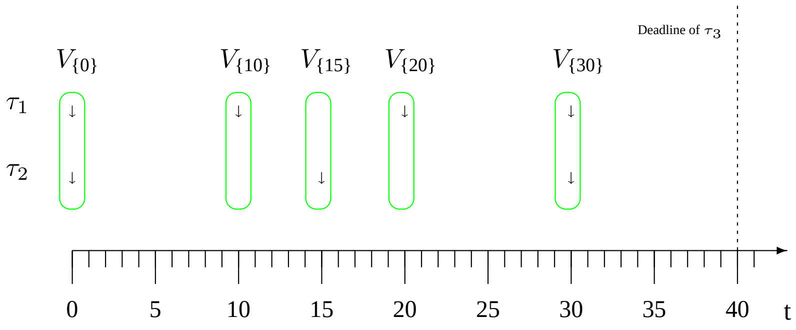

Figure 5.2: *Five vertically-composed tasks are shown using vertically long ovals at time instants 0, 10, 15, 20, and 30. Each vertically-composed task at time t abstracts all the newly released higher-priority jobs of task τ_3 that are released at time t within the time interval* $[0, 40)$.

than the priority of task τ_i are released within the interval $[0, T_i)$ is given by Eq. (5.11) as follows:

$$S_i = \{k \cdot T_j \mid j = 1 \ldots (i-1), k = 0 \ldots \lfloor \frac{T_i}{T_j} \rfloor\} - \{T_i\} \tag{5.11}$$

The time points in set S_i are less than T_i and are nonnegative integer multiples of the periods of the higher-priority task τ_j for $j = 1, 2, \ldots (i-1)$. Since higher-priority jobs released at or beyond time instant T_i will not execute prior to time instant T_i, it is necessary that all the time points in set S_i are less than T_i (that is, deadline of the first job of task τ_i). At each of the time points in set S_i, new higher-priority jobs are released.

Example: *Consider the task set given in Table 5.1. Using Eq.* (5.11) *we have,*

$$\begin{aligned} S_1 &= \{\} \\ S_2 &= \{0, 10\} - \{15\} = \{0, 10\} \\ S_3 &= \{0, 10, 15, 20, 30, 40\} - \{40\} = \{0, 10, 15, 20, 30\} \end{aligned} \tag{5.12}$$ ∎

The jobs having higher priority than that of task τ_i are released at each of the time points in set S_i. Remember that there are at most N jobs released within any interval of length T_{max}. The time points in S_i are integer multiples of the periods of the higher-priority tasks. Therefore, the run-time complexity to compute S_i is $O(N)$.

When considering the feasibility of task τ_i, at each time point in set S_i some new higher-priority jobs of task τ_i are released. For each time point $s \in S_i$, a vertically-composed task $V_{\{\mathrm{s}\}}$ is formed. In the case of our running example, when considering the feasibility condition for task τ_3, one vertically-composed task for each time point $s \in S_3$ ={0,10,15,20,30} is formed (see the five vertically-composed tasks in Figure 5.2).

The vertically-composed task $V_{\{\mathrm{s}\}}$ at time $s \in S_i$ abstracts the set of higher-priority jobs (that is, jobs from set HP_i) that are released at time s. To find the execution time of a vertically-composed task at time $s \in S_i$, we need to know the higher-priority jobs in set HP_i that are released at time instant s. The set $\mathtt{Rel}_{i,s}$ denotes the higher-priority jobs of task τ_i that are released at time s. The set $\mathtt{Rel}_{i,s}$ is given in Eq. (5.13) as follows:

$$\mathtt{Rel}_{i,s} = \{\tau_{p,q} |\ \tau_{p,q} \in \mathsf{HP}_i \text{ and } r_{p,q} = s\} \tag{5.13}$$

The set $\mathtt{Rel}_{i,s}$ contains the jobs that are released at time s and are of higher priority than task τ_i. If job $\tau_{p,q}$ is in set $\mathtt{Rel}_{i,s}$, then job $\tau_{p,q}$ is in set HP_i and the released time of job $\tau_{p,q}$ is equal to time instant s, that is, s is equal to $r_{p,q}$. The condition in Eq. (5.13) is to be evaluated for each job in set HP_i. Since there are at most N jobs released within any time interval of length T_{max}, the number of jobs in set HP_i is $O(N)$. The job $\tau_{p,q} \in \mathsf{HP}_i$ is stored in set $\mathtt{Rel}_{i,s}$ if the released time $r_{p,q}$ is equal to s. By selecting one by one job $\tau_{p,q}$ from set HP_i, the job $\tau_{p,q}$ can be stored in the appropriate set $\mathtt{Rel}_{i,s}$ such that the released time $r_{p,q}$ of job $\tau_{p,q}$ is equal to s. Therefore, the time complexity to find $\mathtt{Rel}_{i,s}$ for all $s \in S_i$ is equal to $O(N)$.

Example: *Consider the example task set in Table 5.1. Since there are no higher-priority jobs of task τ_1, the set $\mathsf{HP}_1 = \emptyset$. For tasks τ_2 and τ_3 we have S_2 ={0, 10} and S_3 ={0, 10, 15, 20, 30}, respectively, according to Eq.* (5.12). *The set, $\mathtt{Rel}_{i,s}$, of higher-priority jobs released at different time instant $s \in S_i$ for $i = 2$ and $i = 3$ are given in Eq.* (5.14) *as follows:*

$$\begin{aligned}
\mathtt{Rel}_{2,0} &= \{\tau_{1,1}\} \\
\mathtt{Rel}_{2,10} &= \{\tau_{1,2}\} \\
\mathtt{Rel}_{3,0} &= \{\tau_{1,1}, \tau_{2,1}\} \\
\mathtt{Rel}_{3,10} &= \{\tau_{1,2}\} \\
\mathtt{Rel}_{3,15} &= \{\tau_{2,2}\} \\
\mathtt{Rel}_{3,20} &= \{\tau_{1,3}\} \\
\mathtt{Rel}_{3,30} &= \{\tau_{2,3}, \tau_{1,4}\}
\end{aligned} \tag{5.14}$$

■

The jobs in set $\texttt{Rel}_{i,s}$ are of higher priority than that of the task τ_i and all these higher-priority jobs are released at time s. For each $s \in S_i$, the vertically-composed task $V_{\{s\}}$ abstracts the jobs in set $\texttt{Rel}_{i,s}$. We now concentrate on calculating the execution time of a vertically-composed task $V_{\{s\}}$.

The execution time of the vertically-composed task $V_{\{s\}}$ is denoted by the function `w(k,{s})` for a k-fault pattern affecting the jobs in set $\texttt{Rel}_{i,s}$. If no jobs other than the jobs in set $\texttt{Rel}_{i,s}$ are allowed to execute within the interval $[0, T_i)$, then the value of `w(k,{s})` represents the maximum amount of execution that can be completed by the jobs in set $\texttt{Rel}_{i,s}$ within the interval $[0, T_i)$ for a k-fault pattern.

The value of $L_k(\texttt{Rel}_{i,s})$ is the maximum amount of workload requested by the jobs abstracted by the vertically-composed task $V_{\{s\}}$. The set of jobs released at time s can complete, using work conserving algorithm RM, at most $(T_i - s)$ amount of work within $[0, T_i)$ if $L_k(\texttt{Rel}_{i,s})$ is greater than $(T_i - s)$. Otherwise, the maximum amount of work completed by the set of jobs released at time s is $L_k(\texttt{Rel}_{i,s})$. To this end, the execution time of $V_{\{s\}}$ for $k = 0, 1, 2, \ldots f$ is defined in Eq. (5.15) as follows:

$$\texttt{w(k,\{s\})} = min \; \{L_k(\texttt{Rel}_{i,s}) \; , (T_i \; - s) \; \} \tag{5.15}$$

The value `w(k,{s})` represents the maximum amount of execution completed by the jobs released at time s within the interval $[0, T_i)$ if no jobs other than the jobs in set $\texttt{Rel}_{i,s}$ are allowed to execute within the interval $[0, T_i)$. The calculation of `w(k,{s})` is shown next for our running example.

Example: *Consider the task set in Table 5.1. When considering the feasibility of task τ_1, there is no higher-priority jobs of task τ_1. Therefore, no vertically-composed task is formed since set S_1 is empty.*

When considering the feasibility of task τ_2, there are higher-priority jobs that are released within $[0, T_2)$. To find the vertical compositions of the higher-priority jobs, we use the following information:

$S_2 = \{0, 10\}$ from Eq. (5.12)

$T_2 = 15$ from Table 5.1

$\texttt{Rel}_{2,0} = \{ \; \tau_{1,1}\}$ for $s = 0$ from Eq. (5.14)

$\texttt{Rel}_{2,10} = \{ \; \tau_{1,2}\}$ for $s = 10$ from Eq. (5.14)

Two vertically-composed tasks are formed since there are two time points in *set $S_2 = \{0, 10\}$. The two vertically-composed tasks are $V_{\{0\}}$ and $V_{\{10\}}$. For each vertically-composed task, the amount of execution time within the interval $[0, T_2)$ can be determined for $k = 0, 1, 2$ (since $f = 2$) using Eq.* (5.15). *The value of `w(k,{s})` for the composed task $V_{\{s\}}$ using Eq.* (5.15) *is calculated in Table 5.2 for $k = 0, 1, 2$ and $s = 0, 10$. When considering the feasibility*

For $s = 0$ and $k = 0$	For $s = 10$ and $k = 0$
`w(0,0)` $= min\{L_0(\mathtt{Rel}_{2,0}), T_i - 0\}$ $= min\{L_0(\mathtt{Rel}_{2,0}), 15 - 0\}$ $= min\{L_0(\{\tau_{1,1}\}), 15\}$ $= min\{3, 15\} = 3$	`w(0,10)` $= min\{L_0(\mathtt{Rel}_{2,10}), T_i - 10\}$ $= min\{L_0(\mathtt{Rel}_{2,10}), 15 - 10\}$ $= min\{L_0(\{\tau_{1,2}\}), 5\}$ $= min\{3, 5\} = 3$
For $s = 0$ and $k = 1$	For $s = 10$ and $k = 1$
`w(1,0)` $= min\{L_1(\mathtt{Rel}_{2,0}), T_i - 0\}$ $= min\{L_1(\mathtt{Rel}_{2,0}), 15 - 0\}$ $= min\{L_1(\{\tau_{1,1}\}), 15\}$ $= min\{5, 15\} = 5$	`w(1,10)` $= min\{L_1(\mathtt{Rel}_{2,10}), T_i - 10\}$ $= min\{L_1(\mathtt{Rel}_{2,10}), 15 - 10\}$ $= min\{L_1(\{\tau_{1,2}\}), 5\}$ $= min\{5, 5\} = 5$
For $s = 0$ and $k = 2$	For $s = 10$ and $k = 2$
`w(2,0)` $= min\{L_2(\mathtt{Rel}_{2,0}), T_i - 0\}$ $= min\{L_2(\mathtt{Rel}_{2,0}), 15 - 0\}$ $= min\{L_2(\{\tau_{1,1}\}), 15\}$ $= min\{8, 15\} = 8$	`w(2,10)` $= min\{L_2(\mathtt{Rel}_{2,10}), T_i - 10\}$ $= min\{L_2(\mathtt{Rel}_{2,10}), 15 - 10\}$ $= min\{L_2(\{\tau_{1,2}\}), 5\}$ $= min\{8, 5\} = 5$

Table 5.2: *Calculation of `w(k,{s})` for vertical composition at each $s \in S_2$ for $k = 0, 1, 2$. The left column show the execution time `w(k,{0})` of the vertically-composed task $V_{\{0\}}$ for $k = 0, 1, 2$ faults and the right column show the execution time `w(k,{10})` of the vertically-composed task $V_{\{10\}}$ for $k = 0, 1, 2$ faults.*

of task τ_3, there are higher-priority jobs that are eligible for execution within $[0, T_3)$. To find the vertical compositions of the higher-priority jobs, we use the *following information:*

$$S_3 = \{0, 10, 15, 20, 30\} \quad \text{from Eq. (5.12)}$$
$$T_3 = 40 \quad \text{from Table 5.1}$$
$$\texttt{Rel}_{3,0} = \{\tau_{1,1}, \tau_{2,1}\} \quad \text{for } s = 0 \text{ from Eq. (5.14)}$$
$$\texttt{Rel}_{3,10} = \{\tau_{1,2}\} \quad \text{for } s = 10 \text{ from Eq. (5.14)}$$
$$\texttt{Rel}_{3,15} = \{\tau_{2,2}\} \quad \text{for } s = 15 \text{ from Eq. (5.14)}$$
$$\texttt{Rel}_{3,20} = \{\tau_{1,3}\} \quad \text{for } s = 20 \text{ from Eq. (5.14)}$$
$$\texttt{Rel}_{3,30} = \{\tau_{1,4}, \tau_{2,3}\} \quad \text{for } s = 30 \text{ from Eq. (5.14)}$$

Five vertically-composed tasks are formed since there are five time points in S_3 at each of which new higher-priority jobs are released. The five vertically-composed tasks are $V_{\{0\}}$, $V_{\{10\}}$, $V_{\{15\}}$, $V_{\{20\}}$ and $V_{\{30\}}$. For each vertically-composed task $V_{\{s\}}$, the value of `w(k,{s})` for $k = 0, 1, 2$ is given in each row of Table 5.3 for $k = 0, 1, 2$ and $s = 0, 10, 15, 20, 30$. ■

$V_{\{s\}}$	$k = 0$	$k = 1$	$k = 2$
$V_{\{0\}}$	`w(0,0)`=6	`w(1,0)`=10	`w(2,0)`=12
$V_{\{10\}}$	`w(0,10)`=3	`w(1,10)`=5	`w(2,10)`=8
$V_{\{15\}}$	`w(0,15)`=3	`w(1,15)`=7	`w(2,15)`=9
$V_{\{20\}}$	`w(0,20)`=3	`w(1,20)`=5	`w(2,20)`=8
$V_{\{30\}}$	`w(0,30)`=6	`w(1,30)`=10	`w(2,30)`=10

Table 5.3: *The value of `w(k,{s})` for each $s \in S_3$ and for $k = 0, 1, 2$. The k faults affect the higher-priority jobs that are released at time $s \in S_3$.*

Run-time complexity for vertical composition: Calculating $\texttt{Rel}_{i,s}$ for all $s \in S_i$ need total $O(N)$ operations. Calculating $L_k(\texttt{Rel}_{i,s})$ for set $\texttt{Rel}_{i,s}$ requires $O(|\texttt{Rel}_{i,s}| \cdot f^2)$ operations for all $k = 0, 1, 2, \ldots f$. There are at most N jobs that are released within any time interval of length T_{max}. Therefore, the number of total jobs having higher priority than the priority of task τ_i that are released in all the time points in set S_i is equal to $O(N)$. In other words, $\sum_{s \in S_i} |\texttt{Rel}_{i,s}| = O(N)$.

Therefore, the computational complexity of all the vertical compositions in all time points $s \in S_i$ is $[O(N)+O(\sum_{s \in S_i} |\texttt{Rel}_{i,s}| \cdot f^2)]=O(N \cdot f^2)$. ■

For each $s \in S_i$, a vertically-composed task $V_{\{s\}}$ is formed. The vertically-composed task $V_{\{s\}}$ has execution time `w(k,{s})` considering a k-fault pattern for $k = 0, 1, 2 \dots f$. Within the interval $[0, T_i)$, there may be more than one vertically-composed task. In our running example, there are five vertically-composed task within $[0, T_3)$ as shown in Figure 5.2. The higher-priority jobs represented by two or more vertically-composed tasks will execute within the interval $[0, T_i)$. Now, we observed that the execution of the jobs represented by two or more vertically-composed tasks may not be completely independent. Some jobs in one vertically-composed task may interfere or be interfered by the execution of some jobs in another vertically-composed task within $[0, T_i)$. By considering such effect of one composed task over another, the vertically-composed tasks can be further composed using horizontal composition so as to calculate the value of `Load-Factor-HPi`.

Horizontal Composition

A horizontally-composed task is formed by composing two or more vertically-composed tasks. To see how this composition works, consider two different time points s_1 and s_2 in set S_i such that $s_1 < s_2$. For these two time points, two vertically-composed tasks $V_{\{s_1\}}$ and $V_{\{s_2\}}$ are formed. A horizontally-composed task, denoted by $H_{\{s_1, s_2\}}$, is formed by composing the two vertically composed tasks $V_{\{s_1\}}$ and $V_{\{s_2\}}$. The task $H_{\{s_1, s_2\}}$ abstracts all the jobs of the higher-priority tasks than the priority of task τ_i that are released at time s_1 and s_2.

The execution time of this new horizontally-composed task $H_{\{s_1, s_2\}}$ is denoted by `w(k,`$\{s_1, s_2\}$`)` and must not be greater than ($T_i - s_1$). This is because the earliest time at which the jobs represented by the the composed task $H_{\{s_1, s_2\}}$ can start execution is at time s_1 since $s_1 < s_2$. Note that, if $0 \in \{s_1, s_2\}$, then `w(k,`$\{s_1, s_2\}$`)` must not be greater than T_i. The value of `w(k,`$\{s_1, s_2\}$`)` represents the maximum execution exclusively by the jobs released at time s_1 and s_2 within the time interval $[0, T_i)$.

When considering the feasibility of task τ_i, there are a total of $|S_i|$ time instants at each of which a vertically-composed task is formed. To calculate `Load-Factor-HPi`, we have to find the final horizontally-composed task H_{S_i} with execution time `w(k,`S_i`)` for $k = 0, 1, 2 \dots f$. The value of `w(k,`S_i`)` is the amount of execution completed by the higher-priority jobs that are released within the time instants in set S_i. Since set S_i contains all the time instants where jobs of higher-priority task are released, the value of `w(k,`S_i`)` is `Load-Factor-HPi`.

To find the horizontally-composed task H_{S_i}, we need ($|S_i| - 1$) horizontal compositions. Starting with two vertically-composed tasks a new horizontally-composed task is formed. This horizontally-composed task is further composed with a third vertically-composed task into a new horizontally-composed task. This process continues until all the vertically-composed tasks are considered in the horizontal compositions. Note that a vertically-composed task has no priority associated with it. The jobs abstracted by a vertically-composed tasks have RM priority. Therefore, the order of execution of the jobs abstracted by a horizontally-composed task is determined by the RM priorities of the jobs that are abstracted by the constituent vertically-composed tasks.

The first horizontally-composed task abstracts all higher-priority jobs released at two points in set S_i. The last (final) horizontally-composed task abstracts all the jobs that are released at all time points in set S_i. For example, the five vertically-composed tasks in Figure 5.2 are composed horizontally and the four horizontally-composed tasks are shown in Figure 5.3.

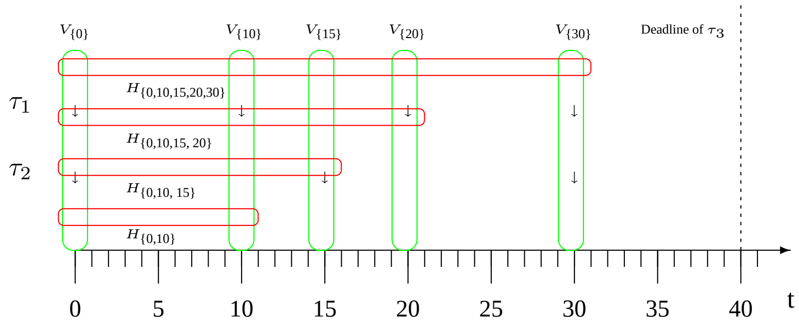

Figure 5.3: *Four horizontal compositions (horizontally longer ovals) are shown for the five vertically-composed tasks (vertically longer ovals). The four horizontally-composed tasks are $H_{\{0,10\}}$, $H_{\{0,10, 15\}}$, $H_{\{0,10, 15, 20\}}$ and $H_{\{0,10,15,20,30\}}$. The composed task $H_{\{0,10\}}$ represents the first horizontally-composed task and the composed task $H_{\{0,10,15,20,30\}}$ represents the final horizontally-composed task. The execution time of $H_{\{0,10,15,20,30\}}$ is the value of `Load-Factor-HPi`.*

We now concentrate on finding the execution time of a horizontally-composed task. If there are c time points in the set S_i, then we denote the set S_i as S_i=$\{s_1, s_2 \dots s_c\}$. According to Eq. (5.11), the set S_i contains the time point 0. Without loss of generality, we assume $s_1 = 0$. We denote the first x time points in S_i by set

$$\texttt{p(x)} = \{s_l \mid l \leq x \text{ and } s_l \in S_i\}$$

Therefore, the set `p(x)` =$\{s_1, s_2 \dots s_x\}$ for $x = 1, 2 \dots c$. For example, we have `p(1)` =$\{s_1\}$=$\{0\}$ and `p(c)` =$\{s_1, s_2 \dots s_c\}$=S_i.

We start composing the first two vertically-composed tasks horizontally. The horizontal composition of the first two vertically-composed tasks $V_{\{s_1\}}$ and $V_{\{s_2\}}$ is denoted by the composed task $H_{\texttt{p(2)}}$=$H_{\{s_1,s_2\}}$. The execution time of $V_{\{s_1\}}$ and $V_{\{s_2\}}$ are $\texttt{w(k,}\{s_1\}\texttt{)}$ and $\texttt{w(k,}\{s_2\}\texttt{)}$, respectively (given by Eq.(5.15)). The execution time of $H_{\texttt{p(2)}}$ is denoted by $\texttt{w(k,p(2))}$ = $\texttt{w(k,}\{s_1, s_2\}\texttt{)}$ and is given in Eq. (5.16), for $k = 0, 1, 2, \ldots f$, as follows:

$$\texttt{w(k,p(2))} = \max_{q=0}^{k} \left\{ min \Big\{ [\texttt{w(q,}\{s_1\}\texttt{)} + \texttt{w(k-q,}\{s_2\}\texttt{)}],\ T_i \Big\} \right\} \tag{5.16}$$

The calculation of the value of $\texttt{w(k,p(2))}$ in Eq. (5.16) considers the sum of the execution time of tasks $V_{\{s_1\}}$ and $V_{\{s_2\}}$ for q and $(k - q)$ fault pattern, respectively, such that the sum is maximized for some q, $0 \leq q \leq k$. Since the amount of execution within the interval $[0, T_i)$ by the higher-priority jobs released at time s_1 and s_2 can not be greater than ($T_i - s_1$) = ($T_i - 0$) = T_i, the minimum of this sum for some q and T_i is determined to be the value of $\texttt{w(k,p(2))}$ in Eq. (5.16). This is because the earliest time that higher-priority jobs can start execution is at time $s_1 = 0$.

By assuming that the value of $\texttt{w(k,p(x))}$ is known for the horizontally-composed tasks $H_{\texttt{p(x)}}$, we define a new horizontally-composed task $H_{\texttt{p(x+1)}}$ which is equivalent to $H_{\texttt{p(x)} \cup \{s_{x+1}\}}$. The execution time $\texttt{w(k,p(x+1))}$ of the horizontally-composed task $H_{\texttt{p(x+1)}}$ is given in Eq. (5.17), for $k = 0, 1, 2, \ldots f$, as follows:

$$\texttt{w(k,p(x+1))} = \max_{q=0}^{k} \left\{ min \Big\{ [\texttt{w(q,p(x))} + \texttt{w(k-q,}\{s_{x+1}\}\texttt{)}],\ T_i \Big\} \right\} \tag{5.17}$$

The execution time $\texttt{w(k,p(x+1))}$ of the new horizontally-composed task $H_{\texttt{p(x+1)}}$ is calculated by finding the sum of the execution time of the horizontally composed task $H_{\texttt{p(x)}}$ and the execution time of the new vertically-composed task $V_{\{s_{x+1}\}}$. The value of this sum is maximized by considering q faults in task $H_{\texttt{p(x)}}$ and ($k - q$) faults in task $V_{\{s_{x+1}\}}$, for some q, $0 \leq q \leq k$. Since the amount of execution within the interval $[0, T_i)$ can not be greater than ($T_i - s_1$) = ($T_i - 0$) = T_i, the minimum of this sum (for some q) and T_i is determined to be the value of $\texttt{w(k,p(x+1))}$ in Eq. (5.17).

Using Eq. (5.17), we can find the execution time $\texttt{w(k,}S_i\texttt{)}$ of the final horizontally-composed task H_{S_i}=$H_{\texttt{p(}|S_i|\texttt{)}}$ for $k = 0, 1, 2 \ldots f$. The value of $\texttt{w(k,}S_i\texttt{)}$ is the value of $\texttt{Load-Factor-HPi}$ for $k = 0, 1, 2 \ldots f$. Before we demonstrate the calculation of the execution time of horizontally-composed task using an example, we analyze the run time complexity of calculating the execution time of the horizontally composed tasks.

Run time complexity of horizontal compositions: There are total $|S_i| - 1$ horizontal composition for $|S_i|$ vertically-composed tasks when considering the feasibility of task τ_i. When considering the feasibility of a task τ_i, for each horizontal composition, there are $(k+1)$ possibilities for q, $0 \leq q \leq k$, in Eq. (5.17). For each value of q, there is one addition and one comparison operation. Therefore, total $(2 \cdot (k+1))$ operations are needed for one horizontal composition for each k. For all $k = 0, 1, 2 \ldots f$, each horizontal composition requires total $[2 + 4 + 6 + \ldots 2 \cdot (f+1)]$=$O(f^2)$ operations. Given all the $|S_i|$ vertical compositions, there are a total of $[(|S_i| - 1) \cdot O(f^2)]$= $O(|S_i| \cdot f^2)$ operations for all the $(|S_i| - 1)$ horizontal compositions. Note that $|S_i|$=$O(N)$ since there are at most N time instants where new higher-priority jobs are released. Therefore, finding the `Load-Factor-HPi` for one task τ_i is $O(N \cdot f^2)$.

The time complexity to find the execution time of vertically-composed tasks is $O(N \cdot f^2)$. Therefore, total time complexity for the vertical and horizontal composition when considering the feasibility of task τ_i is $O(N \cdot f^2 + N \cdot f^2)$=$O(N \cdot f^2)$. ∎

We now present the calculation of `Load-Factor-HPi` (that is, the value of `w(k,`S_i`))` using our running example.

Example: *For task τ_1, we have $S_1 = \emptyset$ from Eq.* (5.12). *Therefore, no vertical composition, and hence no horizontal composition is needed.*

For task τ_2, we have $S_2 = \{0, 10\}$. Using vertical composition, we have two vertically-composed tasks $V_{\{0\}}$ and $V_{\{10\}}$. The execution time `w(k,{s})` of the vertically-composed task for $s = 0$ and $k = 0, 1, 2$ fault patterns are `w(0,0)`=3, `w(1,0)`=5, and `w(2,0)`=8 (given in the first column of Table 5.2). Similarly, the execution time `w(k,{s})` of the vertically-composed task for $s = 10$ and $k = 0, 1, 2$ fault patterns are `w(0,10)`=3, `w(1,10)`=5 and `w(2,10)`=5 (given in the second column of Table 5.2). Using Eq.(5.16), *the two vertically-composed tasks $V_{\{0\}}$ and $V_{\{10\}}$ are horizontally-composed as $H_{\{0, 10\}}$ with execution time `w(k,{0,10})` that is calculated in Table 5.4 for $k = 0, 1, 2$. Form Table 5.4, when considering the feasibility of task τ_2, the amount of execution completed by the higher-priority jobs within $[0, 15)$ is 6, 8 and 11 for k=0, 1 and 2 faults affecting only the jobs of the higher-priority task, respectively.*

For task τ_3, we have $S_3 = \{0, 10, 15, 20, 30\}$. Using vertical composition, we have five vertically-composed tasks $V_{\{0\}}$, $V_{\{10\}}$, $V_{\{15\}}$, $V_{\{20\}}$ and $V_{\{30\}}$. The execution time of the vertically-composed tasks for $k = 0, 1, 2$ are given in Table 5.3. Using Eq. (5.16) *and Eq.* (5.17), *we horizontally compose the five vertically-composed tasks $V_{\{0\}}$, $V_{\{10\}}$,$V_{\{15\}}$ $V_{\{20\}}$ and $V_{\{30\}}$. For the five*

For $H_{\{0,\,10\}}$ and $k = 0$
$\texttt{w(0,\{0,10\})} = \texttt{w(0,\{0\}}\cup\texttt{\{10\})}$
$=\max_{q=0}^{0} \Big\{ min\{\texttt{w(q,}\{0\}\texttt{)} + \texttt{w(k-q,}\{10\}\texttt{)}, T_i \}\Big\}$
$= min\Big\{[\texttt{w(0,\{0\})} + \texttt{w(0,\{10\})}], T_i \Big\}$
$= min\{[3+3], 15\} = min\{6, 15\}\Big\} = 6$
For $H_{\{0,10\}}$ and $k = 1$
$\texttt{w(1,\{0,10\})} = \texttt{w(1,\{0\}}\cup\texttt{\{10\})}$
$=\max_{q=0}^{1} \Big\{ min\{\texttt{w(q,}\{0\}\texttt{)} + \texttt{w(1-q,}\{10\}\texttt{)}, T_i \}\Big\}$
$= max\Big\{ min\{[\texttt{w(0,\{0\})} + \texttt{w(1,\{10\})}], T_i \},$
$\quad min\Big\{[\texttt{w(1,\{0\})} + \texttt{w(0,\{10\})}], T_i \}\Big\}$
$= max\Big\{ min\{[3+5], 15\}, min\{[5+3], 15\}\Big\}$
$= max\Big\{ min\{8, 15\}, min\{8, 15\}\Big\} = 8$
For $H_{\{0,10\}}$ and $k = 2$
$\texttt{w(2,\{0,10\})} = \texttt{w(2,\{0\}}\cup\texttt{\{10\})}$
$=\max_{q=0}^{2} \Big\{ min\{\texttt{w(q,}\{0\}\texttt{)} + \texttt{w(2-q,}\{10\}\texttt{)}, T_i \}\Big\}$
$= max\Big\{ min\{[\texttt{w(0,\{0\})} + \texttt{w(2,\{10\})}], T_i \},$
$\quad min\{[\texttt{w(1,\{0\})} + \texttt{w(1,\{10\})}], T_i \}$
$\quad min\{[\texttt{w(2,\{0\})} + \texttt{w(0,\{10\})}], T_i \}\Big\}$
$= max\Big\{ min\{[3+5], 15\}, min\{[5+5], 15\}, min\{[8+3], 15\}\Big\}$
$= max\Big\{ min\{8, 15\}, min\{10, 15\}, min\{11, 15\}\Big\} = 11$

Table 5.4: *Calculation of* $\texttt{w(k,\{0,10\})}$ *for horizontally-composed task* $H_{\{0,\,10\}}$ *for* $k = 0, 1, 2.$

vertically-composed tasks, four horizontally-composed tasks are formed. We start with composing $V_{\{0\}}$ *and* $V_{\{10\}}$ *horizontally using Eq.* (5.16). *The new horizontally-composed task is* $H_{\{0,10\}}$. *The execution time of the horizontally-composed task* $H_{\{0,10\}}$ *is* $\texttt{w(k,\{0,10\})}$ *and calculated using Eq.* (5.16) *for* $k = 0, 1, 2$ *(given in the first row of each Table 5.5-Table 5.7). Then, the horizontally-composed task* $H_{\{0,\,10\}}$ *and the vertically-composed task* $V_{\{15\}}$ *are composed in to a new horizontally-composed task* $H_{\{0,\,10,\,15\}}$. *The execution* *time of $H_{\{0,10,15\}}$ is `w(k,{0,10,15})` and calculated using Eq.* (5.17) *for* $k = 0, 1, 2$ *(given in the second row of each Table 5.5-Table 5.7). This process continues and finally the horizontally-composed task $H_{\{0,10,15,20\}}$ and the vertically-composed task $V_{\{30\}}$ are composed into a new horizontally-composed task that is $H_{\{0,10,15,20,30\}}$. The execution time of the horizontally-composed task $H_{\{0,10,15,20,30\}}$ is `w(k,{0,10,15,20,30})` that is calculated using Eq.* (5.17)*, for* $k = 0, 1, 2$ *(given in the fourth row of each Table 5.5-Table 5.7). The execution time of the four horizontally-composed tasks $H_{\{0,10\}}$, $H_{\{0,10,15\}}$, $H_{\{0,10,15,20\}}$, and $H_{\{0,10,15,20,30\}}$ are given in Table 5.5, Table 5.6 and Table 5.7 for $k = 0$, $k = 1$ and $k = 2$ fault patterns, respectively.*

Composed task	Execution time for 0-fault pattern
$V_{\{0, 10\}}$	`w(0,{0,10})`=9
$V_{\{0,10,15\}}$	`w(0,{0,10,15})`=12
$V_{\{0,10,15,20\}}$	`w(0,{0,10,15,20})`=15
$V_{\{0,10,15,20,30\}}$	`w(0,{0,10,15,20,30})`=21

Table 5.5: *The execution time due to 0-fault pattern of the four horizontally-composed tasks $H_{\{0,10\}}$, $H_{\{0,10,15\}}$, $H_{\{0,10,15,20\}}$, and $H_{\{0,10,15,20,30\}}$*

Composed task	Execution time for 1-fault pattern
$V_{\{0, 10\}}$	`w(1,{0,10})`=13
$V_{\{0,10,15\}}$	`w(1,{0,10,15})`=16
$V_{\{0,10,15,20\}}$	`w(1,{0,10,15,20})`=19
$V_{\{0,10,15,20,30\}}$	`w(1,{0,10,15,20,30})`=25

Table 5.6: *The execution time due to 1-fault pattern of the four horizontally-composed tasks $H_{\{0,10\}}$, $H_{\{0,10,15\}}$, $H_{\{0,10,15,20\}}$, and $H_{\{0,10,15,20,30\}}$*

Composed task	Execution time for 2-fault pattern
$V_{\{0, 10\}}$	`w(2,{0,10})`=18
$V_{\{0,10,15\}}$	`w(2,{0,10,15})`=21
$V_{\{0,10,15,20\}}$	`w(2,{0,10,15,20})`=24
$V_{\{0,10,15,20,30\}}$	`w(2,{0,10,15,20,30})`=30

Table 5.7: *The execution time due to 2-fault pattern of the four horizontally-composed tasks $H_{\{0,10\}}$, $H_{\{0,10,15\}}$, $H_{\{0,10,15,20\}}$, and $H_{\{0,10,15,20,30\}}$*

The amount of execution time `w(k,{0,10,15,20,30})` of the final *horizontally-composed task H_{S_i} is the exact value of* `Load-Factor-HPi` *due to a k-fault-pattern. The value of* `w(k,{0,10,15,20,30})` *represents the amount of execution time within $[0, 40)$ by all the higher-priority jobs due to the k-fault-pattern. Table 5.5-Table 5.7 show that the execution completed by the higher-priority jobs within $[0, 40)$ is 21, 25, and 30 for k=0,1 and 2-fault patterns, respectively (shown in the shaded fourth row in each of the Table 5.5-Table 5.7).* ▮

It is easy to realize at this point that the way the composition technique is applied to calculate the execution time of the final horizontally composed task can also be applied to any fixed-priority task system.

Based on the value of the `Load-Factor-HPi`, we now derive the exact RM feasibility condition of task τ_i in Section 5.6.

5.6 Exact Feasibility Condition

The exact feasibility condition for RM fault-tolerant uniprocessor scheduling for a periodic task set Γ is derived based on the exact feasibility condition of each task τ_i for $i = 1, 2 \ldots n$. The exact feasibility condition of task τ_i depends on the amount of execution required by task τ_i and its higher-priority jobs within the interval $[0, T_i)$ considering f faults that could occur within $[0, T_i)$.

By considering ($f - k$) faults exclusively affecting task τ_i and the k-fault pattern affecting its higher-priority jobs within the interval $[0, T_i)$, the sum of `Load-Factor-i` and `Load-Factor-HPi` can be calculated such that it is maximized for some k, $0 \leq k \leq f$. This sum is consequently the worst-case workload within $[0, T_i)$. The value of `Load-Factor-i` is $C_i^{(f-k)}$ and can be calculated using Eq. (5.5), for $k = 0, 1, 2, \ldots f$. The value of `Load-Factor-HPi` is `w(k,`S_i`)` and can be calculated using Eq. (5.17), for $k = 0, 1, 2, \ldots f$.

We denote the maximum total workload within $[0, T_i)$ by `TLoad`$_i$ which is equal to the sum of `Load-Factor-i` and `Load-Factor-HPi` such that the sum is maximum for some k, $0 \leq k \leq f$. The function `TLoad`$_i$ is thus defined in Eq. (5.18) as follows:

$$\texttt{TLoad}_i = \max_{k=0}^{f} \{ C_i^{(f-k)} + \texttt{w(k,}S_i\texttt{)} \} \tag{5.18}$$

Using Eq. (5.18), the maximum total workload within the interval $[0, T_i)$ can be determined. The total load is equal to the sum of the execution time required by task τ_i if $(f - k)$ faults exclusively affect the task τ_i and the execution time within the interval $[0, T_i)$ by the jobs having higher priority than the task τ_i due to k-fault pattern, such that, the sum is maximum for some k, $0 \leq k \leq f$.

Run-time complexity to compute the total load: Calculating the value of $C_i^{(f-k)}$ for all $k = 0, 1, 2, \ldots f$ can be done in $O(f)$ steps. The value of `w(k,`S_i`)` is the execution time of the final horizontally-composed task and can be calculate in $O(N \cdot f^2)$ time. In Eq. (5.18), there are $(f + 1)$ possible values for the selection of k, $0 \leq k \leq f$. Evaluating `TLoad`$_i$ in Eq. (5.18) requires a total of $f + 1$ addition operations and f comparisons to find the maximum. Given the values of $C_i^{(f-k)}$ and `w(k,`S_i`)` for all $k = 0, 1, 2, \ldots f$, finding the value of `TLoad`$_i$ requires $O(f)$ steps. Therefore, the total time complexity for evaluating `TLoad`$_i$ is [$O(f)$+$O(N \cdot f^2)$+$O(f)$]=$O(N \cdot f^2)$.

Based on the value of `TLoad`$_i$, the necessary and sufficient condition for RM scheduling is now proved in Theorem 5.1.

Theorem 5.1 *Task $\tau_i \in \Gamma$ is fault-tolerant RM-schedulable if, and only if, `TLoad`$_i$$\leq T_i$.*

Proof *(if part)* We prove that, if `TLoad`$_i$$\leq T_i$, then task τ_i is fault-tolerant RM-schedulable using proof by contradiction. The value of `TLoad`$_i$ as given in Eq. (5.18) is the sum of `Load-Factor-i` and `Load-Factor-HPi`. The value of `Load-Factor-i` is the maximum execution time required by the task τ_i if $(f - k)$ faults exclusively occur in the first job of task τ_i. The value of `Load-Factor-i` is given by $C_i^{(f-k)}$ in Eq. (5.5) for $k = 0, 1, 2, \ldots f$. The value of `Load-Factor-HPi` is the execution completed within the interval $[0, T_i)$ by the jobs having higher priority than the priority of task τ_i. The value of `Load-Factor-HPi` is given by `w(k,`S_i`)` which is equal to the execution time of the final horizontally-composed task H_{S_i} considering a k-fault pattern affecting the jobs of the higher-priority tasks within the interval $[0, T_i)$, for $k = 0, 1, 2, \ldots f$. The value of `w(k,`S_i`)` is the maximum amount of work that can be completed by the higher-priority jobs within $[0, T_i)$.

Now, assume a contradiction, that is, that some job of task τ_i misses it deadline when `TLoad`$_i$ $\leq T_i$. This assumption implies that the first job of task τ_i misses its deadline (due to it being a critical instant). When the first job of task τ_i misses its deadline at time T_i, the processor must be continuously busy within the entire interval $[0, T_i)$. This is because, if the processor was idle at some time instant within $[0, T_i)$, then τ_i cannot have missed its deadline since RM is a work-conserving algorithm.

In case that τ_i misses its deadline, the processor either executes task τ_i or its higher-priority jobs at each time instant within $[0, T_i)$. The time required for executing the higher-priority jobs within $[0, T_i)$ is `Load-Factor-HPi` which is given by `w(k,`S_i`)`. Note that `w(k,`S_i`)` is less than or equal to T_i according to Eq. (5.17). The total time required for completing the execution of task τ_i is `Load-Factor-i` considering $(f - k)$ faults that could affect the first job of task τ_i. Since τ_i misses it deadline at T_i, the complete execution of task τ_i cannot have finished by time T_i. Therefore, the sum of `Load-Factor-i` and `Load-Factor-HPi`, denoted by `TLoad`$_i$, must have been greater than T_i (which is a contradiction). Therefore, if `TLoad`$_i \leq T_i$, then task τ_i is fault-tolerant RM-schedulable.

(only if part) We prove that, if τ_i is RM-schedulable, then `TLoad`$_i \leq T_i$. The amount of work on behalf of task τ_i (including execution of its recovery copy) within the interval $[0, T_i)$ that is completed by RM is `Load-Factor-i`. Since τ_i is the lowest priority task, the amount of execution on behalf of the jobs (including execution of their recovery copies) having higher priority than task τ_i that is completed by RM is exactly equal to `Load-Factor-HPi` within $[0, T_i)$.

Since the amount of work completed by the algorithm RM completes within $[0, T_i)$ is equal to `Load-Factor-i` plus `Load-Factor-HPi`, the total load `TLoad`$_i$ is less than or equal to T_i. Therefore, if task τ_i is fault-tolerant RM-schedulable, then we have `TLoad`$_i \leq T_i$. Since the first jobs of task τ_i is RM-schedulable, all the jobs of task τ_i are also RM-schedulable (due to our scheduling analysis considering critical instant). ▮

The exact feasibility condition for RM scheduling of task τ_i is given in Theorem 5.1. The time complexity for evaluating this exact condition is the same as the time complexity for evaluating Eq. (5.18). Therefore, the necessary and sufficient condition for checking the feasibility of task τ_i can be evaluated in time $O(N \cdot f^2)$.

The exact feasibility condition for the entire task set Γ is now given in the following Corollary 5.2.

Corollary 5.2 *Task set* $\Gamma = \{\tau_1, \tau_2, \ldots, \tau_n\}$ *is fault-tolerant RM schedulable if, and only if, task* τ_i *is RM-schedulable using Theorem 5.1 for all* $i = 1, 2, \ldots n$.

Proof Obvious from Theorem 5.1.

Note that Corollary 5.2 is the application of Theorem 5.1 for each one of the n tasks in set Γ. Therefore, the exact feasibility condition for the entire task set can be evaluated in $O(n \cdot N \cdot f^2)$ time. We now determine the RM-schedulability of the running example task set given in Table 5.1.

Example: *We have to apply Theorem 5.1 to all the three tasks given in Table 5.1. For task τ_i we have to find the value of* `TLoad`$_i$ *for all $i = 1, 2, 3$. The task τ_1 is trivially RM-schedulable because it is the highest priority task and we assume Eq.* (5.4) *is true for all tasks.*

Consider the feasibility of task τ_2. Remember that, `w(k,`S_i`)` *is the execution time of the final horizontally-composed task and is equal to the value of* `Load-Factor-HPi`*. For task τ_2, we have $S_2 = \{0, 10\}$. By horizontal composition, the final horizontally-composed task $H_{\{0,10\}}$ has execution time equal to* `w(0,`S_2`)` *= 6,* `w(1,`S_2`)` = 8, and `w(2,`S_2`)` = 11 for $k = 0$, $k = 1$ and $k = 2$ faults, respectively, within interval $[0, 15)$ (given in Table 5.4). For task τ_2, we also have C_2^0 =3, C_2^1 =7 and C_2^2 =9 for $k = 0$, $k = 1$ and $k = 2$ faults, respectively, which are the values of `Load-Factor-i` using Eq.(5.6). For task τ_2 and $f = 2$, the calculation of `TLoad`$_2$ using Eq. (5.18) is given below:

$$\begin{aligned}
\texttt{TLoad}_2 &= \max_{q=0}^{2} \left\{ C_2^{(2-q)} + \texttt{w(q,\{0,10\})} \right\} \\
&= max\Big\{ [C_2^2 + \texttt{w(0,\{0,10\})}], \\
&\qquad [C_2^1 + \texttt{w(1,\{0,10\})}], \\
&\qquad [C_2^0 + \texttt{w(2,\{0,10\})}] \Big\} \\
&= max\Big\{ [9+6], [7+8], [3+11] \Big\} = 15
\end{aligned}$$

Since `TLoad`$_2$$= 15 \leq T_2 = 15$*, task τ_2 is RM-schedulable using Theorem 5.1.*

Consider the feasibility of task τ_3. We have $S_3 = \{0, 10, 15, 20, 30\}$. By horizontal composition, the final horizontally-composed task $H_{\{0,10,15,20,30\}}$ has execution time equal to `w(0,`S_3`)`*=21,* `w(1,`S_3`)`*=25, and* `w(2,`S_3`)`*=30 for $k = 0$, $k = 1$ and $k = 2$ faults, respectively, within interval $[0, 40)$ (given in the fourth shaded row in Table 5.5–Table 5.7). For task τ_3, we also have C_3^0 =9, C_3^1 =17 and C_3^2 =23 for $k = 0$, $k = 1$ and $k = 2$ faults, respectively, which are the values of* `Load-Factor-i` *using Eq.*(5.6)*. For task τ_3 and $f = 2$, the* *calculation of* `TLoad`$_3$ *using Eq.* (5.18) *is given below:*

$$\begin{aligned}
\texttt{TLoad}_3 &= \max_{q=0}^{2} \left\{ C_3^{(2-q)} + \texttt{w(q,\{0,10,15,20,30\})} \right\} \\
&= max \Big\{ [C_3^2 + \texttt{w(0,\{0,10,15,20,30\})}], \\
&\qquad [C_3^1 + \texttt{w(1,\{0,10,15,20,30\})}], \\
&\qquad [C_3^0 + \texttt{w(2,\{0,10,15,20,30\})}], \Big\} \\
&= max \Big\{ [21+23], [25+17], [30+9] \Big\} = 44
\end{aligned}$$

Since `TLoad`$_3$$= 44 \geq T_3 = 40$*, task* τ_3 *is not RM-schedulable using Theorem 5.1. Therefore, the task set given in Table 5.1 is not RM -schedulable using Corollary 5.2.*

Based on the necessary and sufficient feasibility condition in Corollary 5.2, the algorithm `FTRM` is now presented in Section 5.7.

5.7 Algorithm FTRM

In this section, we present the fault-tolerant uniprocessor algorithm `FTRM` based on the exact feasibility condition derived in Corollary 5.2. First, the pseudocode of the algorithm `CheckFeasibility(`τ_i, f`)`is given in Figure 5.4. The algorithm `CheckFeasibility(`τ_i, f`)`checks the RM feasibility of a task τ_i by considering occurrences of f faults in any jobs of the tasks in set $\{\tau_1, \tau_2, \ldots \tau_i\}$ released within the interval $[0, T_i)$. Next, the Algorithm `FTRM` is presented in Figure 5.5. Algorithm `FTRM` checks the feasibility of the entire task set Γ based on the feasibility of each task $\tau_i \in \Gamma$ using the algorithm `CheckFeasibility(`τ_i, f`)`.

In line 1 of Algorithm `CheckFeasibility(`τ_i, f`)`in Figure 5.4, the jobs having higher priority than the priority of task τ_i are determined using Eq. (5.7). In line 2, the time instants at each of which higher-priority jobs are released within the interval $[0, T_i)$ are determined using Eq. (5.11). Using the loop in line 3–7, the execution time `w(k,{`s`})` of each vertically-composed task $V_{\{s\}}$ is derived for each point $s \in S_i$. The value of `w(k,{`s`})` is determined for each $k = 0, 1, 2, \ldots f$ at line 5 using Eq. (5.15).

Using the loop in line 8–12, the vertically-composed tasks are composed further using horizontal compositions. The loop at line 8 iterates total $|S_i| - 1$ times. Each iteration of this loop calculates the execution time of one horizontally composed task $H_{\texttt{p}(l)}$=$H_{\texttt{p}(l\texttt{-1})\cup\{s_l\}}$, for $l = 2, 3, \ldots |S_i|$. The execution

```
Algorithm CheckFeasibility($\tau_i, f$)

1.  Find the HP$_i$ using Eq. (5.7)
2.  Find the $S_i$ using Eq. (5.11)
3.  For all $s \in S_i$
4.    For $k = 1$ to $f$
5.      Find w(k,$\{s\}$) using Eq. (5.15)
6.    End For
7.  End For
8.  For $l = 2$ to $|S_i|$
9.    for $k = 1$ to $f$
10.     Find w(k,p($l$-1)$\cup\{s_l\}$) using Eq. (5.17)
11.   End For
12. End For
13.   For $k = f$ to $0$
14.     Find $C_i^{(f\,-\,k)}$ using Eq. (5.5)
15.   End For
16. For $k = 0$ to $f$
17.   If [$C_i^{(f\,-\,k)}$ +w(k,$S_i$)] $> T_i$ then
18.     return False
19.   End If
20. End For
21. return True
```

Figure 5.4: *Pseudocode of Algorithm* CheckFeasibility(τ_i, f)

time w(k,p(l-1)$\cup\{s_l\}$) of the horizontally-composed task $H_{\mathtt{p}(l\text{-}1)\cup\{s_l\}}$ is calculated at line 10 using Eq. (5.17) for a k-fault pattern, $k = 0, 1, 2, \ldots f$. The execution time w(k,S_i) of the final horizontally-composed task H_{S_i} is the value of Load-Factor-HPi, for $k = 0, 1, 2 \ldots f$.

Using the loop in line 13–15, the value of $C_i^{(f\,-\,k)}$ is determined in line 14 using Eq. (5.5) for $k = 0, 1, \ldots f$. Remember that the value of $C_i^{(f\,-\,k)}$ is Load-Factor-i. In line 16–21, the exact feasibility condition for τ_i is checked by considering k faults affecting the jobs of the higher-priority tasks and $(f - k)$ faults exclusively affecting the task τ_i, for $k = 0, 1, 2, \ldots f$. In line 17, the value of TLoad$_i$ is calculated by summing Load-Factor-i and Load-Factor-HPi and this sum is compared against the period of task τ_i. If this sum is greater than T_i, then task τ_i is not RM schedulable and the algorithm CheckFeasibility(τ_i, f)returns FALSE at line 18. If the condition at line 17 is false for all $k = 0, 1, 2 \ldots f$, then task τ_i is RM schedulable and the algorithm CheckFeasibility(τ_i, f)returns TRUE at line 21.

Next, using the algorithm `CheckFeasibility(`τ_i, f`)`, we present the algorithm `FTRM` in Figure 5.5.

```
Algorithm   FTRM ($\Gamma$, $f$)

1. For all $\tau_i \in \{\tau_1, \tau_2, \ldots, \tau_n\}$
2.   If CheckFeasibility($\tau_i, f$)= False then
3.     return False
4.   End If
5. End For
6. return True
```

Figure 5.5: *Pseudocode of Algorithm `FTRM` (Γ, f)*

Using the loop in line 1–5 of algorithm `FTRM` (Γ, f), the RM-feasibility of each task τ_i in set Γ is checked. The algorithm `FTRM` (Γ, f) checks the RM feasibility of task $\tau_i \in \Gamma$ using the algorithm `CheckFeasibility(`τ_i, f`)` at line 2. If the condition at line 2 is true for any task τ_i (that is, the algorithm `CheckFeasibility(`τ_i, f`)`returns FALSE), then the task set Γ is not RM-schedulable. In such case, the algorithm `FTRM` (Γ, f) returns FALSE (line 3). If the condition at line 2 is false for task τ_i , for all $i = 1, 2, \ldots n$ (that is, `CheckFeasibility(`τ_i, f`)`returns TRUE for each task), then the task set Γ is RM-schedulable. In such case, the algorithm `FTRM` (Γ, f) returns TRUE (line 6).

Given a task set Γ and the number of faults f that can occur within any interval of length T_{max}, the fault-tolerant RM feasibility of the task set can be determined using algorithm `FTRM` (Γ, f) in $O(n \cdot N \cdot f^2)$ time.

Next we discuss the applicability of our exact uniprocessor feasibility analysis for multiprocessor platform in subsection 5.7.1.

5.7.1 Multiprocessor Scheduling

The uniprocessor `FTRM` scheduling analysis is applicable to multiprocessor partitioned scheduling. To that end, the exact analysis of `FTRM` can be applied during the task assignment phase of a partitioned multiprocessor scheduling algorithm in which the run time dispatcher in each processor executes tasks in RM priority order.

Consider a multiprocessor platform consisting of m processors. The question addressed is as follows:

> Is there an assignment of the tasks in set Γ on m processors such that each processor can tolerate f faults?

Partitioned multiprocessor task scheduling is typically based on a bin-packing algorithm for task assignment to the processors. When assigning a new task to a processor, a uniprocessor feasibility condition is used to check whether or not an unassigned task and all the previously assigned tasks in a particular processor are RM schedulable. If the answer is yes, the unassigned task can be assigned to the processor. In order to extend the partitioned multiprocessor scheduling to fault-tolerant scheduling, we can apply the exact feasibility condition derived in Corollary 5.2 when trying to assign a new task to a processor in partitioned scheduling. The following example discusses how the exact feasibility condition derived in Corollary 5.2 can be applied to the First-Fit heuristic for task assignment on multiprocessors.

Example: *Consider the First-Fit heuristic for task assignment to processors. Given a task set $\{\tau_1, \tau_2, \ldots, \tau_n\}$, we consider the tasks to be assigned to m processors in increasing order of task index. That is, τ_1 is considered first, and then τ_2 is considered and so on. Using the First-Fit heuristics, the processors of the multiprocessor platform are also indexed from $1 \ldots m$. An unassigned task is considered to be assigned to processor in increasing order of processor index. An unassigned task is assigned to the processor with the smallest index for which it is feasible. Following the First-Fit heuristic, task τ_1 is trivially assigned to the first processor. For task τ_2, the necessary and sufficient feasibility condition in Corollary 5.2 is applied to a set of tasks $\{\tau_1$, $\tau_2\}$ considering at most f faults that could occur in an interval of length T_{max} (where T_{max} is the maximum period of the tasks in set $\{\tau_1$, $\tau_2\}$). If the feasibility condition is satisfied, then τ_2 is assigned to the first processor. Otherwise, τ_2 is trivially assigned to the second processor. Similarly, for a task τ_i, the feasibility condition in Corollary 5.2 is checked for the already assigned tasks and task τ_i on the first processor. If task τ_i and all the previously assigned tasks to the first processor are RM schedulable using the exact condition in Corollary 5.2, then τ_i is assigned to the first processor. If the exact condition is not satisfied, the feasibility condition is checked for the second processor and so on. If task τ_i cannot be assigned to any processor, then task set Γ cannot be partitioned on the given multiprocessor platform. If all the tasks are assigned to the multiprocessor platform, then task set Γ is RM schedulable. For a successful partition of the task set Γ, each processor can tolerate f faults that can occur in any tasks within a time interval equal to the maximum period of the tasks assigned to a particular processor.* ▮

Similarly, the exact condition in Corollary 5.2 can be used during task assignment to the processors of a multiprocessor platform in which each processor executes tasks using uniprocessor RM scheduling algorithm.

5.8 Discussion and Summary

This chapter presented the analysis of RM fault-tolerant scheduling that can be used to guarantee the correctness and timeliness property of real-time applications on uniprocessor. The correctness property of the system is addressed by means of fault-tolerance so that the system functions correctly even in the presence of faults. The timeliness property is addressed by deriving a necessary and sufficient feasibility condition for the RM scheduling on uniprocessor. The proposed algorithm `FTRM` can verify the feasibility of a task set Γ using the fault-tolerant RM scheduling on uniprocessor. The time complexity of `FTRM` is $O(n \cdot N \cdot f^2)$, where n is the number of tasks in the periodic task set, N is the maximum number of jobs released within any time interval of length T_{max}, and f is the maximum number of faults that can occur within any time interval of length T_{max}.

The schedulability analysis used in this chapter is directly applicable to any static-priority scheduling algorithm for periodic task systems in which the relative deadline of each task is less than or equal to its period and the time instant zero can be considered as the critical instant. To check the feasibility of the first job of task τ_i, the higher priority jobs within the interval $[0, D_i)$, where D_i is the relative deadline of task τ_i, are to be determined. Using the composition techniques proposed in this chapter the value of maximum total workload within $[0, D_i)$ can be calculated. If the value of the total load is less than or equal to D_i, then task τ_i is schedulable. And conversely if the task τ_i is schedulable, then the value of total load is less than or equal to D_i. It is not difficult to see that, the novelty of our composition technique is applicable to determine the exact feasibility of a static-priority aperiodic task set on uniprocessor.

The fault model considered for the exact RM analysis is general enough in the sense that multiple faults can occur in any jobs, at any time and even during recovery operation. There is no restriction posed on the occurrences of two consecutive faults. The only restriction considering our fault model is that a maximum of f faults could occur within any time interval of length T_{max}, where T_{max} is the maximum period of any tasks in set Γ. The fault-tolerance mechanism proposed in this chapter can tolerate a variety of hardware and software faults. Transient faults in hardware that are frequent, short-lived and do not reappear if a task is re-executed can be tolerated using our proposed fault-tolerant mechanism. Software faults that do not reappear when the same software is re-executed can also be tolerated using our proposed scheme. Moreover, `FTRM` can tolerate software faults that do appear again when the same software is simply re-executed. To tolerate such software faults, a different implementation of the specification of the software can be executed when an error is detected. To account for the execution of a different version of software, the exact analysis of `FTRM` considers WCET of the recovery copy that may not be equal to the WCET of the primary copy of a task.

The variety of faults considered in our fault model can also be tolerated using spatial redundancy, for example, executing the task in two different processor. When an error is detected at the end of execution of a task in one processor, the result of the task execution from another processor can be used to tolerate the fault. However, considering the highly frequent transient faults, use of spatial redundancy may not a cost-efficient approach to achieve fault-tolerance. Moreover, in many safety-critical systems, like applications in space, avionics and automotive systems, there is always a space and weight constraints. In such systems, time redundancy is more cost-efficient and preferable over spatial redundancy to achieve fault-tolerance. The fault-tolerance mechanisms proposed in this chapter exploits time redundancy.

To the best of my knowledge, no other work has proposed an exact fault-tolerant feasibility analysis for RM scheduling of periodic tasks considering such a general fault model as ours. If an efficient (in terms of time complexity) and exact feasibility test is needed, then the scheduling algorithm `FTRM` provides better computational efficiency than a recently proposed fault-tolerant EDF scheduling algorithm in [Ayd07].

Our proposed exact uniprocessor feasibility condition can be applied to task scheduling on multiprocessors. The exact uniprocessor feasibility condition of `FTRM` can be used for partitioned multiprocessor scheduling, in which, tasks are assigned to the processors using uniprocessor RM feasibility condition. The fault-tolerant exact feasibility condition proposed in this chapter for uniprocessor can be used during task assignment to the processors considering equal or different values for f for different processors. The bin-packing heuristic and the parameter f determines the ways tasks are assigned to the processors. Consequently, the task assignment algorithm for partitioned method for multiprocessor scheduling can be driven by the reliability requirement of the system.

6
Multiprocessor Scheduling

This chapter proposes a fixed-priority partitioned scheduling algorithm for periodic tasks on multiprocessors. A new technique for assigning tasks to processors is developed and the schedulability of the algorithm is analyzed for worst-case performance. We prove that, if the workload (utilization) of a given task set is less than or equal to 55.2% of the total processing capacity on *m* processors, then all tasks meet their deadlines. During task assignment, the total work load is regulated to the processors in such a way that a subset of the processors are guaranteed to have an individual processor load of at least 55.2%. Due to such load regulation, our algorithm can be used efficiently as an admission controller for online task scheduling. And this online algorithm is scalable with increasing number of processors.

6.1 Introduction

In recent years, applications of many embedded systems run on multiprocessors, in particular, chip multiprocessors [KAO05, CCE$^+$09]. The main reasons for doing this is to reduce power consumption and heat generation. Many of these embedded systems are also hard real-time systems in nature and meeting the task deadlines of the application is a major challenge. Since many well-known uniprocessor scheduling algorithms, like Rate-Monotonic (RM) or Earliest Deadline First (EDF) [LL73], are no longer optimal for multiprocessors [DL78], developing new scheduling algorithms for multiprocessor platform have received considerable attention. In this chapter, we address the problem of meeting deadlines for a set of *n* implicit deadline periodic tasks using RM scheduling on m processors based on task-splitting paradigm. We also propose an extension of our scheduling algorithm that can be efficiently used as an admission controller for online scheduling. In addition, our scheduling algorithm possesses two properties that may be important for the system designer. The first one guarantees that if task priorities are fixed before task assignment they do not change during task assignment and execution, thereby facilitating debugging during development and maintenance of the system. The second property guarantees that at most $m/2$ tasks are split, thereby keeping the runtime overhead as caused by task splitting low.

Static-priority preemptive task scheduling on multiprocessors can be classified as *global* or *partitioned* scheduling. In global scheduling, at any time m highest-priority tasks from a global queue are scheduled on m processors. In partitioned scheduling, a task is allowed to execute only on one fixed, assigned, processor. That is, tasks are grouped first and each group of tasks executes in one fixed processor without any migration. In global scheduling, tasks are allowed to migrate while in partitioned scheduling, tasks are never allowed to migrate. Many static-priority scheduling policies for both global [ABJ01, Lun02, Bak06, BCL05] and partitioned [DL78, LBOS95, AJ03, FBB06, LGDG03, LMM98, OB98] approaches have been well studied. We address a variation of partitioned scheduling technique in which a bounded number of tasks can migrate to a different processor.

It has already been proved that there exists some task set with load slightly greater than 50% of the capacity of a multiprocessor platform on which a deadline miss must occur for both global and partitioned static-priority scheduling [ABJ01, OB98]. To achieve a utilization bound higher than 50%, some recent work proposes techniques where *migratory* [ABD05] or *split* [AT06, ABB08, KY08a] tasks are allowed to migrate using a variation of partitioned scheduling for dynamic-priority tasks. Very little work [KY08b, LRL09] have addressed the scheduling problem for static-priority tasks using task splitting to overcome the 50% utilization bound. We propose a static-priority scheduling algorithm, called *Interval Based Partitioned Scheduling* (`IBPS`), for periodic tasks using the task splitting approach. By *task splitting*, we mean that some tasks are allowed to migrate their execution to a different processor during execution[1]. We call the task that is splitted a 'split task' and its pieces 'subtasks'. No task or the subtasks of a split task can run in parallel (sequential execution of the code of a task). In IBPS , rate-monotonic (RM) prioritization [LL73] is used both during task assignment and during run-time scheduling of tasks on a processor. One of the main contributions in this chapter is to prove that *if the total utilization (or workload) of a set of n periodic tasks is less than or equal to 55.2% of the capacity of m processors, the task set is RM schedulable on m processors using IBPS* .

Apart from the guarantee bound, the important features of IBPS are:

- During task assignment, the individual processor loads are regulated in a way that makes on-line scheduling (task addition and removal) more efficient than for other existing task-splitting algorithms. Due to load regulation, a bounded number of processors have load less than 55.2%. So, the percentage of processors with load greater than 55.2% increases as the number of processors in a system increases. Therefore, with an increasing number of cores in a chip multiprocessor (for example, Sun's Rock processor with 16 cores [CCE+09]), our proposed on-line scheduler is more effective and scales very well.

- The priority of a task given before task assignment is not changed to another priority during task assignment and execution, which facilitates debugging during system development and maintenance.

- The task splitting algorithm split tasks in such a way that the number of migrations is lower than for other existing task-splitting algorithms.

The rest of the chapter is organized as follows. In Section 6.2, the important features of IBPS are further elaborated. In Section 6.3, we present the assumed system model. In Section 6.4, we briefly discuss the basic idea of our task assignment algorithms and also present our task splitting approach. Then, in Sections 6.5 through 6.7, we present the IBPS task-assignment algorithms in detail. The performance of IBPS and its online version is presented in Section 6.8 and 6.9. In Section 6.10, we discuss other work related to ours. The approaches to extend our proposed multiprocessor scheduling algorithm to fault-tolerance are discussed in Section 6.11. Finally, Section 6.12 concludes the chapter with discussion and summary.

[1]Here, we do not mean splitting the code. 'Task-splitting' is migration of the execution of task from one processor to another.

6.2 Important Features of `IBPS`

A real-time task scheduling algorithm does not only need worst-case guarantee for deadlines but also need to be practically implementable. `IBPS` has three other major features:(i) *load regulation*, (ii) *priority traceability property*, and (iii) *low cost of splitting*.

Load Regulation: `IBPS` regulates the load of a processor during task assignment. When assigning tasks to processors, the objective of `IBPS` is to have as many processors as possible with load greater than 55.2% and still meet all the deadlines. In the worst case, the number of processors on which the load is less than or equal to 55.2% is at most *min{m,4}* where m is the number of processors. Thus, the load regulation of `IBPS` bounds the number of under-utilized processors.

Load regulation of `IBPS` enables design of efficient admission controller for online scheduling. In practice, many real-time systems are dynamic in nature, that is, tasks arrive and leave the system online. After accepting an online task, we then need to assign the task to a particular processor. Finding the best processor to assign the task may require disturbing the existing schedule in all m processors by means of a reassignment of tasks (e.g. task assignment algorithms that require sorting).

As will be evident later in this chapter, finding the best processor to which an accepted online task is assigned requires searching at most *min{m,4}* processors (the under-loaded processors) when `IBPS` is used online. Similarly, when a task leaves the system, reassignment of tasks is needed in at most *min{m,5}* processors to regulate the load for future admittance of new tasks. `IBPS` runs in linear time, therefore, reassignment of tasks on a bounded number of processors for load regulation is efficient. Moreover, task reassignment on a bounded number of processors makes our online scheduling algorithm, called `O-IBPS`, scalable with the trend of increasing number of cores in chip multiprocessor.

Priority Traceability Property: From a system designer's point of view it is desirable to facilitate debugging (execution tracing), during the development and maintenance of the system. One explanation for the wide-spread use of RM in industry is the relative ease with which the designer can predict the execution behavior at run-time. The dynamic-priority EDF scheduler has (despite being as mature a scheduling method as RM, but with stronger schedulability properties) not received a corresponding attention in industrial applications. Even for static-priority schedulers, the ease of debugging differs for different algorithms. For example, when studying the recent work in [LRL09] of static-priority partitioned scheduling with task splitting, we see that it is possible that the deadline of a subtask could become smaller (during task assignment) than the given implicit deadline of the original task during task assignment to processors. This, in turn, could make the priorities of the subtasks different from the original RM priority of the tasks and therefore cause a different, less traceable, execution behavior. A similar behavior can be identified in known dynamic-priority task-splitting algorithms [AT06, ABB08, KY08a] where a subtask of a split task may have different priority (that is, executes in specific time slots, or has smaller deadline) than the original task. IBPS is a static-priority partitioned algorithm with a strong *priority traceability property*, in the sense that if the priorities of all tasks are fixed before task assignment they never change during task assignment and execution.

Cost of Splitting: One final property of interest for task-splitting partitioned scheduling in particular is the run-time penalty introduced due to migration. Cost of splitting and its relation to scheduling performance has received attention [KLL09]. Clearly, the number of total split tasks and number of subtasks (resulting from a split tasks) directly affect the amount of preemptions and migrations, both of which in turn may affect cache performance and other migration related overhead. Therefore, it is always desirable to reduce the number of split tasks and reduce the number of subtasks for each split task. For all existing dynamic- and static-priority task-splitting algorithms, the number of split tasks is $(m-1)$ on m processor. In IBPS , total number of split task in the worst-case is at most $m/2$. In IBPS , a split task has only two subtasks and therefore, a split task never suffers more than once due to migration in one period.

We assume tasks are independent. At most one subtask of a split-task is assigned to one processor. Since the two subtasks of a split-task are assigned in two fixed processors, only the threads assigned to these two processors can be considered during debugging independent of the threads in other processors.

6.3 Task Model

The task model presented in Section 4.1 in extended here to incorporate split tasks. We assume a task set Γ consisting of n implicit deadline periodic tasks. Each task $\tau_i \in \Gamma$ arrives repeatedly with a period T_i and requires C_i units of worst-case execution time within each period. Task priorities are assigned according to the RM policy (lower the period, the higher the priority). We define the *utilization* of task τ_i as $u_i = \frac{C_i}{T_i}$. The *load* or *total utilization* of any task set A is $U(A) = \sum_{\tau_i \in A} u_i$.

When a task τ_i is split, it is considered as two subtasks, τ_i' and τ_i'', such that both subtasks has execution time and period equal to $\frac{C_i}{2}$ and T_i, respectively. Note that, since the period of a subtask is equal to the period of the split task τ_i, we must have $u_{i'} = u_{i''} = \frac{u_i}{2}$. When assigning tasks to each processor, we use Liu and Layland's sufficient feasibility condition for RM scheduling [LL73] for determining whether the tasks can be assigned to a processor. The sufficient RM feasibility condition according to [LL73] is given as follows:

> ***if** $U(A) \leq n(2^{\frac{1}{n}} - 1)$**, then all the** n **tasks in set** A **meet deadlines on uniprocessor.***

If $n \rightarrow \infty$, then the value of $[n(2^{\frac{1}{n}} - 1)]$ approaches 0.693. Therefore, if the total utilization of a task set is less than or equal to 0.693, then the task set is always RM-schedulable on uniprocessor. Here, we make the pessimistic assumption that each non-split task has an offset $\phi_i = 0$ since according to [LL73] the worst-case condition for schedulability analysis is when all tasks are released at the same time (we assume all tasks are released at time zero). However, the second subtask τ_i'' of a split task τ_i is given an offset equal to $\phi_{i''} = \frac{C_i}{2}$ to ensure nonparallel execution with first subtask τ_i'. In rest of the chapter, we use the notation LLB(n)=$n(2^{\frac{1}{n}} - 1)$ for n tasks, LLB(∞)=$\ln 2 \approx 0.693$ to represent an unknown number of tasks, and also let Q=($\sqrt{2} - 1$).

6.4 Task Assignment and Splitting Overview

Our proposed task assignment algorithm starts by dividing the utilization interval (0,1] into seven disjoint utilization subintervals I$_1$–I$_7$ (see Table 6.1). For $a < b$, we use I$_a$–I$_b$ to denote all the subintervals I$_a$, I$_{a+1}$, $\ldots$ I$_b$. Note

I$_1$= $(\frac{4Q}{3}, 1]$	I$_2$= $(\frac{8Q}{9}, \frac{4Q}{3}]$	I$_3$= $(\frac{2Q}{3}, \frac{8Q}{9}]$
I$_4$= $(\frac{8Q}{15}, \frac{2Q}{3}]$	I$_5$= $(\frac{4Q}{9}, \frac{8Q}{15}]$	I$_6$= $(\frac{Q}{3}, \frac{4Q}{9}]$
I$_7 = (0, \frac{Q}{3}]$	(*where, Q*= $\sqrt{2} - 1$)	

Table 6.1: *Seven disjoint utilization subintervals* I$_1$–I$_7$

that, each task $\tau_i \in \Gamma$ will have a utilization that belongs to exactly one of the subintervals I$_1$–I$_7$. By overloading the set membership operator "$\in$", we write $\tau_i \in$ I$_k$ to denote "τ_i in I$_k$=(a,b]" for any $k \in \{1 \ldots 7\}$, if $a < u_i \leq b$. For example, if a task τ_i has u_i=$\frac{4Q}{5}$, then $\tau_i \in$ I$_3$. Clearly, the grouping of tasks into subintervals can be completed in linear time.

Why Seven Utilization Subintervals? The seven utilization intervals result from the four main goals of the task assignment: (i) low number of subtasks per split task, (ii) low number of split tasks, (iii) assigning low number of tasks per processor[2], and (iv) load regulation.

To reach these goals, we start by assigning only one task to one processor exclusively. To avoid assigning a task with very small utilization to one processor exclusively, we select a task that belongs to a certain utilization subinterval. If `IBPS` has worst-case utilization bound U$_w$ and load regulation tries to maintain the load on most processors above U$_w$, then one task with utilization greater than U$_w$ is assigned to one processor exclusively. Thus, we obtain our first utilization interval which is (U$_w$, 1]. The actual value of U$_w$ is determined when we assign more than one task having utilization less than U$_w$ to one processor. When we try to assign two tasks with utilization less than U$_w$ to one processor, we find that in case if these two tasks have equal utilization, then each individual task's utilization can not be greater than Q=($\sqrt{}$2–1) according to LLB(n=2). This implies U$_w$ $\leq$ 41% without a task splitting technique. To achieve U$_w$ greater than 50%, we split a task with utilization less than U$_w$ in two subtasks each having equal utilization. We gain no advantage by having unequal utilization for the two subtasks as individual task utilization is bounded from below and our strategy is to assign minimum number of tasks per processor. So, each subtask has utilization at most $\frac{\text{U}_w}{2}$. We assign one such subtask and a non-split task to one processor. For RM schedulability, we must have U$_w$ $+\ \frac{\text{U}_w}{2} \leq 2Q$. This implies the value of U$_w$ $\leq \frac{4Q}{3}$ and we get our first utilization subinterval I$_1$=($\frac{4Q}{3}$,1]. Note that this interval also defines the maximum possible utilization bound of `IBPS`.

In summary, a task with utilization greater than U_w and less than or equal to 1 is exclusively assigned to one processor. Some other tasks with maximum utilization U$_w$ is split into two subtasks such that each subtask has maximum utilization U$_w/2$. If one such non-split task with maximum utilization U$_w$ and one subtask with maximum utilization U$_w/2$ are assigned to one processor, then we must need $\text{U}_w + \text{U}_w/2 \leq 2Q$ using LLB. This implies the maximum value of U$_w$ is $\text{U}_w = 4Q/3$. Thus the actual (maximum possible) value of U$_w$ is hinted (proved later) when more than one task are assigned to one processor. Thus the first interval $(\text{U}_w, 1] = (4Q/3, 1]$ is derived.

The overall goal of the task assignment is to achieve the maximum possi-

[2]According to LLB, the RM scheduling on uniprocessor achieves higher utilization bound if number of tasks assigned to the processor is small [LL73].

ble U$_w$, which is $\frac{4Q}{3}$ ≈55.2%. To achieve this bound, at each stage of task assignment, we bound the utilization of tasks from below for load regulation by selecting a task from a certain utilization interval, split a task in only two subtasks when task splitting is unavoidable and try to assign a minimum number of tasks to one processor. The consequence is that we derive the seven utilization intervals I$_1$-I$_7$ given in Table 6.1. Driven by the four main goals of task assignment, the derivations of I$_2$-I$_6$ are done in a similar fashion, as for I$_1$, while I$_7$ becomes the remaining interval task utilization in which requires no lower bound. More details can be found in the technical report [PJ09].

Task Assignment Overview: `IBPS` assigns tasks to processors in *three* phases. In the *first phase,* tasks from each subinterval I$_k$ are assigned to processors using one particular *policy* for each I$_k$. Any unassigned tasks in I$_2$–I$_6$ after first phase, called *odd tasks,* are assigned to processors in the *second phase*. If tasks are assigned to m' processors during the first and second phases, the total utilization in each of the m' processors will then be greater than $\frac{4Q}{3} \approx 55.2\%$ due to load regulation strategy. Any unassigned tasks after the second phase, called *residue tasks,* are assigned to processors during the *third phase*. If, after the second phase, the total utilization of the residue tasks is smaller than or equal to $\frac{4Qm''}{3}$ for the smallest non-negative integer m'', all the residue tasks are assigned to at most m'' processors. The load on these m'' processors may be smaller than 55.2%. Load regulation in first two phases ensures that $m'' \leq 4$. When task arrives online, we need only to consider these m'' processor for task assignment. The different task assignment algorithms in the three phases constitute the algorithm `IBPS` .

We conclude this section by defining some useful functions that will be used later for the worst-case schedulability analysis in this chapter. We use $U_{\text{MN}}(A)$ and $U_{\text{MX}}(A)$ to denote the lower and upper bound, respectively, on the total utilization of all tasks in an arbitrary task set A:

$$U_{\text{MN}}(A) = \sum_{\tau_i \in A} x \quad \text{where } \tau_i \in \text{I}_k = (x, y] \tag{6.1}$$

$$U_{\text{MX}}(A) = \sum_{\tau_i \in A} y \quad \text{where } \tau_i \in \text{I}_k = (x, y]$$

Clearly, the following inequality holds for any task set A:

$$U_{\text{MN}}(A) < U(A) \leq U_{\text{MX}}(A) \tag{6.2}$$

If `IBPS` assigns a task set to m_x processors and $m_x \leq m$, we declare

SUCCESS for schedulability on m processors. Otherwise, we declare FAILURE. Therefore, `IBPS` can be used to determine: (i) *the number of processors needed to schedule a task set*, and (ii) *whether a task set is schedulable on* m *processors*. When used online, `O-IBPS` either accepts or rejects a new online task. Similarly, when a task leaves the system `O-IBPS` makes necessary changes (that is, reassignment of the existing tasks) so that new online tasks can be accepted to the system in future.

Task Splitting Algorithm: `IBPS` uses the algorithm SPLIT in Fig. 6.1 for task splitting. The input to algorithm SPLIT is a set X, containing an odd number of tasks. Each task in X, except the highest-priority one, is assigned to one of two selected (empty) processors. The highest-priority task is split and assigned to both processors. As will be evident in later sections, no more tasks will be assigned to these processors.

```
Algorithm SPLIT (TaskSet: X)
1. Let $\tau_k \in X$ such that $T_k \leq T_i$ for all $\tau_i \in X$
2. Split the task $\tau_k$ into subtasks $\tau_k'$ and $\tau_k''$ such that
3. ${\phi_k}'$=0,          $C_k' = C_k/2$,    $T_k' = T_k$
4. ${\phi_k}''$=$C_k/2$,   $C_k'' = C_k/2$,   $T_k'' = T_k$
5. Let set $X_1$ contains $\frac{|X-\{\tau_k\}|}{2}$ tasks from X-$\{\tau_k\}$
6. $X_2 = X - X_1 - \{\tau_k\}$
7. Assign ${\tau_k}'$ and all tasks in set $X_1$ to a processor
8. Assign ${\tau_k}''$ and all tasks in set $X_2$ to a processor
```

Figure 6.1: *Task Splitting Algorithm*

The highest priority task $\tau_k \in X$ is determined (line 1) and τ_k is split into two subtasks τ_k' and τ_k'' such that $u_{k'} = u_{k''} = \frac{u_k}{2}$ (line 2-4). Half of the tasks from set $(X - \{\tau_k\})$ is stored in a new set X_1 (line 5), and the remaining half in another new set X_2 (line 6). Note that $|X - \{\tau_k\}|$ is an even number. Subtask ${\tau_k}'$ and all tasks in X_1 are assigned to one processor (line 7), while subtask ${\tau_k}''$ and all tasks in X_2 are assigned to the other processor (line 8). Note that, since τ_k is the highest-priority task and no more tasks will be assigned to the selected processors, the offset of ${\tau_k}''$ ($\phi_{k''}$=$\frac{C_k}{2}$) will ensure that the two subtasks are not executed in parallel.

6.5 Task Assignment: First Phase

During the first phase, `IBPS` assigns tasks using a particular policy for each of the seven subintervals using load regulation strategy. Common for all policies, however, is that each processor to which tasks have been assigned will have an RM schedulable task set with a total utilization strictly greater than $\frac{4Q}{3}$ for load regulation. We now describe the seven policies used during the first task-assignment phase.

Policy 1: Each task $\tau_i \in \mathrm{I}_1$=$(\frac{4Q}{3}, 1\}$ is assigned to one dedicated processor. Clearly, each of these tasks is trivially RM schedulable with a utilization greater than $\frac{4Q}{3}$ on one processor. This policy guarantees that there will be no tasks left in I_1 after the first phase.

Policy 2: Exactly *three* tasks in I_2=$(\frac{8Q}{9}, \frac{4Q}{3}]$ are assigned to *two* processors using algorithm SPLIT given in Fig. 6.1. This process iterates until less than three tasks are left in I_2. Thus, there are 0–2 unassigned tasks in I_2 to declare as *odd* tasks.

The proof that the tasks assigned to each processor are RM schedulable and the utilization on each processor is greater than $\frac{4Q}{3}$ is as follows. Assume that a particular iteration assigns the three tasks τ_k, τ_i and τ_j from I_2 to two processors by calling SPLIT ($\{\tau_k$,τ_i,$\tau_j\}$) such that τ_k is the highest priority tasks. Then, line 3 of algorithm SPLIT ensures that $\frac{4Q}{9} < u_{k'} \leq \frac{2Q}{3}$. Now, without loss of generality, assume $X_1 = \{\tau_i\}$ in line 5 of SPLIT. We have the total minimum and maximum load in one processor $U_{\mathrm{MN}}(\{\tau_i, \tau_k'\}) = \frac{8Q}{9} + \frac{4Q}{9} = \frac{4Q}{3}$ and $U_{\mathrm{MX}}(\{\tau_i, \tau_k'\}) = \frac{4Q}{3} + \frac{2Q}{3} = 2Q = LLB(2)$, respectively, using Eq.(6.1). We have, $\frac{4Q}{3} < U(\{\tau_i, \tau_{k'}\}) \leq LLB(2)$ using Eq. (6.2). Similarly we have, $\frac{4Q}{3} < U(\{\tau_j, \tau_{k''}\}) \leq LLB(2)$.

Policy 3: Exactly *two* tasks from I_3=$(\frac{2Q}{3}, \frac{8Q}{9}]$ are assigned to one processor without any splitting. This process iterates until less than two tasks are left in I_3. Thus, there are 0–1 task left in I_3 to declare as *odd* tasks. Two tasks in I_3 must have a total utilization greater than $\frac{2\times 2Q}{3}$ and less than or equal to $\frac{2\times 8Q}{9}$, assigned to one processor. Since $\frac{16Q}{9}$ $<LLB(2)$=$2Q$, the two tasks from I_3 are RM schedulable with processor utilization greater than $\frac{4Q}{3}$.

Policy 4: Exactly *five* tasks from I_4=$(\frac{8Q}{15}, \frac{2Q}{3}]$ are assigned to *two* processors using algorithm SPLIT in Fig. 6.1. This process iterates until there are less than five tasks left in I_4. Thus, there are 0–4 unassigned tasks in I_4 to declare as *odd* tasks. The proof that the tasks in each processor are RM schedulable and the utilization in each processor is greater than $\frac{4Q}{3}$ is similar to the proof for

Policy 2 and can be found in [PJ09].

Policy 5 : Exactly *three* tasks from I$_5$=($\frac{4Q}{9}$, $\frac{8Q}{15}$] are assigned to one processor. This process iterates until there are less than three tasks left in I$_5$. Thus, there are 0–2 tasks left in I$_5$ to declare as *odd* tasks. Three tasks in I$_5$ must have a total utilization greater than $\frac{3\times 4Q}{9}$ and less than $\frac{3\times 8Q}{15} < LLB(3)$. So, each processor utilization is greater than $\frac{4Q}{3}$ and the three tasks in I$_5$ are RM schedulable on one processor.

Policy 6: Exactly four tasks from I$_6$=($\frac{Q}{3}$, $\frac{4Q}{9}$] are assigned to one processor. This process iterates until there is less than four tasks left in I$_6$. Thus, there are 0–3 tasks left in I$_6$ to declare as *odd* tasks. Four tasks in I$_6$ must have a utilization greater than $\frac{4\times Q}{3}$ and less than $\frac{4\times 4Q}{9} < LLB(4)$. So, each processor utilization is greater than $\frac{4Q}{3}$ and the four tasks in I$_6$ are RM schedulable on one processor.

Policy 7: In this policy, IBPS assigns tasks from I$_7$=$\{0, \frac{Q}{3}]$ using First-Fit (FF) bin packing allocation as in [DL78]. We denote the l^{th} processor by Θ_l and the total utilization (load) of Θ_l by $L(\Theta_l)$. Assume that Θ_p is the first considered processor and that Θ_q is the last considered processor, for $q \geq p$ using FF, in this policy. When a task $\tau_i \in$ I$_7$ cannot be feasibly assigned to Θ_l, for $l = p, (p+1), \ldots (q-1)$ we must have $L(\Theta_l) + u_i > LLB(\infty) = \ln 2$. Since $u_i \leq \frac{Q}{3}$, we have $L(\Theta_l) > \ln 2 - \frac{Q}{3} > \frac{4Q}{3}$. So, the total utilization of the tasks in each processor is greater than $\frac{4Q}{3}$, except possibly the last processor Θ_q. If $L(\Theta_q) \leq \frac{4Q}{3}$, the task assignment to Θ_q is undone and these unassigned tasks are called *residue* tasks in I$_7$. If $L(\Theta_q) > \frac{4Q}{3}$, all the tasks in I$_7$ assigned to processors $\Theta_p, \Theta_{p+1}, \ldots \Theta_q$ are RM schedulable with utilization in each processor greater than $\frac{4Q}{3}$.

Therefore, it is proved that, each processor to which tasks have been assigned in first phase will have an RM schedulable task set with a total utilization strictly greater than $\frac{4Q}{3}$. In rest of this chapter, to denote the set of residue tasks in I$_7$ we use

$$\text{S}=\{\tau_r |\ \tau_r \text{ is a residue task in I}_7\}$$

We have the following Lemma 1.

Lemma 1 *All the residue task in S are RM schedulable in one processor.*

Proof From policy 7, we have $U(S) \leq \frac{4Q}{3}$ (undone task assignment in Θ_q). Since $\frac{4Q}{3} < LLB(\infty) = \ln 2$, all tasks in S are RM schedulable in one processor.

Note that, first phase of task assignment runs in linear time due to its iterative nature.

6.6 Task Assignment: Second Phase

During the second phase, `IBPS` assigns unassigned tasks in subintervals I_2–I_6, referred to as *odd tasks*, using algorithm OddAssign in Fig. 6.2. Unlike the first phase, however, each processor can now be assigned tasks from more than one subinterval.

Algorithm OddAssign (Odd tasks in I_2–I_6):

```
1.  while both $I_2$ and $I_4$ has at least one task
2.    Assign $\tau_i \in I_2$ and $\tau_j \in I_4$ to a processor
3.  while both $I_2$ and $I_5$ has at least one task
4.    Assign $\tau_i \in I_2$ and $\tau_j \in I_5$ to a processor
5   while $I_3$ has one task and $I_6$ has two tasks
6.    Assign $\tau_i \in I_3$, $\tau_j \in I_6$ and $\tau_k \in I_6$ to a processor
7.  while (($I_4$ has one task and $I_5$ has two tasks)
8.          or ($I_4$ has two tasks and $I_5$ has one task))
9.    if ( $I_4$ has one task and $I_5$ has two tasks) then
10.     Assign $\tau_i \in I_4$, $\tau_j \in I_5$ and $\tau_k \in I_5$ to a processor
11.   else
12.     Assign $\tau_i \in I_4$, $\tau_j \in I_4$ and $\tau_k \in I_5$ to a processor
13.   end if
14. while $I_4$ has two tasks and $I_6$ has one task
15.   Assign $\tau_i \in I_4$, $\tau_j \in I_4$ and $\tau_k \in I_6$ to a processor
16. while each $I_3$, $I_5$ and $I_6$ has one task
17.   Assign $\tau_i \in I_3$, $\tau_j \in I_5$ and $\tau_k \in I_6$ to a processor
```

Figure 6.2: *Assignment of odd tasks in* I_2–I_6

Using algorithm OddAssignin Figure 6.2, tasks assigned in each iteration of each `while` loop are RM schedulable with total utilization is greater than $\frac{4Q}{3}$ on each processor due to load regulation strategy. We prove this by showing that the inequality $\frac{4Q}{3} < U(A) \leq LLB(|A|)$ holds (where A is the task set assigned to one processor) in each iteration of each `while` loop (here, named Loop 1–Loop 6). We now analyze each loop separately.

Loop 1 (line 1–2): Each iteration of this loop assigns A=$\{\tau_i,\tau_j\}$ to one processor such that $\tau_i \in$ I$_2$ $= (\frac{8Q}{9}, \frac{4Q}{3}]$ and $\tau_j \in$ I$_4$ $= (\frac{8Q}{15}, \frac{2Q}{3}]$. Thus, $U_{\text{MX}}(\text{A}) = \frac{4Q}{3} + \frac{2Q}{3} = 2Q = LLB(2)$ and $U_{\text{MN}}(\text{A}) = \frac{8Q}{9} + \frac{8Q}{15} = \frac{64Q}{45} > \frac{4Q}{3}$.

Loop 2 (line 3–4): Each iteration of this loop assigns A=$\{\tau_i,\tau_j\}$ to one processor such that $\tau_i \in$ I$_2$ $= (\frac{8Q}{9}, \frac{4Q}{3}]$ and $\tau_j \in$ I$_5$ $= (\frac{4Q}{9}, \frac{8Q}{15}]$. Thus, $U_{\text{MX}}(A) = \frac{4Q}{3} + \frac{8Q}{15} = \frac{28Q}{15} < LLB(2)$ and $U_{\text{MN}}(A) = \frac{8Q}{9} + \frac{4Q}{9} = \frac{4Q}{3}$.

Loop 3 (line 5–6): Each iteration of this loop assigns A=$\{\tau_i,\tau_j,\tau_k\}$ to one processor such that $\tau_i \in$ I$_3$ $= (\frac{2Q}{3}, \frac{8Q}{9}]$, $\tau_j \in$ I$_6$ $= (\frac{Q}{3}, \frac{4Q}{9}]$ and $\tau_k \in$ I$_6$ $= (\frac{Q}{3}, \frac{4Q}{9}]$. Thus, we have $U_{\text{MX}}(A) = \frac{8Q}{9} + \frac{4Q}{9} + \frac{4Q}{9} = \frac{16Q}{9} < LLB(3)$ and $U_{\text{MN}}(A) = \frac{2Q}{3} + \frac{Q}{3} + \frac{Q}{3} = \frac{4Q}{3}$.

Loop 4 (line 7–13): Each iteration of this loop assigns exactly three tasks A=$\{\tau_i,\tau_j,\tau_k\}$ to one processor, selecting the tasks from two subintervals I$_4$$=(\frac{8Q}{15}, \frac{2Q}{3}]$ and I$_5$=$(\frac{4Q}{9}, \frac{8Q}{15}]$. Tasks are assigned to one processor either if (i) $\tau_i \in$ I$_4$, $\tau_j \in$ I$_5$ and $\tau_k \in$ I$_5$ or (ii) $\tau_i \in$ I$_4$, $\tau_j \in$ I$_4$ and $\tau_k \in$ I$_5$. When (i) is true, we have $U_{\text{MX}}(A) = \frac{2Q}{3} + \frac{2\times 8Q}{15} = \frac{26Q}{15} < LLB(3)$ and $U_{\text{MN}}(A) = \frac{8Q}{15} + \frac{2\times 4Q}{9} = \frac{64Q}{45} > \frac{4Q}{3}$. When (ii) is true, we have $U_{\text{MX}}(A) = \frac{2\times 2Q}{3} + \frac{8Q}{15} = \frac{28Q}{15} < LLB(3)$ and $U_{\text{MN}}(A) = \frac{2\times 8Q}{15} + \frac{4Q}{9} = \frac{68Q}{45} > \frac{4Q}{3}$.

Loop 5 (line 14–15): Each iteration of this loop assigns A=$\{\tau_i,\tau_j,\tau_k\}$ to one processor such that $\tau_i \in$ I$_4$ $= (\frac{8Q}{15}, \frac{2Q}{3}]$, $\tau_j \in$ I$_4$ $= (\frac{8Q}{15}, \frac{2Q}{3}]$ and $\tau_k \in$ I$_6$ $= (\frac{Q}{3}, \frac{4Q}{9}]$. Thus, $U_{\text{MX}}(A) = \frac{2\times 2Q}{3} + \frac{4Q}{9} = \frac{16Q}{9} < LLB(3)$ and $U_{\text{MN}}(A) = \frac{2\times 8Q}{15} + \frac{Q}{3} = \frac{7Q}{5} > \frac{4Q}{3}$.

Loop 6 (line 16–17): Each iteration of this loop assigns A=$\{\tau_i,\tau_j,\tau_k\}$ to one processor such that $\tau_i \in$ I$_3$ $= (\frac{2Q}{3}, \frac{8Q}{9}]$, $\tau_j \in$ I$_5$ $= (\frac{4Q}{9}, \frac{8Q}{15}]$ and $\tau_k \in$ I$_6$ $= (\frac{Q}{3}, \frac{4Q}{9}]$. Thus, $U_{\text{MX}}(A) = \frac{8Q}{9} + \frac{8Q}{15} + \frac{4Q}{9} = \frac{28Q}{15} < LLB(3)$ and $U_{\text{MN}}(A) = \frac{2Q}{3} + \frac{4Q}{9} + \frac{Q}{3} = \frac{13Q}{9} > \frac{4Q}{3}$.

Using Eq. (6.2), we can thus conclude that, for each iteration of each loop if task set A is assigned to one processor, we have $\frac{4Q}{3} < U(A) \leq LLB(|A|)$. Therefore, it is proved that, each processor to which tasks have been assigned in second phase will have an RM schedulable task set with a total utilization strictly greater than $\frac{4Q}{3}$. The second task-assignment phase also runs in linear time due to its iterative nature.

Residue tasks

Unassigned tasks in I$_2$–I$_6$ after the second phase are called *residue tasks*. For example, if there are only two unassigned tasks in I$_2$ after the first phase, these two odd tasks cannot be assigned to a processor in the second phase. Such a scenario, henceforth referred to as a *possibility*, of residue tasks will have to be handled during the third phase. We identify all such possibilities of residue tasks in subintervals I_2–I_6 for any task set. In particular, we determine the number of residue tasks in each of the subintervals I_2–I_6 for each identified possibility.

After the first phase, I_2 has 0–2 odd tasks, I_3 has 0–1 odd task, I_4 has 0–4 odd tasks, I_5 has 0–2 odd tasks, and I_6 has 0–3 odd tasks (see Section 6.5). Odd tasks thus exist, in subintervals I_2–I_6, as one of $(3 \times 2 \times 5 \times 3 \times 4 =)360$ possibilities after the first phase. During the second phase, algorithm ODDASSIGN is able to assign *all* the odd tasks in subintervals I_2–I_6 for 316 out of the 360 possibilities of odd tasks after the first phase. It is easy to see that, this fact can be verified by running the algorithm ODDASSIGN for each the 360 possibilities of odd tasks after first phase and by counting how many possibilities remain unassigned (please also see Appendix B in [PJ09] for a formal proof). Therefore, for any task set, *those residue tasks in subintervals* I_2–I_6 *that need to be handled in the third phase exist as one of the remaining 44 different possibilities.* During the third phase, `IBPS` considers assigning any such possibility of residue tasks from I_2–I_6 including all residue tasks from I_7 to processors.

We now define some functions that will be used in the next sections. Function U_{RT} denotes the total utilization of all the residue tasks in I_2–I_7:

$$U_{\text{RT}} = \sum_{k=2}^{7} \sum_{\tau_i \in \text{I}_k} u_i \quad [\tau_i \text{ is residue task in } \text{I}_k]$$

Using function U_{RT} in Eq. (6.6), the utilization of a set of residue tasks is calculated. For the purpose of schedulability analysis, the next two functions U_{RMN} and U_{RMX} are defined to bound the utilization of a set of residue tasks from below and above, respectively. Functions U_{RMN} and U_{RMX} denote the lower and upper bound, respectively, on total utilization of all residue tasks $\tau_i \in \text{I}_k$ for $i = 2, \ldots 6$:

$$U_{\text{RMN}} = \sum_{k=2}^{6} \sum_{\tau_i \in \text{I}_k} x \quad \text{where } \tau_i \in \text{I}_k = (x, y]$$

$$U_{\text{RMX}} = \sum_{k=2}^{6} \sum_{\tau_i \in \text{I}_k} y \quad \text{where } \tau_i \in \text{I}_k = (x, y]$$

If one of I_2–I_6 is nonempty and task set S is the residue tasks in I_7, we have:

$$U_{\text{RMN}} < U_{\text{RT}} - U(S) \leq U_{\text{RMX}} \tag{6.3}$$

6.7 Task Assignment: Third Phase

During the third phase, `IBPS` assigns all residue tasks to processors, thereby completing the task assignment. Each of the 44 *possibilities* of residue tasks in I_2–I_6 is listed in a separate row of Table 6.2 (row 1-20), Table 6.3 (row 21-41), Table 6.4 (row 42-44). The columns of these table are organized as follows. The first column represents the *possibility* number. Columns two through six represent the number of residue tasks in each of the subintervals I_2–I_6, while the seventh column represents the total number of residue tasks in these subintervals. The eighth and ninth columns represent U_{RMN} and U_{RMX}, respectively. The rows of the table are divided into three categories, based on three value ranges of U_{RMN} as in the following equations:

$$\begin{array}{ll} \textbf{CAT-0 (row no 1–20):} & 0 < U_{\text{RMN}} \leq \frac{4Q}{3} \\ \textbf{CAT-1 (row no 21–41):} & \frac{4Q}{3} < U_{\text{RMN}} \leq \frac{8Q}{3} \\ \textbf{CAT-2 (row no 42–44):} & \frac{8Q}{3} < U_{\text{RMN}} \leq 4Q \end{array} \tag{6.4}$$

We now present the task-assignment algorithms for the three categories (given in Eq. (6.4))of residue tasks in I_2–I_6 and the residue tasks in I_7, whose collective purpose is to guarantee that, *if $U_{RT} \leq \frac{4Qm''}{3}$ for the smallest non-negative integer m'', all residue tasks are assigned to at most m'' processors*.

6.7.1 Residue Task Assignment: CAT-0

Consider the first 20 possibilities (rows 1–20 in Table 6.2) of CAT-0 residue tasks and all residue task in I_7. Assign tasks to processor as follows. *If $U_{RT} \leq \frac{4Q}{3}$, all residue tasks in* I_2–I_7 *are assigned to one processor. If $U_{RT} > \frac{4Q}{3}$, all residue tasks in* I_7 *are assigned to one processor and all CAT-0 residue tasks in* I_2–I_6 *are assigned to another processor.*

We prove the RM schedulability by considering two cases—case (i): $U_{\text{RT}} \leq \frac{4Q}{3}$, and case (ii): $U_{\text{RT}} > \frac{4Q}{3}$. If case (i) is true, all residue tasks in I_2–I_7 are assigned to one processor. Since $U_{\text{RT}} \leq \frac{4Q}{3} < LLB(\infty) = \ln 2$, all residue tasks are RM schedulable on one processor. If case (ii) applies, all residue tasks in I_7 assigned to one processor are RM schedulable using Lemma 1. Next, the CAT-0 residue tasks in I_2–I_6 are assigned to another processor.

No.	I_2	I_3	I_4	I_5	I_6	Total	U_{RMN}	U_{RMX}
CAT-0								
1	0	0	0	0	1	1	$\frac{Q}{3}$	$\frac{4Q}{9}$
2	0	0	0	1	0	1	$\frac{4Q}{9}$	$\frac{8Q}{15}$
3	0	0	1	0	0	1	$\frac{8Q}{15}$	$\frac{2Q}{3}$
4	0	1	0	0	0	1	$\frac{2Q}{3}$	$\frac{8Q}{9}$
5	0	0	0	0	2	2	$\frac{2Q}{3}$	$\frac{8Q}{9}$
6	0	0	0	1	1	2	$\frac{7Q}{9}$	$\frac{44Q}{45}$
7	0	0	0	2	0	2	$\frac{8Q}{9}$	$\frac{16Q}{15}$
8	0	0	1	0	1	2	$\frac{13Q}{15}$	$\frac{10Q}{9}$
9	0	0	1	1	0	2	$\frac{44Q}{45}$	$\frac{18Q}{15}$
10	0	1	0	0	1	2	Q	$\frac{4Q}{3}$
11	1	0	0	0	0	1	$\frac{8Q}{9}$	$\frac{4Q}{3}$
12	0	0	0	0	3	3	Q	$\frac{4Q}{3}$
13	0	0	2	0	0	2	$\frac{16Q}{15}$	$\frac{4Q}{3}$
14	0	1	0	1	0	2	$\frac{10Q}{9}$	$\frac{64Q}{45}$
15	0	0	0	1	2	3	$\frac{10Q}{9}$	$\frac{64Q}{45}$
16	0	0	0	2	1	3	$\frac{11Q}{9}$	$\frac{68Q}{45}$
17	0	1	1	0	0	2	$\frac{18Q}{15}$	$\frac{14Q}{9}$
18	0	0	1	0	2	3	$\frac{18Q}{15}$	$\frac{14Q}{9}$
19	0	0	1	1	1	3	$\frac{59Q}{45}$	$\frac{74Q}{45}$
20	1	0	0	0	1	2	$\frac{11Q}{9}$	$\frac{16Q}{9}$

Table 6.2: *All 20 possibilities (row 1-20) of CAT-0 Residue tasks in* I_2–I_6

According to column seven of Table 6.2, the number of CAT-0 residue tasks in I_2–I_6 is at most 3. And observing the ninth column, we find that the maximum total utilization (U_{RMX}) of all CAT-0 residue tasks in I_2–I_6 for any row 1–20 is $\frac{16Q}{9} \approx 0.736$ (see row 20). Since $\frac{16Q}{9} < LLB(3) \approx 0.779$, all CAT-0 residue tasks from I_2–I_6 are RM schedulable on the second processor. In summary, *for CAT-0 residue tasks, if $U_{RT} \leq \frac{4Q}{3}$, we need one processor, otherwise, we need at most two processors to assign all the residue tasks in* I_2–I_7.

6.7.2 Residue Task Assignment: CAT-1

Before we propose the task assignment algorithms for this category, consider the following Theorem from [LGDG03] that is used to assign total t tasks in s processors using RMFF algorithm[3].

[3]Symbols n and m in [LGDG03] are renamed as s and t in this thesis for clarity.

No.	I$_2$	I$_3$	I$_4$	I$_5$	I$_6$	Total	U_{RMN}	U_{RMX}
CAT-1								
21	0	0	0	1	3	4	$\frac{13Q}{9}$	$\frac{28Q}{15}$
22	0	1	0	2	0	3	$\frac{14Q}{9}$	$\frac{88Q}{45}$
23	0	0	0	2	2	4	$\frac{14Q}{9}$	$\frac{88Q}{45}$
24	0	0	1	0	3	4	$\frac{23Q}{15}$	2Q
25	0	0	3	0	0	3	$\frac{24Q}{15}$	2Q
26	0	1	1	0	1	3	$\frac{23Q}{15}$	2Q
27	0	1	1	1	0	3	$\frac{74Q}{45}$	$\frac{94Q}{45}$
28	0	0	1	1	2	4	$\frac{74Q}{45}$	$\frac{94Q}{45}$
29	1	1	0	0	0	2	$\frac{14Q}{9}$	$\frac{20Q}{9}$
30	0	1	2	0	0	3	$\frac{26Q}{15}$	$\frac{20Q}{9}$
31	1	0	0	0	2	3	$\frac{14Q}{9}$	$\frac{20Q}{9}$
32	0	0	0	2	3	5	$\frac{17Q}{9}$	$\frac{12Q}{5}$
33	0	0	1	1	3	5	$\frac{89Q}{45}$	$\frac{38Q}{15}$
34	1	0	0	0	3	4	$\frac{17Q}{9}$	$\frac{8Q}{3}$
35	0	0	4	0	0	4	$\frac{32Q}{15}$	$\frac{8Q}{3}$
36	1	1	0	0	1	3	$\frac{17Q}{9}$	$\frac{8Q}{3}$
37	2	0	0	0	0	2	$\frac{16Q}{9}$	$\frac{8Q}{3}$
38	2	0	0	0	1	3	$\frac{19Q}{9}$	$\frac{28Q}{9}$
39	2	1	0	0	0	3	$\frac{22Q}{9}$	$\frac{32Q}{9}$
40	0	1	3	0	0	4	$\frac{34Q}{15}$	$\frac{26Q}{9}$
41	2	0	0	0	2	4	$\frac{22Q}{9}$	$\frac{32Q}{9}$

Table 6.3: *All 21 possibilities (row 21-41) of CAT-1 Residue tasks in* I$_2$–I$_6$

No.	I$_2$	I$_3$	I$_4$	I$_5$	I$_6$	Total	U_{RMN}	U_{RMX}
CAT-2								
42	2	1	0	0	1	4	$\frac{25Q}{9}$	4Q
43	2	0	0	0	3	5	$\frac{25Q}{9}$	4Q
44	0	1	4	0	0	5	$\frac{14Q}{5}$	$\frac{32Q}{9}$

Table 6.4: *All 3 possibilities (row 42-44) of CAT-2 Residue tasks in* I$_2$–I$_6$

Theorem 1 (from [LGDG03]) *All t tasks in set A are schedulable on s processors using RMFF, if $U(A) \leq (s-1)Q + (t-s+1)(2^{\frac{1}{t-s+1}} - 1)$.*

We denote the bound given in Theorem 1 as follows:

$$U(s,t) = (s-1)Q + (t-s+1)(2^{\frac{1}{t-s+1}} - 1)$$

Consider the next 21 possibilities (rows 21–41 in Table 6.3) of CAT-1 residue tasks and residue tasks in I$_7$. If $U_{\text{RT}} \leq \frac{8Q}{3}$, we assign all CAT-1 residue tasks in I_2–I_6 for rows 21–41 and the residue tasks in I_7 to at most two processors, otherwise, to at most three processors. For the first case, *if $U_{RT} \leq \frac{8Q}{3}$, all CAT-1 residue tasks in I_2–I_7 are assigned using RMFF allocation to two processors.* We prove the RM schedulability in Lemma 2.

Lemma 2 *If $U_{RT} \leq \frac{8Q}{3}$, then all the CAT-1 residue tasks in I_2–I_6 and the residue tasks in I_7 are RM schedulable on two processors using FF allocation.*

Proof According to Theorem 1, the value of $U(s,t)$ for $s = 2$ is given as $U(2,t) = Q + (t-1)(2^{\frac{1}{(t-1)}} - 1)$. Note that, for rows 21–41, the number of residue tasks $t \geq 2$. The function $U(2,t)$ is monotonically non-increasing as t increases. The minimum of $U(2,t)$ is $Q + \ln 2$ as $t \to \infty$. Therefore, $U(2,t) \geq Q + \ln 2 = 1.10736$ for any t. Since $U_{\mathrm{RT}} \leq \frac{8Q}{3} = 1.10456$, we have $U_{\mathrm{RT}} < U(2,t)$. Using Theorem 1, all CAT-1 residue tasks in I_2–I_6 and the residue tasks in I_7 are RM schedulable on two processors if $U_{\mathrm{RT}} \leq \frac{8Q}{3}$.

For the second case, *if $U_{RT} > \frac{8Q}{3}$, we assign all residue tasks in I_7 to one processor (RM schedulable using Lemma 1). And, all residue tasks in I_2–I_6 are assigned to at most two processors using algorithms `R21_37` , `R38` , `R39` , and `R40_41` for row 21–37, row 38, row 39 and row 40–41 in Table 6.3, respectively.* Next, we present each of these algorithms and show that all CAT-1 residue tasks in I_2–I_6 are RM schedulable on at most two processors.

Algorithm `R21_37` : All residue tasks in I_2–I_6 for rows 21–37 are assigned to two processors using FF allocation. Such tasks are RM schedulable using Lemma 3.

Lemma 3 *All the residue tasks in I_2–I_6 given in any row of 21–37 of Table 6.3 are RM schedulable on two processors using FF allocation.*

Proof According to column seven in Table 6.3 for rows 21–37, the number of residue tasks in I_2–I_6 is at most 5. Therefore, $U(2,t)$ is minimized for rows 21–37 when $t = 5$, and we have, $U(2,5) = Q + 4(2^{\frac{1}{4}} - 1) \approx 1.17$. Observing the ninth column of rows 21–37, we find that the maximum total utilization (U_{RMX}) of the residue tasks in I_2–I_6 is at most $\frac{8Q}{3} \approx 1.105$ (see row 37). Since $\frac{8Q}{3} < U(2,5)$, all the $t \leq 5$ residue tasks in I_2–I_6 for any row 21–37 are RM schedulable on two processors using FF allocation.

Algorithm `R38` : For row 38, there are two tasks in I_2 and one task in I_6. Assume that $\tau_a \in \mathrm{I}_2$, $\tau_b \in \mathrm{I}_2$ and $\tau_c \in \mathrm{I}_6$. Task $\tau_a \in \mathrm{I}_2$ is assigned to one dedicated processor and therefore trivially RM schedulable. Then, $\tau_b \in \mathrm{I}_2 = (\frac{8Q}{9}, \frac{4Q}{3}]$ and $\tau_c \in \mathrm{I}_6 = (\frac{Q}{3}, \frac{4Q}{9}]$ are assigned to another processor. Since

$u_b \leq \frac{4Q}{3}$ and $u_c \leq \frac{4Q}{9}$, we have $u_b + u_c \leq \frac{4Q}{3} + \frac{4Q}{9} = \frac{16Q}{9} < 2Q = LLB(2)$. So, τ_b and τ_c are schedulable on one processor. All three residue tasks in row 38 are thus RM schedulable on two processors.

Algorithm `R39` : For row 39, there are two tasks in I_2 and one task in I_3. Assume that $\tau_a \in \mathrm{I}_2$, $\tau_b \in \mathrm{I}_2$ and $\tau_c \in \mathrm{I}_3$. We assign these tasks by calling SPLIT ($\{\tau_a,\tau_b,\tau_c\}$) where the highest-priority task is split. We prove that task τ_a, τ_b and τ_c are RM schedulable on two processors in Lemma 4.

Lemma 4 *If $\tau_a \in \mathrm{I}_2$, $\tau_b \in \mathrm{I}_2$ and $\tau_c \in \mathrm{I}_3$, then all three tasks are RM schedulable in two processors using* SPLIT *($\{\tau_a,\tau_b,\tau_c\}$).*

Proof It has already been proven (Policy 2 in Section 6.5) that three tasks in I_2 are RM schedulable on two processors using SPLIT . The utilization of a task from I_3 is smaller than that of a task in I_2. Hence, two task from I_2 and one from I_3 are also RM schedulable using SPLIT on two processors.

Algorithm `R40_41` : Four residue tasks either in row 40 or in row 41 are scheduled on two processors as in Fig. 6.3. We prove the RM schedulability of these tasks in Lemma 5.

Algorithm `R40_41` (Residue tasks for row 40 or 41)

```
1. Select $\tau_a$ and $\tau_b$ from two different subintervals
2. Let $\tau_c$ and $\tau_d$ be the remaining residue tasks
3. Assign $\tau_a$, $\tau_b$ to one processor
4. Assign $\tau_c$, $\tau_d$ to one processor
```

Figure 6.3: *Residue Task Assignment (row 40 or row 41)*

Lemma 5 *The residue tasks in row 40 or 41 in Table 6.3 are RM schedulable on two processors using algorithm `R40_41` .*

Proof The four residue tasks either in row 40 or row 41 are from exactly two subintervals. For row 40, there is one residue task in I_3=($\frac{2Q}{3}$, $\frac{8Q}{9}$] and three residue tasks in I_4=($\frac{8Q}{15}$, $\frac{2Q}{3}$]. For row 41, there are two residue tasks in each of I_2=($\frac{8Q}{9}$, $\frac{4Q}{3}$] and I_6=($\frac{Q}{3}$, $\frac{4Q}{9}$]. Now, consider two cases: case (i) for row 40 and case (ii) for row 41.

Case (i): Two tasks from two different subintervals of row 40 (line 1) satisfy $\tau_a \in \mathrm{I}_3$ and $\tau_b \in \mathrm{I}_4$. We then have $\tau_c \in \mathrm{I}_4$ and $\tau_d \in \mathrm{I}_4$ (line 2). So, $u_a \leq \frac{8Q}{9}$,

$u_b \leq \frac{2Q}{3}$, $u_c \leq \frac{2Q}{3}$ and $u_d \leq \frac{2Q}{3}$. Task τ_a and τ_b are assigned to one processor (line 3); we have $u_a + u_b \leq \frac{8Q}{9} + \frac{2Q}{3} = \frac{14Q}{9} < 2Q = LLB(2)$. Task τ_c and τ_d are assigned to one processor (line 4); we have $u_c + u_d \leq \frac{2Q}{3} + \frac{2Q}{3} = \frac{4Q}{3} < 2Q = LLB(2)$. Thus, all residue tasks in row 40 are RM schedulable on two processors.

Case (ii): Two tasks from two different subintervals of row 41 (line 1), satisfy $\tau_a \in \mathrm{I}_2$ and $\tau_b \in \mathrm{I}_6$. We then have $\tau_c \in \mathrm{I}_2$ and $\tau_d \in \mathrm{I}_6$ (line 2). So, $u_a \leq \frac{4Q}{3}$, $u_b \leq \frac{4Q}{9}$, $u_c \leq \frac{4Q}{3}$ and $u_d \leq \frac{4Q}{9}$. Task τ_a and τ_b are assigned to one processor (line 3); we have $u_a + u_b \leq \frac{4Q}{3} + \frac{4Q}{9} = \frac{16Q}{9} < 2Q = LLB(2)$. Similarly, task τ_c and τ_d are assigned to one processor (line 4); we have $u_c + u_d \leq LLB(2)$. Thus, all residue tasks in row 41 are RM schedulable on two processors.

In summary, for CAT-1 residue tasks *If $U_{RT} \leq \frac{8Q}{3}$, then* `IBPS` *needs to assign all residue tasks in* I_2–I_7 *to at most two processors, and otherwise* `IBPS` *needs at most three processors.*

6.7.3 Residue Task Assignment: CAT-2

We now consider the last three possibilities (rows 42–44 in Table 6.4) of CAT-2 residue tasks in I_2–I_6 and the residue tasks in I_7. We propose two task assignment algorithms `R42` and `R43_44` for residue tasks in row 42 and rows 43–44, respectively, along with all residue tasks in I_7.

Algorithm `R42` : The algorithm `R42` presented in Figure 6.4 assigns the four CAT-2 residue tasks in row 42 and residue tasks in I_7. The RM schedulability using algorithm `R42` is ensured as follows. There are four residue tasks in row 42 such that $\tau_a \in \mathrm{I}_2$, $\tau_b \in \mathrm{I}_2$, $\tau_c \in \mathrm{I}_3$ and $\tau_d \in \mathrm{I}_6$ (line 1 in Fig. 6.4). Tasks τ_a, τ_b and τ_c are assigned to two processors (line 2) by calling SPLIT ($\{\tau_a$,τ_b,$\tau_c\}$). Such three tasks are RM schedulable on two processors according to Lemma 4.

Next, residue task $\tau_d \in \mathrm{I}_6$ and all residue tasks in I_7 are assigned to one or two processors depending on two cases: case (i) $U_{\text{RT}} \leq 4Q$ and, case (ii) $U_{\text{RT}} > 4Q$ respectively.

Case (i) $U_{\text{RT}} \leq 4Q$: When $U_{\text{RT}} \leq 4Q$ (line 3), task τ_d and all residue tasks in I_7 are assigned to a third processor (line 4). So, to ensure RM schedulability on one processor, we prove that, $u_d + U(S) \leq LLB(\infty) \approx 0.693$ where S is the set of residue tasks in I_7. For row 42, we have $U_{\text{RT}} = u_a + u_b + u_c + u_d + U(S)$ and $U_{\text{RMN}} < u_a + u_b + u_c + u_d$. Therefore, $U_{\text{RT}} > U_{\text{RMN}} + U(S)$. Since,

Algorithm R42 (Residue task in I$_2$–I$_7$ in row 42)

```
1. Let $\tau_a \in$ I$_2$, $\tau_b \in$ I$_2$, $\tau_c \in$ I$_3$ and $\tau_d \in$ I$_6$
2. SPLIT ($\{\tau_a,\tau_b,\tau_c\}$)
3. if ($U_{\text{RT}} \leq 4Q$) then
4.    Assign $\tau_d$ and all tasks of I$_7$ to one processor
5. else
6.    Assign $\tau_d$ to one processor.
7.    Assign all tasks of I$_7$ (if any) to one processor
8. end if
```

Figure 6.4: *Residue Assignment(row 42)*

for row 42, $U_{\text{RMN}} = \frac{25Q}{9}$ (see eighth column of row 42) and $U_{\text{RT}} \leq 4Q$ (case assumption), we have $U(S) < 4Q - \frac{25Q}{9} = \frac{11Q}{9}$. Since $\tau_d \in$ I$_6$ $= (\frac{Q}{3}, \frac{4Q}{9}]$, we have $u_d + U(S) \leq \frac{4Q}{9} + \frac{11Q}{9} = \frac{15Q}{9} \approx 0.69035$. Therefore, $u_d + U(S) \leq LLB(\infty) \approx 0.693$.

Case (ii) $U_{\textbf{RT}} > 4Q$**:** When $U_{\text{RT}} > 4Q$, task τ_d is assigned to the third processor (line 6), which is trivially RM schedulable. All residue tasks in I$_7$ are scheduled on another processor (line 7), which are also RM schedulable according to Lemma 1. So, task τ_d and all residue tasks in I$_7$ are RM schedulable on two processor for row 42 whenever $U_{\text{RT}} > 4Q$.

In summary, *If $U_{RT} \leq 4Q$, the four CAT-2 residue tasks in row 42 and all residue tasks in* I$_7$ *are RM schedulable on at most three processors; otherwise, the tasks are schedulable on at most four processors using R42.*

Algorithm R43_44 : Algorithm R43_44 (see Fig. 6.5) assigns the five CAT-2 residue tasks in row 43 or row 44 and residue tasks in I$_7$.

The RM schedulability using algorithm R43_44 is ensured as follows. The five tasks in row 43 or in row 44 are denoted by τ_a, τ_b, τ_c, τ_d and τ_e. Tasks τ_a and τ_b are from two different subintervals (line 1). For row 43, there are two residue tasks in I$_2$=$(\frac{8Q}{9}, \frac{4Q}{3}]$ and three residue tasks in I$_6$=$(\frac{Q}{3}, \frac{4Q}{9}]$. For row 44, there is one residue task in I$_3$=$(\frac{2Q}{3}, \frac{8Q}{9}]$ and four residue tasks in I$_4$=$(\frac{8Q}{15}, \frac{2Q}{3}]$. Tasks τ_a and τ_b are assigned to one processor (line 4). For row 43, $\tau_a \in$ I$_2$ and $\tau_b \in$ I$_6$ and we have $u_a + u_b \leq \frac{4Q}{3} + \frac{4Q}{9} = \frac{16Q}{9} < LLB(2)$. For row 44, $\tau_a \in$ I$_3$ and $\tau_b \in$ I$_4$, we have $u_a + u_b \leq \frac{8Q}{9} + \frac{2Q}{3} = \frac{14Q}{9} < LLB(2)$. So, tasks τ_a and τ_b are RM schedulable on one processor for any row 43 or 44. To prove the RM schedulability of τ_c, τ_d, τ_e and all residue tasks in I$_7$, we consider two cases: case (i): $u_c + u_d + u_e \leq LLB(3)$ (line 5), case (ii): $u_c + u_d + u_e > LLB(3)$ (line 8).

Algorithm `R43_44` (Residue tasks for row 43 or 44)

```
1.  Select $\tau_a$ and $\tau_b$ in different subintervals of I$_2$–I$_6$
2.  Let $\tau_c$, $\tau_d$ and $\tau_e$ be the remaining tasks in I$_2$–I$_6$
3.    such that, $u_c \geq u_d$ and $u_c \geq u_e$
4.  Assign $\tau_a$ and $\tau_b$ to one processor.
5.  if $(U(\tau_c) + U(\tau_d) + U(\tau_e)) \leq LLB(3)$ then
6.    Assign $\tau_c$, $\tau_d$ and $\tau_e$ to a processor.
7.    Assign all tasks of I$_7$ (if any) to a processor
8.  else
9.    Assign $\tau_c$ and $\tau_d$ to a processor.
10.   if $(U_{\text{RT}} \leq 4Q)$ then
11.     Assign $\tau_e$ and all tasks of I$_7$ to a processor
12.   else
13.     Assign $\tau_e$ to one processor.
14.     Assign all tasks of I$_7$ (if any) to a processor
15.   end if
16. end if
```

Figure 6.5: *Residue Assignment (row 43-44)*

Case (i): Here, τ_c, τ_d and τ_e are assigned to a second processor (line 6) and they are RM schedulable (case assumption). All residue tasks in I$_7$ are assigned to a third processor (line 7) and are RM schedulable using Lemma 1.

Case (ii): Here, τ_c and τ_d are assigned to a second processor (line 9). For row 43, $\tau_c \in \text{I}_2 = (\frac{8Q}{9}, \frac{4Q}{3}]$ since $u_c \geq u_d$ and $u_c \geq u_e$ (line 3). Then obviously, $\tau_d \in \text{I}_6 = (\frac{Q}{3}, \frac{4Q}{9}]$ for row 43.

We have $u_c + u_d \leq \frac{4Q}{3} + \frac{4Q}{9} = \frac{16Q}{9} < LLB(2)$. For row 44, both τ_c and τ_d are in I$_4$=$(\frac{8Q}{15}, \frac{2Q}{3}]$, and we have $u_c + u_d \leq \frac{2Q}{3} + \frac{2Q}{3} = \frac{4Q}{3} < LLB(2)$. So, τ_c and τ_d assigned to one processor (line 9) are RM schedulable for row 43 or 44. Next, task τ_e and all residue tasks in I$_7$ are scheduled on one or two processors depending on two subcases: subcase (i): $U_{RT} \leq 4Q$ (line 10) and, subcase (ii): $U_{RT} > 4Q$ (line 12).

Subcase(i): When $U_{RT} \leq 4Q$, task τ_e and all residue tasks in I$_7$ are assigned to a third processor (line 11). For RM schedulability, we show that $u_e + U(S) \leq LLB(\infty) = \ln 2$ where S is the set of residue tasks in I$_7$. Note that $U_{\text{RMN}} + U(S) < U_{\text{RT}} \leq 4Q$ for this subcase. For row 43 $U_{\text{RMN}} = \frac{25Q}{9}$ (see column 8) and we have $U(S) \leq 4Q - \frac{25Q}{9} = \frac{11Q}{9}$. Since $\tau_e \in \text{I}_6 = (\frac{Q}{3}, \frac{4Q}{9}]$ for row 43, we have $u_e + U(S) \leq \frac{4Q}{9} + \frac{11Q}{9} = \frac{15Q}{9} \approx 0.6903 < \ln 2$.

For row 44 in Table 6.4, since $\tau_a \in \text{I}_3 = (\frac{2Q}{3}, \frac{8Q}{9}]$, $\tau_b \in \text{I}_4 = (\frac{8Q}{15}, \frac{2Q}{3}]$ and

$u_c + u_d + u_e > LLB(3) = 3(2^{\frac{1}{3}} - 1)$ for case (ii) we have, $u_a + u_b + u_c + u_d + u_e > \frac{2Q}{3} + \frac{8Q}{15} + 3(2^{\frac{1}{3}} - 1) = \frac{6Q}{5} + 3(2^{\frac{1}{3}} - 1)$. Since $U(S) + u_a + u_b + u_c + u_d + u_e = U_{\text{RT}}$ and $U_{\text{RT}} \leq 4Q$ (subcase assumption), we have $U(S) \leq 4Q - (\frac{6Q}{5} + 3(2^{\frac{1}{3}} - 1)) = \frac{14Q}{5} - 3(2^{\frac{1}{3}} - 1)$. Since $\tau_e \in \text{I}_4 = (\frac{8Q}{15}, \frac{2Q}{3}]$ for row 44, we have $u_e + U(S) \leq \frac{2Q}{3} + \frac{14Q}{5} - 3(2^{\frac{1}{3}} - 1) \approx 0.6561 < \ln 2 = LLB(\infty)$. So, τ_e and all residue tasks from I_7 are schedulable on one processor for row 43 or row 44 if $U_{RT} \leq 4Q$. So, in order to assign τ_a, τ_b, τ_c, τ_d and τ_e and residue tasks in I_7 we need at most three processors if $U_{RT} \leq 4Q$.

Subcase(ii): When $U_{RT} > 4Q$, task τ_e is assigned to a third processor (line 13) and is trivially RM schedulable. All residue tasks in I_7 are assigned to a fourth processor (line 14) and RM schedulable using Lemma 1. So, in order to assign τ_a, τ_b, τ_c, τ_d and τ_e and residue tasks in I_7, we need at most four processors if $U_{RT} > 4Q$.

In summary, *if $U_{\text{RT}} \leq 4Q$, we assign residue tasks in* I_2–I_6 *and residue tasks for* I_7 *to at most three processors; otherwise, these residue tasks are assigned to at most four processors.*

From the scheduling analysis in this section, we have the following fact.

Fact-1. Any CAT-x residue tasks, for $x = 0, 1, 2$, and residue tasks from I_7 are RM schedulable on at most $(x+1)$ processors if $U_{\text{RT}} \leq \frac{4Q(x+1)}{3}$; otherwise, these residue tasks are schedulable on at most $(x + 2)$ processors. We have the following Theorem 2.

Theorem 2 *If $U_{RT} \leq \frac{4Qm''}{3}$, for the smallest non-negative integer m'', all the residue tasks are RM schedulable on at most m'' processors.*

Proof Note that, if residue tasks only exist in I_7, our theorem is true because of Lemma 1. Now, consider any CAT-x residue tasks and residue tasks from I_7. For CAT-x residue tasks, we have $U_{\text{RT}} > U_{\text{RMN}} > \frac{4Qx}{3}$ using Eq. (6.3)–(6.4). If $U_{\text{RT}} \leq \frac{4Q(x+1)}{3}$, then all the residue tasks are RM schedulable on $(x + 1)$ processors (using Fact-1). Note that $m'' = (x + 1)$ is the smallest non-negative integer such that $U_{RT} \leq \frac{4Qm''}{3}$. Now, if $U_{\text{RT}} > \frac{4Q(x+1)}{3}$, then $U_{\text{RT}} \leq \frac{4Qm''}{3}$ for some $m'' \geq (x + 2)$. Since, using Fact-1 in such case, all residue tasks are assigned to at most $(x + 2)$ processors, our theorem is true for the smallest nonnegative integer $m'' \geq (x + 2)$ such that $U_{\text{RT}} \leq \frac{4Qm''}{3}$.

Task assignment to processor completes here. The task assignment algorithms in this phase also runs in linear time.

6.8 Performance of `IBPS`

Utilization Bound: The worst-case utilization bound of `IBPS` is given in Theorem 3.

Theorem 3 *If $U(\Gamma) \leq \frac{4Qm}{3}$, all tasks meet deadlines on at most m processors using `IBPS` .*

Proof As shown in Section 6.5–6.6, each processor, to which tasks have been assigned during the first two phases, will have an RM schedulable task set with a total utilization strictly greater than $\frac{4Q}{3}$ due to load regulation. Let $m' \geq 0$ be the number of processors to which tasks are assigned during the first two phases. Theorem 2 states that, if $U_{RT} \leq \frac{4Qm''}{3}$ for the smallest non-negative integer m'', then all residue tasks are RM schedulable on at most m'' processors during the third phase. We must show that, if $U(\Gamma) \leq \frac{4Qm}{3}$, then $(m' + m'') \leq m$. Since $U_{RT} \leq \frac{4Qm''}{3}$ for the smallest integer m'', we have $\frac{4Q(m''-1)}{3} < U_{\mathrm{RT}}$. The total utilization of tasks assigned to m' processors during the first two phases is $U(\Gamma) - U_{\mathrm{RT}}$. Therefore, $U(\Gamma) - U_{\mathrm{RT}} > \frac{4Qm'}{3}$ and we have, $\frac{4Q(m''-1)}{3} + \frac{4Qm'}{3} < U(\Gamma)$. If we have $U(\Gamma) \leq \frac{4Qm}{3}$, then $\frac{4Q(m'+m''-1)}{3} < \frac{4Qm}{3}$ which implies $(m' + m'' - 1) < m$. Because m', m'', and m are non-negative integers, we have $(m' + m'') \leq m$. So, if $U(\Gamma) \leq \frac{4Qm}{3}$, all tasks in set Γ are RM schedulable on at most m processors. Since $Q = (\sqrt{2} - 1)$, the utilization bound on m processors is $\frac{4(\sqrt{2}-1)m}{3} \approx 55.2\%$.

Resource Augmentation: Resource augmentation compares a given algorithm against an optimal algorithm by determining the factor by which if a given multiprocessor platform is augmented, then the given algorithm has equal performance to the optimal. We find the resource augmentation factor for `IBPS` as follows: given a task set Γ known to be feasible on m processors each having speed ζ, we determine the multiplicative factor of this speed by which the platform of `IBPS` can be augmented so that Γ is schedulable using `IBPS` . Baruah and Fisher [BF05] proved that, *if task system Γ is feasible (under either partitioned or global paradigm) on an identical multiprocessor platform comprised of m processors each with speed ζ, then we must have $m\zeta \geq U(\Gamma)$.* Now, if $\frac{4Q}{3} \geq \zeta$, then $m \geq \frac{3m\zeta}{4Q}$. Using the necessary condition $m\zeta \geq U(\Gamma)$ for feasibility in [BF05], we have $m \geq \frac{3U(\Gamma)}{4Q} \Leftrightarrow U(\Gamma) \leq \frac{4Qm}{3}$. According to Theorem 3, Γ is schedulable on m unit-capacity processors. Therefore, the processor speed-up factor for `IBPS` is $\frac{1}{\zeta} \geq \frac{3}{4Q} \approx 1.81$.

6.9 Admission Controller OIBPS

In this section, an efficient admission controller for online task scheduling, called O-IBPS , is presented. When a multiprocessor scheduling algorithm is used on-line, the challenge is to determine how a new on-line task τ_{new} is accepted and assigned to a processor. In O-IBPS , *if* $U(\Gamma) + u_{new} \leq \frac{4Qm}{3}$, *the new task* τ_{new} *is accepted to the system.* Theorem 3 ensures that we have sufficient capacity to assign the new task τ_{new} using IBPS . If $u_{new} \in \mathrm{I}_1$, we assign this new task to a dedicated processor. Otherwise, if $\tau_{new} \in \mathrm{I}_k$ for some $k = 2, 3, \ldots 7$, we have $u_{new} \leq \frac{4Q}{3}$. Let Γ_{R} denote the set of residue tasks before τ_{new} is assigned to a processor such that $\frac{4Q(m''-1)}{3} \leq U(\Gamma_{\mathrm{R}}) \leq \frac{4Qm''}{3}$. These residue tasks in Γ_{R} were assigned on at most m'' processors (using Theorem 2) before τ_{new} arrives to the system. O-IBPS then forms a new task set $\Gamma_{\mathrm{new}} = \Gamma_{\mathrm{R}} \cup \{\tau_{new}\}$. If $U(\Gamma_{new}) \leq \frac{4Qm''}{3}$, Γ_{new} is assigned to m'' processors using the IBPS task assignment phases. If $U(\Gamma_{new}) > \frac{4Qm''}{3}$, then we have, $U(\Gamma_{new}) = U_{\mathrm{R}} + u_{new} \leq \frac{4Qm''}{3} + \frac{4Q}{3} = \frac{4Q(m''+1)}{3}$. The task set Γ_{new} is assigned to at most $(m'' + 1)$ processors using IBPS (one new processor is introduced).

When a task leaves the system, say from processor Θ_x, then for load regulation we re-execute the assignment algorithm on Θ_x as well as on all m'' processors. Since residue tasks never require more than four processors (See Section 6.7)during third phase of IBPS , we have $m'' \leq 4$. Thus, when τ_{new} is admitted to the system using O-IBPS , the number of processors that require reassignment of task is upper bounded by $min\{4, m\}$. And when a task leaves the system, the number of processors that require reassignment of task for load regulation is upper bounded by $min\{5, m\}$. Remember that IBPS runs in linear time. This, together with the trend in processor industry to have chip multiprocessors with many cores (16, 32, 64 cores or more), means that our scheduling algorithm is efficient and scalable with increasing number of cores in CMPs for online scheduling of real-time tasks.

6.10 Related Work

While the RM scheduling algorithm is optimal for uniprocessor scheduling [LL73], it is not optimal for multiprocessor scheduling because of the well-known *Dhall's effect* limiting the utilization bound [DL78]. Recent work circumvents this effect by restricting the utilization of individual tasks in multiprocessor scheduling [ABJ01, Lun02, And08, Bak06, BG03, BCL05]. To that end, the worst-case utilization bound for static- and dynamic-priority partitioned scheduling is 50% [CFH$^+$04]. The corresponding bound for dynamic-priority global scheduling is 100% using the *pfair* family of algorithms [BCPV96], while static-priority global scheduling is 50% [AJ03] even when using the *pfair* strategy.

Partitioned approaches for RM scheduling is based on different bin-packing heuristics. The performance of earlier works on partitioned RM scheduling were measured in terms of $\Re_A^\infty = \frac{N}{N_o}$ where N is the number of processor for algorithm A under investigation and N_o is the optimal number of processors [DL78, DD86, OS95, LBOS95]. While metric $\Re_A^\infty$ expresses the resource requirement of a task allocation scheme, it does not express its schedulability performance. In [OB98], it is shown that the worst case utilization bound for partitioned RM First-Fit (FF) scheduling is $m(\sqrt{2} - 1) \approx$ 41%. In [LGDG03], this bound is improved by also including the number of tasks in the schedulability condition. The algorithm `R-BOUND-MP` [LMM98] uses R-BOUND test that exploits harmonicity in task periods. In [AJ03], an algorithm `R-BOUND-MP-NFR` (based on the R-BOUND test) is proposed that has a utilization bound of 50%. The work in [FBB06] assigns tasks to processors according to FF with a decreasing deadline order and decides uniprocessor schedulability using a *demand-bound* function (DBF). Although these works do not present a utilization bound, their worst-case performance is characterized using *resource augmentation*.

In order to achieve a utilization bound for the partitioned approach that exceeds 50%, a new type of scheduling algorithms using task splitting has evolved [ABD05, AT06, AB08, ABB08, KY07, KY08a, KY09a, LRL09]. Most of these works address task splitting for EDF priority. In [ABD05], a task splitting algorithm `EDF-fm` is proposed that has no scheduling guarantee but instead offers bounded task tardiness. An algorithm, called `EKG` [AT06], for dynamic-priorities using task splitting has a utilization bound between 66% and 100% depending on a design parameter k which trade-off utilization bound and preemption count. Using a time slot-based technique, sporadic task scheduling for constrained and arbitrary deadline are developed in [AB08, ABB08]. In [KY07, KY08a], the EDF-based partitioning algorithms `Ehd2Sip` and `EDDP` are developed using a concept similar to task splitting called ‘portioning’. In `Ehd2Sip`, the second portion of a task gets highest priority over other non-split task if the first portion is not executing. In `EDDP`, the deadline of a split task is changed to a smaller deadline called ‘virtual deadline’.

Common for all these dynamic-priority task splitting algorithms is the absence of priority traceability property and load regulation. As many of the algorithms requires sorting task before assignment, online scheduling may be inefficient. The static-priority task-splitting algorithm in [KY09a, KY08b] has utilization bound that does not exceed 50%. In [LRL09], an implicit deadline task set is converted to a constrained deadline static-priority task set during task assignment. Even if this algorithm has more than 50% utilization bound, it does not have the priority traceability property and does not consider its online applicability as we do with `IBPS` . Our scheduling algorithm has utilization bound 55.2% and the algorithm of Lakshmanan, Rajkumar and Lehoczky [LRL09] has utilization bound 60% (non-sorted-version) to 65% (sorted-version). Although the utilization bound in [LRL09] is higher than that of ours, the algorithm in [LRL09] has higher overhead in terms of online scheduling and number of migrations.

First, the sorted-version of algorithm presented in [LRL09] is not suitable for online scheduling since sorting of all tasks is required whenever task enters/leaves dynamically. If unsorted-version of algorithm in [LRL09] is used for online scheduling and when task leaves the system, then the subtasks of some existing split-task (for example, that run across all processors) may need to be recombined and reassigned. This can disturb the schedule in all processors. Our algorithm ensures online rescheduling of tasks is required in at most 4 to 5 processors. Second, our algorithm has lower migration overhead due to splitting than that of in [LRL09]. Our algorithm has at most m/2 split tasks in comparison to $(m-1)$ split tasks in [LRL09]. The relationship between number of subtasks of a split-task on preemption overhead and scheduling performance (in terms of makespan) has been considered in [KLL09].

If trade-off between theoretical utilization bound and practical overheads for online scheduling is to be made, then we believe our algorithm outweighs the 5%-10% higher utilization bound of algorithm in [LRL09] for large systems(CMPs with many cores).

In addition, the algorithm `IBPS` is priority-traceable (that is, task priorities are not changed to another fixed-priority during task assignment) while the algorithm in [LRL09] is not. Priority-traceability is important for system designer who wants that during task assignment his assigned task priorities be preserved for (i) application requirement, (ii) better predictability of the run-time system and (iii) ease of debugging during development.

6.11 Fault-Tolerant Scheduling

We now turn our attention to tolerating faults in a multiprocessor system. Using the sufficient and necessary feasibility condition of `FTRM` derived in Chapter 5, tasks can be assigned to the processors via partitioned multiprocessor scheduling. During task assignment to each processor, we can determine whether an unassigned task can be feasibly assigned to a processor using Corollary 5.2. If all the tasks of a task set can be feasibility assigned to the processors, then each processor can tolerate f faults. Note that in `IBPS`, a processor can have a subtask of a split task. If an error is detected at the end of execution of a split task, the error is in fact detected in the processor onto which the second subtask of the split task is assigned. However, the recovery copy of the task will be executed on two processors as a split recovery task. The calculation of the execution time of the composed task in Eq. (5.17) can take into account the execution of a split-task across two processors in order to find the value of `Load-Factor-HPi`as defined in Chapter 5. However, the time complexity of the exact test in `FTRM` will then be significant for on-line multiprocessor scheduling.

As discussed in Section 6.9, using the sufficient schedulability condition of `IBPS`, an efficient admission controller for online multiprocessor scheduling can be designed. If an online task is accepted, this task can be assigned to the system using `O-IBPS` very efficiently. If during run time of the system, an error is detected at the end of execution of a task, the required execution of the recovery copy of the task can be regarded as an online request by a new task. Based on the `O-IBPS` admission controller, if the recovery request is accepted to the system, then the recovery operation can be completed before the deadline of the task. If the recovery request cannot be accepted using the sufficient condition of the admission controller, we propose three possible alternatives for handling the recovery request:

- **Direct Rejection:** Simply reject the request without any further consideration.
- **Criticality-Based Eviction:** Evict some low-criticality task from the system to accept the new request.
- **Imprecise Computation:** Accept the new request and execute as much as possible of the recovery copy without compromising the timeliness of other tasks.

6.11.1 Direct Rejection

If an error is detected and the recovery request cannot be accepted by the admission controller of `O-IBPS`, the approach is as simple as just rejecting the recovery request. Therefore, if the system utilization is greater than or equal to

55.2%, then no recovery request would be accepted by the admission controller. In such a highly-loaded system the reliability is compromised in favor of timeliness for the already accepted tasks in the multiprocessor system. Therefore, reliability is degraded so as to guarantee schedulability of the existing tasks.

6.11.2 Criticality-Based Eviction

If an error is detected and the recovery request cannot be accepted by the admission controller of `O-IBPS`, then we can employ *criticality-based eviction*. In this approach, some already-admitted task, having lower criticality than the criticality of the recovery request, is temporarily terminated and the recovery request is accepted. The termination of the lower-criticality task is temporary in the sense that, when the recovery copy of a faulty task finishes execution, the evicted lower criticality task can be re-admitted into the system. In such case, the lower-criticality task may be unable to execute its jobs that are released while recovery operation is performed.

By criticality of a task we mean the user-perceived importance of the applications tasks in meeting the deadlines. The criticality of the tasks in a task set can be determined independent of the priorities of the tasks [MAM99]. Such criticality-based eviction is applicable for applications in which execution of some jobs of a task can be skipped. In [CB98], scheduling of hard and firm periodic tasks are considered. A firm task can occasionally skip one of its jobs based on some predetermined quality-of-service agreement while the hard periodic task must execute all of its jobs.

Criticality-based scheduling for non-deterministic workloads is addressed by Alvarez and Mossé in [MAM99]. They analyzed the schedulability of a fixed-priority system using a concept called responsiveness [MAM99]. Their analysis is best suited for systems with nondeterministic workload in which recovery operations caused by faults are serviced at different responsiveness levels. By responsiveness level, the authors mean whether the recovery operation is run in a non-intrusive (without affecting schedulability of other tasks) or intrusive (affecting schedulability of existing tasks) manner. In case of intrusive recovery, timeliness of the less-critical tasks are compromised and the system suffers degraded service. Thus, the eviction of lower-criticality task degrades schedulability performance but provides higher reliability.

Note that, such criticality-based eviction may not work if there are no lower-criticality tasks to evict in order to accept a recovery request, or if the computation time of the lower-criticality tasks may not be enough for executing the recovery copy. This problem can be addressed using imprecise computation paradigm.

6.11.3 Imprecise Computation

If partial computation of the recovery request is useful, then the recovery request can be accepted into the system even though a complete recovery request cannot be serviced due insufficient processing capacity. When the result of a complete execution of a recovery request cannot be produced before the deadline, faults can be tolerated using *imprecise computation* of the recovery copy. Imprecise computation models are considered in [CLL90, LSL$^+$94, MAAMM00] and are appropriate for monotone processes where result produced by a task will have increasingly higher quality the more time is spent in executing the task. Such monotone processes are considered to have a mandatory part and an optional part [LSL$^+$94]. The mandatory part of each task has a hard deadline and must complete its execution. However, the optional part of a task can be skipped if enough processing power is not available.

The imprecise computational model is applicable if the recovery copy of a faulty task is modeled as a monotone process. Therefore, even if the full execution of the recovery request cannot be completed, the result of the partial computation of the recovery request can ensure a certain quality to the application. Hence, when the admission controller cannot guarantee complete execution of a recovery request, the request can still be accepted to the system and imprecise result can be delivered to the application consequently. By considering the recovery request as a monotone process, the imprecise computation technique to serve a recovery requests can be seen as providing a balance between schedulability performance and reliability.

It is easy to realize that eviction of a low-criticality task and imprecise computation can be combined so as to offer a solution to the problem where the mandatory part of a task does not have enough time to finish before its hard deadline. In such case by evicting a lower criticality task could enable the complete execution of the mandatory part of a highly-critical recovery request to be serviced.

6.12 Discussion and Summary

In this chapter, we have proposed a task-assignment algorithm, called `IBPS` that uses the utilization of static-priority tasks in different subintervals as a guideline during assignment. The algorithm `IBPS` is based on a task splitting approach. The worst-case utilization bound of `IBPS` is 55.2%. The task-assignment algorithm of `IBPS` requires no a priori sorting of the tasks, and the time complexity for assigning n periodic tasks to the processor is $O(n)$. In addition to having linear time complexity for task assignment, the algorithm `IBPS` possesses many important practical features.

First, the load regulation technique of `IBPS` enables the design of an efficient admission controller for on-line task scheduling in multiprocessor systems. `IBPS` guarantees that, as the number of processors in the system increases, the percentage of processors having a load greater than 55.2% also increases, since at most four processor could have a load lower than 55.2%. Therefore, online scheduling of tasks using `O-IBPS` scales very well with the current trend of having an increasing number of cores in chip multiprocessors. Second, our algorithm `IBPS` possesses a priority-traceability property which facilitates the system designer's ability to debug and moniotr a system during development and maintenance. Third, the task-splitting algorithm causes a lower number of migrations compared to any other task-splitting algorithm for static and dynamic priority systems. All these salient features make our scheduling algorithm efficient for practical implementation for chip multiprocessors with an increasing number of cores.

When an error is detected at the end of execution of a task, the recovery request generated as a means to achieve online fault-tolerance can be considered as an online request of a new task. If the sufficient schedulability condition of `IBPS` can guarantee the schedulability of the recovery request, the online admission controller accepts the request and can execute the recovery to tolerate the fault. If the sufficient schedulability condition of `IBPS` cannot guarantee the schedulability of the recovery request, this thesis proposes three different alternatives to handle the non-accepted request. Each of these alternatives makes a particular trade-off between schedulability performance and reliability requirements for the real-time application.

7 Conclusion

The research presented in this thesis deals with designing scheduling algorithms with the objective of meeting deadlines for a set of periodic tasks on uni- and multiprocessor systems. The feasibility of the two proposed scheduling algorithms — FTRM and IBPS — are analyzed for uni- and multiprocessor platforms with the main goal at achieving fault-tolerance and high processor utilization, respectively. Both algorithms are designed for a static-priority system, more specifically, the RM priority policy for a set of implicit-deadline periodic tasks.

A very general fault model is considered in the analysis of the uniprocessor scheduling algorithm FTRM for achieving fault-tolerance. The fault model covers a variety of hardware and software faults that can occur at any time, in any task, and even during execution of a recovery operation. The recovery copy of a task that runs to tolerate a fault may simply be the re-execution of the faulty task or it may be the execution of a recovery block (that is, a different implementation of the task). The possession of a fault-tolerant scheduling algorithm that considers such a general fault-model makes FTRM a viable candidate for development of a wide ranges of hard-real time applications.

The schedulability analysis of FTRM uses a novel composability technique to compute the worst-case workload requested by jobs of the periodic tasks released within a particular time interval. By calculating the worst-case workload within an interval defined by the released time and deadline of each task, the necessary and sufficient (exact) RM feasibility condition can be derived for uniprocessor systems. It is proved that this exact condition is true if, and only if, a set of implicit-deadline periodic tasks is fault-tolerant RM-schedulable. In addition, the efficiency in terms of time complexity of `FTRM` is also preferable over the same for existing EDF scheduling.

Another facet of the proposed composability technique used in the feasibility analysis of `FTRM` is that it is not only applicable for tasks with implicit deadline and RM priority, but also for tasks with constrained deadlines and any fixed-priority policy. Therefore, the proposed feasibility analysis technique would enable the derivation of an exact feasibility condition for any fixed-priority scheduling of constrained or implicit deadline task systems assuming the general fault model considered in this thesis.

The exact feasibility condition of `FTRM` is directly applicable to partitioned multiprocessor scheduling during assignment of task to the processors, for example, during assignment of tasks using `IBPS` algorithm proposed in this thesis. However, in oder to be able to design an efficient admission controller for online multiprocessor scheduling, the time-complexity of `FTRM` may be a concern if the system has a large number of tasks. Therefore, a simple and sufficient feasibility condition for offline partitioned multiprocessor scheduling, called `IBPS`, is derived, based on a task-splitting technique. It is proved that, all task deadlines are met using `IBPS` if at most 55.2% capacity of the processor computation capacity is requested. This sufficient condition is used to design the admission controller of an online multiprocessor scheduling algorithm, called `O-IBPS`. The algorithm `IBPS` is one of the first two works to overcome the fundamental limitation of a 50% minimum achievable utilization bound of the traditional, non-task-splitting partitioned multiprocessor scheduling for static-priority systems. The other algorithm, proposed by Lakshmanan, Rajkumar, and Lehoczky at Carnegie Mellon University, has an utilization bound of 60% (unsorted version) to 65% (sorted version). While the scheduling algorithm proposed by Lakshmanan *et al.* has better utilization bound for offline systems, it is not suitable for online scheduling on multiprocessors due to the overhaed associated with assignment and re-assignment of tasks that arrive and leave online, respectively.

Using the task-splitting paradigm, the proposed task assignment algorithm for `IBPS` runs in linear time. During task assignment in `IBPS`, the load of the processors are regulated such that at most $min\{m, 4\}$ processors (where m is the number of processors) may have an individual processor load less than 55.2%. This means that, when a new task arrives, the number of processors that has to be considered for assigning a new task is at most $min\{m, 4\}$. Likewise, when a task leaves the system, the number of processors that has to be considered for reassignment of some of the remaining tasks (for load regulation) is at most $min\{m, 5\}$. Consequently, O-IBPS is an efficient scheduling algorithm for online scheduling of real-time tasks on large systems with many processors (for example, Chip-Multiprocessors with many cores).

The algorithm IBPS possesses two additional properties, namely priority traceability and a low number of migrations due to task splitting. In IBPS, the initial priorities of the application tasks do not get changed during task assignment. This is in contrast to the algorithm proposed by Lakshmanan *et al.* where static-priorities are assigned to tasks during task assignment. The priorities of the tasks are independent of the task assignment algorithm in IBPS. This property of IBPS enables the application designer to easily come up with his preferred choice of task priorities just by selecting appropriate task periods (that is, the shorter is the task period selected, the higher is the assigned task priority). Such priority assignments that are independent of the task assignment algorithm provides a better traceability and predictability of the run-time systems and also could facilitates debugging and monitoring of the system during development.

Another positive property of IBPS is that the total number of migrations caused by the split tasks is lower than that of any other task-splitting algorithm. A migration of a task from one processor to another causes one preemption (the task stops execution in one processor and then starts execution in another processor). Therefore, the lower the number of migrations, the lower the number of preemptions and its associated cost. If such overhead cost, due to task migration is accounted for, then our proposed algorithm IBPS has yet another positive feature over other task-splitting algorithms.

In IBPS, first tasks are grouped into seven utilization subintervals and then the tasks from these groups are assigned to processors. A split-task has only two subtasks that are assigned on two different processors in IBPS. I believe that, by splitting a task more than once and grouping the tasks of a given task set into more than seven subintervals, new task assignment algorithms can be derived such that it may be possible to achieve a 69% minimum achievable utilization bound, which is the maximum possible utilization bound for partitioned RM scheduling on multiprocessors.

In the future, the exact feasibility condition of FTRM can be used for joint scheduling of periodic and aperiodic tasks on uni- and multiprocessor platform. To compare the average performance of such joint scheduling, simulation experiments can be conducted to schedule randomly generated task sets on a particular platform. Moreover, it would be interesting to look into the aspects of how the algorithms FTRM and IBPS can be adapted for scheduling of periodic tasks that are not independent.

For the fault-tolerant point of view, the admission controller of O-IBPS can be augmented with the appropriate mechanisms for tolerating faults. I believe such an augmentation would provide good average scheduling performance of many soft-real time applications. For example, when a frame of an online video is dropped occasionally due to network problems, this frame can be re-transmitted using appropriate fault-tolerant mechanisms by the sender in order to maintain the quality of the video perceived by the receiver.

In summary, if the preciseness of a feasibility condition is of main concern, then the algorithm FTRM for a uniprocessor provides a faster schedulability decision than EDF scheduling assuming the fault model used in this thesis. The composability technique of the uniprocessor fault-tolerant scheduling algorithm can be easily applied to derive the exact feasibility condition for any fixed-priority implicit or constrained deadline periodic task system (for example, deadline-monotonic scheduling). The algorithm IBPS can be used to assign and execute the tasks on the processors of a multiprocessor platform based on the application designer's selected task periods according to RM priority. If IBPS is used online, then scheduling of the online application tasks can be done efficiently using the algorithm O-IBPS. By selecting appropriate fault-tolerant mechanisms, O-IBPS can be augmented with the capability of fault-tolerance. To conclude, FTRM and O-IBPS can be used more efficiently to schedule the real-time tasks on uni- and multiprocessor platforms compared to many other competing algorithms.

Bibliography

[AB08] B. Andersson and K. Bletsas. Sporadic Multiprocessor Scheduling with Few Preemptions. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 243–252, 2008.

[ABB08] B. Andersson, K. Bletsas, and S. Baruah. Scheduling Arbitrary-Deadline Sporadic Task Systems on Multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 385–394, 2008.

[ABD$^+$95] N. C. Audsley, A. Burns, R. I. Davis, K. W. Tindell, and A. J. Wellings. Fixed priority pre-emptive scheduling: an historical perspective. *Real-Time Systems*, 8(2-3):173–198, 1995.

[ABD05] J. H. Anderson, V. Bud, and U. C. Devi. An EDF-based Scheduling Algorithm for Multiprocessor Soft Real-Time Systems. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 199–208, 2005.

[ABJ01] B. Andersson, S. Baruah, and J. Jonsson. Static-Priority Scheduling on Multiprocessors. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 193–202, 2001.

[ABR$^+$93] N. Audsley, A. Burns, M. Richardson, K. Tindell, and A.J. Wellings. Applying new scheduling theory to static priority pre-emptive scheduling. *Software Engineering Journal*, 8(5):284–292, 1993.

[AFK05] J. Aidemark, P. Folkesson, and J. Karlsson. A Framework for Node-Level Fault Tolerance in Distributed Real-Time Systems. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 656–665, 2005.

[AH06] G. Attiya and Y. Hamam. Task allocation for maximizing reliability of distributed systems: a simulated annealing approach. *Journal of Parallel and Distributed Computing*, 66(10):1259–1266, 2006.

[AJ03] B. Andersson and J. Jonsson. The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 33–40, 2003.

[ALRL04] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1(1):11–33, 2004.

[And08] Björn Andersson. Global Static-Priority Preemptive Multiprocessor Scheduling with Utilization Bound 38%. In *Proceedings of the International Conference on Principles of Distributed Systems*, pages 73–88, 2008.

[AOSM01] R. Al-Omari, Arun K. Somani, and G. Manimaran. A New Fault-Tolerant Technique for Improving Schedulability in Multiprocessor Real-time Systems. In *Proceedings of the IEEE Parallel and Distributed Processing Symposium*, page 8, 2001.

[AT06] B. Andersson and E. Tovar. Multiprocessor Scheduling with Few Preemptions. In *Proceedings of the IEEE Conference on Embedded and Real-Time Computing Systems and Applications*, pages 322–334, 2006.

[Avi85] A. Avižienis. The N-Version Approach to Fault-Tolerant Software. *IEEE Transactions on Software Engineering*, 11(12):1491–1501, 1985.

[Ayd07] H. Aydin. Exact Fault-Sensitive Feasibility Analysis of Real-Time Tasks. *IEEE Transactions on Computers*, 56(10):1372–1386, 2007.

[Bak06] T. P. Baker. An Analysis of Fixed-Priority Schedulability on a Multiprocessor. *Real-Time Systems*, 32(1-2):49–71, 2006.

[Bar08] R. Barbosa. Layered Fault Tolerance for Distributed Embedded Systems. *PhD Thesis, Department of Computer Science and Engineering, Chalmers University of Technology*, 2008.

[Bau05] R. Baumann. Soft errors in advanced computer systems. *IEEE Design and Test of Computers*, 22(3):258–266, 2005.

[BBB03] E. Bini, G. C. Buttazzo, and G. M. Buttazzo. Rate Monotonic Analysis: The Hyperbolic Bound. *IEEE Transactions on Computers*, 52(7):933–942, 2003.

[BCL05] M. Bertogna, M. Cirinei, and G. Lipari. New Schedulability Tests for Real-Time Task Sets Scheduled by Deadline Monotonic on Multiprocessors . In *Proceedings of the Conference on Principles of Distributed Systems*, pages 306–321, 2005.

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

[BDP96] A. Burns, R. Davis, and S. Punnekkat. Feasibility Analysis of Fault-Tolerant Real-Time Task Sets. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 522–527, 1996.

[BF97] A. A. Bertossi and A. Fusiello. Rate-monotonic scheduling for hard-real-time systems. *European Journal of Operational Research*, 96(3):429–443, 1997.

[BF05] S. Baruah and N. Fisher. The partitioned multiprocessor scheduling of sporadic task systems. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 321–329, 2005.

[BG03] S. Baruah and J. Goossens. The Static-priority Scheduling of Periodic Task Systems upon Identical Multiprocessor Platforms. *in Proc. of the IASTED Int. Conf. on PDCS*, pages 427–432, 2003.

[BMR99] A. A. Bertossi, L. V. Mancini, and F. Rossini. Fault-Tolerant Rate-Monotonic First-Fit Scheduling in Hard-Real-Time Systems. *IEEE Transactions on Parallel and Distributed Systems*, 10(9):934–945, 1999.

[BPSW99] A. Burns, S. Punnekkat, L. Strigini, and D.R. Wright. Probabilistic scheduling guarantees for fault-tolerant real-time systems. In *Dependable Computing for Critical Applications*, pages 361–378, 1999.

[BRH90] S. Baruah, L. E. Rosier, and R. R. Howell. Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. *Real-Time Systems.*, 2(4):301–324, 1990.

[BT83] J. A. Bannister and K. S. Trivedi. Task allocation in fault-tolerant distributed systems. *Acta Informatica*, 20:261–281, 1983.

[CB98] M. Caccamo and G. Buttazzo. Optimal scheduling for fault-tolerant and firm real-time systems . In *Proceedings of the IEEE Conference on Real-Time Computing Systems and Applications*, pages 223–231, 1998.

[CC89] H. Chetto and M. Chetto. Some Results of the Earliest Deadline Scheduling Algorithm. *IEEE Transactions on Software Engineering*, 15(10):1261–1269, 1989.

[CCE$^+$09] S. Chaudhry, R. Cypher, M. Ekman, M. Karlsson, A. Landin, S. Yip, H. Zeffer, and M. Tremblay. Rock: A High-Performance Sparc CMT Processor. *IEEE Micro*, 29(2):6–16, 2009.

[CFH$^+$04] J. Carpenter, S. Funk, P. Holman, J. H. Anderson, and S. Baruah. A categorization of real-time multiprocessor scheduling problems and algorithms. *Handbook on Scheduling Algorithms, Methods, and Models*, 2004.

[CLL90] J.-Y. Chung, J.W.S. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers*, 39(9):1156–1174, 1990.

[CLRS01] T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. MIT Press, 2001.

[CMR92] A. Campbell, P. McDonald, and K. Ray. Single event upset rates in space. *IEEE Transactions on Nuclear Science*, 39(6):1828–1835, Dec 1992.

[CMS82] X. Castillo, S. R. McConnel, and D. P. Siewiorek. Derivation and Calibration of a Transient Error Reliability Model. *IEEE Transactions on Computers*, 31(7):658–671, 1982.

[DD86] Sadegh Davari and Sudarshan K. Dhall. An On Line Algorithm for Real-Time Tasks Allocation. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 194–200, 1986.

[Dha77] S. K. Dhall. Scheduling periodic-time - critical jobs on single processor and multiprocessor computing systems. *PhD Thesis, University of Illinois at Urbana-Champaign*, 1977.

[DL78] S. K. Dhall and C. L. Liu. On a Real-Time Scheduling Problem. *Operations Research*, 26(1):127–140, 1978.

[FBB06] N. Fisher, S. Baruah, and T. P. Baker. The Partitioned Scheduling of Sporadic Tasks According to Static-Priorities. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 118–127, 2006.

[Gho] Fault-tolerant scheduling on a hard real-time multiprocessor system , author=Ghosh, S. and Melhem, R. and Mossé, D., booktitle=Proceedings of the International Parallel Processing Symposium, pages=775-782, year=1994,.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA, 1979.

[GMM95] S. Ghosh, R. Melhem, and D. Mossé. Enhancing real-time schedules to tolerate transient faults. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 120–129, 1995.

[GMM97] S. Ghosh, R. Melhem, and D. Mossé. Fault-Tolerance Through Scheduling of Aperiodic Tasks in Hard Real-Time Multiprocessor Systems. *IEEE Transactions on Parallel and Distributed Systems*, 8(3):272–284, 1997.

[GMMS98a] S. Ghosh, R. Melhem, D. Mossé, and J. S. Sarma. Fault-Tolerant Rate-Monotonic Scheduling. *Real-Time Systems.*, 15(2):149–181, 1998.

[GMMS98b] S. Ghosh, Rami Melhem, Daniel Mossé, and Joydeep Sen Sarma. Fault-Tolerant Rate-Monotonic Scheduling. *Real-Time Systems.*, 15(2):149–181, 1998.

[HSW03] C.-C. Han, K. G. Shin, and J. Wu. A Fault-Tolerant Scheduling Algorithm for Real-Time Periodic Tasks with Possible Software Faults. *IEEE Transactions on Computers*, 52(3):362–372, 2003.

[IRH86] R. K. Iyer, D. J. Rossetti, and M. C. Hsueh. Measurement and modeling of computer reliability as affected by system activity. *ACM Transactions on Computer Systems*, 4(3):214–237, 1986.

[JHCS02] A. Jhumka, M. Hiller, V. Claesson, and N. Suri. On systematic design of globally consistent executable assertions in embedded software. In *Proceedings of the joint conference on Languages, compilers and tools for embedded systems*, pages 75–84, 2002.

[Joh88] B. W. Johnson. *Design & analysis of fault tolerant digital systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1988.

[JP86] M. Joseph and P. Pandya. Finding Response Times in a Real-Time System. *The Computer Journal*, 29(5):390–395, 1986.

[KAO05] P. Kongetira, K. Aingaran, and K. Olukotun. Niagara: a 32-way multithreaded Sparc processor. *IEEE Micro*, 25(2):21–29, 2005.

[KLL09] K. Klonowska, L. Lundberg, and H. Lennerstad. The maximum gain of increasing the number of preemptions in multiprocessor scheduling. *Acta Informatica.*, 46(4):285–295, 2009.

[KLLS05a] K. Klonowska, H. Lennerstad, L. Lundberg, and C. Svahnberg. Optimal recovery schemes in fault tolerant distributed computing. *Acta Informatica.*, 41(6):341–365, 2005.

[KLLS05b] K. Klonowska, L. Lundberg, H. Lennerstad, and C. Svahnberg. Extended Golomb rulers as the new recovery schemes in distributed dependable computing. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, page 8, April 2005.

[KS86] C.M. Krishna and K.G. Shin. On Scheduling Tasks with a Quick Recovery from Failure. *IEEE Transactions on Computers*, C-35(5):448–455, 1986.

[KY07] S. Kato and N. Yamasaki. Real-Time Scheduling with Task Splitting on Multiprocessors. In *Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 441–450, 2007.

[KY08a] S. Kato and N. Yamasaki. Portioned EDF-based scheduling on multiprocessors. In *Proceeding of the International Conference on Embedded Software*, pages 139–148, 2008.

[KY08b] S. Kato and N. Yamasaki. Portioned static-priority scheduling on multiprocessors. In *Proceedings of the IEEE International Parallel and Distributed Processing Symposium*, pages 1–12, 2008.

[KY09a] S. Kato and N. Yamasaki. Semi-Partitining Fixed-Priority Scheduling on Multiprocessor. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 23–32, 2009.

[KY09b] S. Kato and N. Yamasaki. Semi-Partitioned Scheduling of Sporadic Task Systems on Multiprocessors. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 249–258, 2009.

[LBOS95] J. Liebeherr, A. Burchard, Y. Oh, and S. H. Son. New Strategies for Assigning Real-Time Tasks to Multiprocessor Systems. *IEEE Transactions on Computers*, 44(12):1429–1442, 1995.

[LDG04] J. M. López, J. L. Díaz, and D. F. García. Minimum and Maximum Utilization Bounds for Multiprocessor Rate Monotonic Scheduling. *IEEE Transactions on Parallel and Distributed Systems*, 15(7):642–653, 2004.

[LGDG03] J. M. López, M. García, J. L. Díaz, and D. F. García. Utilization Bounds for Multiprocessor Rate-Monotonic Scheduling. *Real-Time Systems*, 24(1):5–28, 2003.

[LL73] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM*, 20(1):46–61, 1973.

[LL08] Lars Lundberg and Hakan Lennerstad. Slack-based global multiprocessor scheduling of aperiodic tasks in parallel embedded real-time systems. In *Proceedings of the IEEE/ACS International Conference on Computer Systems and Applications*, pages 465–472, 2008.

[LLWS08] W.-C. Lu, K.-J. Lin, H.-W. Wei, and W.-K. Shih. Efficient Exact Test for Rate-Monotonic Schedulability Using Large Period-Dependent Initial Values. *IEEE Transactions on Computers*, 57(5):648–659, 2008.

[LMM98] S. Lauzac, R. Melhem, and D. Mossé. An Efficient RMS Control and Its Application to Multiprocessor Scheduling. In *Proceedings of the International Parallel Processing Symposium*, pages 511–518, 1998.

[LMM00] F. Liberato, R. Melhem, and D. Mossé. Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-Time Systems. *IEEE Transactions on Computers*, 49(9):906–914, 2000.

[LRL09] K. Lakshmanan, R. Rajkumar, and J. Lehoczky. Partitioned Fixed-Priority Preemptive Scheduling for Multi-core Processors. In *Proceedings of the EuroMicro Conference on Real-Time Systems*, pages 239–248, 2009.

[LSD89] J. Lehoczky, L. Sha, and Y. Ding. The rate monotonic scheduling algorithm: exact characterization and average case behavior. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 166–171, 1989.

[LSL$^+$94] J.W.S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung. Imprecise computations. *Proceedings of the IEEE*, 82(1):83–94, 1994.

[Lun02] L. Lundberg. Analyzing Fixed-Priority Global Multiprocessor Scheduling. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 145–153, 2002.

[MAAMM00] P. Mejia-Alvarez, H. Aydin, D. Mossé, and R. Melhem. Scheduling optional computations in fault-tolerant real-time systems. In *Proceedings* *of the Conference on Real-Time Computing Systems and Applications*, page 323, 2000.

[MAM99] P. Mejia-Alvarez and D. Mossé. A responsiveness approach for scheduling fault recovery in real-time systems. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, pages 4–13, 1999.

[MBS07] A. Meixner, M.E. Bauer, and D.J. Sorin. Argus: Low-Cost, Comprehensive Error Detection in Simple Cores. In *Proceedings of the Annual IEEE/ACM International Symposium on Microarchitecture*, pages 210–222, 2007.

[MCS91] H. Madeira, J. Camoes, and J. G. Silva. A watchdog processor for concurrent error detection in multiple processor systems. *Microprocessors and Microsystems*, 15(3):123–130, 1991.

[MdALB03] G. M. de A. Lima and A. Burns. An optimal fixed-priority assignment algorithm for supporting fault-tolerant hard real-time systems. *IEEE Transactions on Computers*, 52(10):1332–1346, 2003.

[MM98] G. Manimaran and C. S. R. Murthy. A Fault-Tolerant Dynamic Scheduling Algorithm for Multiprocessor Real-Time Systems and Its Analysis. *IEEE Transactions on Parallel and Distributed Systems*, 9(11):1137–1152, 1998.

[OB98] D.-I. Oh and T. P. Baker. Utilization Bounds for N-Processor Rate Monotone Scheduling with Static Processor Assignment. *Real-Time Systems*, 15(2):183–192, 1998.

[OS94] Y. Oh and S. H. Son. Enhancing fault-tolerance in rate-monotonic scheduling. *Real-Time Systems.*, 7(3):315–329, 1994.

[OS95] Yingfeng Oh and Sang H. Son. Allocating fixed-priority periodic tasks on multiprocessor systems. *Real-Time Systems.*, 9(3):207–239, 1995.

[Pat06] R. M. Pathan. Probabilistic Analysis of Real-Time Scheduling of Systems Tolerating Multiple Transient Faults. In *Proceedings of the International Conference of Computer and Information Technology*, 2006.

[PBD01] S. Punnekkat, A. Burns, and R. Davis. Analysis of Checkpointing for Real-Time Systems. *Real-Time Systems.*, 20(1):83–102, 2001.

[PJ09] R. M. Pathan and J. Jonsson. Load Regulating Algorithm for Static-Priority Task Scheduling on Multiprocessors. *Technical Report, Chalmers University of Technology, Sweden*, May 2009. `http://www.cse.chalmers.se/~risat/lraspts.pdf`.

[PM98] M. Pandya and M. Malek. Minimum Achievable Utilization for Fault-Tolerant Processing of Periodic Tasks. *IEEE Transactions on Computers*, 47(10):1102–1112, 1998.

[Pra96] D. K. Pradhan. *Fault-tolerant computer system design*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1996.

[SAA$^+$04] L. Sha, T. Abdelzaher, K.-E. Årzén, A. Cervin, T. Baker, A. Burns, G. Buttazzo, M. Caccamo, J. Lehoczky, and A. K. Mok. Real Time Scheduling Theory: A Historical Perspective. *Real-Time Systems*, 28(2-3):101–155, 2004.

[SABR04] J. Srinivasan, S. V. Adve, P. Bose, and J. A. Rivers. The Impact of Technology Scaling on Lifetime Reliability. In *Proceedings of the International Conference on Dependable Systems and Networks*, pages 177–186, 2004.

[SH98] M. Sjödin and H. Hansson. Improved Response-Time Analysis Calculations. In *Proceedings of the IEEE Real-Time Systems Symposium*, page 399, 1998.

[SKK$^+$08] P. N. Sanda, J. W. Kellington, P. Kudva, R. Kalla, R. B. McBeth, J. Ackaret, R. Lockwood, J. Schumann, and C. R. Jones. Soft-error resilience of the IBM POWER6 processor. *IBM J. Res. Dev.*, 52(3):275–284, 2008.

[SKM$^+$78] D.P. Siewiorek, V. Kini, H. Mashburn, S. McConnel, and M. Tsao. A case study of C.mmp, Cm*, and C.vmp: Part I–Experiences with fault tolerance in multiprocessor systems. *Proceedings of the IEEE*, 66(10):1178–1199, 1978.

[SLR86] L. Sha, J. P. Lehoczky, and R. Rajkumar. Solutions for Some Practical Problems in Prioritized Preemptive Scheduling. In *Proceedings of the IEEE Real-Time Systems Symposium*, pages 181–191, 1986.

[SRL90] L. Sha, R. Rajkumar, and J.P. Lehoczky. Priority Inheritance Protocols: An Approach to Real-Time Synchronization. *IEEE Transactions on Computers*, 39(9):1175–1185, 1990.

[SSO05] R. M. Santos, J. Santos, and J. D. Orozco. A least upper bound on the fault tolerance of real-time systems. *Journal of Systems and Software.*, 78(1):47–55, 2005.

[SUSO04] R. M. Santos, J. Urriza, J. Santos, and J. Orozco. New methods for redistributing slack time in real-time systems: applications and comparative evaluations. *Journal of Systems and Software*, 69(1-2):115–128, 2004.

[SWG92] S. M. Shatz, J.-P. Wang, and M. Goto. Task allocation for maximizing reliability of distributed computer systems. *IEEE Transactions on Computers*, 41(9):1156–1168, Sep 1992.

[WEMR04] C. Weaver, J. Emer, S. S. Mukherjee, and S. K. Reinhardt. Techniques to Reduce the Soft Error Rate of a High-Performance Microprocessor. In *Proceedings of the annual international symposium on Computer architecture*, pages 264–275, 2004.