Enhancing Performance and Fault Tolerance
In Reward-Based Scheduling

by

Hakan Aydın

B.S., Istanbul Technical University, 1991

M.S., Istanbul Technical University, 1994

Submitted to the Graduate Faculty of

Arts and Sciences in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

University of Pittsburgh

2001

UNIVERSITY OF PITTSBURGH

FACULTY OF ARTS AND SCIENCES

This thesis was presented

by

Hakan Aydın

It was defended on

July 23rd, 2001

and approved by

Prof. Daniel Mossé

Prof. Rami Melhem

Prof. Bala Kalyanasundaram

Prof. John P. Lehoczky

Committee Chairperson

Enhancing Performance and Fault Tolerance
In Reward-Based Scheduling

Hakan Aydın,
University of Pittsburgh, 2001

Reward-based scheduling is based on the principle of trading precision for timeliness and it is particularly suitable for real-time applications where worst-case guarantees can not be provided due to overload conditions and/or faults. In this framework, a real-time task comprises a mandatory part and an optional part. The mandatory part produces an approximate result, which is subsequently refined by the optional part. Further, a nondecreasing reward function is associated with the optional execution.

In this dissertation, we explore reward-based scheduling as a comprehensive and cohesive framework for researching real-time scheduling problems with different metrics and goals. In particular, we solve some open problems, and study the application of the reward-based framework to power management and fault tolerance. In *periodic reward-based scheduling*, we provide an efficient and optimal solution to compute service times of optional parts, which can be used with scheduling policies that fully utilize the processor(s). We also investigate relaxing some fundamental assumptions.

Power-aware scheduling consists of dynamic adjustments of the CPU speed, aiming at obtaining energy savings at the expense of increased response time. We devise three schemes that still satisfy the application timing constraints, namely: (a) Static Optimal, based on the reward-based framework, when tasks consume the *worst-case workload*, (b) Dynamic Reclaiming, based on detecting early completions in the *actual workload*, and (c) Aggressive, which anticipates early completions by taking into account the *expected workload*.

Finally, we develop a *Fault Tolerant (FT) Optimality* framework for reward-based real-time tasks. Within this framework, our approach is to schedule the optional parts intelligently to allow recovery time for mandatory parts in case of errors, while maximizing the total reward. We provide FT-Optimal algorithms for independent tasks and tasks having linear precedence constraints, analyzing both single deadline and distinct deadlines cases.

As a whole, the dissertation provides a justification of the reward-based scheduling framework as a general paradigm to express and incorporate multiple requirement dimensions for real-

time computing. Our results show that the framework is particularly promising for emerging applications where timeliness, utility (reward), power-efficiency and fault tolerance dimensions need to be simultaneously addressed.

Acknowledgments

I am grateful to my advisors, Dr. Daniel Mossé and Dr. Rami Melhem, for their orientation, help and support throughout my Ph.D. studies. Even though I had come to the United States aiming at doing Ph.D. in another field of Computer Science, I was introduced to the Real-Time Systems research through their initiatives. They were always willing to discuss and provide help on every matter, despite their busy schedules. When I look back now, I increasingly appreciate their success in maintaining a high-quality research group, while keeping the work atmosphere as pleasant as possible. A Ph.D. dissertation can be also seen as the ultimate training in a well-defined research area: solutions to specific technical problems are important, yet I believe that the research methodology, rationale and strategies are the most enduring components of this training. In this aspect, the guidance of my advisors was invaluable. Having selected an academic career, I cannot help thinking how much I would be happy if I were able to contribute to my (future) students as much as they did.

While working as a graduate student researcher in FORTS and PARTS groups, I have had the chance of meeting and working with several noteworthy researchers. From the professional point of view, I benefited a lot from the interactions with Libin Dong and Sylvain Lauzac. They were also great friends, ready to provide help whenever I needed. I had also the privilege of co-operating with Dr. Pedro-Mejia Alvarez during and after his visit to University of Pittsburgh, we were able to explore together several experimental aspects of Reward-Based scheduling. Joydeep Sensarma, Hugo Varotto, Frank Liberato and Anthony Egan are among FORTS members that I will always remember as very good friends and remarkable researchers.

I am grateful to Dr. Bala Kalyanasundaram and Dr. John P. Lehoczky for graciously agreeing to be in my dissertation committee and for providing many helpful comments. I would like to thank also Drs. Nilufer and Soner Onder for their invaluable help and orientation during my first years at the University of Pittsburgh, their support greatly reduced the adaptation problems of the new cultural and academic settings.

The very last year of my studies at the University of Pittsburgh was also a tense one, due to stress-inducing factors such as military service and job hunting as well as dissertation writing. It was a great joy and happiness to feel the unconditional support and love of Itır in good and difficult days. Finally, I can not emphasize enough the role of my parents' and grandparents' support to my long education that is culminating in this Ph.D. degree. Without their help and sacrifices, none of these achievements would be possible. While acknowledging that their support was priceless, I dedicate this work to my family.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

In recent years, real-time systems have become increasingly important in numerous application domains, such as automated control and manufacturing, robotics, telecommunication networks, multimedia and military systems. In these applications, *timeliness* is as crucial as the correctness of the results produced. One can even argue that the growing emphasis on temporal predictability is continuously marking the design and analysis of complex computing systems as well as embedded systems.

A *hard* real-time system should satisfy stringent timing constraints of individual tasks: the consequences of missing a deadline may be serious, even catastrophic. Nuclear plants and air-traffic control systems are well-known examples of this category. These systems usually consider worst-case scenarios to guarantee timeliness. In contrast, in *soft* real-time systems missing one or a few deadlines usually degrades the overall performance but does not cause any catastrophic effect. On-line transactions and multimedia applications are examples of such systems. Numerous algorithms and techniques developed in hard and soft real-time theories tacitly assume that a task's output is of no value if it is not executed completely. However, in many application areas a partial or approximate but timely result is usually acceptable. Examples of such areas are multimedia, image and speech processing, time-dependent planning, robot control/navigation systems, medical decision making, information gathering, real-time heuristic search and database query processing.

Reward-based (RB) scheduling is a relatively new approach, aiming to trade precision for timeliness when the available resources are not sufficient to provide worst-case guarantees. Such a situation can also occur as a result of transient overload and/or faults. In this framework, each task is logically decomposed into two subtasks: The *mandatory* part produces a result of acceptable quality and the *optional* part refines the result within the limits of available computing capacity. A nondecreasing *reward function*, associated with the execution of the optional part, captures its contribution to the overall system utility. The primary objective of reward-based scheduling is thus assuring timely completion of mandatory parts, while maximizing a performance metric — usually the total system reward.

The apparent complexity of reward-based scheduling is partly due to the fact that it is a general framework encompassing both hard and soft real-time scheduling theories. In fact, a task with no optional part corresponds to a traditional hard real-time task, and a task with no mandatory part can model a soft real-time task. Although efficient algorithms (and sometimes intractability results) have been obtained by the research community, some problems of great practical and theoretical interest remain relatively unexplored in this field.

Our goal in this dissertation is to address some open problems in the area including *periodic* reward-based scheduling problem and some tractability problems, as well as to investigate and justify the use of the RB scheduling framework as a general paradigm to simultaneously express and incorporate multiple requirement dimensions for real-time computing. This is especially relevant for emerging applications where the *utility, power-efficiency* and *fault tolerance* dimensions will be prime requirements, in addition to *temporal predictability*.

Many real-time tasks are invoked periodically. A considerable number of them such as multimedia and information retrieval applications allow the mandatory/optional subdivision in a natural fashion. An effective solution to the periodic RB scheduling problem should therefore improve the scheduling flexibility and enhance user-perceived system utility.

In recent years, we have also observed a growing interest in *power management* issues, especially for embedded and/or mobile devices which have to rely on limited battery power. An emerging technique, also boosted by the advances in VLSI design, is to dynamically adjust the CPU speed with the aim of saving energy at the expense of increased response time. Needless to say, the temporal predictability may not be compromised in real-time embedded systems, despite the fact that saving power is still highly desirable. We show in this work that the *real-time power-aware* and *reward-based* scheduling techniques are closely related, and that the RB framework can be used to address power management issues. Though a real-time system should be designed with worst case scenarios in mind, the ability to react to the *actual* workload and predict the expected workload allows us to improve the performance of the system. The power-aware scheduling framework, in fact, offers a perfect model to investigate the dynamic aspects of reward-based scheduling, since *reclaiming unused computation time* and *anticipating early completions* can be the key to power savings which would not be achieved otherwise.

The need for *fault tolerance* has been long recognized in hard real-time systems. A critical application should be able to provide correct (at least, acceptable) and timely results even in the presence of faults. Fault tolerance is also a prime issue in reward-based computing systems, since mandatory parts still impose stringent hard real-time requirements. Yet, designing an RB system with fault tolerance techniques common in hard real-time systems can result in an unacceptably low reward. Hence, it is natural to envisage that the next important step in reward-based scheduling will be to demand fault tolerance capabilities, *in addition* to feasibility and maximization of the reward.

This dissertation is organized as follows. In Chapter 2, we first give the basic definitions and examine the current status in real-time systems, fault tolerance, reward-based and power-aware scheduling research areas. In Chapter 3, we present the system model and a summary of dissertation contributions.

In Chapter 4, we present the results of our research on *periodic reward-based scheduling*. This includes the optimal and efficient solution to the problem for the most common (linear and concave) reward functions, extension to multiprocessors, performance studies including a comparison with previous sub-optimal policies, and the consequences of considering more general models. We also present the details of a polynomial-time algorithm to solve a non-linear optimization problem which is the key to obtain an efficient solution.

In Chapter 5, we address the *periodic power-aware scheduling* as an application of the reward-based scheduling framework and as a particularly important area to investigate the dynamic behavior of reward-based systems. We formally show how to relate two problems, and present schemes to improve power savings when the actual workload is lower than the worst-case. We evaluate our schemes on the basis of simulation results.

In Chapter 6, we develop our Fault-Tolerant (FT) Optimality framework for aperiodic RB tasks. The framework aims at incorporating error recovery to the model in order to provision for the faults of mandatory parts, while still keeping the reward as high as possible. We first investigate the (simpler) case where tasks share a common, single deadline before considering multiple deadlines. For each of these models, we address independent tasks and tasks with linear precedence constraints separately.

Chapter 7 concludes the thesis, and in Chapter 8 we elaborate on research prospects to extend this work in the future.

Chapter 2

Background

2.1 Real-Time Systems

In *real-time systems*, tasks are defined with ready times and deadlines. The *ready time* of a task is the time at which it becomes available for the execution, it must produce a correct result by the *deadline*.

An *aperiodic* task is a single invocation of computation. In contrast, a *periodic* task is composed of a sequence of instances which are invoked periodically. Usually, the relative deadline of a periodic task coincides with its period. In addition, tasks can be preemptive or non-preemptive. A *preemptive* task can be interrupted during execution and later resumed. However, a *non-preemptive* task can not be interrupted before it completes.

Tasks may have also *precedence constraints*, which impose a partial order on task executions. Finally, a schedule is said to be *feasible* if all the tasks start after their ready times, complete before their deadlines and precedence constraints are satisfied.

2.1.1 Previous Work on Hard Real-Time Scheduling

For *periodic* scheduling, the seminal work of Liu and Layland [67] established the optimality of EDF (Earliest Deadline First) and RMS (Rate Monotonic Scheduling) among dynamic and fixed priority policies, respectively. EDF dispatches the task with the earliest deadline in the ready queue, while RMS statically sets higher priorities to tasks with smaller periods. Let us denote by C_i and P_i the worst-case execution time and the period of task T_i, respectively. In [67], it is shown that a task set is schedulable by EDF if and only if the total utilization $\sum U_i = \sum \frac{C_i}{P_i} \leq 1$. For RMS, a sufficient condition for schedulability is that the total utilization should not be larger than $m\,(2^{\frac{1}{m}} - 1)$, where m is the number of tasks. However, in the same work it is proven that 100% utilization is achievable under RMS if the tasks' periods are multiple of each other. Later, Lehoczky et al. provided a necessary and sufficient schedulability test (exact characterization) for

RMS [53]. In his Ph.D. thesis [72], Mok proved that LLF (Least Laxity First) is also capable of achieving 100% utilization and hence is optimal among dynamic policies.

If the resources are shared among tasks, mutual exclusion mechanism such as *semaphores* [24] must be used. However, in the case of periodic real-time tasks, this may lead to *priority inversion* and subsequently to unbounded blocking times. Several resource access protocols such as Priority Inheritance Protocol [87], Priority Ceiling Protocol [87] and Stack Resource Policy [3] have been proposed to solve this problem. In our research, we assume that periodic tasks use no shared resources.

In *aperiodic* scheduling, optimality of EDF for a set of (preemptive) independent tasks was established by Jackson [43] and Dertouzos [21] for synchronous and asynchronous arrivals respectively. Lawler presented a polynomial-time algorithm that minimizes the maximum lateness for (hence, decides the feasibility of) a set of tasks with *precedence constraints* and identical arrival times [52]. Chetto et al. elegantly solved the case of arbitrary arrivals by showing how to transform a set of dependent tasks to another independent set of tasks by modifying timing constraints and then applying EDF [18].

Non-preemptive aperiodic scheduling problem with asynchronous arrivals was shown to be NP-Hard [58]. However, EDF is still optimal for this problem if the processor is not allowed to remain idle while there are ready tasks [44]. [94] is a collection of several topics related to scheduling with EDF.

On-line scheduling of real-time tasks in the face of overload has been also addressed by various researchers. A generalization of EDF, addressing overload scheduling using the value density approach (the ratio of a value denoting the importance of the task to the task execution time) was provided by Locke in his Ph.D. thesis [68]. The algorithm was using a best-effort EDF scheduling technique with a rejection policy based on removing tasks with the minimum value density. Baruah et al. proved that, in general, no-online algorithm can guarantee a competitive ratio smaller than 4 with respect to a clairvoyant algorithm[4], when task values are proportional to task execution times. Koren and Shasha developed an algorithm called D^{over}, which can indeed achieve this ratio even under worst-case scenario[47]. If the overload is of infinite duration, then no on-line algorithm can guarantee a constant competitive ratio[94]. Yet, recently several results have been obtained pointing to the possibility of achieving constant competitive factor under the *resource augmentation* assumption. Kalyanasundaram and Pruhs proved in [46] that there are simple on-line scheduling algorithms that are constant-competitive if the on-line scheduler is equipped with a slightly faster processor (i.e. by a factor of $1 + \epsilon$ over the clairvoyant adversary). Phillips et al. extended this line of research in [82], showing that well-known uniprocessor algorithms EDF and LLF can guarantee a constant competitive ratio even for multiprocessors, provided that the CPU speeds are increased by a factor of $2 - \frac{1}{m}$, where m is the number of processors. As it can be seen , this ratio is always smaller than 2[82].

2.2 Reward-Based Scheduling

Reward-Based(RB) scheduling theory can be considered as a generalization of hard and soft real-time system frameworks. In the RB model, each task T_i comprises a mandatory part M_i and an optional part O_i. The worst-case execution time of M_i and O_i are denoted by m_i and o_i, respectively. The mandatory part M_i must complete successfully by the task's deadline, the optional part O_i can only execute after M_i completes. During its execution, O_i *refines* the approximate result produced by M_i while still before the deadline. However, the optional part may be left uncompleted by the deadline if more important/urgent objectives require so. To quantify the 'utility' (or contribution) of the refinement process, a reward function $R_i(t_i)$ is associated with the execution of the optional part O_i, where t_i is the amount of *optional* service time T_i receives beyond the mandatory part.

All previous studies in RB scheduling [20, 22, 23, 66, 85] assume that the reward functions are nondecreasing, which is based on the observation that the utility of a task does not decrease by allowing it to run longer in almost every practical application.

A schedule of RB tasks is *feasible* if all the mandatory parts complete in a timely fashion. Furthermore, it is *optimal* if the reward accrued maximizes a performance metric. The performance metric most often considered is to maximize the (weighted) total reward of RB tasks [66, 20, 89, 22].

An interesting question concerns types of reward functions which represent realistic application areas. Figure 2.1 shows the form of the most common reward function types. A *linear* reward function [65, 64, 66, 20, 89, 91] models the case where the benefit to the overall system increases *uniformly* during the optional execution. The simplest case is to have *identical* linear functions regardless of the task currently executing. *Nonidentical* (or weighted) linear functions allow us to distinguish between the importance of different tasks' optional parts.

Concave reward functions [22, 23, 50, 85, 54] go further by addressing the case where the greatest increase/refinement in the output quality is obtained during the first portions of optional executions. A function $f(x)$ is concave if and only if for all x, y and $0 \leq \alpha \leq 1$, $f(\alpha x + [1-\alpha]y) \geq \alpha f(x) + (1-\alpha)f(y)$. Geometrically, this condition means that the line joining any two points of a concave curve *may not be above the curve.* Examples of concave functions are linear functions $(kx + c)$, logarithmic functions $(\ln[kx + c])$, some exponential functions $(c \cdot [1 - e^{-kx}])$ and k^{th} root functions $(x^{1/k})$. Note that the first derivative of a nondecreasing concave function is nonincreasing (diminishing returns). Having nondecreasing concave reward functions means that although the reward monotonically increases beyond M_i, its *rate of increase* decreases or remains constant with

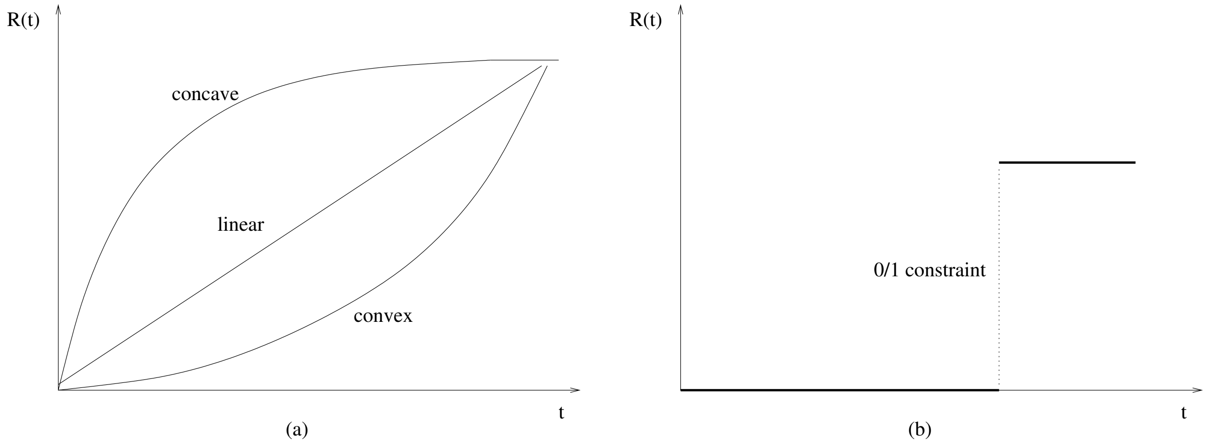


Figure 2.1: Reward functions (a) Continuous concave, convex and linear (b) 0-1 constraint

time. Hence, linear and concave reward functions successfully model the applications where earlier slots of optional executions are *not* less valuable than the later ones.

Convex reward functions (a function f is convex if $-f$ is concave) represent the case where a considerable benefit is obtained only later during the optional execution. It should be noted that reward functions with 0-1 constraints, where no reward is accrued unless the *entire* optional part is executed, have also received interest in the literature [66, 35, 89]. Unfortunately, this problem has been shown to be NP-Complete in [89].

A generalization of the 0-1 type reward functions in the form of increasing functions with multiple steps was proposed and used in the context of discrete QoS Options in the Q-RAM model in [55, 56]. Although the problem is NP-Complete in nature, Lee et al. provided approximation schemes and pseudo-polynomial time algorithms to address this case. More details about the work on the Q-RAM model are presented in Section 2.2.1.3. In this dissertation, we consider only continuous reward functions.

2.2.1 Previous Work on Reward-Based Scheduling

2.2.1.1 Imprecise Computation Model

A large body of work on RB scheduling originated from *Imprecise Computation* model, where [64] can be considered as a starting point. In this work, mandatory/optional semantic distinction and imprecise computation model were introduced. The objective of the scheduling problem was defined to guarantee the timeliness of mandatory parts, while minimizing the total error. The *error* of a task is the amount of optional work left uncompleted. Later, the concept was generalized to weighted error functions. Notice that the error functions in this context are analogous to (dual of) the reward functions of our framework.

In [88], optimal preemptive algorithms based on network-flow algorithms, were first proposed. Faster algorithms have been devised later in [89] to optimally schedule independent tasks with identical weights (in time $O(n \log n)$) and different weights (in time $O(n^2)$). In the same work, minimizing the total error with 0-1 constraints have been shown to be NP-Complete even with identical weights. Subsequently, Ho et al. [35] proposed a $O(n^2)$ heuristic for this last problem, along with polynomial-time algorithms for maximizing the *number* of optional tasks that are entirely executed under 0-1 constraints.

One should note that the imprecise computation problem with precedence constraints is not harder than the independent tasks model, since one can first modify the ready times/deadlines of tasks to 'reflect' precedence constraints and then solve that problem [66].

Recently, the *extended imprecise computation model* was introduced by Feng and Liu in [26]. This study describes several heuristics regarding the problem of assigning service times to tasks within chains. All the tasks in a chain share a common, single temporal constraint. Each chain is composed of imprecise computation tasks with linear precedence constraints and the input quality of each task depends upon the output quality of its predecessor. This work assumes linear error functions.

The periodic imprecise computation problem was first addressed in [65, 19]. A more detailed study appeared in [20] where the possible application areas are classified as "error non-cumulative" and "error cumulative". In "error cumulative" applications, such as radar tracking, an optional instance must be executed completely at every (predetermined) k invocations. The authors further proved that the case of error cumulative jobs is an NP-Complete problem. On the other hand, in "error non-cumulative" applications, errors (or optional parts left unexecuted) have no effect on the future instances of the same task. Well-known examples of this category are image/speech processing, information retrieval and display tasks. For these jobs, they showed how to guarantee a feasible schedule by constantly favoring mandatory parts and proposed a class of heuristics to minimize the total error. All heuristics except one operate independently from the nature of error functions. We restrict our work in this research to error non-cumulative jobs.

On-line scheduling of imprecise computation tasks has been examined in [91]. The algorithms address separately the cases where the workload consists solely of on-line tasks and of a mix of on-line and off-line tasks. Only identical linear error functions are considered. However, Baruah and Hickey later proved [7] that an on-line algorithm for imprecise computation tasks in general may perform arbitrarily badly when compared to a clairvoyant scheduler.

Performance metrics other than the (weighted) total error have been also studied. In [90], two polynomial-time algorithms are presented for unit-weight error functions to obtain a 'balanced' distribution of error among imprecise computation tasks. In other words, the aim is to minimize the maximum error.

Leung and Wong investigated the problem of minimizing the number of tardy tasks for a given maximum, tolerable error and showed it to be NP-Complete [59]. Minimizing the average response time, while keeping the total error less than a threshold has been also shown to be NP-Hard for almost every reasonable model in [60].

2.2.1.2 Increased-Reward-With-Increased-Service (IRIS) model

An alternative model allows tasks to get increasing reward with increasing service, without an upper bound on the execution times of the tasks and without the separation between mandatory and optional parts [22]. A task executes for as long as the scheduler allows. Typically, a nondecreasing *concave* reward function is associated with each task's execution time. In [22, 23] the problem of maximizing the total reward in a system of independent aperiodic tasks is addressed. An optimal polynomial-time solution with static task sets (identical ready times) is presented, as well as two extensions that include mandatory parts and on-line policies for dynamic task arrivals.

Note that imprecise computation and IRIS models are closely related, since the performance metrics can be defined as duals (maximizing the total reward vs. minimizing the total error). Similarly, a concave reward function corresponds to a convex error function, and vice versa.

2.2.1.3 Quality of Service (QoS) Management

A QoS-based resource allocation model has been proposed in [85, 54, 55, 56]. In that study, the problem is to optimally allocate multiple resources to the various applications such that they simultaneously meet their minimum requirements along multiple QoS dimensions and the total system utility is maximized. In one aspect, this can be viewed as a generalization of optimal CPU allocation problem in RB scheduling, to multiple resources and multiple reward metrics (quality dimensions). Further, dependent and independent quality dimensions are separately addressed for the first time in this work.

In [85], Rajkumar et al. addressed the case of a single resource and multiple QoS (reward) dimensions and provided a pseudo-polynomial time algorithm for continuous concave utility functions.

In [54], the problem of a single resource and two or more QoS dimensions (with one dimension being discrete and dependent on another) is explored. The problem is proven to be NP-Hard, and an approximation algorithm based on Concave Majorant Optimization is presented. Then, the authors show that the problem of multiple resources and single QoS dimension can be reduced to a mixed integer programming problem; they also report the run-time of example problems when solved by commercial software packages.

Lee et al. further considered discrete QoS dimensions in [55], as opposed to the continuous QoS dimension assumption of their previous research in [85]. They proposed two approximation algorithms for the case of single resource and multiple quality dimensions. The first algorithm yields a solution that is within a known bounded distance from the optimal solution, while the second is capable of returning an allocation whose distance from the optimal solution can be reduced by giving more time to the allocation routine.

Finally, the more general case of multiple resources and QoS dimensions was addressed in [56]. Due to the intractable nature of the problem, a mixed integer programming formulation is developed to yield near-optimal results, followed by an approximation algorithm based on local search techniques. The execution time and quality distance to the optimal solution of these algorithms are experimentally evaluated.

We must underline that a fundamental assumption of Q-RAM model is that the resource allocation is in terms of *utilization of resources*, usually expresssed as a fraction (percentage) of the available resource capacity. In the CPU allocation case for the periodic real-time tasks, this translates to identical service time allocation assumption for each instance of a given task. Our work falls rather along the lines of Imprecise Computation and IRIS models, where aperiodic scheduling is also an issue and the reward accrued has to be computed separately over *each* instance for the periodic case. In periodic RB scheduling, we will use the term *instance-based* for our (and Imprecise Computation) framework to distinguish it from the *utilization-based* approach of QRAM model.

2.3 Power-Aware Scheduling

With the recent advent of mobile, wireless and embedded computing, the *power management* issues, especially for devices relying on battery power, became a major research avenue. The problem has necessarily multiple aspects, including VLSI design, memory management, operating systems and user interface dimensions, among others [33, 79]. Such on-going research has important implications for real-time systems design, simply because many applications running on power-limited systems inherently impose temporal constraints on the response time such as real-time communication (e.g. in satellites transmitting video and audio data).

One powerful and increasingly popular technique to reduce CPU power consumption is based on dynamically adjusting the supply voltage and the frequency of the CPU on-the-fly. In fact, the power consumption of an on-chip system is a strictly increasing *convex* function of the supply voltage V_{dd}, but its exact form depends on the technology. For example, the dominant component of energy consumption in widely popular CMOS technology is the *dynamic power dissipation* P_d, which is given by:

$$P_d = C_{ef} \cdot V_{dd}^2 \cdot f$$

where C_{ef} is the effective switched capacitance and f is the frequency of the clock. On the other hand, the gate delay D is inversely related to the supply voltage V_{dd}:

$$D = k \cdot \frac{V_{dd}}{(V_{dd} - V_t)^2}$$

where k is a constant, and V_t is the threshold voltage.

From these equations, it can be seen that one can obtain striking (cubic) power savings if the supply voltage and the clock frequency are reduced simultaneously. However, by doing so, the response time will increase in a linear fashion. Thus, the *power-aware scheduling* technique (known also as *variable voltage scheduling* or *dynamic voltage scaling*) is based on obtaining energy savings by simultaneously reducing the supply voltage and the clock frequency, at the expense of increased latency. The exact form of the speed/power relation can be obtained by resorting to normalization [38]; in general, it is a strictly increasing polynomial of at least second degree.

The latency resulting from the CPU speed reduction may or may not be acceptable for general user or non-real-time applications; however, it should be clear that provisions must be taken to prevent the violation of any (hard) real-time constraints. Thus, the *Real-Time Power-Aware Scheduling* problem aims at minimizing the CPU energy consumption through speed reduction while still meeting the deadlines for all the tasks.

2.3.1 Previous work on Power-Aware Scheduling

The idea of trading speed for energy savings was first proposed in the absence of stringent timing constraints. In its simplest form, the power/speed relation can be exploited by shutting down the CPU whenever the system is (likely going to stay) idle for an acceptably long period. However, this *predictive shutdown* technique [93] remains inefficient and sub-optimal even for a scheduler with perfect knowledge of idle intervals, due to the convex dependence between the speed and the power consumption.

Weiser et al. adopted an approach where time is divided into 10-50 msec intervals, and the CPU clock speed (and the supply voltage) is adjusted using predictions based on processor utilization in recent intervals [100]. Govil et al. proposed and experimentally evaluated several predictive and non-predictive approaches for voltage changes [30].

For real-time systems, the proposed voltage scaling techniques focus on minimizing energy consumption in the system, while still meeting the deadlines. Yao et. al [101] provided a polynomial-time static off-line scheduling algorithm, assuming aperiodic tasks and worst-case execution times. Heuristics for on-line scheduling of aperiodic tasks while not hurting the feasibility of off-line periodic requests are proposed in [37]. Non-preemptive power aware scheduling is investigated in [36]. Concentrating on a periodic task set with *identical periods*, the effects of having an upper bound on the voltage change rate are examined in [38]. After proven the intractability of the problem, the authors proposed a fast heuristic. Slowing down the processor whenever there is a single task eligible for execution was explored in [92].

Krishna and Lee addressed the power-aware scheduling problem, in the context of systems with two (discrete) voltage levels. In [57], they showed the problem to be NP-Complete, and then investigated the power-aware Rate Monotonic Scheduling. Cyclic and Earliest Deadline First based general periodic tasks were the subject of [49], where dynamic energy reclaiming issues were also addressed. However, systems which are able to operate on a (more or less) continuous voltage spectrum are rapidly becoming a reality thanks to advances in power-supply electronics and CPU design [32, 74]. For example, the Crusoe processor is able to dynamically adjust clock frequency in 33 MHz steps [97].

2.4 Fault Tolerance

Faults can be generally classified as software or hardware faults. A *hardware fault* is some physical defect that can cause a component to malfunction. A *software fault* is a specification or implementation deficiency that can cause the program to fail for a given set of inputs. An *error* is a manifestation of a fault and error(s) may further result in *failures*.

Faults can be further divided according to their temporal behaviors: *Permanent faults* may result in total failure of the computing unit and remain until it is repaired or replaced. On the other hand, *transient faults* affect temporarily the unit, and disappear after a relatively short duration. Between these extremes, there are *intermittent faults* that disappear and re-occur in time.

There are three main techniques to maintain a system's normal performance in the presence or possibility of faults [84]. *Fault avoidance* is a technique that attempts to prevent the occurrence of faults. Design reviews, extensive testing and other quality control methods are the most common fault-avoidance approaches. *Fault masking* does not prevent faults per se, but hinders them from causing errors; *modular redundancy* coupled with *voting* is a widely used fault masking technique which relies on concurrently doing the computation on multiple units and choosing the result agreed by the majority. *Fault tolerance* aims to prevent system failures even in the presence of faults/errors. A fault-tolerant system should be able to continue its basic/essential functions so as to maintain the integrity of the system.

Almost every fault-tolerant system relies on some form of *redundancy*, which implies the incorporation of resources, information or time in addition to what is required for normal operation. Hardware redundancy, software redundancy, time redundancy and information redundancy are the main types of redundancy used in contemporary systems. Fault-tolerance usually requires several *steps* in order to prevent system failure/serious damage:

1. Fault detection: Before undertaking any recovery procedure, we must be able to recognize the occurrence of the fault.

2. Fault location is the process of determining where the fault has occurred so that any further action can be taken.

3. Fault containment: Usually, it is crucial to isolate the fault and prevent its effects from propagating throughout the system.

4. Error recovery is the final step and involves the process of regaining operational status and/or the integrity of the system.

Two main approaches to error recovery are backward and forward recovery techniques. In *backward error recovery*, the system is rolled back to a moment in time before the error is believed to have occurred; that is, the state of the system is restored to a previous safe state. Then, the computation is repeated. Instead of repeating the computation from scratch, *checkpointing* techniques periodically save the important state information to stable storage and simultaneously run an error detection routine [11]. For real-time applications, the error detection and any subsequent recovery can be carried out at the end of the computation. Recovery can be in the form of *re-execution* of the computation or the task. Alternatively, one may execute a *recovery block* with similar or identical functionality. Several recovery blocks may also exist; in this case they are run one after another as long as the error persists. Obviously, the functionality and the number of recovery blocks are limited by the amount of time redundancy available.

In *forward error recovery* schemes, the system is *not* rolled back to a previous safe state, for example the recovery may be performed by another processor while the processor where the fault is detected continues with (other) processing.

2.4.1 Previous work on Fault-Tolerant Hard Real-Time Systems

Permanent faults in hard real-time systems are addressed by the hardware redundancy, usually by means of running concurrent/clone tasks on multiprocessors. For instance, the Integrated Multiplex System uses dual-redundancy to achieve fault tolerance [45]. In [76], an algorithm for fault tolerant scheduling of periodic preemptive tasks on multiprocessors is presented.

Among software redundancy techniques one needs to mention the *recovery block* approach of Randell [86]. *N-version programming* [1] involves the concurrent execution of several versions of the same application and then voting on them to decide on the correct output.

Many fault-tolerant scheduling studies so far were concerned with the transient faults of periodic real-time tasks. In [63], two instances of each periodic real-time task (a primary and a backup) are scheduled on a uniprocessor system, with the goal of maximizing the number of backups scheduled. An admission control algorithm using RMS in conjunction with recovery blocks is presented by Burns et al. [16]. Ramos-Thuel adapted the Slack Stealing Algorithm to serve re-execution requests in [96]. Ghosh et al. assured dependability in RMS by reserving a slack for re-execution in case of transient faults [29]. Pandya and Malek show that one can always re-execute all instances that did not complete at the time of detection of a single fault using RMS if the total utilization is less than 50% [77]. The multiprocessor scheduling of periodic real-time tasks with error recovery is investigated by Lauzac in [51].

For aperiodic case, optimality of EDF in the presence of transient faults was shown in [61]. An efficient algorithm to decide if a given set of aperiodic tasks can tolerate k faults with EDF scheduling is provided in [62]. In [28], two algorithms provide minimum extra slack for a queue of real-time tasks to tolerate transient faults.

2.4.2 Previous work on Fault-Tolerant Reward-Based Scheduling

A study to tolerate transient faults of mandatory parts in Reward-Based Scheduling applies a backward error recovery mechanism in conjunction with a checkpointing mechanism [12]. An extension to on-line scheduling was described in [13]. An important assumption of these works is the a priori knowledge of worst-case fault profile *per task*. That is, a fault vector $V = \{k_1, k_2, \ldots, k_n\}$ is determined by the user, where k_i is the number of failures that the mandatory part M_i may suffer during the execution. This information is much more difficult to obtain/predict as opposed to the maximum number of faults of the *entire task set* and forces the provisioning for simultaneous occurrences of all the $(\sum k_i)$ faults for all the tasks, which may yield the rejection of many task sets and low CPU utilization. Furthermore, even when the task set is accepted, the schedule needs to be updated *on-line* every time a mandatory part completes successfully (without failure). As the authors observe in [12], this approach becomes prohibitively expensive in systems with low fault rates, which is usually the case. Also, the error functions are assumed to be all linear and all tasks independent.

In his Ph.D. thesis [102], Yu investigated replicating *periodic* imprecise computation tasks on multiprocessors in order to tolerate permanent processor faults. The number of the required "clones" is determined by the user. Each clone is assigned to a different processor and should be guaranteed to complete (at least) the mandatory part by the deadline. Heuristic algorithms to achieve that goal while maximizing the *total weighted utilization* are described in [102, 103]. Only linear error functions and independent tasks are considered.

Chapter 3

Research Scope and Contributions

In this dissertation, we concentrate on uniprocessor systems. However, one of our main results extends easily to identical shared-memory multiprocessors and it will be presented in Section 4.1.1.

3.1 Models

3.1.1 Task Model

We consider a set **T** composed of n RB tasks $T_1, T_2, \ldots, T_n$. Each task T_i is composed of a mandatory part M_i and an optional part O_i, with worst-case execution times denoted by m_i and o_i, respectively. Similarly, d_i and r_i denote the deadline and the ready time of task T_i. We assume that *preemption* is allowed. We will assume that the overhead of preemption is negligible and that tasks share no resources except CPU.

The reward function $R_i(t_i)$ indicates the reward accrued by T_i when it receives t_i units of *optional* service beyond mandatory part. It is of the form:

$$R_i(t_i) = \begin{cases} F_i(t_i) & if \quad 0 \le t_i \le o_i \\ F_i(o_i) & if \quad t_i > o_i \end{cases} \tag{3.1}$$

Throughout this dissertation, we will assume that each F_i is a nondecreasing, concave and continuously differentiable function over nonnegative real numbers, unless stated otherwise. However, we indicate whenever a more efficient solution exists by the assumption of linearity.

Linear and general concave functions are considered as the most realistic and typical in the literature since they have non-increasing marginal returns and adequately capture the behavior of many application areas like image and speech processing [17, 9, 27, 98], multimedia applications [85], time-dependent planning [14], robot control/navigation systems [104], real-time heuristic search [48], information gathering [31] and database query processing [99].

In order to cover a wide range of applications, we consider (possible) co-existence of linear and general concave functions with upper bounds. To our knowledge, this general problem has not been efficiently solved before.

In this research, we address periodic and aperiodic scheduling problems separately.

For the *periodic* case, we assume that all tasks $T_1, \ldots, T_n$ are independent, in phase and error non-cumulative[20]. The period of T_i is denoted by P_i, which is also equal to the relative deadline of the current invocation. We refer to the j^{th} invocation of task T_i as T_{ij}.

A schedule of periodic RB tasks is **feasible** if mandatory parts meet their deadlines at every invocation. Given a feasible schedule of the task set **T**, the **average reward** of task T_i is defined as:

$$REW_i = \frac{P_i}{P} \sum_{j=1}^{P/P_i} R_i(t_{ij}) \tag{3.2}$$

where P is the *hyperperiod*, that is, the least common multiple of $P_1, P_2, \ldots, P_n$ and t_{ij} is the service time assigned to the j^{th} instance of optional part of task T_i (i.e, O_{ij}). That is, the average reward of T_i is computed over the number of its invocations during the hyperperiod P, in an analogous way to the definition of average error in [20].

The **average weighted reward** of a feasible schedule is then given by:

$$REW_{PER} = \sum_{i=1}^{n} w_i \; REW_i \tag{3.3}$$

where w_i is a constant in the interval (0,1] indicating the relative importance of optional part O_i.

For *aperiodic* tasks, we address the cases when the tasks are independent and there are (linear) precedence constraints. In general, we assume that all tasks are simultaneously ready to execute; however, we comment on the implications of relaxing this assumption in Section 6.2.1.3.

A schedule of aperiodic RB tasks is said to be **feasible** if every mandatory part completes before its deadline. Moreover, the performance metric that we want to maximize is the *total weighted reward:*

$$REW_{APER} = \sum_{j=1}^{n} w_i \, R_i(t_i) \tag{3.4}$$

where the "weight" w_i is a constant in the interval (0,1].

A schedule is **optimal** if it is feasible and maximizes the reward associated by the task model, that is, the expression (3.4) for the aperiodic case and the expression (3.3) for the periodic case. The determination of the optimal schedule involves clearly the computation of optimal optional service times. Noting that the reward accrued by each optional part O_i does not increase beyond the upperbound o_i, this computation can be expressed as an optimization problem where the objective is to find t_i values so as to:

$$\begin{aligned} &\text{maximize} && \sum_{i=1}^{n} F_i(t_i) && (3.5)\\ &\text{subject to} && 0 \leq t_i \leq o_i \qquad i = 1, \ldots, n && (3.6) \end{aligned}$$

$$\text{There exists a feasible schedule with } \{m_i\} \text{ and } \{t_i\} \text{ values} \qquad (3.7)$$

We note that for the periodic case, each t_{ij} value should be considered as a separate unknown. Finally, although (3.3) and (3.4) are the most general formulations, it is easy to see that the weight w_i can always be incorporated into the reward function $F_i()$, by replacing it by $w_i\, F_i()$. Thus, we will assume that all weight (importance) information is already expressed in the reward function formulation and we will omit w_i in (3.4) and (3.3) henceforth.

3.1.2 Power Model

In our power-aware real-time scheduling research, we consider a processor with variable voltage and speed capability. We assume that the CPU speed S (expressed in the number of CPU cycles per time unit) can be changed between a minimum speed S_{min} (corresponding to a minimum supply voltage necessary to keep the system functional) and a maximum speed S_{max}. For convenience, we normalize the speed values with respect to S_{max}; that is, $0 \leq S_{min} \leq S \leq S_{max} = 1$.

We assume that the voltage/speed change is performed at the context switch time. Recent reports from rapidly advancing VLSI research in dynamic voltage scaling imply that the latency overhead associated with voltage/speed transitions is improving: While Namgoong et al. reported the time taken to reach steady state at the new voltage level as under 6 ms/V as of 1996 [74], Burd et al. indicate that this delay is limited by 70 μs in the new ARM V4 processor based system, including frequency transition. For the Low-Power lpARM processor, clock frequency transitions take approximately 25 μs [81]. In a study based on StrongARM SA-1100 processor operating under constant supply voltage, this delay is reported to be less than 150 μs[71]. When modifying the supply voltage and the clock frequency of the StrongARM SA-1100 processor, Pouwelse et al. have found that [83], the voltage and speed increase is rapidly handled (40 μs), but the decrease can take up to 5.5 ms. A full voltage transition can be performed in less than 300 μs in the Transmeta Crusoe 5400 processor[97]. In case that this overhead is not negligible, it can be incorporated into the worst-case workload of each task.

In the variable CPU speed settings, it is clear that the execution time of a task is dependent on both the number of CPU cycles required, and the CPU speed. We will denote by C_i and AC_i, the number of CPU cycles required by T_i in the worst-case and in the actual execution, respectively. Note that the worst-case execution time of a task T_i under the speed S_i is given by $\frac{C_i}{S_i}$.

We define U_{tot} as the worst-case utilization of a periodic task set under maximum speed $S_{max} = 1$, that is, $U_{tot} = \sum_{i=1}^{n} \frac{C_i}{P_i}$. Note that the schedulability theorems for periodic real-time tasks [67] imply that $U_{tot} \leq 1$ is a necessary condition to have at least one feasible schedule; hence, when we address power-aware scheduling for periodic real-time tasks, we will assume that $U_{tot} = \sum_{i=1}^{n} \frac{C_i}{P_i} \leq 1$.

We assume that the process descriptor of the task T_i has two extra fields related to speed settings, in addition to other conventional fields. The first one, S_i, denotes the *current* CPU speed at which T_i is executing. The other field $\widehat{S_i}$ denotes the *nominal* speed of T_i, which is the indicator of the "default" speed of T_i. In other words, for each task that is dispatched, the operating system sets $S_i = \widehat{S_i}$, prior to any dynamic speed adjustment.

The power consumption of the processor under the speed S is given by $g(S)$, which is assumed to be a strictly increasing convex function, represented by a polynomial of at least second degree [101, 38]. If the task T_i occupies the processor during the time interval $[t_1, t_2]$, then the *energy* consumed during this interval is $E(t_1, t_2) = \int_{t_1}^{t_2} g(S(t))dt$. A feasible schedule is **energy-optimal** in time interval $[t_1, t_2]$ if $E(t_1, t_2)$ is minimized.

3.1.3 Fault Model

In the part of our work which incorporates error recovery into the RB scheduling model, we address tolerating *transient faults of mandatory parts*. Transient faults are much more frequent than permanent faults in most systems [42]. Furthermore, it is impossible to tolerate permanent faults in uniprocessor systems without having some form of hardware redundancy/spare processor. We assume that the system has error containment capabilities to prevent the propagation of a specific error to other tasks.

In our approach, the results are produced or committed at the end of mandatory parts and then again at the end of optional parts. We assume the existence of a backward error recovery mechanism for any error detected at the end of a mandatory part M_i. This mechanism may involve either re-executing the mandatory part or invoking a recovery block. If an error is detected at the end of the optional part O_i, then it is ignored and the result of the optional part is not committed.

We assume that at most k faults may occur throughout the execution of the mandatory parts. However, no specific assumption is made about the location or distribution of the faults, except that only user-level tasks are subject to faults. As two extreme scenarios, a given task may fail k times, or k distinct tasks may fail each once.

The recovery block associated with the j^{th} fault of M_i and its worst-case execution time will be denoted by $B_{i,j}$ and $b_{i,j}$, respectively. However, when we deal with the single fault case ($k = 1$), we will freely use the notation B_i (b_i), as opposed to $B_{i,1}$ $(b_{i,1})$.

Note that if the re-execution mechanism is chosen, $b_i = m_i$. Otherwise, it is to be estimated from the specification of the alternate recovery operation. It is worth observing that, if $b_{i,j} = 0 \; \forall i, j$, then the model collapses to the one where all errors are ignored, that is, the traditional reward-based scheduling model.

3.2 Thesis Contributions

In addition to solving some open problems in RB scheduling, the dissertation, as a whole, **investigates and justifies the use of the RB framework as a general paradigm to express and address multiple requirement dimensions in real-time systems**. While *timeliness* is a common requirement in each part of our study, we show that *power-efficiency* and *fault tolerance* can be effectively incorporated to the real-time systems design and operation using the framework, in addition to *utility (reward)* considerations.

More specifically, in this dissertation, **we address the scheduling of periodic RB tasks** and **the incorporation of error recovery to aperiodic RB tasks, while still yielding maximum reward**. We explore also the **periodic power-aware scheduling of real-time tasks** as an application of the periodic RB scheduling framework that we develop and as a setting in which the *dynamic* scheduling issues in RB systems can be further investigated.

- **Periodic Reward-Based Scheduling**: While several efficient algorithms exist for optimal scheduling of *aperiodic* RB tasks, only a class of heuristics was available for the *instance-based* periodic model prior to our research [20]. We address the problem of generating provably optimal schedules for the most common (that is, linear and concave) reward functions and provide an efficient and exact solution. We obtain a fundamental result which states that it is always possible to find an optimal schedule where the optional service time of a task does not change from instance to instance. We formulate the problem as a concave optimization problem and develop a polynomial-time algorithm to compute the optimal service times. We also prove the optimality of Rate Monotonic Scheduling (with harmonic periods), Earliest Deadline First and Least Laxity First policies for the case of uniprocessors when used with the optimal service times we computed. Moreover, we extend our result by showing that any policy which can fully utilize *all* the processors is also optimal for the multiprocessor periodic reward-based scheduling. Although the task model encompasses most of the realistic applications, we additionally investigate the consequences of relaxing some fundamental assumptions. In particular, we show that the optimality of identical service times no longer holds if we relax any of the following assumptions: (a) *Deadline = Period*, (b) *the dynamic* *priority scheduling*, and (c) *the concavity of the reward functions*. Finally, we prove that the Reward-Based Scheduling problem with *convex* reward functions is NP-Hard.

- **Periodic Power-Aware Scheduling:** With the increasing emphasis on power-efficiency issues, minimizing the energy consumption while still meeting all the timing constraints by dynamically adjusting the CPU speed has become a problem of considerable theoretical and practical importance. In our power-aware scheduling research, we first formally establish the *equivalence of the problem* to the reward-based scheduling problem. After providing the *static* optimal solution for the periodic power-aware hard real-time tasks using our optimal reward-based framework, we explore *dynamic* scheduling strategies to adjust the static solution according to the *actual* workload of the system in on-line fashion, as opposed to worst-case assumptions. This is particularly relevant for real-time applications where tasks usually complete earlier than the worst-case. To this aim, we provide a *Generic Dynamic Reclaiming* technique to allow the operating system to detect early completions and reduce the CPU speed further, thereby obtaining additional power savings. Finally, we introduce the *Aggressive Scheduling* technique, where, based on average-case workload information of the system, we *anticipate* early completions and *speculatively reduce the CPU speed*. Both the dynamic reclaiming and aggressive scheduling techniques are formally proven to be sound, that is, no deadline miss will occur even under the worst-case scenario that can happen after any speed adjustment decision. Experimental evaluation points to the fact that energy gains up to 60% *over the static solution* can be reached through a combination of both techniques.

- **Fault Tolerance and Optimality for Aperiodic RB Tasks (FT-Optimality):** Although the RB models allow for greater scheduling flexibility, the execution of mandatory parts still exhibit hard real-time characteristics. Our work is motivated by the observation that optional parts that do not impose stringent real-time requirements provide intrinsically a sort of slack (time redundancy) for recovery/re-execution of mandatory parts. However, one needs to keep track of the optional parts and schedule them appropriately so as to be able to use their CPU allotments for recovery. In this aspect of our research, we develop our *Fault-Tolerant Optimality* framework, in which we present a systematic approach which yields a schedule allowing timely recoveries and having the maximum total reward, while keeping the run-time overhead as low as possible. Focusing on exclusively aperiodic tasks, we investigate both independent and chain tasks, and both the single deadline and multiple deadlines cases. In case of the chain tasks, we provide an exact characterization of Fault-Tolerant schedules. Using this characterization, we show how to generate the FT schedule with the highest reward even for multiple deadlines and general concave functions. For independent tasks, we provide an exact solution for the single deadline case; and propose two recovery models for the case of multiple deadlines. In *Delayed Recovery*, only schedule segments which are originally allocated to optional parts are used for recovery; while in *Immediate Recovery*, the recovery block is executed as soon as the error is detected by postponing the execution of all other tasks and by possibly sacrificing optional parts. We show that the FT-Optimality problem with Delayed Recovery is inherently intractable, while the one with Immediate Recovery admits an optimal solution in pseudo-polynomial time.

Chapter 4

Periodic Reward-Based Scheduling

Many real-time tasks are periodic in nature. Further, an important number of applications is suitable for RB scheduling framework. Well-known examples are periodic tasks which receive, process and transmit video, audio or compressed images [17, 9, 27, 85, 98] and information retrieval tasks [31, 99]. With the advance of multimedia and networking technologies, this type of applications is likely to become more and more widespread. Moreover, these applications readily admit a mandatory/optional semantic distinction: the mandatory part produces an image, frame or output of acceptable quality and the optional part improves on it. Hence, an optimal solution to the periodic RB scheduling problem should yield the maximum utility for the applications at hand, while guaranteeing feasibility.

Aspects of Periodic Reward-Based Scheduling Problem

The difficulty of finding an optimal schedule for a periodic reward-based task set has its origin on two objectives that must be simultaneously achieved, namely:

i. Meeting deadlines of mandatory parts at *every* periodic task invocation.

ii. Scheduling optional parts to maximize the total (or average) reward.

These two objectives are both important, yet often incompatible. In other words, hard deadlines of mandatory parts may require sacrificing optional parts with greatest value to the system.

The analytical treatment of the problem is complicated by the fact that, in an optimal schedule, optional service times of a given task may *vary* from instance to instance which makes the framework of classical periodic scheduling theory inapplicable. Furthermore, this fact introduces a large number of variables in any analytical approach. Moreover, by allowing nonlinear reward functions to better characterize the optional tasks' contribution to the overall system, the optimization problem becomes computationally harder.

In [20], Chung, Liu and Lin proposed the strategy of assigning *statically* higher priorities to mandatory parts. This decision, as proven in that paper, effectively achieves the first objective mentioned above by securing mandatory parts from the potential interference of optional parts.

Optional parts are scheduled whenever no mandatory part is ready in the system. In [20], simulation results regarding the performance of several policies which assign static or dynamic priorities among optional parts are reported. We call the class of algorithms that statically assign higher priorities to mandatory parts *Mandatory-First Algorithms*.

In our solution, we do *not* decouple the objectives of meeting the deadlines of mandatory parts and maximizing the total (or average) reward. We formulate the periodic reward-based scheduling problem as an optimization problem and derive an important and surprising property of the solution for the most common (i.e., linear and concave) reward functions. Namely, *we prove that there is always an optimal schedule where optional service times of a given task do not vary from instance to instance.* This important result immediately implies that the optimality (in terms of achievable utilization) of any policy which can fully use the processor in case of hard-real time periodic tasks also holds in the context of reward-based scheduling (in terms of total reward) *when used with these optimal service times*. Examples of such policies are Rate Monotonic Scheduling with harmonic periods (RMS-h) [67], Earliest Deadline First (EDF) [67] and Least Laxity Lirst (LLF) [72] scheduling disciplines. We also extend the framework to homogeneous multiprocessor settings and prove the optimality of any policy which can fully utilize *all* the processors.

Following these existence proofs, we address the problem of efficiently computing optimal service times and provide polynomial-time algorithms for linear and/or general concave reward functions. Note that using these optimal and constant optimal service times has also important practical advantages: (a) The runtime overhead due to the existence of mandatory/optional dichotomy and reward functions is removed, and (b) existing RMS (with harmonic periods), EDF, and LLF schedulers may be used without any modification with these optimal assignments.

The main result about the optimality of any periodic policy which can fully utilize the processor is obtained in Section 4.1. The extension to identical multiprocessors case is obtained in Section 4.1.1. In Section 4.2, we analyze the worst-case performance of Mandatory-First approaches and show that they can behave arbitrarily badly with respect to the optimal solution. In Section 4.4, we show that the concavity assumption is also necessary for computational efficiency by proving that allowing continuous convex reward functions results in an NP-Hard problem.

A Motivating Example:

Before describing our solution to the problem, we present a simple example based on a synthetic task set, which shows the performance limitations of *any* Mandatory-First algorithm. Consider two tasks where $P_1 = 4, m_1 = 1, o_1 = 1, P_2 = 8, m_2 = 3, o_2 = 5$. Assume that the reward functions associated with optional parts are linear and $f_1(t_1) = k_1 t_1$, $f_2(t_2) = k_2 t_2$. Furthermore, suppose that k_2 associated with the reward accrued by T_2 is negligible when compared to k_1, (i.e., $k_1 \gg k_2$). In this case, the "best" algorithm among "Mandatory-First" approaches should produce the schedule shown in Figure 4.1.

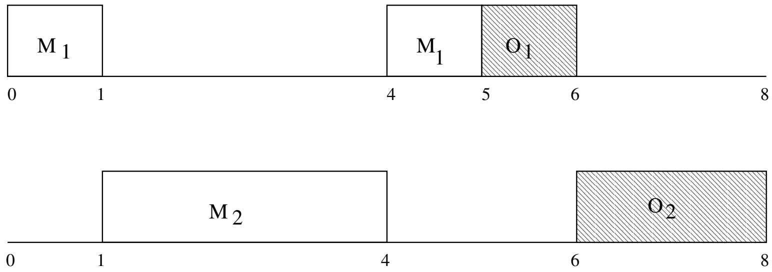


Figure 4.1: A schedule produced by Mandatory-First Algorithm

Above, we assumed that the Rate Monotonic Priority Assignment is used whenever more than one mandatory task are simultaneously ready, as in [20]. Yet, following other (dynamic or static) priority schemes would not change the fact that the processor will be busy executing solely mandatory parts until $t = 5$ under any Mandatory-First approach. During the remaining idle interval [5,8], the best algorithm would have chosen to schedule O_1 completely (which brings most benefit to the system) for 1 time unit and O_2 for 2 time units. However, an optimal algorithm would produce the schedule depicted in Figure 4.2.

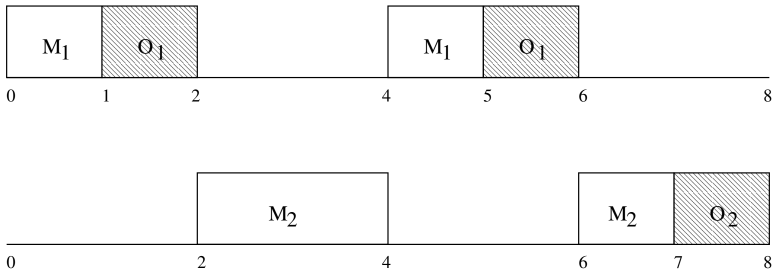


Figure 4.2: An optimal schedule

As it can be seen, the optimal strategy in this case consisted of delaying the execution of M_2 in order to be able to execute 'valuable' O_1 and we would still meet the deadlines of all mandatory parts. By doing so, we would succeed in executing two instances of O_1, in contrast to any Mandatory-First scheme which can execute only one instance of O_1. Remembering that $k_1 \gg k_2$, one can conclude that the reward accrued by the 'best' Mandatory-First scheme may only be around half of that accrued by the optimal one, for this example. Also, observe that in the optimal schedule, the optional execution times of a given task did not vary from instance to instance. In the next section, we prove that this pattern is not a mere coincidence. We further perform an analytical worst-case analysis of Mandatory-First algorithms in Section 4.2.

4.1 Optimality of Full-Utilization Policies for Periodic Reward-Based Scheduling

We first formalize the Periodic Reward-Based Scheduling problem. The objective is finding optimal $\{t_{ij}\}$ values to maximize the average reward. By substituting the average reward expression given by (3.2) in (3.3), we obtain our objective function:

$$\text{maximize} \quad \sum_{i=1}^{n} \frac{P_i}{P} \sum_{j=1}^{P/P_i} R_i(t_{ij})$$

The first constraint that we must enforce is that the total processor demand of mandatory and optional parts during the hyperperiod P, which is the least common multiple of all the task periods, may not exceed the available computing capacity, that is:

$$\sum_{i=1}^{n} \sum_{j=1}^{P/P_i} (m_i + t_{ij}) \leq P$$

Note that this constraint is necessary, but by no means sufficient for feasibility of the task set with $\{m_i\}$ and $\{t_{ij}\}$ values. Next, we observe that optimal t_{ij} values may not be less than zero, since negative service times do not have any physical interpretation. In addition, the service time of an optional instance of T_i does not need to exceed the upper bound o_i of reward function $R_i(t)$, since the reward accrued by T_i ceases to increase after $t_{ij} = o_i$. Hence, we obtain our second constraint set:

$$0 \leq t_{ij} \leq o_i \quad i = 1, \ldots, n \quad j = 1, \ldots, \frac{P}{P_i}$$

The constraint above allows us to readily substitute $f_i()$ for $R_i()$ in the objective function. Finally, we need to express the "full" feasibility constraint, including the requirement that mandatory parts complete in a timely manner at every invocation. Note that it is sufficient to have one feasible schedule with the involved $\{m_i\}$ and optimal $\{t_{ij}\}$ values:

$$\text{There exists a feasible schedule with } \{m_i\} \text{ and } \{t_{ij}\} \text{ values}$$

We express this constraint in English and not through formulas since the policy or algorithm producing this schedule including optimal t_{ij} assignments need not be specified at this point.

To re-capture all the constraints, the periodic reward-based scheduling problem, which we denote by REW-PER, is to find $\{t_{ij}\}$ values so as to:

$$\text{maximize} \quad \sum_{i=1}^{n} \frac{P_i}{P} \sum_{j=1}^{P/P_i} f_i(t_{ij}) \tag{4.1}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{P}{P_i} m_i + \sum_{i=1}^{n} \sum_{j=1}^{P/P_i} t_{ij} \leq P \tag{4.2}$$

$$0 \leq t_{ij} \leq o_i \quad i = 1, \ldots, n \quad j = 1, \ldots, \frac{P}{P_i} \tag{4.3}$$

$$\text{There exists a feasible schedule with } \{m_i\} \text{ and } \{t_{ij}\} \text{ values} \tag{4.4}$$

Before stating our main result, we underline that if $\sum_{i=1}^{n} \frac{P}{P_i} m_i > P$, it is not possible to schedule mandatory parts in a timely manner and the optimization problem has no solution. Note that this condition is equivalent to $\sum_{i=1}^{n} \frac{m_i}{P_i} > 1$, which indicates that the task set would be unschedulable, even if it consisted of only mandatory parts. Hence, hereafter, we assume that $\sum_{i=1}^{n} \frac{m_i}{P_i} \leq 1$.

Theorem 1 *Given an instance of Problem REW-PER with concave reward functions, there exists an optimal solution where the optional parts of a task T_i receive the* same *service time at every instance, that is, $t_{ij} = t_{ik}$ $1 \leq j < k \leq \frac{P}{P_i}$. Furthermore, any periodic hard-real time scheduling policy which can fully utilize the processor (EDF, LLF, RMS-h) can be used to obtain a feasible schedule with these assignments.*

Proof:

Our strategy to prove the theorem will be as follows. We will drop the feasibility condition (4.4) and obtain a new optimization problem whose feasible region strictly contains that of REW-PER. Specifically, we consider a new optimization problem, denoted by MAX-REW, where the objective function is again given by (4.1), but only the constraint sets (4.2) and (4.3) have to be satisfied. Note that the new problem MAX-REW does *not* a priori correspond to any scheduling problem, since the feasibility issue is not addressed. We then show that there exists an optimal solution of MAX-REW where $t_{ij} = t_{ik},\ 1 \leq j < k \leq \frac{P}{P_i}$. Then, we will return to REW-PER and demonstrate the existence of a feasible schedule (i.e., satisfiability of (4.4)) under these assignments. The reward associated with MAX-REW's optimal solution is always greater than or equal to that of REW-PER's optimal solution, for MAX-REW does *not* consider one of the REW-PER's constraints. This will imply that this specific optimal solution of the new problem MAX-REW is also an optimal solution of REW-PER.

Now, we show that there exists an optimal solution of MAX-REW where $t_{ij} = t_{ik},\ 1 \leq j < k \leq \frac{P}{P_i}$.

Claim 1 *Let $\{t_{ij}\}$ be an optimal solution to MAX-REW, $1 \leq i \leq n \quad 1 \leq j \leq \frac{P}{P_i} = q_i$. Then $\{t'_{ij}\}$ where $t'_{i1} = t'_{i2} = \ldots = t'_{iq_i} = t'_i = \frac{t_{i1} + t_{i2} + \ldots + t_{iq_i}}{q_i} \quad 1 \leq i \leq n \quad 1 \leq j \leq q_i$, is also an optimal solution to MAX-REW.*

- We first show that $\{t'_{ij}\}$ values satisfy the constraints (4.2) and (4.3), if $\{t_{ij}\}$ already satisfy them. Since $\sum_{j=1}^{q_i} t_{ij} = \sum_{j=1}^{q_i} t'_{ij} = q_i t'_i$ the constraint (4.2) is not violated by the transformation. Also, by assumption, $t_{ij} \leq o_i \, \forall j$, which implies $\max_j \{t_{ij}\} \leq o_i$. Since t'_i, which is arithmetic mean of $t_{i1}, t_{i2}, \ldots, t_{iq_i}$ is necessarily less than or equal to $\max_j \{t_{ij}\}$, the constraint set (4.3) is not violated either by the transformation.

- Furthermore, the total reward does not decrease by this transformation, since:

$$\sum_{j=1}^{q_i} f_i(t_{ij}) \leq q_i \; f_i(t'_i) \tag{4.5}$$

To see this, first observe that, if f_i is a linear function of the form $f_i(t) = k_i \cdot t$, then $\sum_{j=1}^{q_i} f_i(t_{ij}) = k_i \, (t_{i1} + t_{i2} + \ldots + t_{iq_i}) = k_i \, q_i \, t'_i$ and the inequality (4.5) is immediately established.

If f_i is general concave, we recall that:

$$\alpha f_i(x) + (1-\alpha) f_i(y) \leq f_i(\alpha x + [1-\alpha] y) \tag{4.6}$$

$\forall x, y$ and for every α such that $0 \leq \alpha \leq 1$. In this case, we prove the validity of (4.5) by induction. If $q_i = 1$, (4.5) holds trivially. So assume that it holds for $q_i = 1, 2, \ldots, m-1$. Induction assumption implies that:

$$\sum_{j=1}^{m-1} f_i(t_{ij}) \;\leq\; (m-1)\; f_i(\frac{t_{i1} + \ldots + t_{i(m-1)}}{m-1}) \tag{4.7}$$

Choosing $\alpha = \frac{m-1}{m}$, $x = \frac{t_{i1} + t_{i2} + \ldots + t_{i(m-1)}}{m-1}$, $y = t_{im}$ in (4.6), we can write:

$$\frac{m-1}{m}\, f_i(\frac{t_{i1} + \ldots + t_{i(m-1)}}{m-1}) + \frac{1}{m}\, f_i(t_{im}) \;\leq\; f_i(\frac{t_{i1} + \ldots + t_{im}}{m}) \tag{4.8}$$

Combining (4.7) and (4.8), we get:

$$\frac{1}{m} \sum_{j=1}^{m} f_i(t_{ij}) \;\leq\; f_i(\frac{t_{i1} + \ldots + t_{im}}{m}) \quad \forall m$$

establishing the validity of (4.5).

Using Claim 1, we can commit to finding *an* optimal solution of MAX-REW by setting $t_{i1} = t_{i2} = \ldots = t_{iq_i} = t_i \quad i = 1, \ldots, n$. In this case, $\sum_{j=1}^{P/P_i} f_i(t_{ij}) = \frac{P}{P_i}\, f_i(t_i)$ and $\sum_{j=1}^{P/P_i} t_{ij} = \frac{P}{P_i}\, t_i$. Hence, this version of MAX-REW can be re-written as:

$$\text{maximize} \quad \sum_{i=1}^{n} f_i(t_i) \tag{4.9}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{P}{P_i} t_i \leq P - \sum_{i=1}^{n} \frac{P}{P_i} m_i \tag{4.10}$$

$$0 \leq t_i \leq o_i \quad i = 1, \ldots, n \tag{4.11}$$

Finally, we prove that the optimal solution $t_1, t_2, \ldots, t_n$ of MAX-REW above, automatically satisfies the feasibility constraint (4.4) of our original problem REW-PER. Having equal optional service times for a given task greatly simplifies the verification of this constraint. Since $t_1, t_2, \ldots, t_n$ (by assumption) satisfy (4.10), we can write $\sum_{i=1}^{n} P \cdot \frac{m_i+t_i}{P_i} \leq P$, or equivalently, $\sum_{i=1}^{n} \frac{m_i+t_i}{P_i} \leq 1$.

This implies that any policy which can achieve 100% processor utilization in classical periodic scheduling theory (EDF, LLF, RMS-h) can be used to obtain a feasible schedule for tasks, which have now identical execution times $m_i + t_i$ at every instance. Hence, the "full feasibility" constraint (4.4) of REW-PER is satisfied. Furthermore, this schedule clearly maximizes the average reward since $\{t_i\}$ values maximize MAX-REW whose feasible region encompasses that of REW-PER.

□

Corollary 1 *Optimal t_i values for the Problem REW-PER can be found by solving the optimization problem given by (4.9), (4.10) and (4.11).*

We present the details of the solution of this concave optimization problem in Section 4.5. Before concluding this section, we underline that Theorem 1 will remain valid if one considers maximizing the *total* reward $\sum_{i=1}^{n} \sum_{j=1}^{P/P_i} R_i(t_{ij})$ during the hyperperiod P.

4.1.1 Extension to Multiprocessors

The existence proof of identical service times can be easily extended to homogeneous multiprocessors. The original formulation of REW-PER needs to be modified in order to reflect the multiprocessor environment. Note that the objective function (4.1), the lower and upper bound constraints (4.3) on optional service times and the full feasibility constraint (4.4) can be kept as they are. However, with k processors, the system can potentially have a task set whose total utilization is k instead of 1. Hence, we need to change the first constraint accordingly.

By doing so, we obtain the formulation of periodic reward-based scheduling problem for k processors, denoted as MULTI-REW:

$$\text{maximize} \quad \sum_{i=1}^{n} \frac{P_i}{P} \sum_{j=1}^{P/P_i} f_i(t_{ij}) \tag{4.12}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{P}{P_i} m_i + \sum_{i=1}^{n} \sum_{j=1}^{P/P_i} t_{ij} \leq k \cdot P \tag{4.13}$$

$$0 \leq t_{ij} \leq o_i \quad i = 1, \ldots, n \quad j = 1, \ldots, \frac{P}{P_i} \tag{4.14}$$

$$\text{There exists a feasible schedule on } k \text{ processors with } \{m_i\} \text{ and } \{t_{ij}\} \text{ values} \tag{4.15}$$

In the constraint (4.15), we make use of the the feasibility concept regarding multiprocessor schedules as defined in [6]; in particular, in a multiprocessor feasible schedule there should not be a time instant t where a given task runs simultaneously at multiple processors[11]. Following exactly the same line of reasoning depicted in Theorem 1, we can prove the following:

Theorem 2 *Given an instance of Problem MULTI-REW with concave reward functions, there exists an optimal solution where the optional parts of a task T_i receive the* same *service time at every instance, that is, $t_{ij} = t_{ik}$ $1 \leq j < k \leq \frac{P}{P_i}$. Furthermore, any scheduling policy which can achieve full utilization on k processors can be used to obtain a feasible schedule with these assignments.*

An example of such full-utilization policies for multiprocessors is provided by Bertossi et al. in [11]. We note that the PFair scheduling policy [5] which can also achieve full-utilization, assumes that all the periods are multiples of the slot length and hence it can not be used in this context.

Corollary 2 *Optimal t_i values for the Problem MULTI-REW can be found by solving the following optimization problem:*

$$\text{maximize} \quad \sum_{i=1}^{n} f_i(t_i) \tag{4.16}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{P}{P_i} t_i \leq k \cdot P - \sum_{i=1}^{n} \frac{P}{P_i} m_i \tag{4.17}$$

$$0 \leq t_i \leq o_i \quad i = 1, \ldots, n \tag{4.18}$$

Again, the details of the solution of this concave optimization problem are given in Section 4.5.

4.2 Evaluation and comparison with other approaches

We showed through the example in the beginning of Chapter 4 that the reward accrued by *any* Mandatory-First scheme [20] may only be approximately half of that of the optimal algorithm. We now prove that, under worst-case scenario, the ratio of the reward accrued by a Mandatory-First approach to the reward of the optimal algorithm may be arbitrarily close to zero. This is a consequence of Theorem 3, which allows us to construct a specific instance of the problem, by choosing an arbitrarily large integer r.

Theorem 3 *There is an instance of periodic reward-based scheduling problem where the ratio $\frac{\text{Reward of the best mandatory-first scheme}}{\text{Reward of the optimal scheme}} = \frac{2}{r}$ for any integer $r \geq 2$.*

Proof: Consider two tasks T_1 and T_2 such that $P_2/P_1 = r$, $f_1(t_1) = k_1 t_1$, $f_2(t_2) = k_2 t_2$ and $k_1/k_2 = r(r-1)$. Furthermore, let $m_2 = \frac{1}{2}(r\, o_2)$ and

$$P_1 = m_1 + o_1 + \frac{m_2}{r} = m_1 + \frac{m_2}{r-1}$$

which implies that $o_1 = \frac{m_2}{r\,(r-1)}$.

This setting suggests that during any period of T_1, a scheduler has the choice of executing (parts of) O_1 and/or M_2, in addition to M_1.

Note that under any Mandatory-First policy, during the hyperperiod $[0, P_2]$, the processor will be continuously busy executing mandatory parts until $t = P_2 - P_1 + m_1$. Furthermore, the best algorithm among Mandatory-First policies should use the remaining idle times by scheduling O_1 entirely (since $k_1 > k_2$) and $t_2 = \frac{m_2}{r} = \frac{o_2}{2}$ units of O_2. The resulting schedule is shown in Figure 4.3.

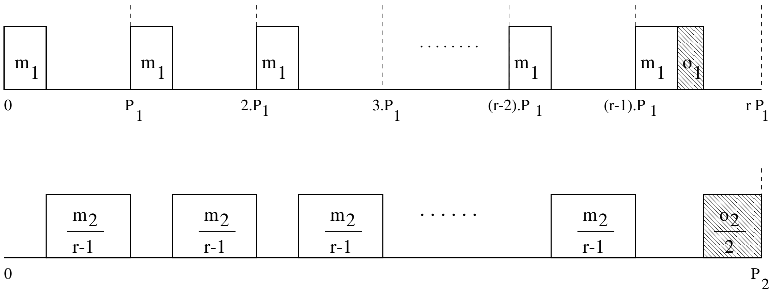


Figure 4.3: A schedule produced by Mandatory-First Algorithm

The average reward that the best mandatory-first algorithm (MFA) can accrue is therefore:

$$R_{MFA} = \frac{f_1(o_1)}{r} + f_2(t_2)$$

However, an optimal algorithm (shown in Figure 4.4) would choose delaying the execution of M_2 for o_1 units of time, at *every* period of T_1. By doing so, it would have the opportunity of accruing the reward of O_1 at every instance.

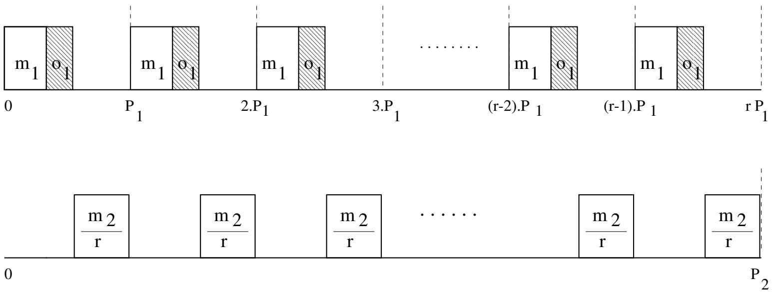


Figure 4.4: An optimal schedule

The total reward of the above schedule is:

$$R_{OPT} = \frac{r\,f_1(o_1)}{r} = f_1(o_1)$$

The ratio of rewards for the two policies turns out to be (for any $r \geq 2$)

$$\frac{R_{MFA}}{R_{OPT}} = \frac{1}{r} + \frac{f_2(t_2)}{f_1(o_1)} = \frac{1}{r} + \frac{1}{r(r-1)} \frac{m_2}{r} \frac{r(r-1)}{m_2} = \frac{2}{r}$$

which can be made as close as possible to 0 by appropriately choosing r.

□

Theorem 3 gives the worst-case performance ratio of Mandatory-First schemes. We also ran simulations using a synthetic task set to investigate the relative performance of Mandatory-First schemes proposed in [20] with different types of reward functions and different mandatory/optional workload ratios.

The Mandatory-First schemes differ by the policy according to which optional parts are scheduled when there is no mandatory part ready to execute. *Rate-Monotonic (RMSO)* and *Least-Utilization (LU)* schemes assign statically higher priorities to *optional parts* with smaller periods and least utilizations respectively. Among dynamic priority schemes are *Earliest-Deadline-First (EDFO)* and *Least-Laxity-First (LLFO)* which consider the deadline and laxity of optional parts when assigning priorities. *Least Attained Time (LAT)* aims at balancing execution times of optional parts that are ready, by dispatching the one that executed *least* so far. Finally, *Best Incremental Return (BIR)* is an on-line policy which chooses the optional task contributing most to the total reward, at a given *slot*. In other words, *at every slot* BIR selects the optional part O_{ij} such that the difference $f_i(t_{ij} + \Delta) - f_i(t_{ij})$ is the largest (here t_{ij} is the optional service time O_{ij} has already received and Δ is the minimum time slot that the scheduler assigns to any optional task). However, it is still a sub-optimal policy since it does not consider the laxity information. The authors indicate in [20] that BIR is too computationally complex to be actually implemented. However, since the total reward accrued by BIR is usually much higher than the other five policies, BIR is used as a yardstick for measuring the performance of other algorithms.

We have used a synthetic task set comprising 11 tasks whose total (mandatory + optional) utilization is 2.3. Individual task utilizations vary from 0.03 to 0.6. We implemented all the MFA schemes mentioned above and generated the schedule during the hyperperiod. Considering exponential, logarithmic and linear reward functions as separate cases, we measured the reward ratio of six Mandatory-First schemes with respect to our optimal algorithm. The tasks' characteristics (including reward functions) are given in the Table 4.1. In our experiments, we first set mandatory utilization to 0 (which corresponds to the case of all-optional workload), then increased it to 0.25, 0.4, 0.6, 0.8 and 0.91 subsequently.

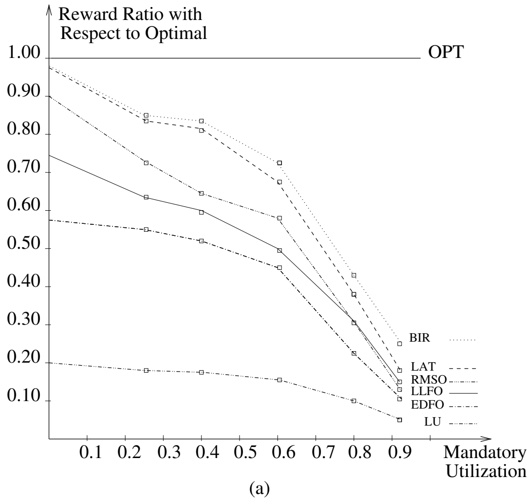
Figures 4.5 and 4.6 show the reward ratio of six Mandatory-First schemes with respect to our optimal algorithm as a function of mandatory utilization, for different types of reward functions. A common pattern appears: the optimal algorithm improves more dramatically with the increase in

Task id	P_i	$m_i + o_i$	$f_i^1(t)$	$f_i^2(t)$	$f_i^3(t)$
1	20	10	$15\,(1-e^{-t})$	$7\,\ln(20\,t+1)$	$5\,t$
2	30	18	$20\,(1-e^{-3t})$	$10\,\ln(50\,t+1)$	$7\,t$
3	40	5	$4\,(1-e^{-t})$	$2\,\ln(10\,t+1)$	$2\,t$
4	60	2	$10\,(1-e^{-0.5t})$	$5\,\ln(5\,t+1)$	$4\,t$
5	60	2	$10\,(1-e^{-0.2t})$	$5\,\ln(25\,t+1)$	$4\,t$
6	80	12	$5\,(1-e^{-t})$	$3\,\ln(30\,t+1)$	$2\,t$
7	90	18	$17\,(1-e^{-t})$	$8\,\ln(8\,t+1)$	$6\,t$
8	120	15	$8\,(1-e^{-t})$	$4\,\ln(6\,t+1)$	$3\,t$
9	240	28	$8\,(1-e^{-t})$	$4\,\ln(9\,t+1)$	$3\,t$
10	270	60	$12\,(1-e^{-0.5t})$	$6\,\ln(12\,t+1)$	$5\,t$
11	2160	300	$5\,(1-e^{-t})$	$3\,\ln(15\,t+1)$	$2\,t$

Table 4.1: Three task sets used in the evaluation

mandatory utilization. The other schemes miss the opportunities of executing "valuable" optional parts while constantly favoring mandatory parts. The reward loss becomes striking as the mandatory workload increases. Figures 4.5.a and 4.5.b show the reward ratio for the case of exponential and logarithmic reward functions, respectively. The curves for these strictly concave reward functions are fairly similar: BIR performs best among Mandatory-First schemes, and its performance decreases as the mandatory utilization increases; for instance the ratio falls below 0.75 (for both curves) when mandatory utilization is 0.6. Other algorithms which are more amenable to practical implementations (in terms of runtime overhead) than BIR perform even worse. However, it is worth noting that the performance of LAT is close to that of BIR. This is to be expected, since task sets with strictly concave reward functions usually benefit from "balanced" optional service times. Finally, at high mandatory utilizations, the CPU time that can be used for optional executions is naturally diminished, which results in a convergence of all MFA schemes for strictly concave reward functions: the total reward is only slightly affected by the choice of optional task to dispatch.

Figure 4.6 shows the reward ratio for linear reward functions. Although the reward ratio of Mandatory-First schemes again decreases with the mandatory utilization, the decrease is less dramatic than in the case of concave functions (see above). However, note that the ratio is typically less than 0.5 for the five practical schemes. It is interesting to observe that the reward of (impractical) BIR method now remains comparable to that of optimal, even in the higher mandatory utilizations: the difference is less than 15%. In our opinion, the main reason for this behavior change lies on the fact that, for a given task, the reward of optional execution slots in different instances does not make a difference in the linear case. In contrast, not executing the "valuable" *first slot(s)* of a given instance creates a tremendous effect for nonlinear concave functions. The improvement of the optimal algorithm would be larger for a larger range of k_i values (where k_i is the coefficient of



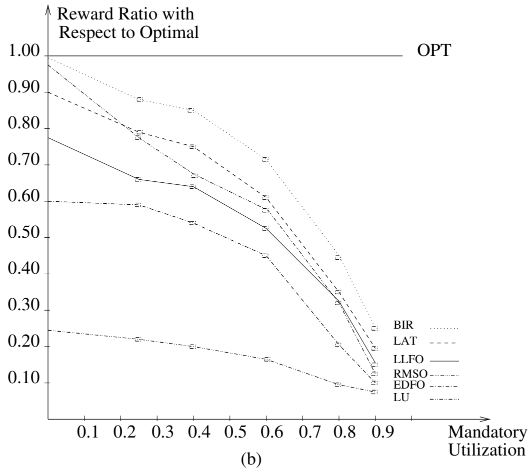


Figure 4.5: The Reward Ratio of Mandatory-First schemes: strictly concave reward functions (a) Exponential (b) Logarithmic functions

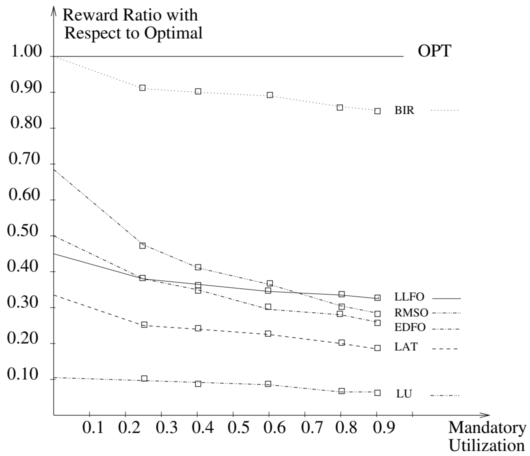


Figure 4.6: The Reward Ratio of Mandatory-First schemes: linear reward functions

id	m_i	o_i	P_i	$F_i(t)$
1	3	3	8	t
2	7	5	16	t

Table 4.2: 2-task system

the linear reward function). We note that the worst-case performance of BIR may be arbitrarily bad with respect to the optimal one for linear functions, as Theorem 3 suggests.

4.3 Limits on the optimality of identical service times

We underline that Theorem 1 was the key to eliminate a potentially exponential number of unknowns (t_{ij} values) and thereby to obtain an optimization problem of n variables. One is naturally tempted to ask whether the optimality of identical service times is *still* preserved if some fundamental assumptions of the model are relaxed. Unfortunately, attempts to reach further optimality results for extended / different models remain inconclusive as the following propositions suggest.

Proposition 1 *The optimality of identical service times no longer holds if the* Deadline = Period *assumption is relaxed.*

Proof:

We will prove the statement by providing a counter-example. Assume that we allow the deadline of a task to be less than its period. Let us consider the 2-task system given in Table 4.2.

Further, assume that the deadline of T_2 is $d_2 = 10 < P_2$, while the deadline of T_1 coincides with its period, (i.e., $d_1 = 8$). Note that the tight deadline of T_2 makes it impossible to schedule any optional part until $t = 13$, after which we are able to schedule O_1 for 3 units. This optimal schedule is shown in Figure 4.7. On the other hand, if one commits to identical service times per instance, it is clear that we may not schedule *any* optional part, since we could not execute O_1 in the first instance of T_1 (Figure 4.8).

□

Next, suppose that the deadlines are equal to the periods, but we have to adopt a static priority scheduling policy. It was already mentioned in Section 4.1, that if the periods are harmonic, then we can use RMS without compromising optimality. But, in the general case where the periods are not necessarily harmonic, this is not true *even if we are investigating the 'best' schedule within the context of a given static priority assignment.*

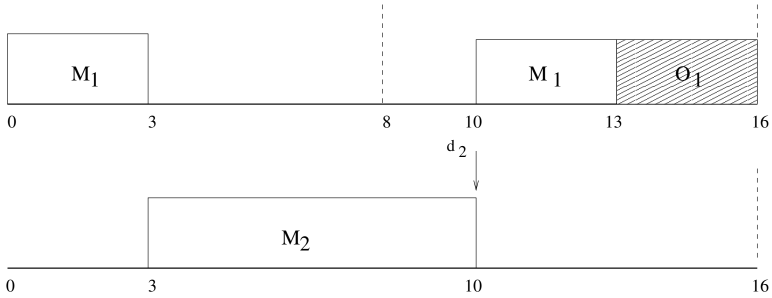


Figure 4.7: The optimal schedule

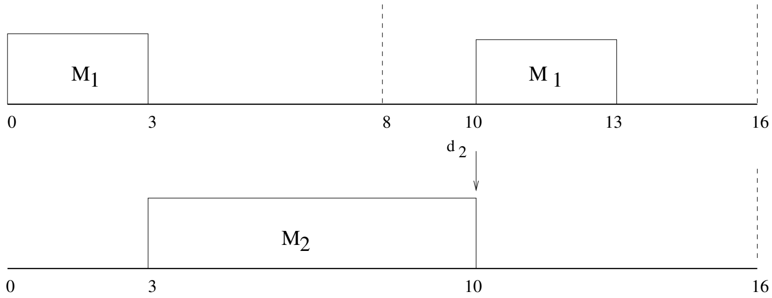


Figure 4.8: Best schedule with identical service times

Proposition 2 *In the general case, the optimality of identical service times no longer holds if we commit to a static priority assignment.*

Proof:

Again, to prove the statement with a counterexample, consider the task set whose parameters are presented in Table 4.3:

As we have only two tasks, we will consider the cases where T_1 or T_2 has higher priority and show that in every case, the reward of the optimal schedule differs from the one obtained with identical service times per instance assumption.

Case 1 - T_1 has higher priority: It is easy to see that we can construct a schedule which *fully* utilizes the timeline during the interval $[0, lcm(6, 8) = 24]$ (Figure 4.9a). This schedule is also immediately optimal since the reward function is linear (observe that $o_2 = 0$ and hence we do not receive any reward for executing O_2). But, we remark that we can not execute O_1 for more than 1

id	m_i	o_i	P_i	$F_i(t)$
1	2	4	6	t
2	3	0	8	t

Table 4.3: Task set for fixed priority scheduling

unit on its first instance in *any* feasible schedule — without violating the deadline of T_2. Therefore, we would have ended up with a lower reward after executing 1 unit of O_1 at each instance, if we had committed to identical service times (Figure 4.9b).

Case 2 - T_2 has higher priority: The optimality is still compromised if T_2 has higher priority. While the optimal schedule (Figure 4.10a) fully utilizes the timeline, the best schedule with $t_{1i} = 1$ $(i = 1, \ldots, 4)$ (Figure 4.10b) remains sub-optimal.

□

We remark that the Proposition 2 has also implications for Q-RAM model [85], since it points to the impossibility of achieving optimality with identical service times by using a static priority assignment. However, it is interesting to explore the performance of the (best) schedules produced by committing to identical service times for the fixed priority policies. Let us define the following:

- $FIX - ID(\mathbf{T})$: the reward accrued by a rate monotonic scheduler when each (optional) task receives the same service time at every instance; these service times can be (efficiently) computed by committing to the processor utilization bound given by Liu and Layland [67] and then solving (appropriately modified) REW-PER problem.
- $OPT(\mathbf{T})$: the reward accrued by an optimal rate monotonic reward-based scheduler.

We conjecture that the ratio $\frac{FIX-ID(\mathbf{T})}{OPT(\mathbf{T})}$ can not be smaller than the (Liu and Layland) utilization bound $n\ (2^{1/n}-1)$ for a task set **T** containg n tasks, provided that all the reward functions are concave.

The final proposition in this section illustrates that the optimality of identical service times is also sensitive to the concavity of reward functions.

Proposition 3 *The optimality of identical service times no longer holds if the concavity assumption about the reward functions is relaxed.*

Proof:

Consider two harmonic tasks without mandatory parts whose parameters are given in Table 4.4. Note that t_2 should be assigned its maximum possible value (i.e., the upper bound 6), since the marginal return of F_2 is larger than F_1 everywhere. An optimal schedule maximizing the average reward for these two tasks is depicted in Figure 4.11.

The sum of the average rewards in the optimal schedule is $2.6^2 + \frac{2^2+8^2}{2} = 106$. However, if we commit ourselves to the equal service times per instances, we can find no better schedule than the one shown in Figure 4.12, whose reward is only $5^2 + 2 \cdot 6^2 = 97$.

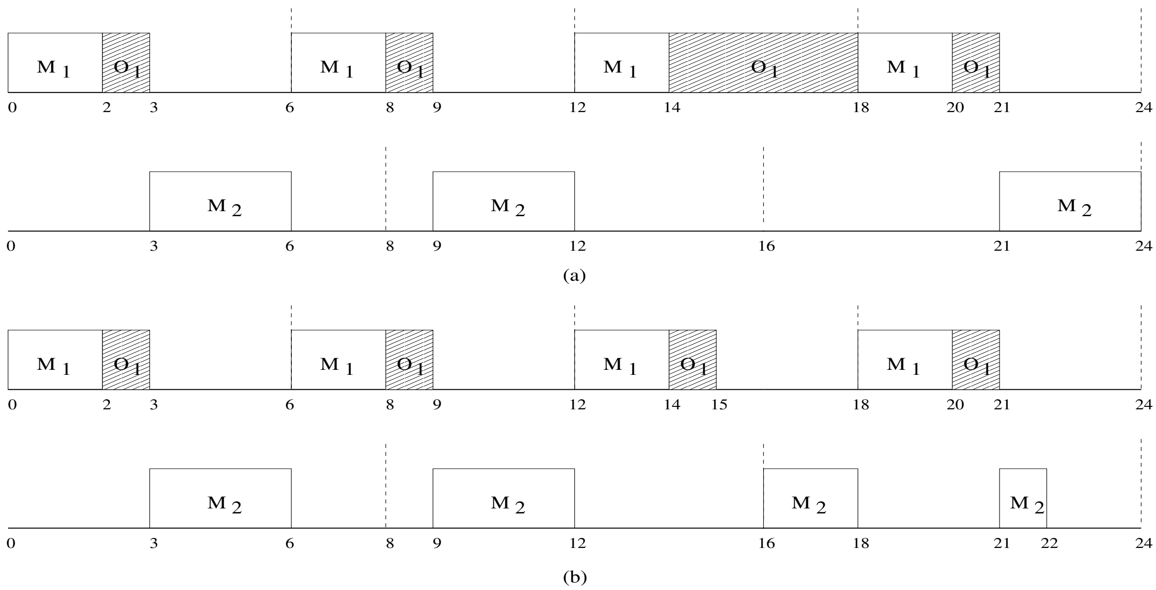


Figure 4.9: T_1 has the higher priority (a) The optimal schedule (b) The best schedule with identical service times

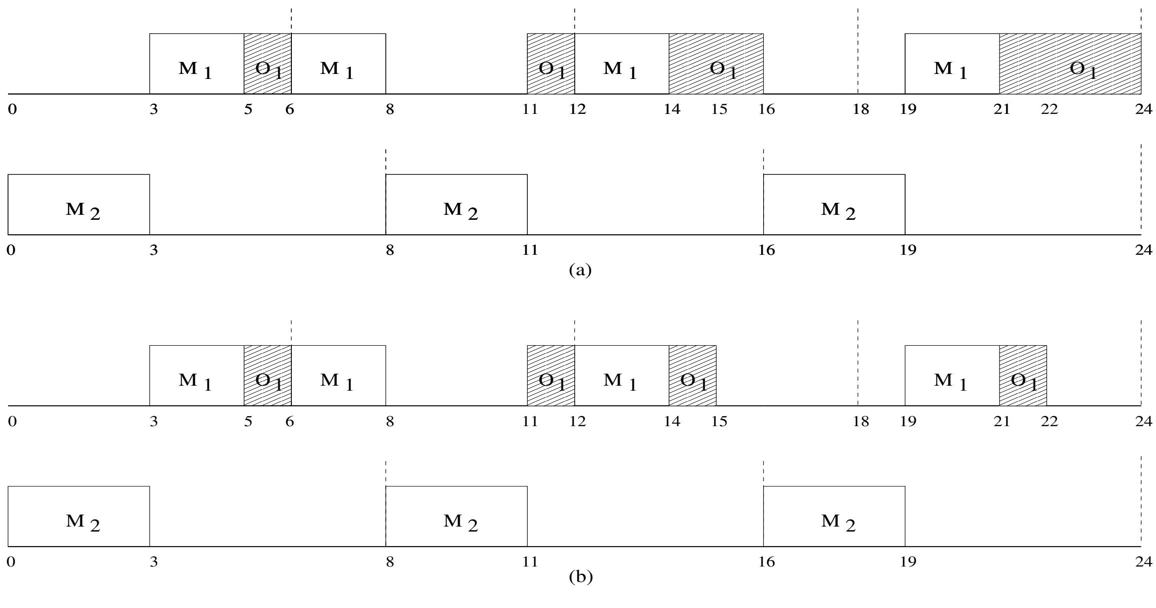


Figure 4.10: T_2 has the higher priority (a) The optimal schedule (b) The best schedule with identical service times

id	m_i	o_i	P_i	$F_i(t)$
1	0	8	8	t^2
2	0	6	16	$2\,t^2$

Table 4.4: A task set with convex reward functions

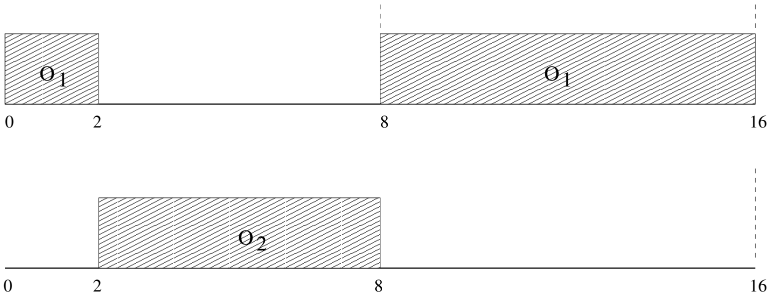


Figure 4.11: An optimal schedule

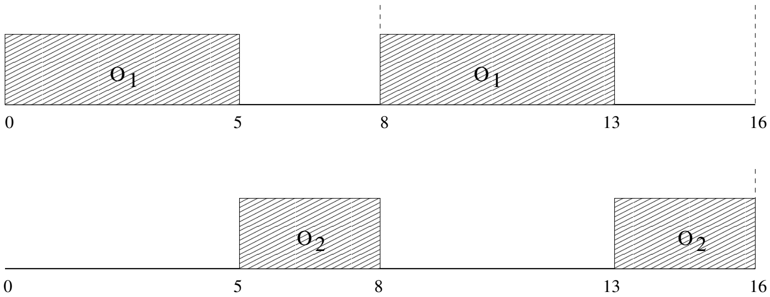


Figure 4.12: Best schedule with identical service times

It is not difficult to construct a similar example for the tasks with 0-1 constraints as well, which implies that even *the number* of variables to deal with (the t_{ij}'s) may be prohibitively large in these problems.

□

4.4 Intractability of Convex Reward-Based Scheduling Problem

As we mentioned before, maximizing the total (or average) reward with 0/1 constraints case had already been proven to be NP-Complete in [66]. Similarly, in Section 4.3 we showed that, if the reward functions are convex, the optimality of identical service times is not preserved. In this section, we show that, in fact convex reward functions result in an NP-Hard problem, even with identical periods[1].

We now show how to transform the SUBSET-SUM problem, which is known to be NP-Complete, to REW-PER with convex reward functions.

SUBSET-SUM: Given a set $S = \{s_1, s_2, \ldots, s_n\}$ of positive integers and the integer M, is there a set $S_A \subseteq S$ such that $\sum\limits_{s_i \in S_A} s_i = M$?

[1] We underline that, as of today, most of the applications we investigated have non-increasing marginal returns, hence they can be represented best by concave reward functions. However, we include convex reward functions' case and show its intractability in order to complete the theoretical investigation of the periodic reward based scheduling problem.

We construct the corresponding REW-PER instance as follows. Let $W = \sum_{i=1}^{n} s_i$. Now consider a set of n periodic tasks with the same period M and mandatory parts $m_i = 0 \ \forall i$. The reward function associated with T_i is given by:

$$R_i(t_i) = \begin{cases} f_i(t_i) & if \quad 0 \leq t \leq o_i = s_i \\ f_i(o_i) & if \quad t > o_i = s_i \end{cases}$$

where $f_i(t_i) = t_i^2 + (W - s_i)t_i$ is a strictly convex and increasing function on nonnegative real numbers.

Notice that $f_i(t_i)$ can be re-written as $t_i(t_i - s_i) + W t_i$. Also we underline that having the same periods for all tasks implies that REW-PER can be formulated as:

$$\text{maximize} \qquad \sum_{i=1}^{n} t_i \, (t_i - s_i) + W \sum_{i=1}^{n} t_i \tag{4.19}$$

$$\text{subject to} \qquad \sum_{i=1}^{n} t_i \leq M \tag{4.20}$$

$$0 \leq t_i \leq s_i \tag{4.21}$$

Let us denote by $MaxRew$ the total reward of the optimal schedule. Observe that for $0 < t_i < s_i$, the quantity $t_i \, (t_i - s_i) < 0$. Otherwise, at either of the boundary values 0 or s_i, $t_i \, (t_i - s_i) = 0$. Hence, $MaxRew \leq WM$.

Now, consider the question: "Is $MaxRew$ equal to WM?". Clearly, this question can be answered quickly if there is a polynomial-time algorithm for REW-PER where reward functions are allowed to be convex. Furthermore, the answer can be positive only when $\sum_{i=1}^{n} t_i = M$ *and* each t_i is equal to either 0 or s_i. Therefore, $MaxRew$ equal to WM, if and only if there is a set $S_A \subseteq S$ such that $\sum_{s_i \in S_A} s_i = M$, which implies that REW-PER with convex reward functions is NP-Hard.

4.5 Solution of Periodic Concave Reward-Based Scheduling Problem

Corollaries 1 and 2 reveal that the two optimization problems whose solutions provide optimal service times for uniprocessor and multiprocessor systems share a *common form*:

$$\text{maximize} \qquad \sum_{i=1}^{n} f_i(t_i)$$

$$\text{subject to} \qquad \sum_{i=1}^{n} b_i \, t_i \leq d$$

$$t_i \leq o_i \quad i = 1, 2, \ldots n$$

$$0 \leq t_i \quad i = 1, 2, \ldots n$$

where d (the 'slack' available for optional executions) and $b_1, b_2, \ldots, b_n$ are positive rational numbers. In this section, we present polynomial-time solutions for this problem, where each f_i is a non-decreasing, concave and differentiable[2] function.

First note that, if the available slack is large enough to accommodate every optional part entirely (i.e., if $\sum_{i=1}^{n} b_i \; o_i \leq d$), then the choice $t_i = o_i \; \forall \; i$ clearly maximizes the objective function due to the non-decreasing nature of reward functions.

Otherwise, the slack d should be used in its entirety since the total reward never decreases by doing so (again due to the non-decreasing nature of the reward functions). In this case, we obtain a concave optimization problem with lower and upper bounds, denoted by OPT-LU. An instance of OPT-LU is specified by the set of non-decreasing concave functions $\mathbf{F} = \{f_1, \ldots, f_n\}$, the set of upper bounds $\mathbf{O} = \{o_1, \ldots, o_n\}$ and the available slack d. The aim is to:

$$\text{maximize} \quad \sum_{i=1}^{n} f_i(t_i) \tag{4.22}$$

$$\text{subject to} \quad \sum_{i=1}^{n} b_i \; t_i = d \tag{4.23}$$

$$t_i \leq o_i \quad i = 1, 2, ...n \tag{4.24}$$

$$0 \leq t_i \quad i = 1, 2, ...n \tag{4.25}$$

where $0 < d < \sum_{i=1}^{n} b_i \cdot o_i$.

Special Case of Linear functions: We address separately the case when **F** comprises solely linear functions, since the time complexity can be considerably reduced by using this information. Note that for a function $f_i(t_i) = k_i \cdot t_i$, if we increase t_i by Δ then total reward increases by $k_i \, \Delta$. However by doing so, we make use of $b_i \, \Delta$ units of slack (d is reduced by $b_i \, \Delta$ due to (4.23)). Hence, the "marginal return" of task T_i per slack unit is $w_i = \frac{k_i}{b_i}$. Now consider another function $f_j(t_j) = k_j \cdot t_j$. Clearly, $w_j = \frac{k_j}{b_j}$. If $w_j > w_i$, then task T_j *should be always favored* with respect to T_i since the marginal return of f_j is strictly greater than f_i *everywhere*. Repeating the argument for every pair of tasks, we can obtain the following optimal strategy.

We first order the functions according to their marginal returns $w_i = \frac{k_i}{b_i} \; i = 1, 2, \ldots, n$. Let f_1 be the function with the largest marginal return, f_2 the second and so on (ties are broken arbitrarily). If $b_1 \, o_1 \geq d$, then we set $t_1 = d/b_1$ and we are done (we are using the entire slack for T_1). If $b_1 \, o_1 < d$ then we set $t_1 = o_1$ and d is reduced accordingly ($d = d - b_1 \, o_1$). Next, we repeat the same step for the next "most valuable" function, f_2. After at most n iterations, the slack d is completely consumed. We note that this solution for linear functions is analogous to the one presented in [85]. The dominant factor in the time complexity comes from the initial

[2] In the auxiliary optimization problems which will be introduced shortly, the differentiability assumption holds as well.

sorting procedure, hence in the special case of all-linear functions, OPT-LU can be solved in time $O(n \log n)$.

When **F** contains nonlinear functions, the procedure becomes more involved. In the next two subsections, we introduce two auxiliary optimization problems, namely Problem OPT (which considers only the equality constraint) and Problem OPT-L (which considers only the equality and lower bound constraints), which will be used to solve OPT-LU.

4.5.1 Problem OPT: Case of the Equality Constraint

An instance of the problem OPT is characterized by the set $\mathbf{F} = \{f_1, \ldots, f_n\}$ of non-decreasing concave functions and the slack d:

$$\begin{aligned} \text{maximize} \quad & \sum_{i=1}^{n} f_i(t_i) \\ \text{subject to} \quad & \sum_{i=1}^{n} b_i\, t_i = d \end{aligned}$$

As it can be seen, OPT does not take into account the lower and upper bound constraints of Problem OPT-LU. The algorithm which returns the solution of Problem OPT, is denoted by "Algorithm OPT".

When **F** is composed solely of *non-linear* reward functions, the application of Lagrange multipliers technique [69] to the Problem OPT, yields:

$$\frac{1}{b_i} f_i'(t_i) - \lambda = 0 \qquad i = 1, 2, \ldots, n \tag{4.26}$$

where λ is the common Lagrange multiplier and $f_i'(t_i)$ is the derivative of the reward function f_i. The quantity $\frac{1}{b_i} f_i'(t_i)$ in (4.26) actually represents the *marginal return* contributed by T_i to the total reward, which we will denote as $w_i(t_i)$. Observe that since $f_i(t_i)$ is non-decreasing and concave by assumption, both $w_i(t_i)$ and $f_i'(t_i)$ are non-increasing and positive valued. Equation (4.26) implies that the marginal returns $w_i(t_i) = \frac{1}{b_i} f_i'(t_i)$ *should* be equal for all reward functions in the optimal solution $\{t_1, \ldots, t_n\}$. Considering that the equality $\sum_{i=1}^{n} b_i\, t_i = d$ should also hold, one can obtain closed formulas in most of the cases which occur in practice. The closed formulas presented below are obtained by this method.

- For *logarithmic* reward functions of the form $f_i(t_i) = \ln(k_i \cdot t_i + c_i)$,
$$t_1 = \frac{d + \sum_{j=1}^{n} \frac{c_j}{k_j} - \frac{c_1}{k_1\, b_1} \sum_{j=1}^{n} b_j}{\frac{1}{b_1} \sum_{j=1}^{n} b_j} \text{ and } t_j = b_1\, t_1 + \frac{c_1}{k_1\, b_1} - \frac{c_j}{k_j\, b_j} \quad \forall j \;\; 1 < j \le n.$$
- For *exponential* reward functions of the form $f_i(t_i) = c_i(1 - e^{-k_i t_i})$,
$$t_1 = \frac{d - \sum_{j=1}^{n} \frac{1}{k_j}[\ln(\frac{c_j\, b_1\, k_j}{c_1\, b_j\, k_1})]}{\sum_{j=1}^{n} \frac{k_1}{k_j}} \text{ and } t_j = \frac{1}{k_j}[k_1\, t_1 + \ln(\frac{c_j b_1 k_j}{c_1 b_j k_1})] \quad \forall j \;\; 1 < j \le n.$$

- For "k^{th} root" reward functions of the form $f_i(t_i) = c_i\ t_i^{1/k}\ (k > 1)$,
$t_1 = \frac{d}{\sum\limits_{j=1}^{n} (\frac{b_j c_1}{b_1 c_j})^{\frac{1}{k-1}}}$ and $t_j = t_1\,(\frac{b_j c_1}{b_1 c_j})^{\frac{1}{k-1}} \quad \forall j\ \ 1 < j \leq n.$

When it is not possible to find a closed formula, following exactly the approach presented in [22, 23, 50], we solve for λ in the equation $\sum_{i=1}^{n} b_i\ h_i(\lambda) = d$, where $h_i(k)$ is the inverse function of $\frac{1}{b_i} f_i'(t_i) = w_i(t_i)$ (we assume the existence of the derivative's inverse function whenever f_i is nonlinear, complying with [22, 23, 50]). Once λ is determined, $t_i = h_i(\lambda),\ i = 1, \ldots, n$ is the optimal solution.

We now examine the case where **F** is a *mix* of linear and nonlinear functions. Consider two linear functions $f_i(t) = k_i \cdot t$ and $f_j(t) = k_j \cdot t$. The marginal return of $f_i(t_i)$ is $w_i(t_i) = \frac{k_i}{b_i} = w_i$ and that of f_j is $w_j(t_j) = \frac{k_j}{b_j} = w_j$. Now, if $w_j > w_i$ then the service time t_i should be definitely zero, since marginal return of f_i is strictly less than f_j everywhere. After this elimination process, if there are $p > 1$ linear functions with the same (largest) marginal return w_{max} then we will consider them as a single linear function in the procedure below and evenly divide the returned service time t_{max} among t_j values corresponding to these p functions.

Hence, without loss of generality, we assume that $f_n(t) = k_n \cdot t$ is the only linear function in **F**. Its marginal return is $w_n(t_n) = \frac{k_n}{b_n} = w_{max}$. We first compute the optimal distribution of slack d among tasks with nonlinear reward functions $f_1, \ldots, f_{n-1}$. By the Lagrange multipliers technique, $w_i(t_i) - \lambda = 0$ and thus $w_1(t_1^*) = w_2(t_2^*) = \ldots = w_{n-1}(t_{n-1}^*) = \lambda$ at the optimal solution $t_1^*, t_2^*, \ldots, t_{n-1}^*$.

Now we distinguish two cases:

- $\lambda >= w_{max}$. In this case, $t_1^*, t_2^*, \ldots, t_{n-1}^*$ and $t_n = 0$ is the optimal solution to OPT. To see this, first remember that all the reward functions are concave and non-decreasing, hence $w_i(t_i^* - \epsilon) \geq w_i(t_i^*) \geq w_n(\epsilon) = w_{max}\ i = 1, \ldots, n-1$ for all $\epsilon >= 0$. This implies that transferring some service time from another task T_i to T_n would mean favoring the task which has the smaller marginal reward rate and would not be optimal.

- $\lambda < w_{max}$. In this case, reserving the slack d solely to tasks with nonlinear reward functions means violating the best marginal rate principle and hence is not optimal. Therefore, we should decrease service time t_i until $w_i(t_i)$ reaches the level of w_{max} for $i = 1, 2, \ldots, n-1$, *but not beyond*. Solving $h_i(w_{max}) = t_i$ for $i = 1, 2, \ldots, n-1$ and assigning any remaining slack $\frac{d - \sum_{i=1}^{n-1} t_i}{b_n}$ to t_n (the service time of unique task with linear reward function) clearly satisfies the best marginal rate principle and achieves optimality.

4.5.2 Problem OPT-L: Case of Lower Bounds

In this section we present a solution for problem OPT-L and we improve on this solution in Section 4.5.2.1. An instance of Problem OPT-L is characterized by the set **F**=$\{f_1, f_2, .., f_n\}$ of non-decreasing concave reward functions, and the available slack d:

$$\text{maximize} \quad \sum_{i=1}^{n} f_i(t_i) \tag{4.27}$$

$$\text{subject to} \quad \sum_{i=1}^{n} b_i\, t_i = d \tag{4.28}$$

$$0 \leq t_i \quad i = 1, 2, ...n \tag{4.29}$$

To solve OPT-L, we first evaluate the solution set S_{OPT} to the corresponding problem OPT and check whether all inequality constraints are automatically satisfied. If this is the case, the solution set S_{OPT-L} of Problem OPT-L is clearly the solution S_{OPT}. Otherwise, we will construct S_{OPT-L} iteratively as described below.

A well-known result of nonlinear optimization theory states that the solution S_{OPT-L} of Problem OPT-L should satisfy so-called *Kuhn-Tucker conditions* [69, 78]. Furthermore, Kuhn-Tucker conditions are also sufficient in the case of concave reward functions [69, 78]. For Problem OPT-L, Kuhn-Tucker conditions comprise Equations (4.28), (4.29) and:

$$-\frac{1}{b_i} f_i'(t_i) + \lambda - \frac{\mu_i}{b_i} = 0 \quad i = 1, 2 \ldots n \tag{4.30}$$

$$-\mu_i\, t_i = 0 \quad i = 1, 2 \ldots n \tag{4.31}$$

$$\mu_i \geq 0 \quad i = 1, 2 \ldots n \tag{4.32}$$

where $\lambda, \mu_1, \mu_2, \ldots, \mu_n$ are Lagrange multipliers. The necessary and sufficient character of Kuhn-Tucker conditions indicates that any $2n+1$ tuple $(t_1, t_2, \ldots, t_n, \mu_1, \mu_2, \ldots, \mu_n, \lambda)$ which satisfies conditions (4.28) through (4.32) provides optimal t_i values for OPT-L.

One method of solving the optimization problem OPT-L is to find a solution to the $2n+1$ equations (4.28), (4.30) and (4.31) which satisfies constraint sets (4.29) and (4.32). Iteratively solving the $2n+1$ nonlinear equations is a complex process which is not guaranteed to converge. In this work, we follow a different approach. Namely, we use the Kuhn-Tucker conditions (4.30), (4.31) and (4.32) to prove some useful properties of the optimal solution. Our method is based on carefully using the properties that we derive in order to *refine* the solution of the optimization problem OPT.

Claim 2 *If S_{OPT} violates some inequality constraints given by (4.29) then $\exists i\ \mu_i > 0$.*

Proof: Assume to the contrary that $\forall i\ \mu_i = 0$. In this case Kuhn-Tucker conditions reduce to the equality constraint (4.28), the set of inequality constraints (4.29) plus the Lagrangian condition given in (4.26). On the other hand, S_{OPT}, which is the solution of OPT, should satisfy (4.28) and the Lagrangian condition (4.26). In other words, solving OPT is always equivalent to solving a set of non-linear equations which are identical to Kuhn-Tucker conditions of OPT-L *except inequality constraints, by setting* $\forall i \; \mu_i = 0$. Hence, if there were a solution to OPT-L where $\forall i \; \mu_i = 0$, then that solution would be returned by the algorithm solving OPT *and* would not violate inequality constraints. However, given that the solution S_{OPT} failed to satisfy all the inequality constraints we reach a contradiction. Therefore, there exists at least one Lagrange multiplier μ_i which is strictly greater than 0.

□

Claim 3 $\exists j \; \mu_j = 0$.

Proof: For the sake of contradiction, assume that $\forall j \; \mu_j > 0$. In this case Equation (4.31) enforces that $\forall i \; t_i = 0$. If this were true, $\sum_{i=1}^{n} b_i \; t_i$ will be equal to 0, leaving the slack d totally unutilized. In this case, this clearly would not be the optimal solution due to the non-decreasing nature of reward functions.

□

In the rest of this section, we use the expression ”the set of functions” instead of ”the set of **indices** of functions” unless confusion arises. Let:

$$\Pi = \{x | \frac{1}{b_x} f_x'(0) \leq \frac{1}{b_i} f_i'(0) \; \forall i\} \tag{4.33}$$

Remember that $\frac{1}{b_x} f_x'(t_x)$ is the marginal return associated with $f_x(t_x)$ and is denoted by $w_x(t_x)$. Informally, Π contains the functions $f_x \in \mathbf{F}$ with the smallest marginal returns at the lower bound 0, $w_x(0)$.

Lemma 1 *If S_{OPT} violates some inequality constraints then, in S_{OPT-L}, $t_m = 0 \; \forall m \in \Pi$.*

Proof: Assume that $\exists m \in \Pi$ such that $t_m > 0$. In this case, Equation (4.31) implies that the corresponding $\mu_m = 0$. By Claim 2 we know that $\exists j$ such that $\mu_j > 0$. By Equation (4.31), $t_j = 0$. Using Equation (4.30), we can write

$$\frac{1}{b_m} f_m'(t_m) - \frac{1}{b_j} f_j'(0) = \frac{\mu_j}{b_j} > 0$$

Since $t_m > 0$, the concavity property of f_m suggests that $\frac{1}{b_m} f_m'(t_m) \leq \frac{1}{b_m} f_m'(0)$. But in this case we obtain $\frac{1}{b_m} f_m'(0) - \frac{1}{b_j} f_j'(0) \geq \frac{\mu_j}{b_j} > 0$, contradicting the assumption that $m \in \Pi$. Hence $\mu_m > 0$ and by Equation (4.32) $t_m = 0$.

Algorithm OPT-L(F,d)

1 Evaluate the solution S_{OPT} of the optimization problem (without inequality constraints)

2 If all the inequality constraints are satisfied then S_{OPT-L}=S_{OPT}; exit

3 Compute $\mathbf{\Pi}$ from equation (4.33)

4 Set $t_m = 0 \, \forall m \, \in \mathbf{\Pi}$

5 Set $\mathbf{F} = \, \mathbf{F} - \mathbf{\Pi}$

6 goto Step 1

Figure 4.13: Algorithm to solve Problem OPT-L

□

In view of Lemma 1, in Figure 4.13 we present the algorithm to solve Problem OPT-L.

Complexity: The time complexity $C_{OPT}(n)$ of Algorithm OPT is $O(n)$: if the mentioned closed formulas apply, then the complexity is clearly linear; otherwise the unique unknown λ can be solved in linear time under concavity assumptions, as indicated in [22, 23, 50]. Lemma 1 immediately implies the existence of an algorithm which sets $t_m = 0 \; \forall m \in \Pi$, and then re-invokes Algorithm OPT for the remaining tasks and slack (in case that some inequality constraints are violated by S_{OPT}). Since the number of invocations is bounded by n, the complexity of the algorithm which solves OPT-LU is $O(n^2)$.

4.5.2.1 Faster Solution for Problem OPT-L

In this section, we present a faster algorithm of time complexity $O(n \cdot \log n)$, to solve OPT-L. We will make use of the new (faster) algorithm in the final solution of OPT-LU.

Consider Algorithm OPT-L depicted in Fig. 4.13. Let F_k be the set of functions passed to OPT during the k^{th} iteration of Algorithm OPT-L, and Π_k be the set of functions with minimum marginal returns at the lower bounds (minimum $w_i(0)$ values) during the k^{th} iteration (formally, $\Pi_k = \{x | \frac{1}{b_x} f'_x(0) \leq \frac{1}{b_y} f'_y(0) \; \forall y | y \in F_k\}$). Let also $n^* \leq n$ be the number of *distinct* $w_i(0)$ values among functions in **F**, and $m^* \leq n^*$ be the iteration number where Algorithm OPT returns a solution set which satisfies the constraint set given by (4.29) for the remaining t_i values. Note that the elements of $\Pi_i \quad i = 1, \ldots, n^*$ can be produced in $O(n \cdot \log n)$ time during a pre-processing phase. Clearly, Algorithm OPT-L sequentially sets $t_i = 0 \;\; \forall i \;\; \in \Pi_1 \cup \Pi_2 \ldots \cup \Pi_{m^*-1}$ until Algorithm OPT returns a solution which does not violate the constraint set for the remaining unknowns at the $(m^*)^{th}$ iteration.

A tempting idea is to use binary search in the range $[1, n^*]$ to locate the critical index m^* in a faster way. However, to justify the correctness of such a procedure one needs to prove that if one had further set $t_i = 0 \;\; \forall i \;\in \Pi_1 \cup \Pi_2 \ldots \cup \Pi_y$, where $y \geq m^*$, and invoked subsequently the algorithm OPT, then S_{OPT} obtained in this way would have still satisfied the constraint set given by (4.29). Notice that if this property does not hold, then it is not possible to determine the "direction" of the search by simply testing S_{OPT} at a given index i, since we must be assured that there exists a *unique* index m^* such that:

- setting $t_i = 0 \; \forall i \in \Pi_1 \cup \Pi_2 \ldots \cup \Pi_y$ and invoking OPT *does not provide* a solution S_{OPT} which satisfies the inequality constraints whenever $1 \leq y < m^* - 1$
- setting $t_i = 0 \; \forall i \in \Pi_1 \cup \Pi_2 \ldots \cup \Pi_y$ and invoking OPT *does provide* a solution S_{OPT} which satisfies the inequality constraints whenever $m^* - 1 \leq y \leq n$.

The first of these properties follows directly from the correctness of Algorithm OPT-L. It turns out that the second property also holds for *concave* objective functions as proven below. Hence, the time complexity $C_{OPT-L}(n)$ may be reduced to $O(n \cdot \log n)$ by using a binary search like technique. Algorithm FAST-L which solves Problem OPT-L in time $O(n \cdot \log n)$ is shown in Figure 4.14.

4.5.2.2 Correctness proof of the Fast Algorithm

We begin by introducing the following additional notation regarding the k^{th} iteration of Algorithm OPT-L.

- $t_{i,k}$: service time assigned to the optional part of task T_i by OPT during the k^{th} iteration of Algorithm OPT-L
- $S_{OPT,k} = \{t_{1,k}, \ldots, t_{n,k}\}$: solution produced by OPT during the k^{th} iteration of Algorithm OPT-L
- $V_k = \{x | t_{x,k} < 0\}$: the set of indices for which the solution $S_{OPT,k}$ *violates* inequality constraints.

Clearly, Algorithm OPT-L successively sets $t_i = 0 \;\; \forall i \;\in \Pi_1 \cup \Pi_2 \ldots \cup \Pi_{m^*-1}$ until Algorithm OPT returns a solution which does not violate any constraints for functions in F_{m^*} at the $(m^*)^{th}$ iteration. Algorithm FAST-L uses binary search to determine the critical index m^* efficiently. The correctness of Algorithm OPT-L assures that $\forall \; y < m^* - 1$, setting $t_i = 0 \;\; \forall i \;\in \Pi_1 \cup \Pi_2 \ldots \cup \Pi_y$ and invoking OPT would yield a non-empty violating set V_y for the remaining tasks. Finally, Proposition 4 establishes that $\forall \; y \geq m^* - 1$, V_y will always remain empty after

Algorithm FAST-L(F,d)

Evaluate S_{OPT} of the corresponding Problem OPT
If all the constraints are satisfied then
    S_{OPT-L}=S_{OPT}; exit
Enumerate the functions in **F** according to the
non-decreasing order of $w_j(0)$ values and
construct the sets Π_i $i = 1 \ldots n^*$
$m = r = \lfloor n^*/2 \rfloor$
Do forever
    Evaluate S_{OPT} by invoking
    OPT($\Pi_{m+1} \cup \Pi_{m+2} \ldots \cup \Pi_{n^*}, d$)
    if a constraint is violated
        $m = m + \lfloor r/2 \rfloor$; $r = \lfloor r/2 \rfloor$
    else {
        $S_1 = S_{OPT}$
        Evaluate S_{OPT} by invoking
        OPT($\Pi_m \cup \Pi_{m+1} \ldots \cup \Pi_{n^*}, d$)
        if a constraint is violated
            $S_{OPT-L} = \{t_i = 0 \;\; \forall i \; \in \Pi_1 \cup \Pi_2 \ldots \cup \Pi_{m+1}\} \cup S_1$;
            exit
        else {
            $m = m - \lfloor r/2 \rfloor$; $r = \lfloor r/2 \rfloor$
        }
    }

Figure 4.14: Fast Algorithm for Problem OPT-L

setting $t_i = 0 \ \ \forall i \ \ \in \Pi_1 \cup \ldots \Pi_{m^*-1} \ldots \cup \Pi_y$ and invoking OPT, since this would leave even *more* slack for the remaining tasks. In the algorithm FAST-L, a specific index m is tested at each iteration to check whether it satisfies the property $V_m = \emptyset$ and $V_{m-1} \neq \emptyset$. If this is the case, then $m = m^* - 1$, since there is only one index m^* satisfying this property. However, in case that $V_m \neq \emptyset$ then we can infer that $m < m^* - 1$ and the next probe is determined in the range (m, n^*). Finally, if both $V_m = \emptyset$ and $V_{m-1} = \emptyset$ then we restrict the search in the range $(0, m-1)$.

Proposition 4 *Suppose that, during the execution of Algorithm OPT-L, $S_{OPT,k}$ does not violate some inequality constraints (i.e., $V_k = \emptyset$), yet we set $t_j = 0$ for $\forall j \ \in \Pi_k$. Then the $(k+1)^{th}$ invocation of Algorithm OPT for the remaining tasks yields $S_{OPT,k+1}$ such that $t_{i,k+1} \geq t_{i,k}$ for all $i \in \ F_{k+1} = F_k - \Pi_k$ (which implies that V_{k+1} is still empty).*

Proof: Note that $t_{i,k} \geq 0 \ \forall \ i \in F_k$ by assumption. Based on the optimality property of subproblems, if the k^{th} invocation of Algorithm OPT yields an optimal solution, it will also generate the *optimal* distribution of $d - \sum_{j \in \Pi_k} b_j \ t_{j,k} = d_1 \leq d$ time units among functions in $F_k - \Pi_k$. However, the $(k+1)^{th}$ invocation provides optimal distribution of $d - 0 = d$ among functions in $F_k - \Pi_k$ as well (by setting $t_j = 0$ for $\forall j \in \Pi_k$). Thus, two successive invocations of Algorithm OPT can be written as:

$$\text{maximize} \quad \sum f_i(t_i)$$
$$\text{subject to} \quad \sum b_i \ t_i = d_1 \qquad i \in F_k - \Pi_k$$

and

$$\text{maximize} \quad \sum f_i(t_i)$$
$$\text{subject to} \quad \sum b_i \ t_i = d_1 + e \quad i \in F_{k+1} = F_k - \Pi_k$$

where $e \geq 0$. Hence the proof will be complete if we show that $\forall i \in \ F_k - \Pi_k, \ t_{i,k+1} \geq t_{i,k}$.

Any solution $S_{OPT,k}$ should satisfy first-order necessary conditions for Lagrangian [69]:

$$\sum b_i \ t_{i,k} = d \tag{4.34}$$
$$\frac{1}{b_i} f_i'(t_{i,k}) - \lambda = 0 \qquad \forall i \text{ such that } f_i \in F_k \tag{4.35}$$

The necessary conditions (4.35) give

$$\frac{1}{b_c} f_c'(t_{c,k}) = \frac{1}{b_d} f_d'(t_{d,k}) \qquad \forall c, d \in \ F_k - \Pi_k \tag{4.36}$$
$$\frac{1}{b_c} f_c'(t_{c,k+1}) = \frac{1}{b_d} f_d'(t_{d,k+1}) \qquad \forall c, d \in \ F_k - \Pi_k \tag{4.37}$$

For the sake of contradiction, assume that $\exists w \in F_k - \Pi_k$ such that $t_{w,k+1} < t_{w,k}$. Since $e \geq 0$, there must be some $y \in F_k - \Pi_k$ such that $t_{y,k+1} > t_{y,k}$. We distinguish two cases:

1. f_w is nonlinear, that is, its derivative is *strictly* decreasing. Since f_y is also concave, we can write $\frac{1}{b_w} f'_w(t_{w,k+1}) > \frac{1}{b_w} f'_w(t_{w,k})$ and $\frac{1}{b_y} f'_y(t_{y,k+1}) \leq \frac{1}{b_y} f'_y(t_{y,k})$ which are clearly inconsistent with Equations (4.36) and (4.37). This can be easily seen by substituting w for c and y for d in Equations (4.36) and (4.37), respectively.

2. f_w is linear, which implies that $f'_w(t_{w,k+1}) = f'_w(t_{w,k}) = j$. In this case, to satisfy Equations (4.36) and (4.37) f'_y should be also linear of the form $f_y(t) = h \cdot t$ such that $h = \frac{b_y\, j}{b_w}$. Hence, two functions have the *same* marginal return $w_w = w_y = \frac{j}{b_w} = \frac{h}{b_y}$ But remembering our assumption from Section 4.5.1 that Algorithm OPT treats all linear functions of the same marginal return "fairly" (that is, assigns them the same amount of service time), we reach a contradiction since $t_{w,k}$ was supposed to be greater than $t_{y,k}$.

□

Complexity: At most $O(\log n)$ probes are made during binary search and at each probe Algorithm OPT is called twice. Recall that Algorithm OPT has the time complexity $O(n)$. The initial cost of sorting the derivative values is $O(n \log n)$. Hence the total complexity is $C_{OPT-L}(n) = O(C_{OPT}(n) \cdot \log n + n \cdot \log n)$, which is $O(n \cdot \log n)$.

4.5.3 Combining All Constraints: Solution of Problem OPT-LU

An instance of Problem OPT-LU is characterized by the set **F**$= \{f_1, f_2, \ldots, f_n\}$ of non-decreasing, differentiable, and concave reward functions, the set **O**$= \{o_1, o_2, .., o_n\}$ of upper bounds on the length of optional execution parts, and available slack d:

$$\text{maximize} \quad \sum_{i=1}^{n} f_i(t_i) \tag{4.38}$$

$$\text{subject to} \quad \sum_{i=1}^{n} b_i\, t_i = d \tag{4.39}$$

$$t_i \leq o_i \quad i = 1, 2, \ldots n \tag{4.40}$$

$$0 \leq t_i \quad i = 1, 2, \ldots n \tag{4.41}$$

We recall that $0 < d < \sum_{i=1}^{n} o_i$ and $0 < o_i\ \forall i$ in the specification of OPT-LU.

We first observe the close relationship between the problems OPT-LU and OPT-L. Indeed, OPT-LU has only an additional set of upper bound constraints. It is not difficult to see that if S_{OPT-L} satisfies the constraints given by Equation (4.40), then the solution S_{OPT-LU} of problem OPT-LU is the same as S_{OPT-L}. However, if an upper bound constraint is violated then we will construct the solution iteratively in a way analogous to that described in the solution of Problem OPT-L.

Let $\Gamma = \{x | \frac{1}{b_x} f'_x(o_x) \geq \frac{1}{b_i} f'_i(o_i)\ \forall i\}$. Informally, Γ contains the functions $f_x \in \mathbf{F}$ with the largest marginal returns at the upper bounds, $w_x(o_x)$.

The algorithm ALG-OPT-LU (see Figure 4.15) which solves the problem OPT-LU is based on the successive invocations of FAST-L. First, we find the solution S_{OPT-L} of the corresponding problem OPT-L. We know that this solution is optimal for the simpler problem which does not take into account upper bounds. If the upper bound constraints are automatically satisfied, then we are done. However, if this is not the case, we set $t_q = o_q\ \forall q \in \Gamma$. Finally, we update the sets **F,** **O** and the slack d before going through the next iteration.

Algorithm OPT-LU(F,O,d)

```
1    Set S_{OPT-LU} = ∅
2    if F = ∅ then exit
3    Evaluate S_{OPT-L} by invoking Algorithm FAST-L
4    if all upper bound constraints are satisfied then
     S_{OPT-LU} = (S_{OPT-LU} ∪ S_{OPT-L}); exit
5    compute Γ
6    set t_q = o_q ∀q ∈ Γ in S_{OPT-LU}
7    set d = d − Σ_{x∈Γ} b_x o_x
8    set F=F−Γ
9    set O=O−{o_x | x ∈ Γ}
10   Goto Step 2
```

Figure 4.15: Algorithm to solve Problem OPT-LU

Correctness: Most of the algorithm is self-explanatory in view of the results obtained in previous sections. However, lines 5 and 6 of ALG-OPT-LU require further elaboration. In addition to constraints (4.39), (4.40) and (4.41), the necessary and sufficient Kuhn-Tucker conditions for Problem OPT-LU can be expressed as:

$$
\begin{aligned}
-\frac{1}{b_i} f'_i(t_i) + \lambda + \frac{\bar{\mu}_i}{b_i} - \frac{\mu_i}{b_i} &= 0 \quad i = 1, 2, .., n && (4.42)\\
\bar{\mu}_i (t_i - o_i) &= 0 \quad i = 1, 2, ., .n && (4.43)\\
-\mu_i\, t_i &= 0 \quad i = 1, 2, ., .n && (4.44)\\
\bar{\mu}_i &\geq 0 \quad i = 1, 2, .., n && (4.45)\\
\mu_i &\geq 0 \quad i = 1, 2, .., n && (4.46)
\end{aligned}
$$

where $\lambda, \bar{\mu}_1, \bar{\mu}_2, \ldots, \bar{\mu}_n, \mu_1, \mu_2, \ldots, \mu_n$ are Lagrange multipliers.

Claim 4 *If S_{OPT-L} violates upper bound constraints given by Equation (4.40) then $\exists i\ \bar{\mu}_i > 0$.*

Proof: Assume that $\forall i\ \bar{\mu}_i = 0$. In this case, Kuhn-Tucker conditions (4.43) and (4.45) vanish. Also conditions (4.39), (4.41), (4.42), (4.44) and (4.46) become exactly identical to Kuhn-Tucker conditions of Problem **OPT-L**. Thus, S_{OPT-L} returned by Algorithm FAST-L is also equal to S_{OPT-LU} if and only if it satisfies the extra constraint set given by (4.40).

□

Claim 5 $\forall i\ \bar{\mu}_i > 0$ *implies* $\mu_i = 0$.

Proof: Assume that $\exists i$ such that $\bar{\mu}_i > 0$ and $\mu_i > 0$. In this case, (4.43) and (4.44) force us to choose $t_i = o_i$ and $t_i = 0$, which implies that $o_i = 0$. But this is contrary to our assumption that $0 < o_i\ \forall i$ in the specification of the problem.

□

Now we are ready to justify line 5 of the algorithm:

Lemma 2 *If S_{OPT-L} violates upper bound constraints given by (4.40) then , in S_{OPT-LU}, $t_q = o_q\ \ \forall q \in \Gamma$.*

Proof: We will prove that the Lagrange multipliers $\bar{\mu}_i,\ \ i \in \Gamma$, are all non-zero, which will imply (by (4.43)) that $t_q = o_q,\ \forall q \in \Gamma$. Suppose $\exists m \in \Gamma$ such that $\bar{\mu}_m = 0$. We know that $\exists j$ such that $\bar{\mu}_j > 0$ (Proposition 4). Furthermore, $\mu_j = 0$ by Claim 5. Using (4.42) we can write $-\frac{1}{b_m} f'_m(t_m) - \frac{\mu_m}{b_m} = -\frac{1}{b_j} f'_j(t_j) + \frac{\bar{\mu}_j}{b_j}$ which gives (since $t_j = o_j$):

$$\frac{1}{b_m} f'_m(t_m) = \frac{1}{b_j} f'_j(o_j) - \frac{\bar{\mu}_j}{b_j} - \frac{\mu_m}{b_m} \quad (\bar{\mu}_j > 0,\ \mu_m \geq 0)$$

Since t_m is necessarily less than or equal to o_m, we can deduce $\frac{1}{b_m} f'_m(o_m) \leq \frac{1}{b_m} f'_m(t_m) < \frac{1}{b_j} f'_j(o_j)$ which contradicts our assumption that $m \in\ \Gamma$.

□

Complexity: Notice that the worst case time complexity of each iteration is equal to that of Algorithm FAST-L, which is $O(n \cdot \log n)$. Observe that the cardinality of **F** decreases by at least 1 after each iteration. Hence, the number of iterations is bounded by n. It follows that the total time complexity $C_{OPT-LU}(n)$ is $O(n^2 \cdot \log n)$.

4.6 Chapter Summary

In this chapter, we addressed the scheduling of periodic reward-based real-time tasks. We showed that, for the linear and concave reward functions, there exists always an optimal schedule where the mandatory parts meet their deadlines and the optional service time of a given task remains constant at each instance. This, in turn, implied the optimality of any policy which can achieve 100% processor utilization. Examples of such policies are Rate Monotonic Scheduling (with harmonic periods), Earliest Deadline First and Least Laxity First. We then extended the result to identical shared-memory multiprocessors.

Next, we addressed the computation of optimal service times by solving a non-linear optimization problem. We derived an iterative algorithm based on Kuhn-Tucker optimality conditions for non-linear programs. The worst-case complexity of the algorithm is $O(n^2 \log n)$, where n is the number of tasks. We showed that the reward-based scheduling problem becomes NP-Hard once one allows convex reward functions.

To quantify the improvement we obtained through this optimal algorithm, we analyzed Mandatory-First policies, previously proposed in the literature. We showed that the performance of these algorithms may be arbitrarily bad when compared to the optimal policy under a worst-case scenario. We also conducted experiments with synthetic task sets. The results indicate that the advantage of the optimal policy becomes striking especially at high mandatory utilization values.

Finally, to investigate the limits of our results regarding the optimality of identical service times, we analyzed the cases where some fundamental assumptions are relaxed. We showed that the optimality of identical service times no longer holds, if one allows convex reward functions, commits to Static Priority Scheduling or considers tasks in which the relative deadline is not equal to the period.

Optimality of identical service times with concave reward functions in our *instance-based* CPU allocation model, implies also an interesting equivalence to *utilization-based* resource allocation model[85]. However, the equivalence no longer holds if one relax any of the assumptions mentioned above.

Chapter 5
Periodic Power-Aware Scheduling

Power management issues have recently moved to the front of research in Computer Science and Engineering, with the proliferation of mobile and embedded systems that have to rely on battery power during the operation. The motivation behind *Power-Aware Scheduling* is the fact that the CPU power consumption is related to the CPU voltage/speed in a convex (usually cubic) fashion. Thus, in principle, striking energy savings are possible, if an increase in response time is affordable. Nevertheless, in real-time systems, such speed reduction decisions should be taken very carefully in order to avoid any compromise in the temporal predictability of the system.

5.1 The Reward-Based Approach to Power-Aware Scheduling

Before analyzing the periodic model in depth, we formally correlate the reward-based scheduling framework to the power-aware scheduling for hard real-time tasks. We underline that this equivalence is preserved regardless of the task model, as long as our aim is to reach a solution for a given (worst-case) workload. We first formally define the real-time power-aware scheduling problem.

Real-Time Power-Aware Scheduling (RT-PAS) problem: Consider a CPU with variable voltage/speed S $(S_{min} \leq S \leq S_{max})$ facility, where the power consumption is given by a strictly increasing convex function $g(S)$, which is also a polynomial of at least second degree (Further details about the power model are given in Section 3.1.2). Given a set **T** of real-time tasks, in which each task T_i is subject to a worst-case workload of C_i expressed in the number of required CPU cycles, and a time point t_f, determine *the schedule* and *the processor speed* $S(t)$ so as to minimize the total energy consumption $E(0, t_f) = \int_0^{t_f} g(S(t))dt$ in the interval $[0, t_f]$.

Proposition 5 *One can safely commit to a constant CPU speed during the execution of a task instance T_i requiring C_i CPU cycles, without increasing the energy consumption.*

Proof:

Suppose that T_i receives Δ units of CPU time before releasing the CPU either by preemption, or by completion. Further, assume that the CPU speed is changed m times while the CPU is continuously allocated to T_i. The speed changes occur $\Delta_0 = 0, \Delta_1, \Delta_2, \ldots, \Delta_m = \Delta$ time units after T_i starts executing at $t = t_i$; and the CPU speed is S_{Δ_i} during the time interval $[t_i + \Delta_i, t_i + \Delta_{i+1}]$.

Now, consider the *average* speed of T_i during this execution:

$$S_i = \frac{\int_{t_i+\Delta_0}^{t_i+\Delta_1} S_{\Delta_0}\,dt + \ldots + \int_{t_i+\Delta_{m-1}}^{t_i+\Delta_m} S_{\Delta_{m-1}}\,dt}{(\Delta_1 - \Delta_0) + \ldots + (\Delta_m - \Delta_{m-1})}$$

By committing to the constant speed S_i, T_i would be allocated exactly the same amount of CPU time as before. Further, the total energy consumption can only be decreased after this transformation since $S_i \cdot [(\Delta_1 - \Delta_0) + \ldots + (\Delta_m - \Delta_{m-1})] = S_i \cdot \Delta \leq \int_{t_i+\Delta_0}^{t_i+\Delta_1} S_{\Delta_0}\,dt + \ldots + \int_{t_i+\Delta_{m-1}}^{t_i+\Delta_m} S_{\Delta_{m-1}}\,dt$, due to the convex nature of power/energy consumption function. By repeating the argument for every other schedule interval where T_i executes, and by considering all the intervals at once, the proposition is justified.

□

Note that, the determination of S_i is effectively equivalent to determining the CPU time allocation to T_i, which will be denoted by x_i ($x_i = \frac{C_i}{S_i}$). We are now ready the establish the connection between the RT-PAS and Reward-Based scheduling problems.

Proposition 6 *The RT-PAS problem is equivalent to Reward-Based Scheduling problem with concave reward functions.*

Proof:

To prove the statement, we will first formulate the computation of optimal speed values as an optimization problem. The total energy consumption, thanks to the constant speed assumption per task, can now be expressed as a sum:

$$E_{tot} = \sum_{i=1}^{n} x_i \cdot g(S_i) = \sum_{i=1}^{n} x_i \cdot g(\frac{C_i}{x_i})$$

Next, observe that the minimum and maximum speed bounds impose natural lower and upper bounds on CPU allocation of T_i:

$$\frac{C_i}{S_{max}} \leq x_i \leq \frac{C_i}{S_{min}}$$

Hence, the computation of optimal CPU allocation assignments can be formalized as an optimization problem:

$$\text{minimize} \quad \sum_{i=1}^{n} x_i \cdot g(\frac{C_i}{x_i}) \tag{5.1}$$

$$\text{subject to} \quad \frac{C_i}{S_{max}} \leq x_i \leq \frac{C_i}{S_{min}} \quad i = 1, \ldots, n \tag{5.2}$$

$$\text{There exists a feasible schedule with } \{x_i\} \text{ values} \tag{5.3}$$

Now, set

$$\begin{aligned} m_i &= \frac{C_i}{S_{max}} \\ t_i &= x_i - m_i \\ o_i &= \frac{C_i}{S_{min}} - \frac{C_i}{S_{max}} \\ F_i(t_i) &= -(t_i + \frac{C_i}{S_{max}})g(\frac{C_i}{t_i + \frac{C_i}{S_{max}}}) \end{aligned}$$

This transformation can be interpreted as follows: First, T_i must be assigned at least $\frac{C_i}{S_{max}}$ units of CPU time ("mandatory" execution). Any allocation exceeding this minimum amount will be considered as "optional" execution, while the total CPU allocation $(m_i + t_i)$ can not go beyond an upper bound $(\frac{C_i}{S_{min}})$. Finally, the more we allocate CPU time to T_i by increasing t_i, the more we increase the energy savings thanks to the speed/power relation. Thus, we obtain an equivalent optimization problem:

$$\text{maximize} \quad \sum_{i=1}^{n} F_i(t_i) \tag{5.4}$$

$$\text{subject to} \quad 0 \leq t_i \leq o_i \qquad i = 1, \ldots, n \tag{5.5}$$

$$\text{There exists a feasible schedule with } \{m_i\} \text{ and } \{t_i\} \text{ values} \tag{5.6}$$

The final formulation is an instance of the general reward-based scheduling problem defined in Section 3.1.1 by Equations (3.5), (3.6) and (3.7). Further, the reward function $F_i(t_i)$ above is clearly concave, since $(t_i + \frac{C_i}{S_{max}})g(\frac{C_i}{t_i+\frac{C_i}{S_{max}}})$ is convex. To see this, we can use the result from [69] stating that if a and b are both convex functions and if a is increasing, then $a(b(x))$ is also convex. Thus, by setting $h(t_i) = \frac{C_i}{t_i+\frac{C_i}{S_{max}}}$, and observing that the multiplication by $(t_i + \frac{C_i}{S_{max}})$ does not affect the convexity, we justify the concavity of $F_i(t_i) = -g(h(t_i))$.

□

5.2 Three Dimensions of Power-Aware Scheduling

As shown in Section 5.1, for a given task model and the worst-case workload, the Power-Aware scheduling problem can be shown to be equivalent to the Reward-Based scheduling problem through an appropriate transformation.

However, an application with timing constraints usually exhibits a large variation in *actual* workload; for example [25] reports that the ratio of the worst-case execution time to the best-case execution time can be as high as 10 in typical applications. Consequently, if we start executing the tasks with the statically computed optimal speeds based on the worst-case assumptions, it is highly likely that we will encounter *early* completions. In this case, *dynamically monitoring and reclaiming the 'unused' computation time* can be a powerful approach to obtain additional power savings and to minimize the effects of designing the system with worst-case workload, which is usually a very conservative prediction of the *actual* behaviour.

Further improvements can be also achieved by using the *statistical* or *average* workload information: this is the basis for the *aggressive* schemes where we *anticipate* early completions and *speculatively* reduce the CPU speed. This approach immediately raises two intertwined questions, namely, (a) the *level* of aggressiveness which justifies speculative moves under a given average-case workload information; and (b) the issue of guaranteeing the timing constraints when operating in aggressive modes. It is obvious that the solutions to these problems should be simultaneously practical and efficient, in order to be applicable on-line.

In the following, we address three dimensions of power-aware scheduling for hard real-time systems and develop efficient algorithms for the periodic model. Effectiveness in reducing the energy consumption can be improved only by a simultaneous consideration of these three dimensions, since they complement each other. Thus, we present:

1. An optimal static (off-line) solution to compute the speed assignments (for each task) that minimizes the system energy consumption, assuming worst-case workload for each arrival (Section 5.3).

2. An on-line speed adjustment mechanism to reclaim energy not used by tasks that complete without consuming their worst-case workload (Section 5.4).

3. An on-line and speculative speed adjustment mechanism to anticipate and compensate probable early completions of future executions (Section 5.5).

We emphasize that, in the context of hard real-time systems, all these components should be designed not to cause any deadlines to be missed even under the worst-case scenario: our aim is to **meet the timing constraints while simultaneously and dynamically reducing energy consumption as much as possible**.

5.3 Optimal Static Solution

In this section, we present the static optimal solution to the periodic power-aware scheduling problem, *assuming that each task presents its worst-case workload to the processor at every instance*. Due to the periodic scheduling settings, we need to determine the optimal speed S_{ij} for each task instance T_{ij}. Proposition 5 established that during the execution of a given task instance T_{ij} the CPU speed does not need be changed. Nevertheless, a stronger result can be asserted.

Proposition 7 *There exists always a schedule with minimum total energy consumption, where each periodic task T_i is executed with the same speed S_i from instance to instance. Further, any periodic scheduling policy which can fully utilize the processor (EDF, LLF or RMS-h) can be used to obtain a feasible schedule with these assignments.*

Proof:

Follows from the equivalence of Periodic Power-Aware Real-Time Scheduling problem and Reward-Based Real-time Scheduling problem (Proposition 6) and Theorem 1 (Section 4.1).

□

We can now provide the Static Optimal solution.

Proposition 8 *The optimal CPU speed to minimize the total energy consumption while meeting all the deadlines is constant and equal to $\bar{S} = max\{S_{min}, U_{tot}\}$, where $U_{tot} = \sum_{i=1}^{n} \frac{C_i}{P_i}$.*

Proof:

First, observe that, if it is possible to meet all the deadlines by executing all the tasks with S_{min}, in other words, if $\sum_{i=1}^{n} \frac{C_i}{S_{min} \cdot P_i} = \frac{U_{tot}}{S_{min}} \leq 1$, then clearly, this choice minimizes the total energy consumption.

For the other case, in which $U_{tot} > S_{min}$, we should fully utilize the CPU in view of the the convex nature of power consumption function, that is, the *effective* utilization ($\sum_{i=1}^{n} \frac{C_i}{S_i \cdot P_i}$) of the task set must be equal to 1. The optimal speed assignments in this case can be computed by solving the following optimization problem[1]:

$$\text{minimize} \quad \sum_{i=1}^{n} \frac{P}{P_i} \frac{C_i}{S_i} \cdot g(S_i) \tag{5.7}$$

$$\text{subject to} \quad \sum_{i=1}^{n} \frac{C_i}{P_i \cdot S_i} = 1 \tag{5.8}$$

$$S_{min} \leq S_i \leq S_{max} \quad i = 1, \ldots, n \tag{5.9}$$

[1] Naturally one can use the Problem OPT-LU to obtain the CPU allocations at task-level, and then compute the speed. However, as it will be seen, obtaining the optimal speed assignments directly is relatively easier for this case.

Let us focus first on satisfying the equality constraint (5.8). By taking the Lagrangian, we get:

$$\frac{P}{P_i}[-\frac{C_i}{S_i^2} \cdot g(S_i) + g'(S_i) \cdot \frac{C_i}{S_i}] - \lambda \cdot \frac{1}{P_i}\frac{C_i}{S_i^2} = 0 \quad \text{for } i = 1 \ldots n$$

The crucial point is that choosing $S_i = S_j = \bar{S}$ will immediately satisfy the Lagrangian for every i, since the power consumption function $g(S)$ is identical in every expression. And, if we commit to a common speed, we can use (5.8) to compute $\bar{S} = \sum_{i=1}^{n} \frac{C_i}{P_i} = U_{tot}$. Further, this will automatically satisfy (5.9), since $\bar{S} = U_{tot} > S_{min}$ and $\bar{S} = U_{tot} \leq S_{max} = 1.0$ by assumption.

□

5.4 Dynamic Reclaiming Algorithm

The dynamic reclaiming algorithm is based on detecting early completions and adjusting (reducing) the speed of other tasks *on-the-fly* in order to provide additional power savings while still meeting the deadlines. To this aim, we perform comparisons between the actual execution history and the canonical schedule $\mathcal{S}^{can}$, which is the static optimal schedule on which every instance presents its worst-case workload to the processor and runs at a constant speed. The CPU speed is adjusted only at task dispatch times: thus, we should be able to say whether the task is being dispatched *earlier* than $\mathcal{S}^{can}$, and if so, to determine the amount of additional CPU time the dispatched task can *safely* use to slow down its execution; we will refer to this additional CPU time as the *earliness* of the dispatched task.

Before providing the details of our approach, we underline that a simple approach that equates earliness with previously unused CPU time and *blindly* slows down the processor is *not* a safe approach. To see this, consider a 3-task system with the following parameters: $C_1 = 4, P_1 = 10, C_2 = 4, P_2 = 10, C_3 = 6, P_3 = 30$. The worst-case utilization of the task set is equal to 1.00, hence the optimal speed for the static version is $\bar{S} = S_{max} = 1.00$ (from Proposition 8). If every task presents its worst-case workload at every instance and we use EDF, then the schedule in Figure 5.1 ($\mathcal{S}^{can}$) would be obtained. Now, suppose that T_3 completes early at $t = 9$, leaving an unused computation time of 5 units before its deadline. If these 5 units of CPU time are used by $T_{1,2}$ (recall that $T_{i,j}$ is the j^{th} instance of task i), $T_{2,2}$ will miss its deadline, if both $T_{1,2}$ and $T_{2,2}$ require their worst-case workload.

As we can see, computing and managing earliness is not a trivial task. Moreover, it is clearly impractical to a priori produce and keep the entire static optimal schedule $\mathcal{S}^{can}$ during the execution. In order to simultaneously address the problems of feasibility and efficiency, while tasks execute, complete, re-arrive dynamically and the actual schedule is produced, we choose to keep

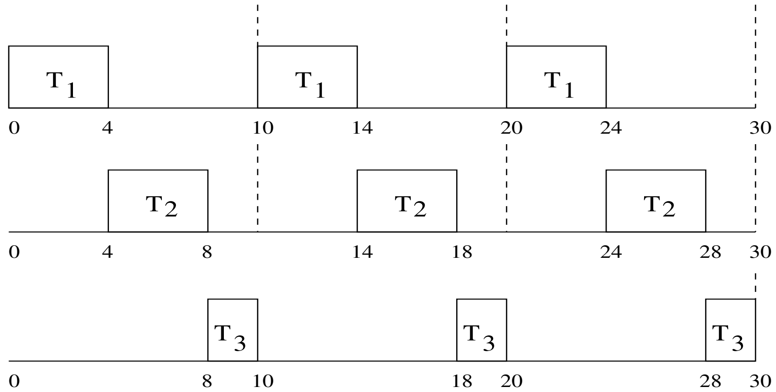


Figure 5.1: The static optimal schedule, $\mathcal{S}^{can}$

and update a data structure (called α**-queue)** that helps to compute the earliness of tasks when they are dispatched. At any time t during actual execution, the α-queue contains information about tasks that would be active (i.e., running or ready) at time t in the worst-case static optimal schedule $\mathcal{S}^{can}$ (in other words, α-queue is the ready queue of $\mathcal{S}^{can}$ at time t). We assume that the following information can be obtained for each task from the α-queue at any time t:

- i, the identity of the task (i.e., task number),
- $r_{i,j}$, the arrival time of the instance (i.e., the period boundary earlier than t),
- $d_{i,j}$, the deadline of the instance (i.e., the period boundary later than t), and
- $rem_{i,j}(t)$, the remaining execution time of $T_{i,j}$ in $\mathcal{S}^{can}$, under the static optimal speed $\bar{S}$.

Clearly, given t, the $r_{i,j}$ and $d_{i,j}$ values can be easily computed for the periodic task model. Note that the α-queue at time t contains information about all instances $T_{i,j}$ such that $r_{i,j} \leq t \leq d_{i,j}$, and $rem_{i,j}(t) > 0$. The α-queue contains at most n elements, since the number of tasks in the ready queue can never exceed the total number of tasks in any schedule. Therefore, we will omit the instance number while referring to α-queue elements, whenever the clarity is not compromised.

Our approach assumes that the tasks are scheduled according to EDF* policy. EDF* is the same as EDF (earliest deadline first [67]), except that, among tasks whose deadlines are the same, the task with the earliest arrival time has the highest priority (FIFO policy); in case that both deadline and arrival times are equal, the task with the lowest index has the highest priority. This EDF* *priority ordering* is essential in our approach because it provides a total order on the priorities. Further, we assume that the α-queue is also ordered according to EDF* priorities. We denote the EDF* priority-level of the task i by d_i^* (low values denote high priorities).

At this point, we are ready to relate the α-queue with the computation of earliness factor. Let $w_i^S(t)$ denote the remaining worst-case execution time of task T_i under the speed S at time t.

Proposition 9 *For any task T_x which is about to execute, any unused computation time (slack) of any task in the α-queue having strictly higher priority than T_x will contribute to the earliness of T_x along with already finished work of T_x in the actual schedule. That is,* total earliness *of T_x is no less than* $\epsilon_x(t) = \sum_{i|d_i^* < d_x^*} rem_i(t) + rem_x(t) - w_x^{\bar{S}}(t) = \sum_{i|d_i^* \leq d_x^*} rem_i(t) - w_x^{\bar{S}}(t)$.

To understand the above result, note that when T_x is being dispatched, tasks with higher priority that are still in the α-queue must be already finished in the actual schedule (since T_x currently has the highest EDF* priority), but they would have not yet finished in $\mathcal{S}^{can}$.

Note that $\epsilon_x(t)$ is only a conservative estimation of the *actual* earliness of T_x. Any task with higher priority that becomes ready, preempts T_x and leaves the system early, before T_x ends, may further contribute to the delay of T_x and also increase the actual earliness of T_x; yet in the absence of clairvoyant capabilities, not much can be done to deterministically take this into consideration. In addition, when T_x returns from preemption after such a future high priority interruption, it will anyhow be able to better evaluate its earliness, and further slow-down, thanks to the above formula. Therefore, we adopt this relatively simple and fast way of inferring the earliness amount *from past events*.

Implementing the α-queue: The α-queue can be easily implemented using the following rules:

R1. Initially the α-queue is empty.

R2. Upon arrival, each task T_j "pushes" its worst-case execution time under nominal speed $\widehat{S_j} = \bar{S}$ to the α-queue in the correct EDF* priority position (This happens only once for each arrival, no re-push at 'return from preemptions').

R3. As time elapses, the elements in the α-queue are updated (consumed) accordingly: the $rem_{i,j}$ field at the head of α-queue is decreased with a rate equal to that of the passage of time. Whenever the $rem_{i,j}$ field of the head reaches zero, that element is removed from α-queue and the update continues with the next element. No update is done when the α-queue is empty.

Observation 1 *At time t, the α-queue, updated according to the rules R1, R2 and R3, contains only the tasks that would be ready at time t in the static optimal schedule $\mathcal{S}^{can}$. Further, the $rem_{i,j}$ field contains the remaining allotted time of each active instance at time t in $\mathcal{S}^{can}$.*

This stems from the following: (a) α-queue is ordered according to EDF* order, (b) every arriving task pushes its remaining worst-case execution time (under nominal speed $\bar{S}$) into the α-queue only once, and (c) the queue is updated only at the head, reflecting the fact that only the task with the highest EDF* priority would be running in $\mathcal{S}^{can}$, and (d) a task that would have finished in

$\mathcal{S}^{can}$ is removed from the α-queue. This effectively yields *a dynamic image* of the ready queue in $\mathcal{S}^{can}$ at time t.

Note that the dynamic reduction of $rem_{i,j}$ in R3 above does not need to be performed at every clock cycle; instead, for efficiency, we perform the reduction whenever a task is preempted or completes, by taking into account the time elapsed since the last update. The above approach relies on two facts: first, the speed adjustment decision will be taken only at arrival/preemption and completion times, and it is necessary to have an accurate α-queue only at these points (If speeds are to be changed at other points, the update of $rem_{i,j}$ must reflect that). Second, between these points, each task is effectively executed non-preemptively.

At this point, we are ready to present our Generic Dynamic Reclaiming Algorithm, GDRA, shown in Figure 5.2. Procedure Speed-Reduce(T_x, B, S), in Figure 5.3, will be used by GDRA to reduce the speed of T_x, by allocating B units of time allocation to its execution under worst-case remaining load, subject to S_{min} constraint. DRA is "generic" in the sense that the amount of additional time allocation Y is not specified, it may assume any value between 0 and $\epsilon_x(t)$ without compromising the correctness.

The following theorem establishes that the schedules produced by GDRA are always *ahead* of $\mathcal{S}^{can}$.

Theorem 4 *At any time t during the execution of GDRA, $w_i^{\widehat{S_i}}(t) \leq rem_i(t)$, for any ready task T_i.*

Proof:

Throughout the proof, we will consider the actual CPU schedule as a *sequence of task execution segments*. A segment starts when a given task T_a is dispatched (say at $t = t_a$), and ends (at $t = t_b$) when T_a releases the CPU either by completion or by preemption. In other words, during the time interval (segment) $[t_a, t_b]$, the CPU is allocated continuously to T_a. We will use induction to prove the validity of the statement for every such execution segment (on which a given task runs). **Note that we will need to show the validity of the proposition for each segment and for all the tasks that are (or incidentally, may become) ready during this segment**.

First, observe that initially (at $t = t_0$) all tasks are ready and the α-queue contains their $w_i^{\widehat{S_i}}$ values in the rem_i fields. Assume that T_1 has the highest EDF* priority at this time, and it will run with the speed $\widehat{S_1}$ until it completes at $t = t_1$ (note that the identical ready times and EDF* imply that it won't be preempted). At any time t in the interval $[t_0, t_1)$, $rem_1 = w_1^{\widehat{S_1}}$ since $S_1 = \widehat{S_1}$. Since by definition T_1 will not execute for more than its worst-case workload, at the time of its completion at $t = t_1$, rem_1 will be at least as large as $w_1^{\widehat{S_1}}$ which is 0 by then. For every other task T_j waiting in the ready queue during this interval, rem_j will remain the same as $w_j^{\widehat{S_j}}$ since T_j did not have a chance to execute. Thus, the proposition holds for the first execution segment $[t_0, t_1]$.

<u>Rules for GDRA</u>

1. Compute $\bar{S}$ (as in Section 5.3) and assign $\widehat{S_{i,j}} = \bar{S} \; \forall \; i, j$.

2. Initialize the α-queue to the empty-list.

3. At every event (arrival/completion), consider the head of the α-queue and decrease its rem_i value by the amount of elapsed-time since the last event. If rem_i is smaller than the time elapsed since the last event, remove the head, update the time elapsed since the last event, and repeat the update with the next element. This is done until all "elapsed time" is used.

4. At every new arrival, insert into the α-queue information regarding the new task T_i with $rem_i(t) = w_i^{\widehat{S_i}}$ value in the correct EDF* order.

5. Whenever T_x is about to be dispatched at time t:

   5.0. Set $S_x = \widehat{S_x}$.

   5.1. Consult the α-queue and compute $\epsilon_x(t)$ (indicator of the earliness amount of T_x)

   5.2. Reduce the speed of task T_x by giving T_x an extra Y time units:
   S_x = Speed-Reduce(T_x, Y, S_x), where $0 \leq Y \leq \epsilon_x(t)$

6. At every event of preemption or completion of a task, say T_i, decrease the value of the remaining execution time: $w_i^{S_i} = w_i^{S_i} - \Delta_t$, where Δ_t is the time elapsed since the task T_i was last dispatched.

Figure 5.2: Generic Dynamic Reclaiming Algorithm

Procedure Speed-Reduce(T_x, B, S):

1. Set $S_x = \frac{w_x^S}{w_x^S + B} \cdot S$
2. If $S_x < S_{min}$, $S_x = S_{min}$
3. return S_x

Figure 5.3: Speed Reduction Procedure

Since the base case is established, now assume that the statement remains valid for the first k execution segments and consider the $(k+1)^{st}$ segment, during which T_x executes. Hence, T_x runs continuously from t_{k-1} to $t_k = t_{k-1} + \Delta_t$, and it relinquishes the CPU either by completion or the arrival of a higher priority task (preemption). From the induction assumption, $rem_i \geq w_i^{\widehat{S_i}}$ for every ready task T_i at $t = t_{k-1}$.

We first focus on the changes in of rem_x and $w_x^{\widehat{S_x}}()$ in interval $[t_{k-1}, t_k]$: When we consider the speed decrease ratio in the procedure 'Speed-Reduce' and $w_x^{\widehat{S_x}}(t_{k-1})$, we can easily conclude that T_x can not run more than $Y + w_x^{\widehat{S_x}}(t_{k-1})$ units, that is $0 \leq \Delta_t \leq Y + w_x^{\widehat{S_x}}(t_{k-1})$.

The value of $w_x^{\widehat{S_x}}(t_{k-1} + \Delta_h)$ where $0 \leq \Delta_h \leq \Delta_t$ can be easily computed. During the interval $[t_{k-1}, t_k]$ we execute with a speed $S'_x = \widehat{S_x} \cdot \frac{w_x^{\widehat{S_x}}(t_{k-1})}{w_x^{\widehat{S_x}}(t_{k-1})+Y}$; hence at $t_h = t_{k-1} + \Delta_h$,

$$
\begin{aligned}
w_x^{\widehat{S_x}}(t_h) &= w_x^{\widehat{S_x}}(t_{k-1}) - \frac{S'_x}{\widehat{S_x}} \Delta_h \\
&= w_x^{\widehat{S_x}}(t_{k-1})[1 - \frac{\Delta_h}{Y + w_x^{\widehat{S_x}}(t_{k-1})}] \geq 0
\end{aligned}
$$

for the preemption case, or $w_x^{\widehat{S_x}}(t_h = t_k) = 0$ for the completion case.

What is the value of $rem_x(t_{k-1} + \Delta_h)$? Let us define by $\beta_h(t, x)$ the sum of all rem_i values in the α-queue at time t, such that T_i has strictly higher EDF* priority than T_x. Observe that, by definition, $\epsilon_x(t) = \beta_h(t, x) + rem_x(t) - w_x^{\widehat{S_x}}(t)$. Hence:

$$
rem_x(t_{k-1}) = \epsilon_x(t_{k-1}) + w_x^{\widehat{S_x}}(t_{k-1}) - \beta_h(t_{k-1}, x) \tag{5.10}
$$

As time elapses, we consume from the α-queue starting from the head. We distinguish two cases:

- If $\Delta_h \leq \beta_h(t_{k-1}, x)$, we will only consume the slack of "higher" priority tasks in the α-queue, and rem_x will remain intact. Hence, using also induction assumption, we can obtain: $rem_x(t_{k-1} + \Delta_h) = rem_x(t_{k-1}) \geq w_x^{\widehat{S_x}}(t_{k-1}) \geq w_x^{\widehat{S_x}}(t_{k-1} + \Delta_h)$ for this special case.

- If $\beta_h(t_{k-1}, x) < \Delta_h \leq \Delta_t \leq Y + w_x^{\widehat{S_x}(t_{k-1})}$, we will start decreasing rem_x value only at $t = t_{k-1} + \beta_h(t_{k-1}, x)$. At $t = t_{k-1} + \Delta_h$, $rem_x(t_{k-1} + \Delta_h) = rem_x(t_{k-1}) - (\Delta_h - \beta_h(t_{k-1}, x))$, and using the equation (5.10), we get:

$$
\begin{aligned}
rem_x(t_{k-1} + \Delta_h) &= \epsilon_x(t_{k-1}) + w_x^{\widehat{S_x}}(t_{k-1}) - \Delta_h \\
&\geq Y + w_x^{\widehat{S_x}}(t_{k-1}) - \Delta_h = (Y + w_x^{\widehat{S_x}}(t_{k-1}))[1 - \frac{\Delta_h}{Y + w_x^{\widehat{S_x}}(t_{k-1})}] \\
&\geq w_x^{\widehat{S_x}}(t_{k-1})[1 - \frac{\Delta_h}{Y + w_x^{\widehat{S_x}}(t_{k-1})}] = w_x^{\widehat{S_x}}(t_{k-1} + \Delta_h)
\end{aligned}
$$

Thus, in both cases, we have $rem_x(t_{k-1} + \Delta_h) \geq w_x^{\widehat{S_x}}(t_{k-1} + \Delta_h) \geq 0$ for $0 \leq \Delta_h \leq \Delta_t \leq Y + w_x^{\widehat{S_x}}(t_{k-1})$ for task T_x. A geometric interpretation of the changes in $rem_x()$ and $w_x^{\widehat{S_x}}$ during the time interval $[t_{k-1}, t_{k-1} + \Delta_t]$ is presented in Figure 5.4.

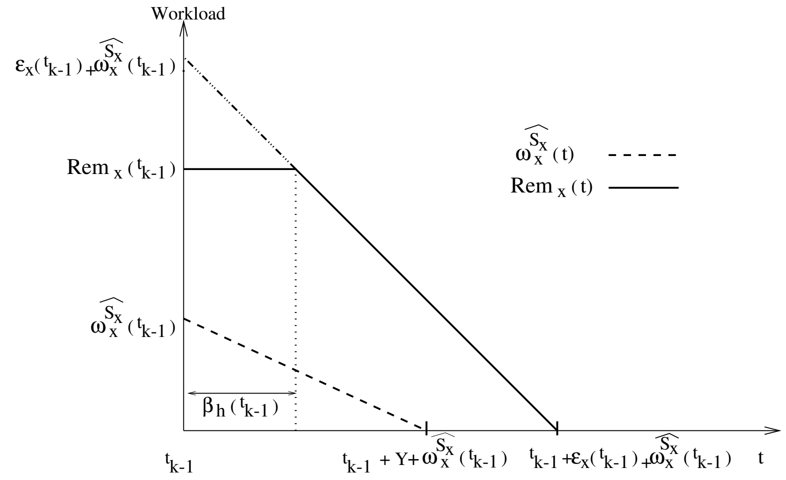


Figure 5.4: The changes in the workload

For any other task T_y which was also ready at $t = t_{k-1}$, but whose execution was delayed due to the high priority computation of T_x, $rem_y(t_{k-1} + \Delta_h) = rem_y(t_{k-1})$, since this value (correctly represented in the α-queue) can not be decreased before rem_x reaches zero (which can not happen before T_x completes, as we have shown above). But T_y was not executed either, hence $w_y^{\widehat{S_y}}(t_{k-1}) = w_y^{\widehat{S_y}}(t_{k-1} + \Delta_h)$. The induction assumption is that $rem_y(t_{k-1}) \geq w_y^{\widehat{S_y}}(t_{k-1})$, hence $rem_y(t_{k-1} + \Delta_h) \geq w_y^{\widehat{S_y}}(t_{k-1} + \Delta_h)$ as well. Finally, for any other task T_q which may arrive (and wait) while T_x is executing, clearly rem_q will remain equal to the $w_q^{\widehat{S_q}}$. From these results, we can infer that at any time t in the $(k+1)^{st}$ execution segment, $rem_i(t) \geq w_i^{\widehat{S_i}(t)}$ for any ready task T_i. Having established the statement for the $(k+1)^{st}$ segment, we prove the theorem.

□

Focusing exclusively on task completion times (in $\mathcal{S}^{can}$), the result above implies that in the actual schedule no task instance completes later than its completion time in $\mathcal{S}^{can}$ (which is feasible), proving the correctness of GDRA:

Corollary 3 *GDRA yields a feasible schedule under EDF* priority for a workload no greater than the worst-case workload.*

Note that any specific algorithm should specify the *exact* amount of earliness parameter Y, to use for speed reduction. One natural choice in Rule 5.2 of Figure 5.2 is to use $Y = \epsilon_x(t)$, that is, to reduce the speed so as to profit from the full earliness. We call this variation simply *Dynamic Reclaiming Algorithm* (DRA).

5.4.1 Incorporating One Task Extension (OTE) Technique

As presented in [92], one can further slow down execution when there is only one task in the ready queue and its worst case completion time (under the current speed) does not extend beyond the *next event* (next arrival/closest deadline of any task). For example, consider the following 2-task system: $C_1 = 100, P_1 = 200, C_2 = 300, P_2 = 600$. Again it is easy to see that $\bar{S} = S_{max} = 1.00$. The worst-case static schedule $\mathcal{S}^{can}$ is shown in Figure 5.5.

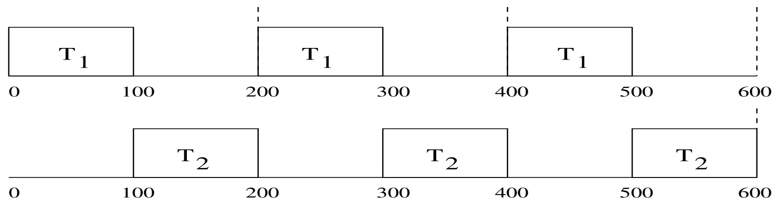


Figure 5.5: The static optimal schedule

In the actual execution, suppose that the schedule follows $\mathcal{S}^{can}$ until $t = 200$ and T_2 completes early at $t = 200$. When the second instance of T_1 arrives at $t = 200$, the α-queue looks as follows:

$$\alpha_1 : \quad id = 1, \;\; deadline = 400, \;\; rem = 100,$$
$$\alpha_2 : \quad id = 2, \;\; deadline = 600, \;\; rem = 200.$$

Note that even though T_2 completed, $rem_2(200)$ is still 200 since the updating of the α-queue only reduced it by 100 units. At this point ($t = 200$), the earliness $\epsilon_1(200) = 0$ and DRA would not reduce the speed. But we can actually reduce the speed S_1 safely to 0.5, effectively adding the interval [300,400] for the execution of $T_{1,2}$, and [500,600] for the execution of $T_{1,3}$ (see Figure 5.6). Note that, at $t_1 = 100$, T_2 is also the only ready task and its earliness $\epsilon_2(100) = 0$, yet, it cannot reduce its speed since $t_1 + w_2^{S_2} > 200$ which is the time of the next event.

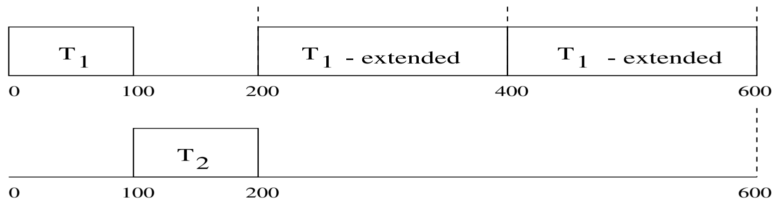


Figure 5.6: The actual schedule with OTE technique

One Task Extension (OTE) technique can be used in conjunction with any scheduling policy: to improve DRA, we add a new rule 5.3. Let NTA be the next arrival time of any task in the system after t, and recall that S_x is the speed from step 5.2 in DRA and t is the time T_x is dispatched.

5.3. If T_x is the only ready task and $Z = NTA - t - w_x^{S_x}(t) > 0$,
S_x = Speed-Reduce(T_x, Z, S_x).

In other words, reduce the speed of T_x so as to use the idle CPU up to time NTA. We call this improved technique DR-OTE. Clearly, the following holds.

Proposition 10 *If all instances meet their deadlines under DRA, they will also meet their deadlines under the algorithm DR-OTE.*

5.4.2 Experimental Results

In order to experimentally evaluate the performance of the DRA, we implemented a periodic scheduling simulator for EDF* policy. We implemented the following schemes for performance evaluation: (a) **Static** uses constant speed $\bar{S}$, and switches to power-down mode (i.e., $S = S_{min}$) whenever there is no ready task; (b) **OTE**: Static optimal speed scheme in conjunction with One Task Extension (but without dynamic reclaiming), and (c) **DRA**, which is implemented in two variations: with or without OTE facility.

In our experiments, we investigated the average performance of the schemes over a large spectrum of worst-case utilization (U_{tot}) and variability in actual workload. In particular, we first focused on the average energy consumption of 100 task sets, each containing 30 tasks. We repeated the experiments for 20- and 10-task sets as well. The periods of the tasks were chosen randomly in the interval [1000, 32000]. The minimum speed S_{min} is set to 0.1. The nominal speed $\bar{S}$ is set to U_{tot}, as the optimality of this choice was shown in Section 5.3. The variability in the actual workload is achieved by modifying the $\frac{WCET}{BCET}$ ratio (that is, the worst-case to best-case execution time ratio). Further, for each task set we simulated the execution up to LCM 10 times, where LCM is the Least Common Multiple of $P_1, \ldots, P_n$, and measured the average energy consumption per experiment using a cubic power/speed function [38]. We ran experiments where the actual execution time follows a normal or uniform probability distribution function. In case of the normal distribution, the mean and the standard deviation are set to $\frac{WCET+BCET}{2}$ and $\frac{WCET-BCET}{6}$ respectively, for a given $\frac{WCET}{BCET}$, as suggested in [92]. The choice of this mean and standard deviation ensures that, on the average, 99.7% of the execution times fall in the interval $[BCET, WCET]$. In case of the uniform distribution, the mean is again set to $\frac{WCET+BCET}{2}$. The results indicate that the general trends and patterns are rather similar for both distribution functions, with occasional minor differences. For the sake of clarity, in each of the following subsections concerning simulation results, we will present and comment on the case of normal distribution. The uniform distribution case will be presented with additional comments when necessary.

The first remarkable result is the following: Although the OTE scheme provides substantial improvements over no reclaiming techniques [92], throughout the entire spectrum, DR-OTE only provides a marginal (less than 1%) improvement over pure DRA. This result indicates that almost the entire power savings is obtained by initially committing to $\bar{S}$ which fully utilizes the CPU *and* to the dynamic reclaiming algorithm itself. To improve the readability of the graphs, we show below only the results of DR-OTE, since the results for the latter are almost identical to pure DRA.

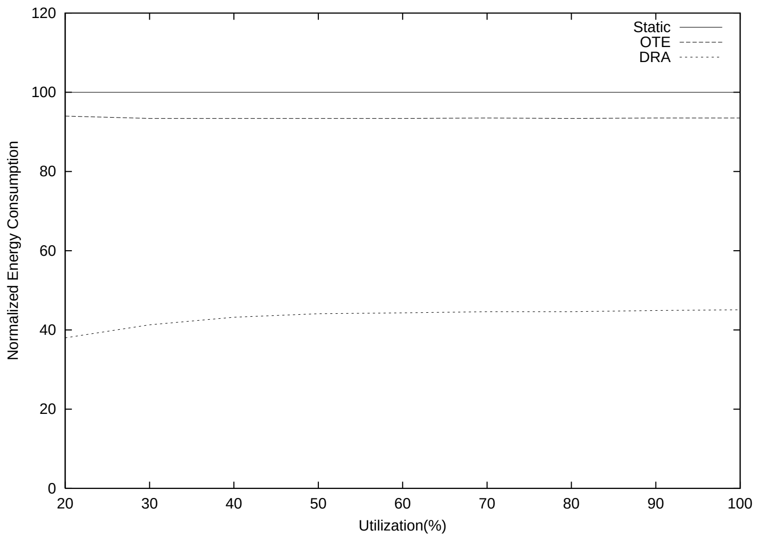


Figure 5.7: Normalized energy consumption (30 tasks). $\frac{WCET}{BCET} = 5$

Effect of Utilization: Figure 5.7 shows the energy consumption of the techniques varying with the utilization of the task set under S_{max}, when $\frac{WCET}{BCET}$ is set to 5. The results are normalized with respect to **Static** which does not reclaim unused computation time. One can observe the following major patterns:

- The normalized energy consumption of all three schemes are rather insensitive to the utilization of the task sets under S_{max}. This is due to the fact that, for a given scheme, the use of optimal nominal speed $\bar{S}$ results in having very similar *effective* utilization, even though the utilization under S_{max} would be different. As an example, the **Static** scheme fully utilizes the CPU under $\frac{WCET}{BCET} = 1$ and $\bar{S}$, throughout the spectrum.

- DRA has a definitive advantage over Static and OTE for all utilization values. Further, the energy consumption of a system using DRA is only around 40% of a system which uses Static or OTE.

- OTE performs better than Static, but the improvement is usually less than 10%.

Effect of $\frac{WCET}{BCET}$ ratio: The simulation results confirmed our prediction that the energy consumption would be highly dependent on the variability of actual workload. The (normalized) average energy consumption of the task sets, as a function of $\frac{WCET}{BCET}$ ratio (with $U_{tot} = 0.6$) is shown in Figure 5.8.

In terms of shape and percentage difference, the curves for other utilization values are fairly similar. From these experiments we arrived at the following observations and conclusions:

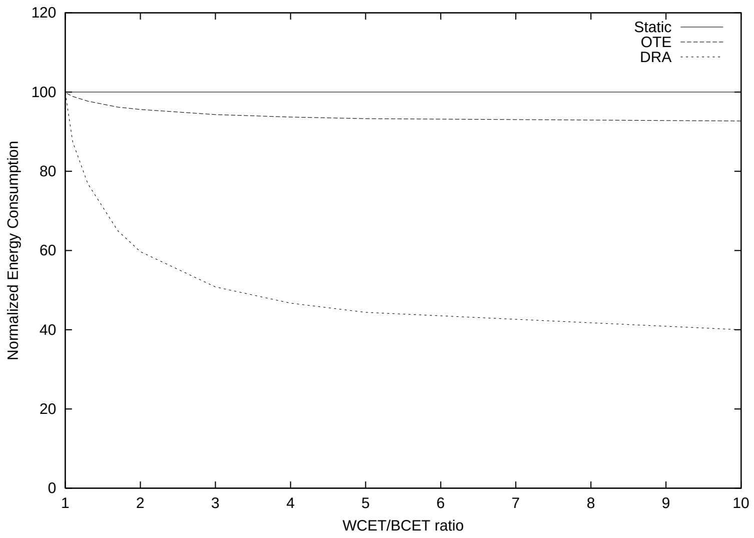


Figure 5.8: Effect of variability in actual workload (30 tasks); load = 60%

- Because there is no CPU time to reclaim dynamically, the energy consumption is the same for all three techniques when $\frac{WCET}{BCET} = 1$, as expected. However, once the actual workload starts differing from the worst-case workload with increasing $\frac{WCET}{BCET}$, OTE and DRA are able to reclaim unused computation time and they are able to save additional energy.
- The DRA is capable of providing considerably higher power savings than OTE; and the difference increases rapidly with $\frac{WCET}{BCET}$ ratio. For instance, the savings of DRA even for $\frac{WCET}{BCET} = 1.1$ is better than the performance of OTE throughout the entire spectrum.
- Once we increase the $\frac{WCET}{BCET}$ beyond 4, power savings of DRA continue to increase, but the improvement is not as impressive as the case where that ratio is ≤ 4. This is due to the fact that, the expected workload of the system converges rapidly to 50% of the worst-case workload with increasing $\frac{WCET}{BCET}$ ratio (remember that the mean of our probability distribution is $\frac{WCET+BCET}{2}$).

Effect of the number of tasks: To examine the effect of the number of tasks, we performed the same simulations for both 10 and 20-task sets (see Figures 5.9 and 5.10).

As it can be seen, the trends are similar to that of 30-task system. This is expected, since the main determinant of the workload is the variability in the actual workload. However, the percentages do differ from one case to the next. In particular, we see that DRA provides a slightly larger advantage when the number of tasks increases, because there are more opportunities for reclaiming unused slack-times with increasing number of tasks. On the other hand, as the number of tasks increases, the performance of OTE degrades as expected, since the likelihood of having a single task in the ready queue decreases.

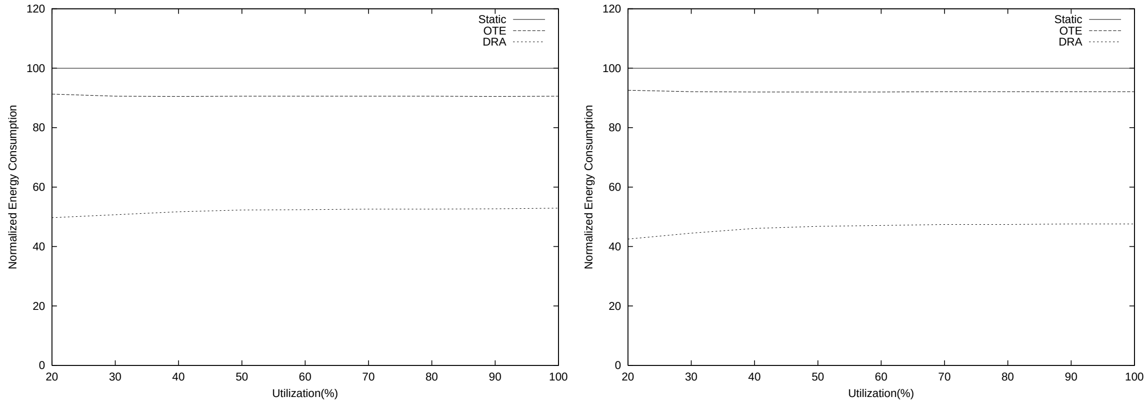


Figure 5.9: Normalized energy consumption for 10 (left) and 20 (right)tasks

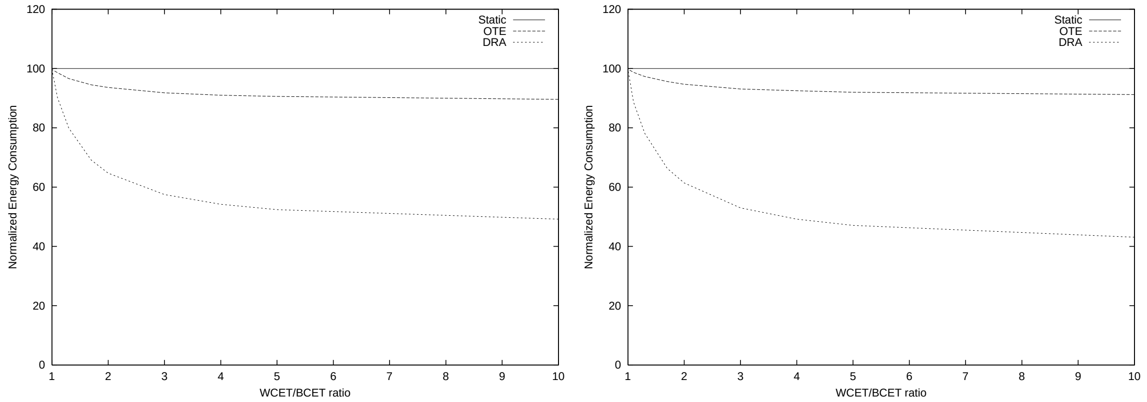


Figure 5.10: Effect of variability in actual workload for 10 (left) and 20 (right)tasks

Uniform Distribution:

When the actual execution times are determined according to the uniform probability distribution, as opposed to a normal distribution, the general patterns remain rather similar. The normalized energy consumption of three schemes with varying utilization and $\frac{WCET}{BCET}$ ratio in the context of 30-, 20- and 10-task sets are shown in Figures 5.11, 5.12 and 5.13. As before, for the utilization based experiments the $\frac{WCET}{BCET}$ was set to 5.0; and for the actual workload based experiments, the utilization was set to 60 %. As it can be seen, for a given number of tasks, at a fixed utilization or $\frac{WCET}{BCET}$ ratio, the average energy consumption is slightly higher for the uniform distribution. This slight increase is due to the fact that the likelihood of having large execution times close to $WCET$

is higher for the uniform distribution. Since the large execution times adversely affect and limit the reclaiming opportunities of **DRA**, the performance is degraded, though marginally.

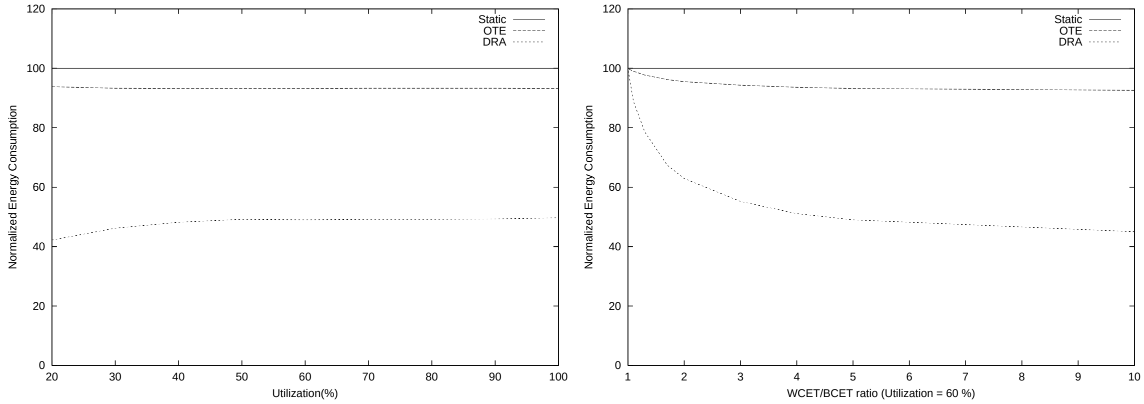


Figure 5.11: The effect of the utilization (left) and the $\frac{WCET}{BCET}$ ratio (right); (30 tasks - uniform distribution).

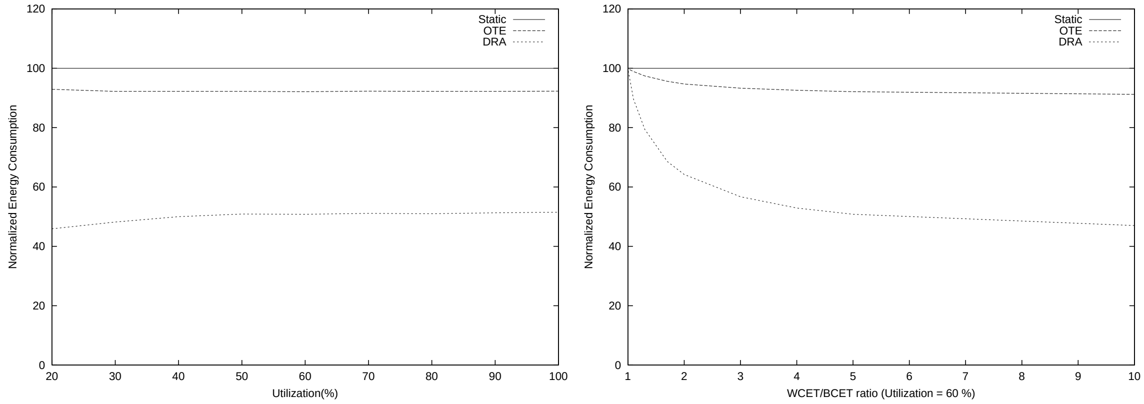
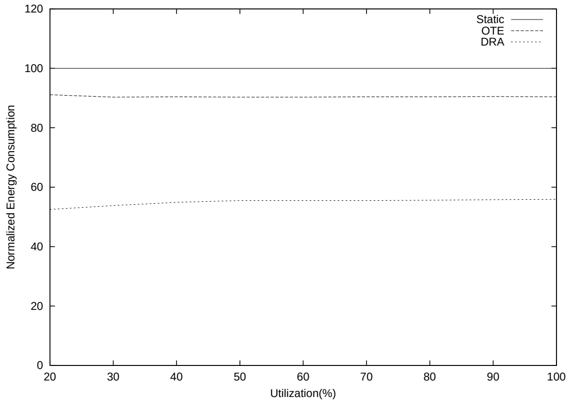


Figure 5.12: The effect of the utilization (left) and the $\frac{WCET}{BCET}$ ratio (right); (20 tasks - uniform distribution).

5.5 Aggressive Dynamic Reclaiming

The DRA and DR-OTE algorithms provide sound dynamic speed reclaiming mechanisms, however they guarantee feasibility by always being 'ahead' of the static worst-case optimal schedule $\mathcal{S}^{can}$ (i.e., tasks never actually start or finish after the scheduled time in $\mathcal{S}^{can}$). $\mathcal{S}^{can}$ is feasible at any time, yet it is optimal only under the assumption that all future instances will present their worst-case workload. Whenever, under constant speed, the actual execution times of a task's instances exhibit large variation, starting a task with this assumption can be too conservative. Instead, whenever the current system state suggests, we may **assume speculatively that the current and**



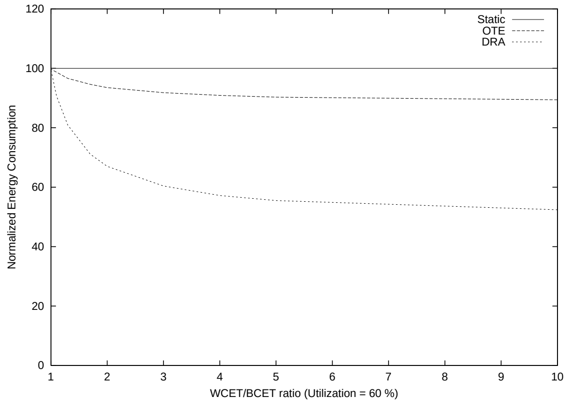


Figure 5.13: The effect of the utilization (left) and the $\frac{WCET}{BCET}$ ratio (right); (10 tasks - uniform distribution).

future instances will most probably present a computational demand which is lower than the worst-case. Hence, we can adopt an "aggressive" approach based on reducing the speed of the running task under certain conditions to a level which is even lower than the one suggested by DR-OTE. But this speculative move might shift the task's worst-case completion time to a point later than the one in $\mathcal{S}^{can}$ under an actual high workload. And if this pessimistic scenario turns out to be true, we should be ready to **increase the CPU speed beyond $\bar{S}$ later to guarantee feasibility of future tasks**. This would hamper significant power savings since the convexity of power/speed curve suggests a uniform speed to achieve a given average speed value over any interval of time. On the other hand, in case that the actual workload turns out to be lower than the worst-case, the actual schedule will *still* be ahead of $\mathcal{S}^{can}$, even with the low speed, thereby achieving even higher power savings.

Let us illustrate the aggressive scheme by a simple example.

$$C_1 = C_2 = 25, \;\; P_1 = P_2 = 100, \;\; g(S) = S^3$$

The static worst-case optimal schedule $\mathcal{S}^{can}$ (Figure 5.14) is obtained by computing $\bar{S} = U_{tot} = 0.5$.

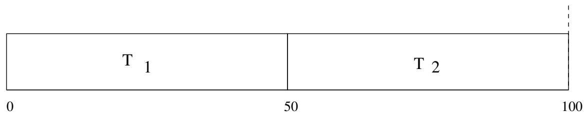


Figure 5.14: The static optimal schedule

Note that $\mathcal{S}^{can}$ is optimal only when the actual required CPU cycles $AC_i = C_i$ for $i = 1, 2$, for which case the total energy consumption is $E_1 = 50(0.5)^2 = 12.5$. In case that $AC_1 < C_1$

and/or $AC_2 < C_2$, $\mathcal{S}^{can}$ is no longer optimal. For example, if $AC_1 = 15$ and $AC_2 = 20$ then starting with $\bar{S}$ and applying the DR-OTE algorithm, yields the schedule shown in Figure 5.15.

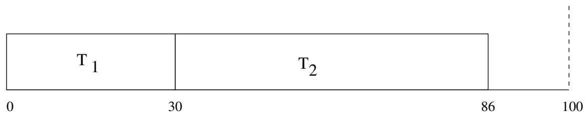


Figure 5.15: The schedule of DR-OTE technique

Note that because of its actual low workload, T_1 completes at $t = 30$. Then the DR-OTE algorithm is able to dynamically reclaim unused 20 units of time, effectively reducing S_2 to $\frac{25}{50+20} = \frac{25}{70} = 0.35$. But since $AC_2 < C_2$, T_2 also completes well before the deadline. In this case, the total energy consumption is[2] $E_2 = 30(0.5)^3 + 56(0.35)^3 + 14(0.1)^3 = 6.16$. However, by adopting an aggressive scheme, we can start executing T_1 with a speed lower than $S_1 = 0.50$, for example $S_1 = 0.35$. In this case, T_1 will complete at $t = 42.8$ and the dynamic reclaiming algorithm will further set S_2 to $\frac{25}{50+7.2} = \frac{25}{57.2} = 0.43$ (note that we cannot, at this point, set the speed S_2 any lower than this and still guarantee completion within the deadline). T_2 will complete at $42.8 + \frac{20}{0.43} = 89.3$. The schedule is depicted in Figure 5.16. The total energy consumption in

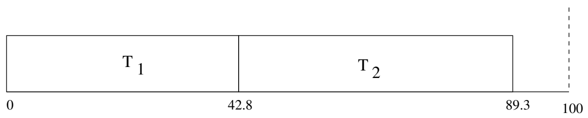


Figure 5.16: The schedule obtained via aggressive speed reduction

this aggressive scheme is $42.8(0.35)^3 + 46.5(0.43)^3 + 10.7(0.1)^3 = 1.83 + 3.7 = 5.55$, providing an additional 10% in power savings with respect to DR-OTE.

However, we should point out that under a very pessimistic scenario where $AC_1 = C_1$, the aggressive scheme would *have* to execute T_2 with high speed to prevent a deadline miss, resulting in a high energy consumption. We conclude this example by noting that it would not be possible to "aggressively" assign a speed lower than $\frac{25}{75} = 0.33$ to S_1, since doing so and having a worst-case workload for T_1 would result in a deadline miss for T_2, even under $S_2 = S_{max}$.

A powerful system design principle is to make the common case more efficient. This translates (in settings where the worst-case workload occurs only rarely) into having a power-efficient schedule for average or close to average cases, which can be achieved by reducing further the CPU speed. After having presented the rationale of aggressive speed management techniques, we should address and provide solutions for two important issues.

[2] Note that we are using the minimum CPU speed $S_{min} = 0.1$ in the interval [86, 100]

The first issue is **feasibility**: when we undertake such an aggressive speed reduction for T_x, we should be ready to guarantee the timing constraints of T_x and that of any other task, since the schedule may no longer be 'ahead' of $\mathcal{S}^{can}$. This guarantee may come at the cost of increasing the CPU speed beyond $\bar{S}$ later. However, any aggressive algorithm should (i) specify the amount of speed increase for different tasks, (ii) allow the use of other dynamic speed management techniques (e.g., DRA and OTE) in order to be able to gather the full benefit of aggressive schemes, and (iii) determine the *critical window* (the interval of time that the schedule is no longer ahead of $\mathcal{S}^{can}$), and prevent deadline misses during this window.

The second issue is the **determination of the aggressiveness level**: even though it may be possible to show the existence of a feasible schedule (under a very aggressive speed reduction for T_x) and of the prospect for high power savings at a given time, if such a move considerably increases the likelihood of creating large critical window(s), it may be reasonable to adopt a more conservative speed reduction, to decrease the probability of speed increases which cause high energy consumption. A natural solution is to use a pre-defined *speed reduction bound* (Sb) below which we never attempt to decrease the CPU speed during an aggressive speed adjustment. Observing that the "average workload" is an appropriate estimator for the actual computational demand, we choose to parameterize the aggressiveness level with respect to the *optimal speed under an average workload* (S_{optavg}). More specifically, S_{optavg} is the optimal speed for the workload where each instance requires exactly its average computational demand (determined by a probability distribution function). Generally, we may set Sb to $k \cdot S_{optavg}$, where k is a constant such that $S_{min} \leq Sb \leq S_{max}$ (i.e., $\frac{S_{min}}{S_{optavg}} \leq k \leq \frac{S_{max}}{S_{optavg}}$). Observe that changing k in this range provides a complete spectrum of "aggressive techniques". At one end of the spectrum, $k = \frac{S_{min}}{S_{optavg}}$ (which is usually much smaller than 1.0) corresponds to the "extreme aggressiveness" where we attempt to obtain the lowest speed level for the running task; this is only subject to feasibility which might be achieved later only by executing the following tasks with very high speeds (i.e., by this choice, we are supposing that the current workload will be well below the worst-case workload). At the other end of the spectrum, setting $k = \frac{S_{max}}{S_{optavg}}$ reflects the DR-OTE algorithm itself. Another main point in the spectrum is the scheme which limits the aggressiveness speed bound by exactly S_{optavg}, that is, $k = 1$; this reflects the view that slowing down the CPU below S_{optavg} will hurt the aggregate power savings in the long run.

5.5.1 Feasibility Guarantees for Aggressive Schemes

As mentioned above, when we attempt to aggressively reduce the CPU speed, we risk exceeding worst-case completion times of $\mathcal{S}^{can}$ in the current schedule, both for the running, ready and yet-to-arrive tasks. In general, to check the consequences of such an aggressive decision is a non-trivial problem (linked with response-time analysis complications of EDF), especially if it is to be addressed in a dynamic fashion, at run-time. In this study, we adopt a simple approach that restricts the aggressive power management to occur only when we can limit their effects upto the next event (arrival/deadline of any task). As the results in Section 5.5.4 below indicate, the aggressive schemes have the potential of providing additional power savings, even with this conservative feasibility test with limited horizon.

Whenever we can predict that the completion time of the currently ready task T_x will not extend beyond the next event (arrival/deadline), we can speculatively reduce the speed of T_x while guaranteeing that it will still complete before the next event (which is, by definition, earlier than or equal to the deadline of T_x). However, care must still be taken in order to guarantee the timely completions of other ready tasks which are waiting on the ready queue at a lower priority level than T_x, since the execution/completion of these tasks will be delayed until T_x completes.

A possible way to guarantee the feasibility in this case is *to increase* the speed of a suitable task T_y which will run after T_x. This is *effectively equivalent* to increasing the time allocation of T_x, while *decreasing* the time allocation of T_y by the same amount. Clearly, from this point on, the system cannot blindly decrease the speed of T_y to its original $\widehat{S_y}$ (i.e. we should also change $\widehat{S_y}$ for that instance).

One can even generalize this with the following: if $T_1, T_2, ..., T_r$ are the tasks that are guaranteed to run consecutively and *all* to complete before the next task arrival time (NTA) even under worst-case workload, we can arbitrarily swap CPU time allocations among them (in particular to reduce the speed of T_1 while increasing the speeds of one or more of $T_2, ..., T_r$). In fact, if this is the case, even the task following T_r (if any), namely T_{r+1}, may also provide a portion of its time allocation, if need arises. However, we must still guarantee that $T_1, T_2, ..., T_r$ will complete before NTA and T_{r+1} will complete no later than before, under the worst-case scenario. Further, in all these computations, we should take into account the slack-time of already completed tasks in the α-queue (with EDF* priority lower than T_1) that may contribute to the worst-case CPU allocation of $T_2, \ldots, T_r, T_{r+1}$ in the future through dynamic reclaiming. Finally, all these speed adjustments should adhere to S_{min}, S_{max} and Sb bounds.

The details of the approach can be best illustrated with a simple example. Suppose at time t_1, T_1 is the ready task with the highest EDF* priority, and that after the reclaiming step (5.2), its CPU allocation is $w_1^{S_1}$, and that its speed S_1 is still greater than both S_{min} and Sb. Further, assume that T_2, T_3 and T_4 are ready tasks with next highest priorities, such that $t_1 + \sum_{i=1}^{3} w_i^{S_i} < NTA$ but $t_1 + \sum_{i=1}^{4} w_i^{S_i} > NTA$ (see Figure 5.17). For the sake of simplicity, we will assume first that at $t = t_1$, the α-queue does not contain an element with EDF* priority between d_1^* and d_4^*; we will comment on the relaxation of this assumption at the end.

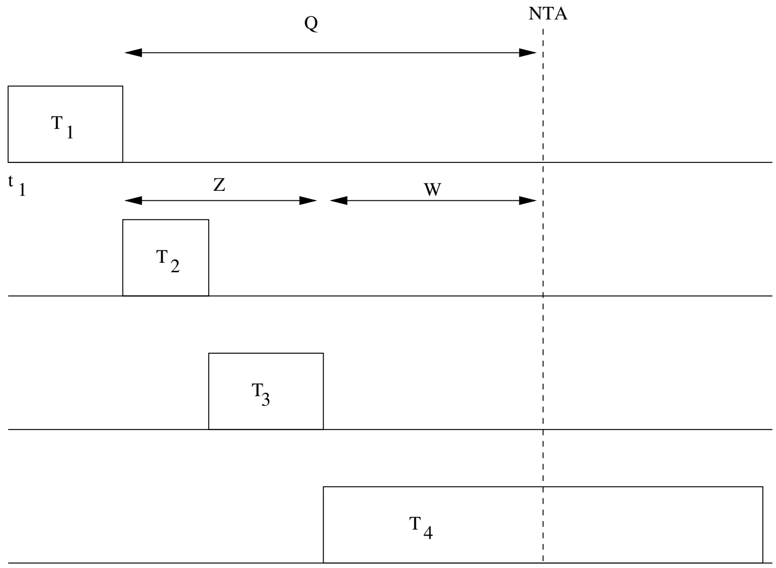


Figure 5.17: The schedule prior to speculative speed adjustment

During an aggressive move, we attempt to transfer up to Q units of CPU time to T_1 (and hence, increase its speed); and it would still complete before NTA. In particular, T_2 and T_3 can provide a fraction of this amount (effectively, up to Z units), as long as their speed remains lower than S_{max} despite the increase. In addition, T_1 can try to transfer CPU time from T_4, following T_2 and T_3. However, this amount can *never exceed* $Q - Z = W$ units, regardless of the amount of previous transfers: Doing otherwise would shift the worst-case completion time of T_3 beyond NTA, with unpredictable consequences. In summary, we may aggressively try to transfer up to Q units of CPU time from T_2 and T_3 (the S_{max} constraint would allow a much smaller amount, in reality); but the amount transferred from T_4 can not exceed $Q - Z$ units.

Now, let us relax the assumption that the α-queue does not contain an element with EDF* priority between d_1^* and d_4^* at $t = t_1$. For example, assume that T_3 is not ready at $t = t_1$, it has already completed though it is in the α-queue with (unused computation time) rem_3. Further, assume that rem_3 is equal to the same amount of worst-case remanining execution time of T_3 in Figure 5.17. According to DR-OTE, T_1 would not be able to use the slack-time of T_3 which has lower EDF* priority. However, as it can be easily seen, T_1 can transfer this amount as well during an aggressive move, and the effects can be checked until the next event. A main difference with the case of previous paragraph is that the S_{max} constraint is irrelevant for the reserved CPU time of already completed tasks: T_1 can consume as much as possible until the next event. Needless to say, utmost care must be still exercised when a slack-time reservation exceeds the NTA boundary (such as T_4 above) and the transfer amount should not shift other ready tasks beyond NTA (it should be still less than $Q - Z$).

To incorporate all these steps, we add a new rule 5.4, to the previous algorithm, thereby obtaining the new algorithm **Aggressive-DR**:

5.4. If $Z = NTA - t - w_x^{S_x}(t) > 0$ and there are other ready tasks in addition to T_x, call `Aggressive-Speed-Adjustment`, shown in Figure 5.18.

Procedure *Speed-Increase* (Figure 5.19) increases the speed of T_x so as to remove at most H units of time allocation under worst-case remaining workload of T_x with respect to the speed S, subject to S_{max}. In procedure Aggressive-Speed-Adjustment, whenever T_1 transfers slack-time from T_j, we perform the speed increase for T_j, increasing the *nominal speed of* T_j. Whenever T_j is about to be dispatched, its current speed will be set to $\widehat{S_j}$ by rule 5.0; rules 5.1 and 5.2 should consider this new (increased) speed when trying to reduce speed due to a (possible) earliness detection. Finally, T_j should assume the new nominal speed $\widehat{S_j}$ when it returns from preemption, since this is the lowest speed known to guarantee a feasible schedule in the case where every task presents its worst-case load to the processor after aggressive speed adjustments. However, we underline that the nominal speed $\widehat{S_j}$ of *future* instances of T_j will remain unchanged and equal to $\bar{S}$.

Proposition 11 *The* `Aggressive-Speed-Adjustment` *routine does not hurt the feasibility of the schedule.*

Proof: First note that, prior to the invocation of the aggressive routine, the dynamic reclaiming procedure was invoked, thanks to which T_1 reclaimed the unused computation time of any task T_j in the α-queue with priority $d_j^* \leq d_1^*$ at Step 5.2 of Figure 5.2. In case that S_1 is already reduced to $max\{S_{min}, k \cdot S_{optavg}\}$ after dynamic reclaiming, or if the worst-case completion time of T_1 extends beyond NTA, then no aggressive speed reduction will be performed, and the schedule will remain feasible in virtue of the correctness of DR-OTE.

Otherwise, the algorithm attempts transferring up to

$$Q = min\{[\frac{S_1}{max(S_{min}, k \cdot S_{optavg})} - 1]w_1^{S_1}(t), NTA - t - w_1^{S_1}(t)\} \tag{5.11}$$

units of CPU time from other tasks thereby adhering to S_{min} and Sb bounds and not exceeding NTA. Let us denote by S_i and S_i' the speed of T_i prior to and after the aggressive speed adjustment, respectively.

The algorithm determines the largest sequence **Z** of tasks $T_2, \ldots, T_r$ of which the total worst-case execution time $Z = \sum_{i=2}^{r} w_i^{S_i}(t)$ does not exceed the requested CPU time Q (see Figure 5.17 for an example where $r = 3$ and $m = 4$). If $w_2^{S_2}$ is already larger than Q, then clearly $Z = 0$ and $r = 1$.

During the *while* loop the CPU-time transfer is performed through the tasks in **Z**, as long as the total transferred amount is less than Q. $T_2, \ldots, T_r$ can provide at most $Z \leq Q$ units of

Procedure Aggressive-Speed-Adjustment

Notation: The algorithm is invoked at time t. The ready task with the highest EDF* priority is denoted by T_1. The other tasks that are ready, or that are completed but have their unused computation time in the α-queue with EDF* priority lower than that of T_1, are denoted by $T_2, ..., T_m,\ 2 \leq m \leq n$, in decreasing order of priorities. Throughout the algorithm, at the cost of a slight abuse of notation, we will also use the expression $w_i^{S_i}(t)$ to refer to $Rem_i(t)$ value of any completed task T_i in the α-queue at time t. The current speed assignments are denoted by $S_1, ..., S_m$, and the next task arrival after t will occur at time NTA.

Algorithm:

- If $S_1 \leq max\{S_{min}, k \cdot S_{opt\,avg}\}$ return; (that is, we should not decrease the speed any further)
- Determine the maximum amount of additional CPU time, Q, that can be assigned to T_1, subject to S_{min} and the aggressiveness level constraints:
$$Q = [\frac{S_1}{max(S_{min}, k \cdot S_{opt\,avg})} - 1] w_1^{S_1}(t).$$
- Adjust Q in order not to extend beyond NTA:
if $NTA - t - w_1^{S_1}(t) < Q$ then $Q = NTA - t - w_1^{S_1}(t)$.
- $Q_{actual} = 0$ (already transferred slack amount).
- If $w_2^{S_2}(t) \geq Q$ then $\{r = 1; Z = 0\}$
else find the largest r $(2 \leq r \leq m)$ such that $Z = \sum_{i=2}^{r} w_i^{S_i}(t) \leq Q$.
- Increase the speed of $T_2, ..., T_{min(m,r+1)}$ while reducing the speed of T_1:
  - $j = 2$
  - while $(j \leq min(m, r+1)\ and\ Q_{actual} < Q)$
    * if $(j < r+1)$ then $Requested\_time = Q - Q_{actual}$
    else $Requested\_time = Q - Z$
    * if T_j is ready then:
      · S_j = Speed-Increase$(T_j, Requested_time, \widehat{S_j})$ (see Figure 5.19)
      · $B = (\frac{S_j}{\widehat{S_j}} - 1) \cdot w_j^{\widehat{S_j}}$
      · $\widehat{S_j} = S_j$ (that is, commit to the new S_j as the default speed of that instance)
    * if T_j is completed but is in the α-queue then $B = min(Requested\_time, Rem_j)$
    * $j = j + 1$
    * $Q_{actual} = Q_{actual} + B$
    * S_1 = Speed-Reduce(T_1, B, S_1)

Figure 5.18: Aggressive Speed Reduction Procedure

Procedure Speed-Increase(T_x, H, S)
1. $S_x = \frac{w_x^S + H}{w_x^S}$
2. If $S_x > S_{max}$ then $S_x = S_{max}$;
3. return S_x

Figure 5.19: Speed Increase Procedure

CPU time. The task T_{r+1} (T_4 in Figure 5.17), whose worst-case completion time in fact exceeds NTA, can provide the remaining $Q - Z$ units if the speed bound is not already attained. After these transfers, T_1 will run in a non-preemptive fashion and complete before the next event.

As to tasks in **Z** (if it is not empty), first observe that

$$(w_1^{S_1'} + \sum_{i=2}^{r} w_i^{S_i'}) - (w_1^{S_1} + \sum_{i=2}^{r} w_i^{S_i}) \leq Q - Z \qquad (5.12)$$

since during the iterations $2 \ldots r$ of the *while* loop, the CPU time of a task in **Z** is reduced by the same amount that T_1's allocation is increased, and finally, the task T_{r+1} can provide at most an additional $Q - Z$ units. So, even the lowest priority task in **Z** will complete at the latest at $t + w_1^{S_1} + \sum_{i=2}^{r} w_i^{S_i} + Q - Z = t + w_1^{S_1} + Q$, which, by definition, does not exceed NTA. Thus, $T_2, \ldots, T_r$ will also be completed by NTA.

On the other hand, the task T_{r+1} will also complete at the latest at the same time as the original schedule even under a worst-case workload, because its speed has been adjusted (increased) accordingly when it provided the CPU-time to T_1. In addition, any task T_x that may become ready at or after NTA cannot be affected in an adverse manner: if $d_x^* < d_{r+1}^*$ then it has high priority and its completion time does not depend on the interference of T_{r+1}. Or else, if $d_x^* > d_{r+1}^*$, T_{r+1} will present at most the same interference as before: this follows from the fact that we effectively transferred T_{r+1}'s CPU allocation *before* NTA to T_1, which definitely completed (by NTA).

Finally, we underline that the consideration of any completed task in the α-queue as opposed to any ready task does not affect the above discussion: its reserved CPU time can be used by T_1 (and without considering the S_{max} constraint) in the aggressive mode. As a similar reasoning as above easily suggests, this may not result in a deadline miss.

□

5.5.2 Experimental Evaluation of the Aggressive Scheme

We conducted experiments to assess the performance of the aggressive scheme (abbreviated by AGR), in the same setting as Section 5.4.2. The speed bound Sb for the speculative speed adjustment is equal to S_{optavg}, that is, the aggressiveness factor k is set to 1. In Figure 5.20, the relative energy consumption of AGR with respect to DRA is shown, for 30-task sets and normal distribution, as a function of the utilization. The results point out to a consistent advantage of AGR over DRA throughout the spectrum (around % 15). The improvement decreases ad the utilization approaches 100%, where all tasks assume a nominal (default) speed $\bar{S} = S_{max} = 1.0$ and aggressive speed reduction at the expense of increasing the speed of others is not always possible.

The effect of variability in actual workload is shown in Figure 5.21. Again, AGR provided better performance than DRA with various $\frac{WCET}{BCET}$ ratios. Increasing this ratio improves the relative performance of AGR, since the speculative moves are justified more frequently. When the number of tasks is smaller (see Figures 5.22 and 5.23), the energy savings due to the reclaiming phase drop for both schemes, as explained in Section 5.4.2; similarly the gains due to the aggressive moves tend to slightly increase with the number of tasks, since the likelihood of finding a suitable candidate from which CPU-time can be transferred is higher.

When the distribution function is uniform, the energy savings decrease when compared with the normal distribution[3]; however, the advantage of AGR over DRA is still consistent (Figures 5.24, 5.25, and 5.26).

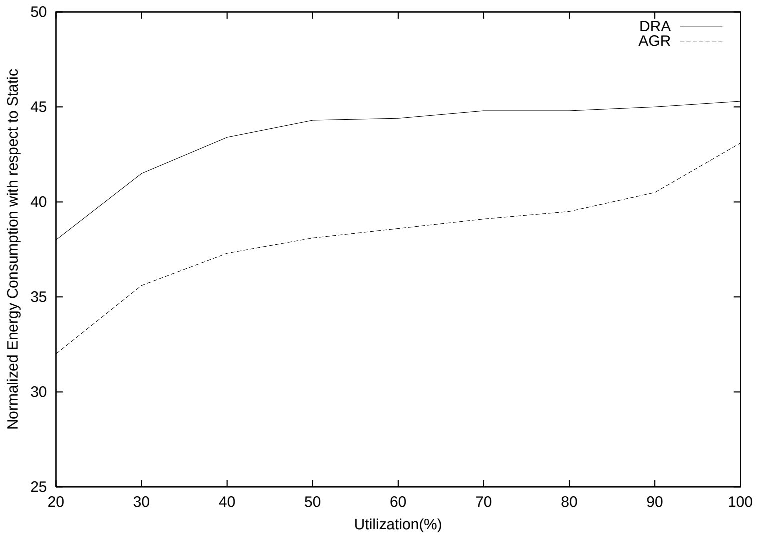


Figure 5.20: Normalized energy consumption (30 tasks). $\frac{WCET}{BCET} = 5$

5.5.3 Further uses of Speed Bound Restrictions

Another possible approach is to use the aggressive scheme, but to adhere to the 'parameterized speed bound' **even when reducing the speed in Step 5.2 through dynamic reclaiming**.

[3] As before, for the utilization based experiments the $\frac{WCET}{BCET}$ was set to 5.0; and for the actual workload based experiments, the utilization was set to 60%

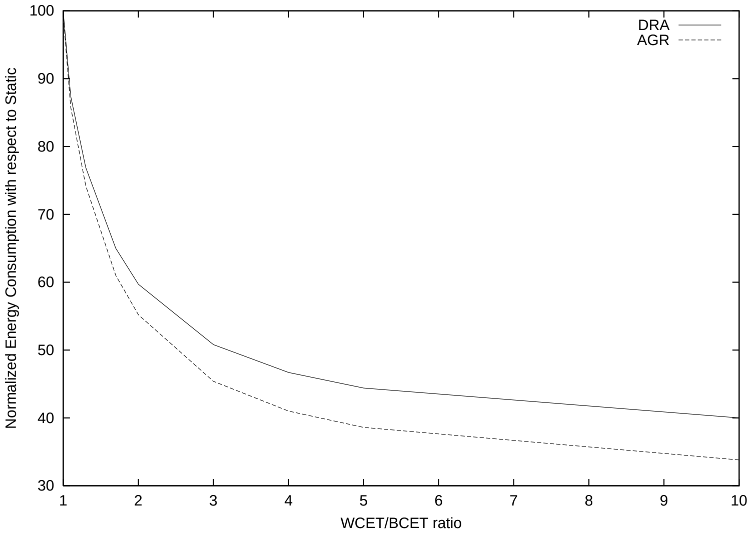


Figure 5.21: Effect of variability in actual workload (30 tasks); load = 60%

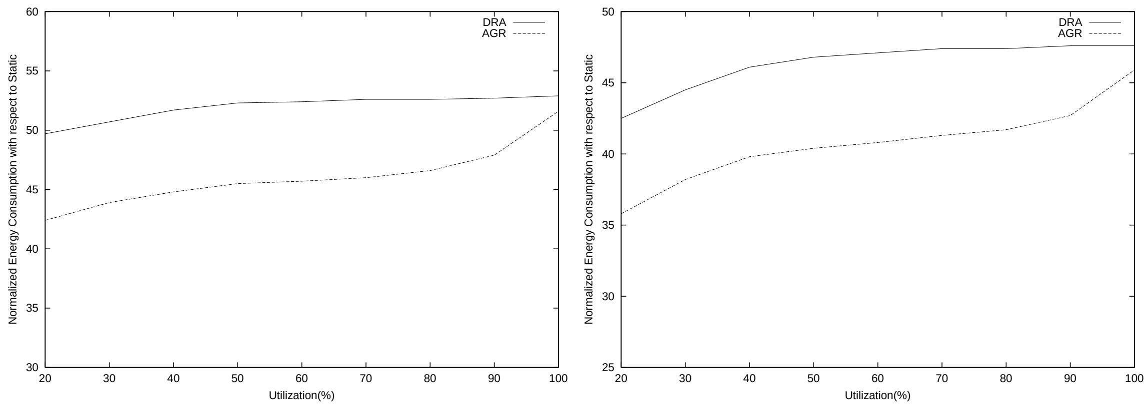


Figure 5.22: Normalized energy consumption for 10 (left) and 20 (right)tasks

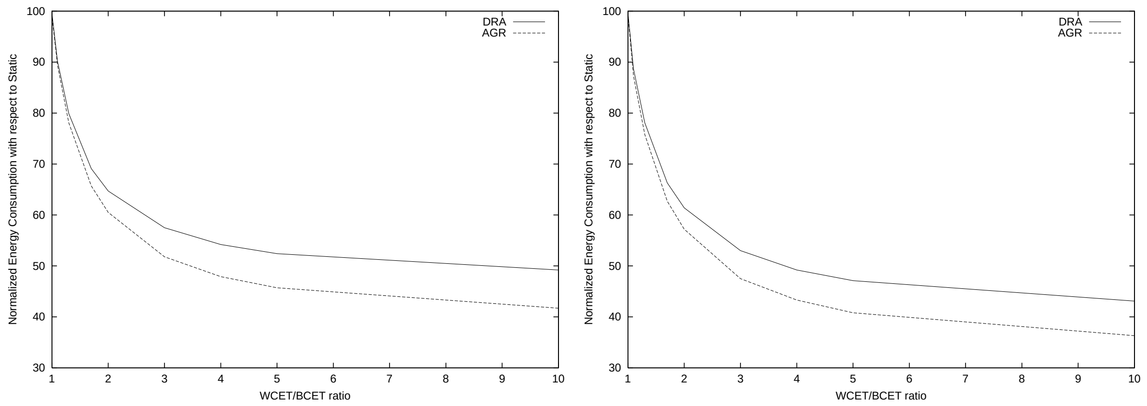


Figure 5.23: Effect of variability in actual workload for 10 (left) and 20 (right)tasks

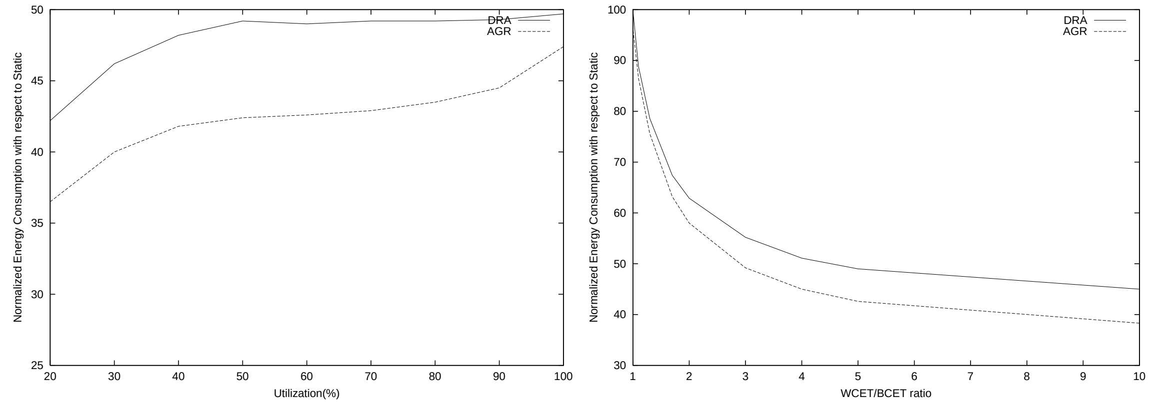


Figure 5.24: The effect of the utilization (left) and the $\frac{WCET}{BCET}$ ratio (right); 30 tasks - uniform distribution.

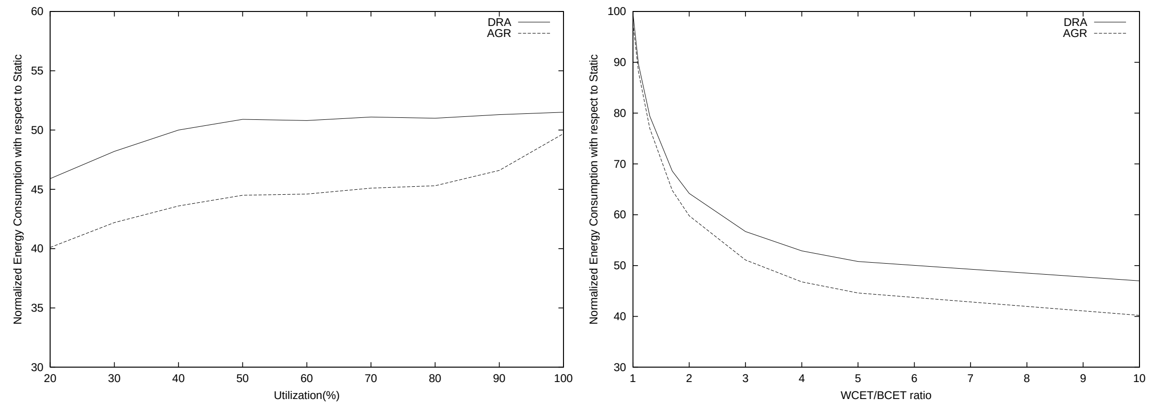


Figure 5.25: The effect of the utilization (left) and the $\frac{WCET}{BCET}$ ratio (right); 20 tasks - uniform distribution.

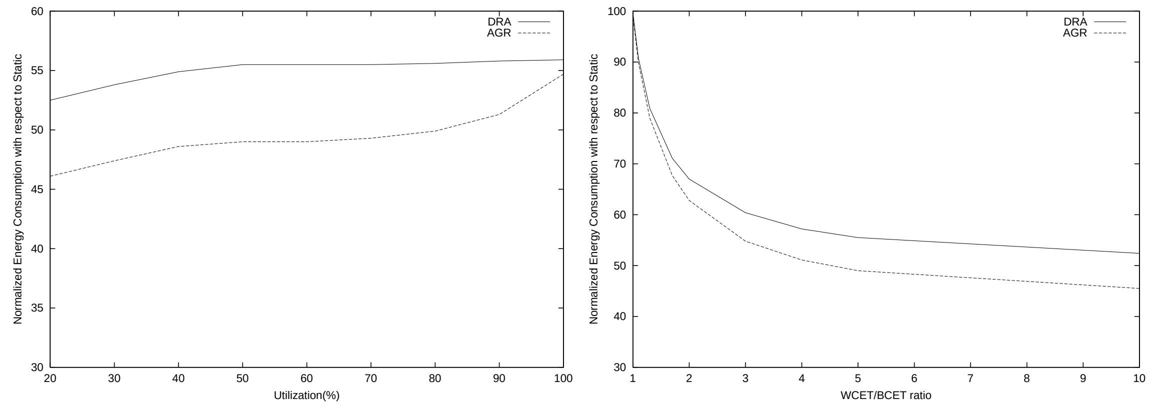


Figure 5.26: The effect of the utilization (left) and the $\frac{WCET}{BCET}$ ratio (right); 10 tasks - uniform distribution.

This approach assumes that reducing the speed below $k \cdot S_{optavg}$ will hurt the total performance in the long run, and prevents doing so even when the earliness factor would justify doing so. To distinguish two variations of the aggressive scheme, we will denote the original scheme and the new variation by **Aggressive-DR-1** and **Aggressive-DR-2**, respectively (or, AGR1 and AGR2, for short).

The correctness of the new scheme follows from the correctness of Aggressive-DR-1, since Aggressive-DR-2 never slows down the processor more than Aggressive-DR-1.

5.5.4 Experimental Evaluation of AGR1 and AGR2

In this section, we present results of simulations performed to compare algorithms AGR-1 and AGR-2. The simulation settings are identical to those of Section 5.4.2.

The effect of the utilization and $\frac{WCET}{BCET}$ ratio (denoted also by r in the graphs) on 30-, 20- and 10-task sets are presented with normal distribution in Figures 5.27, 5.28, 5.29, 5.30, 5.31 and 5.32. The relative performance of AGR1 versus AGR2 is extremely similar to these results for uniform distribution; hence they are omitted. In these graphs, the performance curves of AGR1 are shown by lines, while AGR2 is depicted through points.

As it can be easily seen, the performance of aggressive schemes are hardly distinguishable. We observe that the *relative* performance of AGR2 over AGR1 tends to improve with the increase in the number of tasks and/or utilization. Remember that, in AGR2, a task is not allowed to reclaim or aggressively transferring CPU-time from other tasks if the speed bound Sb is reached, with the implicit assumption that there may be more suitable tasks that can further benefit from reclaiming. However, in low utilization (hence, in low nominal speed values), this decision may not be always justified since other tasks may be already executing at or close to S_{min}. Similarly, when the number of tasks increases, the probability of having another task with suitable (late) deadline to reclaim the unused computation time is higher.

The effect of the aggressiveness level: Unlike the utilization and $\frac{WCET}{BCET}$ ratio, changing the aggressiveness level deeply affects the results, as shown in Figure 5.33 for normal distribution (see Figure 5.34 for uniform distribution). The curves shown are for 60% utilization and $\frac{WCET}{BCET} = 5$; other parameter settings have very similar behavior. The performances of DRA and Static are insensitive to the parameter k. The maximum power savings is are obtained with algorithm Aggressive-DR-2 typically when $k = 0.9$. This represents a further 5% improvement over $k = 1$, yielding a net advantage of 20% over DRA. Aggressive-DR-1 reaches its minimum energy consumption usually with $k = 1$. Further, the curve suggests that unbounded or extreme aggressiveness (small values of k) hinders the power savings: for instance, both schemes consume 60% more energy than DRA for $k \leq 0.2$.

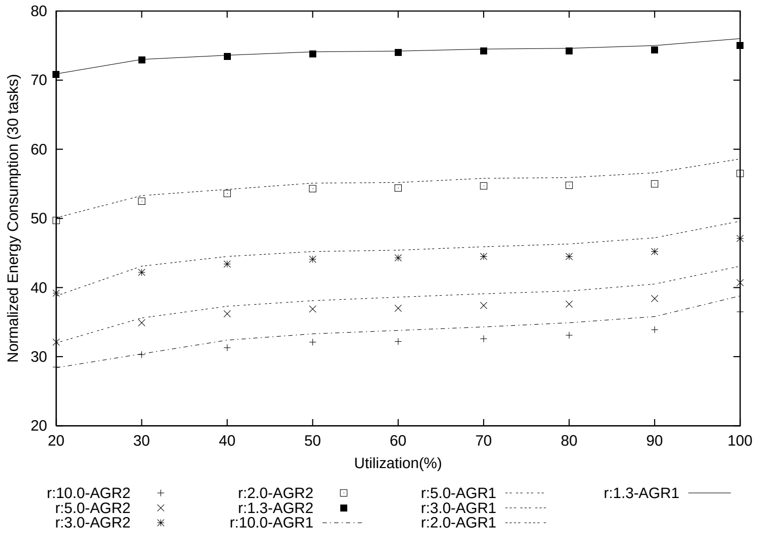


Figure 5.27: Effect of the utilization(30 tasks)

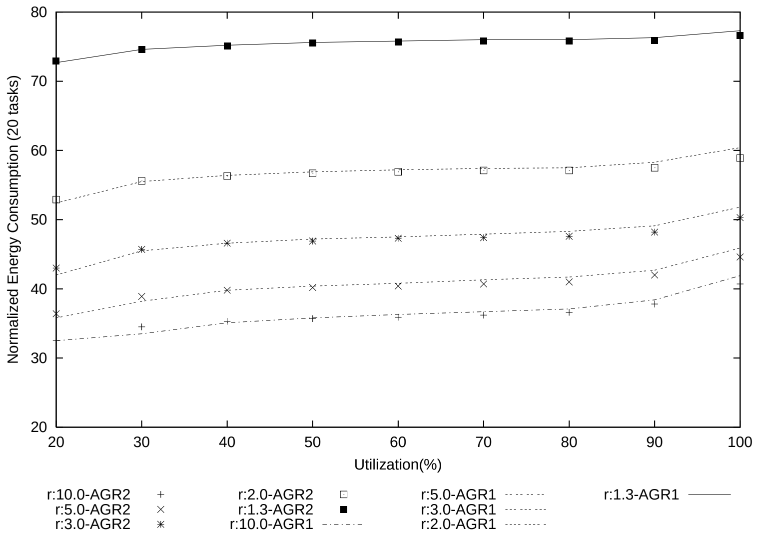


Figure 5.28: Effect of the utilization(20 tasks)

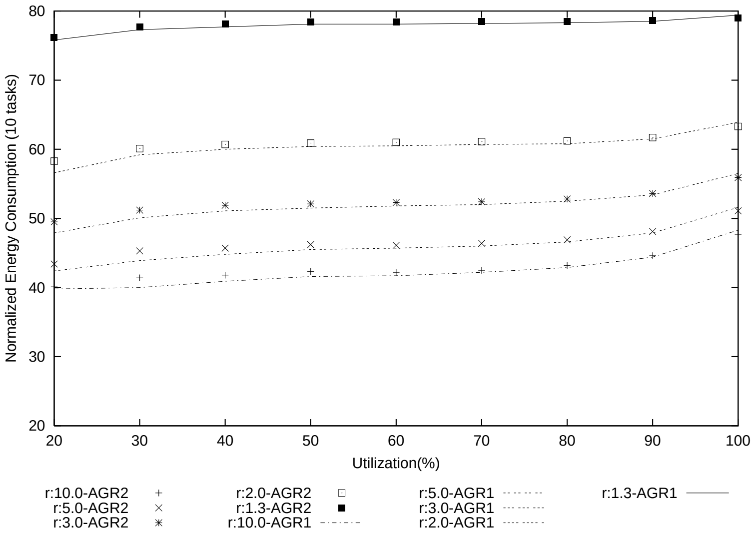


Figure 5.29: Effect of the utilization(10 tasks)

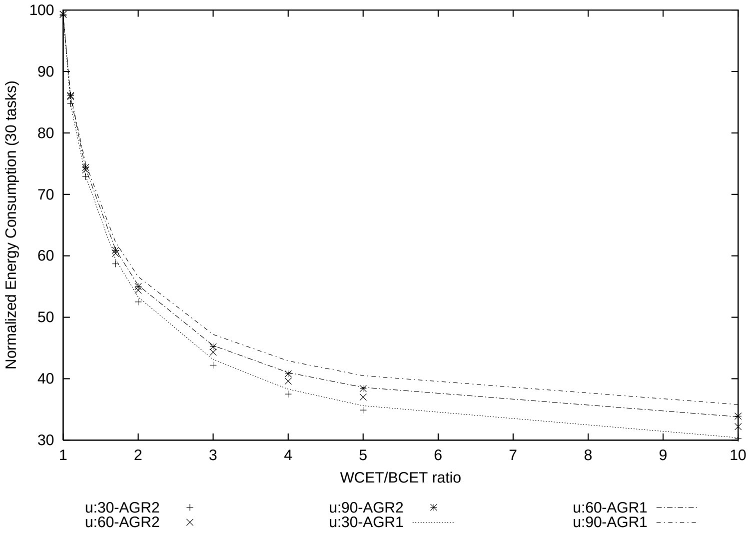


Figure 5.30: Effect of the $r = \frac{WCET}{BCET}$ ratio (30 tasks)

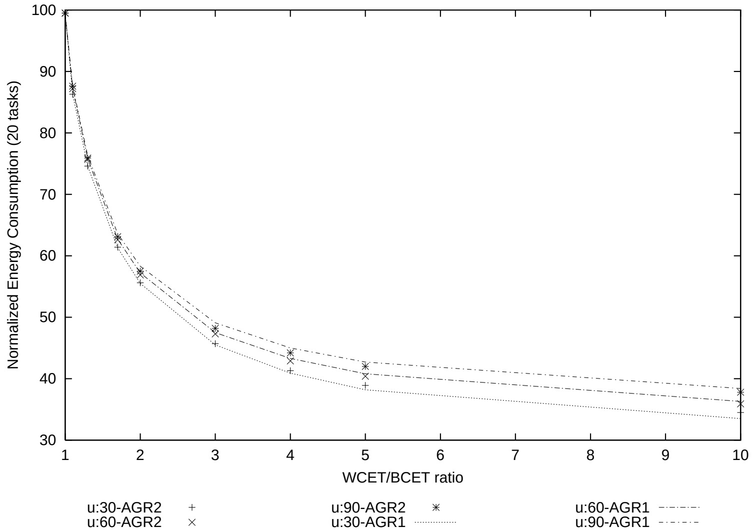


Figure 5.31: Effect of the $r = \frac{WCET}{BCET}$ ratio (20 tasks)

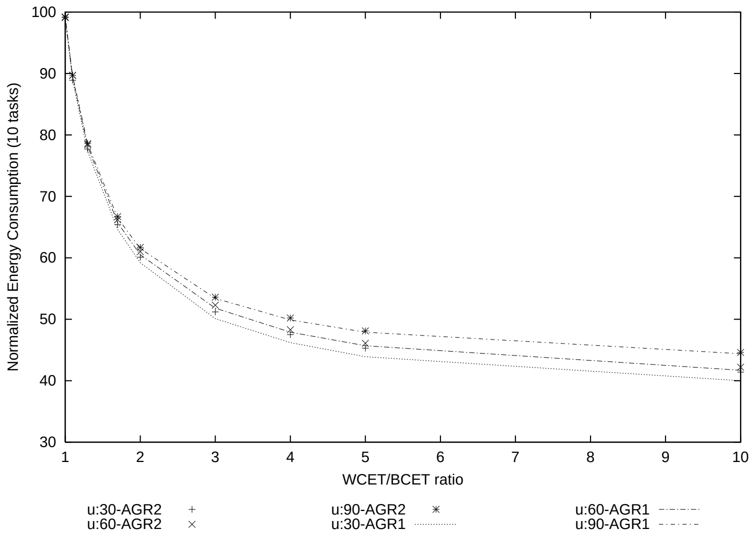


Figure 5.32: Effect of the $r = \frac{WCET}{BCET}$ ratio (10 tasks)

Yet, as we increase the value of k and move towards more 'balanced' aggressiveness levels, the aggressive schemes become preferable to DRA: Aggressive-DR-1 and Aggressive-DR-2 outperforms DRA, for $k \geq 0.75$ and $k \geq 0.7$, respectively. After the power savings reach their maximum at $k = 0.9$ (for Aggressive-DR-2) and $k = 1.0$ (for Aggressive-DR-1), the performance starts to degrade. Remarkably, for $k \geq 1.1$, Aggressive-DR-2 consumes considerably higher energy than Aggressive-DR-1: this is due to the fact that when the algorithm is run with large values of k, the algorithm is reluctant to reclaim or transfer CPU-time, even when the speed is higher than S_{optavg}. Aggressive-DR-1 does not suffer from this effect, since it automatically uses the earliness information to perform an initial speed reduction and considers the speed bound Sb only when aggressively reducing speed. Hence even for large values of k, Aggressive-DR-1 remains better than DRA, and is guaranteed to converge to it for $k = \frac{\bar{S}}{S_{optavg}} = \frac{2}{1+\frac{BCET}{WCET}}$, which is 1.66 for this example. On the other hand, Aggressive-DR-2 converges to OTE (not shown in Figures 5.33 and 5.34) for the same value; this is because, the actual speed starts with $\bar{S}$, and it never goes above Sb, and the CPU speed is reduced only through the OTE technique.

In summary, keeping k in the range $[0.9, 1]$ and committing to an aggressiveness level which aims to achieve very close to S_{optavg} produces best results, yielding *additional* (i.e., beyond DRA or DR-OTE) energy savings which may be as high as 20%.

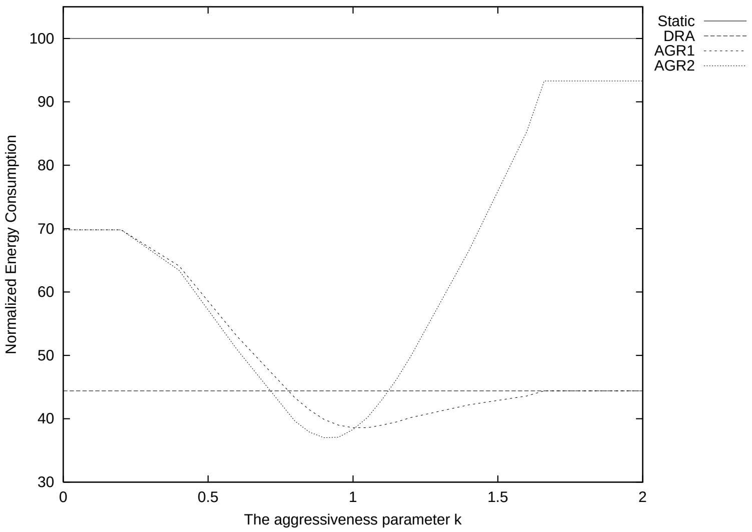


Figure 5.33: Effect of bounding factor k in Aggressive Schemes (30 tasks); utilization = 0.6 and $\frac{WCET}{BCET} = 5$

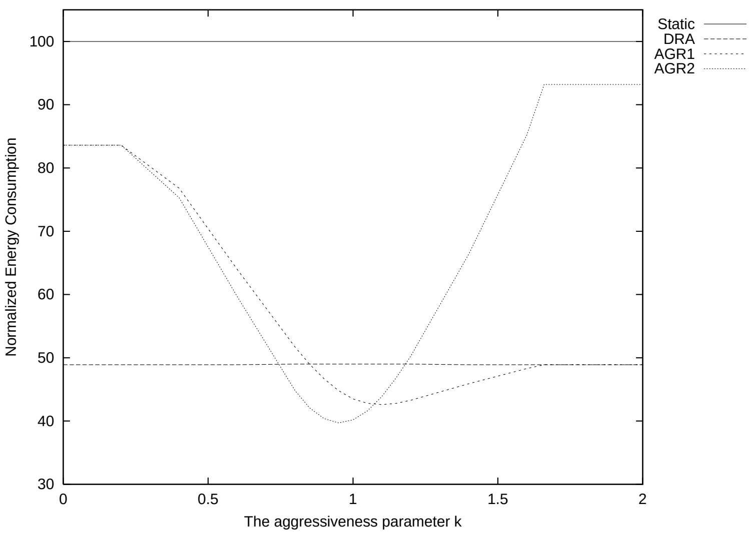


Figure 5.34: Effect of bounding factor k in Aggressive Schemes (30 tasks); utilization = 0.6 and $\frac{WCET}{BCET} = 5$

5.6 Chapter Summary

In this chapter, we first addressed how to approach the real-time power-aware scheduling problem using the reward-based scheduling framework. We showed that the problem is, in fact, equivalent to the reward-based scheduling problem with concave reward functions for a given (worst-case) workload.

However, many real-time tasks usually complete without consuming their worst-case workload. Thus, by dynamically monitoring the early completions and by adjusting the CPU speed, it is possible to obtain additional savings. We proposed a generic *dynamic reclaiming* algorithm, and proved that no task misses its deadline after any speed adjustment decision.

We also proposed an *aggressive scheduling* strategy for systems where the information about the *expected* workload is available. The strategy consists in speculatively reducing the CPU speed, anticipating that the likelihood of a worst-case execution is not high. However, we also showed how to adjust (increase) the CPU speed later to meet the timing constraints, in case that the prediction turns out to be incorrect.

We performed extensive simulations to evaluate our dynamic reclaiming and aggressive scheduling schemes. The results show that it is possible to obtain additional 60% gains over the static solution, by using a combination of both techniques.

Chapter 6

Fault Tolerance and Optimality for Aperiodic Reward-Based Tasks

In this chapter, we address incorporating error recovery into the aperiodic RB scheduling. The solutions we present are optimal in the sense that they address simultaneously the issues of provisioning for errors in mandatory parts and maximizing the total reward of the system. Further, our solution has the desirable property that it eliminates any dynamic re-adjustment of the schedule (hence, run-time overhead), as long as no errors are encountered. This is in contrast with the previous study on incorporating error recovery into the aperiodic RB scheduling [12], where the schedule needs to be updated whenever a mandatory task completes successfully (without failure).

We assume that at most k faults may occur during the execution of the mandatory parts. The recovery block associated with the j^{th} fault of M_i and its worst-case execution time will be denoted by $B_{i,j}$ and $b_{i,j}$, respectively. However, when we deal with the single fault case ($k = 1$), we will freely use the notation B_i (b_i), as opposed to $B_{i,1}$ $(b_{i,1})$. Except the case of chain tasks with multiple deadlines (Section 6.2.2), we will develop and present our methodology first in the context of a *single* fault model (that is $k = 1$) for the sake of simplicity, and show the extension of the framework to the multiple faults' case later.

In general, a feasible schedule is said to be *Fault Tolerant (FT)* if it allows the timely recovery of any single error detected in any of the mandatory parts. In turn, a Fault Tolerant schedule is *FT-Optimal* if its total reward in the fault-free execution is maximum among all FT schedules. A *k-FT-Optimal Schedule* may be defined for k faults in a similar fashion.

An example of the application of aperiodic RB scheduling can be found in the field of robotics, in particular robots that interact to achieve a specific goal or any applications that involve teams of autonomous agents who must carry out effective, goal-directed communication and cooperation with one another in uncertain, dynamic environments. More specifically, consider a team of reconnaissance robots that is tasked with exploring some region and transmitting photographic images back to base. Each robot might begin with a planned route and a set of expectations about what to search. However, unexpected obstacles might require revising the route, and unpredictable

Task Models	Independent tasks	Chain tasks
Single deadline	Section 6.1.1	Section 6.1.2
Multiple deadlines	Section 6.2.1	Section 6.2.2

Table 6.1: Models for FT-Optimality Research

but ”interesting” objects might suggest plan revisions as well. Since at least the basic functionality must be present for the navigation and transmission of data back to the base, a mandatory part will enable the basic results to be collected and transmitted. In addition, tolerance to faults is a necessity in the execution of the mandatory parts of tasks in these robots, to ensure that at least the basic navigation and transmission are taking place.

The work in this research is motivated by the observation that, even when the system is overloaded, the time assigned to optional parts in the RB scheduling model provides opportunities for recovery of tasks, in case an error is detected. However, the optimal assignment of service times to simultaneously guarantee error recovery and maximize total reward is not a trivial problem. In particular, guaranteeing timely error recovery is crucial in the context of tasks with precedence constraints, since an error in a task's mandatory part will prevent all following tasks in the chain from executing correctly. A further objective is to have an efficient algorithm with low run-time overhead.

Well-established results of RB scheduling theory state that the optimal solution for a task set with precedence constraints may be obtained from that of the corresponding independent task set, simply by changing ready times and/or by ”re-adjusting” the final schedule [89]. This approach turns out to be insufficient for fault tolerant versions of the problem and extra complexity and complications arise for a chain of RB tasks, as demonstrated in Section 6.1.2.

In the following, we present the results of our FT-Optimality research by considering independent tasks and tasks with linear precedence constraints (chain tasks) separately. Another line of dichotomy will concern the timing constraints: We will investigate separately the case of a single deadline and that of multiple deadlines. The exposition will be divided as shown in Table 6.1.

6.1 Task Sets with Single Deadline

We begin our FT-Optimality analysis by considering tasks sharing a single deadline D. If we re-iterate the example given above, each reconnaissance robot might have a single (global) goal with a deadline. However, the goal can be further divided into subgoals, such as planning the route, moving towards the target, taking pictures along the path and communicating with the base. Each of these subgoals may have mandatory and optional parts, and furthermore the robot should achieve an acceptable performance by appropriately using optional execution slots, even if errors occur.

We note that an important number of *periodic* applications are *frame-based* in nature, and they can be readily represented by a set of tasks with a common deadline. The tasks (sometimes having precedence constraints) is invoked at regular time intervals, they are to provide an output of acceptable quality/detail before the beginning of the next frame. For example, in tracking/collision avoidance applications, motion detection, recognition/verification, trajectory estimation and computation of time to contact are usually component sub-tasks within a given frame (period) [10]. Similarly, a real-time image magnification task might go through the steps of non-linear image interpolation, contrast enhancement, noise suppression and image extrapolation during each period [75].

Before dealing with the optimality issue, we first present a few fundamental results regarding the *existence* of FT schedules for a given task set **T**.

Let $Slack(i, S)$ be the sum of optional service times scheduled after M_i in schedule S, before the deadline D. We will omit the parameter S whenever the schedule involved is clear from the context. A task T_i will be said to satisfy *the slack constraint* if $Slack(i, S) \geq b_i$, where b_i is the worst-case execution time of the recovery block B_i.

Proposition 12 *A schedule S of RB tasks is* Fault Tolerant *if the slack constraint is satisfied by every task (i.e.,* $\forall i\ Slack(i, S) \geq b_i$*).*

Proof: Consider a system in which the slack constraint is satisfied by each task T_j. If an error is detected when M_j was expected to complete, we initiate the recovery operation which will take at most b_j time units. Due to this additional workload, the execution of *mandatory* parts which are scheduled after M_j now may have to be delayed. However, each of them will be able to complete by the deadline, since the sum of optional parts which will provide slack for recovery of M_j is at least b_j. Notice that the optional parts scheduled after M_j may not execute as a result of the recovery of M_j.

□

Theorem 5 *An FT schedule exists for a set of RB tasks* **T** *if and only if* $d = D - \sum_{i=1}^{n} m_i \geq max\{b_i\}$.

Proof: If S is an FT schedule, then all the slack constraints and in particular, that of the task with the largest recovery time $(max\{b_i\})$ should be satisfied. This implies that the ”total slack” of the system should be at least equal to this amount.

Conversely, if $d \geq max\{b_i\}$, then there exists *at least* one FT schedule, namely the one in which first all the mandatory parts are scheduled and then the optional part of last task T_n with $t_n = d$ is scheduled.

□

We now deepen our analysis by considering two types of potential precedence relations among tasks in **T**.

6.1.1 Independent Tasks

Let **T**$= \{T_1, \ldots, T_n\}$ be a set of RB tasks with no precedence constraints. In other words, we have the flexibility of scheduling the mandatory and optional parts in any order, except for the restriction that an optional part O_i may not start executing before M_i completes successfully.

Consider first the (Non-FT) optimal solution of the problem. Definitely, *all* mandatory parts should complete before D. The optional parts may execute only for d units of time, where d is the total slack. Furthermore, since all the reward functions are nondecreasing by assumption, this slack may be used *fully* by optional parts.

However, when we intend to generate an FT-Optimal schedule, we should make sure that all the slack constraints are satisfied. Consider the schedule in Figure 6.1. Having independent tasks allows us to schedule all optional parts *after* all mandatory parts. It should be obvious that this choice enforces $Slack(i)$ for every task T_i to be d, which is the maximum that can be achieved. So, this *form* of schedule should satisfy the slack constraints automatically, if an FT schedule exists for **T**. In fact, no FT schedule exists if this one is not fault tolerant. Note that one still needs to determine the optimal distribution of the total slack d among optional parts.

M_1	M_2	M_3	.....	M_n	O_1	O_2	.....	O_{n-1}	O_n

Figure 6.1: An FT-optimal schedule for independent tasks

Thus, we obtain the formulation of the following nonlinear optimization problem. Find $t_1, \ldots, t_n$ so as to:

$$\text{maximize} \qquad \sum_{i=1}^{n} R_i(t_i) \tag{6.1}$$

$$\text{subject to} \qquad \sum_{i=1}^{n} t_i = d \tag{6.2}$$

$$0 \leq t_i \quad i = 1 \ldots n \tag{6.3}$$

The distribution of available slack d to several optional parts should satisfy the linear constraints $0 \leq t_i \quad (i = 1, 2 \ldots n)$, since negative service times do not have any physical interpretation. The formulation just obtained is an instance of a generic nonlinear optimization problem (denoted by MAX) in which the inequalities in Equation (6.3) are replaced by a more general form, expressing lower bounds on execution times:

$$l_i \leq t_i \quad i = 1...n \tag{6.4}$$

The algorithm ALG-MAX(R, L, d) solves this problem and is discussed in Section 6.1.3. It takes as input the set $R = \{R_1, \ldots, R_n\}$ of reward functions, the set $L = \{l_1, \ldots, l_n\}$ of lower bounds, and the available slack d and produces optimal t_i values maximizing (6.1) and satisfying (6.2) and (6.4).

Observation 2 *For an independent task set* **T***, the reward of the FT-Optimal schedule is equal to the highest reward we can obtain without considering Fault Tolerance issues, provided that a Fault Tolerant schedule exists for* **T** *(that is, $max\{b_i\} \leq d$).*

The observation is based on the fact that optimal t_i values for Non-FT and FT versions of the problem are the same: both sets of optimal solutions are provided by ALG-MAX. In case of the Non-FT schedules, it is possible to schedule the mandatory and optional parts in any order, subject to the constraint that O_i executes after M_i. However, one needs to commit to the generic schedule of Figure 6.1 for the FT case to guarantee the same reward.

6.1.2 Chain Tasks

In case that $T_1, \ldots, T_n$ have linear precedence constraints among them (i.e., they form a chain), the order of execution is pre-determined as shown in Figure 6.2.

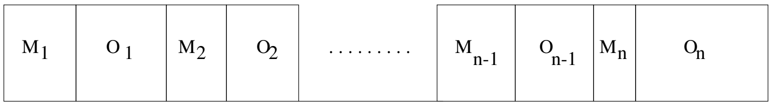


Figure 6.2: A chain of RB tasks

For a chain of RB tasks, achieving fault tolerance is not always without cost. To see this, consider the chain of tasks T_1, T_2 and T_3 with $D = 20$, shown in Table 6.2.

Since O_1 is the optional part with the largest marginal reward, the optimal schedule can be easily found to be the one shown in Figure 6.3a. However, this optimal schedule does not satisfy fault tolerance requirements, since there is insufficient slack to recover from a possible error detected in M_2 or M_3. The "best" schedule which maximizes the total reward while still guaranteeing timely recovery operations is given in Figure 6.3b.

Note that the FT-Optimal schedule has a total reward of 50, in contrast to the reward of 60 for the Non-FT optimal schedule. Unlike the case of independent tasks, in a chain, optional parts

id	m_i	o_i	b_i	$F_i(t)$
1	3	8	3	$10t$
2	6	4	6	$5t$
3	5	5	5	$9t$

Table 6.2: Example Task Set for the Chain Model, with $D = 20$

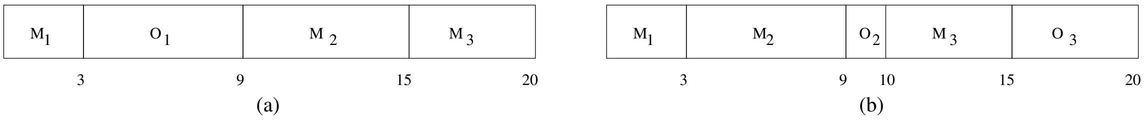


Figure 6.3: (a) An optimal but Non Fault Tolerant schedule (b) FT-Optimal schedule

cannot be arbitrarily swapped with mandatory parts. Hence, we have no freedom of scheduling optional parts late and automatically satisfying the slack constraints. We had already stated that in both hard-real time and RB scheduling problems, precedence constraints could be easily incorporated to the problem by modifying the timing constraints and/or re-arranging the final schedule. This simple example shows that the "equivalence" no longer holds in the Fault-Tolerant RB scheduling problem, as the reward of the optimal schedules are different in FT and Non-FT cases.

Let us determine the conditions that need to be satisfied in any FT schedule. If an error is detected at the end of M_n's execution, then the recovery operation has to rely *solely* on time reserved for O_n. Therefore, t_n should be at least as large as b_n. Similarly, should a fault occur in M_{n-1}, the recovery will succeed if and only if the sum of optional service times t_n and t_{n-1} is *not* less than b_{n-1}. In general, the sum of optional parts scheduled after M_i should be at least as large as b_i.

As a consequence, the problem MAX needs to be modified to explicitly incorporate slack constraints into the optimization problem. Clearly, $Slack(i) = \sum_{j=i}^{n} t_j$ for each task T_i. In turn, slack constraints can be expressed as $Slack(i) \geq b_i$ for $i = 1, 2 \ldots n$. Thus, the problem CHAIN can be formalized as:

$$\text{maximize} \qquad \sum_{i=1}^{n} R_i(t_i) \tag{6.5}$$

$$\text{subject to} \qquad \sum_{i=1}^{n} t_i = d \tag{6.6}$$

$$0 \leq t_i \qquad i = 1, 2, \ldots n \tag{6.7}$$

$$b_i \leq \sum_{j=i}^{n} t_j \quad i = 1, 2, \ldots n \tag{6.8}$$

This is another nonlinear optimization problem with equality and inequality constraints. Even though the addition of new constraints makes the problem more difficult, it is still tractable.

The algorithm ALG-CHAIN (see Figure 6.4) invokes ALG-MAX several times (lines 7 and 15), but with *different* lower bound sets (L) each time.

Informally, the solution is based on the observation that once the slack constraints are satisfied, the problem merely reduces to an instance of MAX. However, care must be taken not to get a sub-optimal solution while trying to satisfy slack constraints.

ALG-CHAIN, which assigns FT-Optimal t_i values, proceeds in two phases. In the first phase, we focus solely on satisfying the slack constraints by processing the chain in a *bottom-up* manner. During this phase, we apply a **least commitment strategy** in that we provide only optimal distribution of **minimum** slack (which is equal to $max\{b_i\}$, as we will prove shortly) which can yield an FT schedule. During the second phase, we make optimal distribution of the total slack d to all the tasks in the chain, *considering the output of the first phase as lower bounds on the execution times*.

Algorithm ALG-CHAIN

```
1    Set $t_1 = t_2 = \ldots t_n = 0$
2    Set $l_1 = l_2 = \ldots l_n = 0$   /*initial lower bounds for MAX   */
3    /* First phase:  Bottom-up processing of the chain */
4    Set $t_n =$ slack $= l_n = b_n$   /* Process $T_n$ first   */
5    For $j = n - 1$ downto 1 do if $b_j >$ slack
6    {
7    Invoke ALG-MAX($\{R_j, \ldots, R_n\}, \{l_j, \ldots, l_n\}, b_j$) /* distribute $b_j$ among
     $T_j \ldots T_n$ */
8    Set slack $= b_j$
9    For $p = j$ to $n$ do set $l_p = t_p$   /* Adjust lower bounds */
10   }
11   /* Second Phase */
12   /* Slack constraints being satisfied,
13      distribute the entire slack $d$ optimally
14      while adhering to current lower bounds $\{l_i\}$ */
15   Invoke ALG-MAX($\{R_1, \ldots, R_n\}, \{l_1, \ldots, l_n\}$, $d$)
16   Output $t_1, t_2 \ldots t_n$
```

Figure 6.4: Algorithm to solve CHAIN

Initially, all t_i values are set to 0. We start by setting $t_n = b_n$ (note that, from Equation (6.8), this is the minimum optional service time that can be given to the last task of the chain to support fault tolerance). We also increase the lower bound l_n for the last task from 0 to $t_n = b_n$, thereby ensuring that t_n will not be decreased in the iterations yet to come. Then we consider the next slack constraint, or rather its "least committed form" which is $t_n + t_{n-1} = b_{n-1}$. If this is already satisfied by the $t_n = b_n$ assignment (i.e., if $b_{n-1} \leq b_n$), we do not make any changes to t_{n-1} or t_n. Otherwise, the total slack should be increased to the minimum acceptable level of b_{n-1}. Hence, we invoke ALG-MAX to distribute b_{n-1} optimally between t_n and t_{n-1} (line 7).

Finally, before processing the next task (T_{n-2}), we commit to the current t_n and t_{n-1} values as the *lower bounds* for the next iteration (line 9), since they can not be reduced because of FT considerations. We continue processing the remaining slack constraints in the same fashion.

Notice that during the first phase, after ALG-CHAIN processes T_j, the value of `slack` is equal to the sum of optional assignments made so far, that is, $\sum_{k=j}^{n} t_k$. This in turn, is equal to $max\{b_j, \ldots, b_n\}$, which is the minimum acceptable slack to obtain an FT schedule for the subchain $T_j, \ldots, T_n$.

When we finish processing the whole chain in this fashion, Phase 2 invokes ALG-MAX to distribute optimally any total slack d among *all* optional parts. However, this distribution adheres to lower bounds provided by the outcome of the first phase (which has satisfied slack constraints while still remaining in the search space containing optimal values).

Before providing a formal proof regarding the optimality of the least commitment strategy, we will first show that ALG-CHAIN effectively solves the two auxiliary optimization problems. Finally, we will prove that solving these two subproblems is in fact equivalent to solving the problem CHAIN.

Proposition 13 ALG-CHAIN is effectively equivalent to first solving P1 which finds $t'_1 \ldots t'_n$ so as to:

$$\begin{aligned}
&\text{maximize} && \sum_{i=1}^{n} R_i(t'_i) && \\
&\text{subject to} && \sum_{i=1}^{n} t'_i = max\{b_1, \ldots, b_n\} && (6.9)\\
& && 0 \leq t'_i \qquad i = 1, 2, \ldots n && (6.10)\\
& && b_i \leq \sum_{j=i}^{n} t'_j \quad i = 1, 2, \ldots n && (6.11)
\end{aligned}$$

and then solving P2 which is computing $t''_1 \ldots t''_n$ so as to:

$$\begin{aligned}
&\text{maximize} && \sum_{i=1}^{n} R_i(t''_i) && \\
&\text{subject to} && \sum_{i=1}^{n} t''_i = d && (6.12)\\
& && t'_i \leq t''_i \qquad i = 1, 2, \ldots n && (6.13)
\end{aligned}$$

Note that the lower bounds $\{t'_i\}$ of P2, used in the constraint set (6.13), are determined by the outcome of P1. We need to prove two lemmas before establishing the validity of Proposition 13.

Lemma 3 *After Phase 1 processes T_i in ALG-CHAIN, the particular slack constraint $b_i \leq \sum_{k=i}^{n} t_k$ is satisfied.*

Proof: This follows from the actions performed by the algorithm ALG-CHAIN whenever it encounters an b_i such that $b_i > \texttt{slack}$ (lines 7-9). Observe that after Phase 1 processes T_j in ALG-CHAIN, $\texttt{slack} = \sum_{k=j}^{n} t_k = max\{b_j, \ldots, b_n\} = Slack(j)$.

□

Lemma 4 *After Phase 1 processes T_j in ALG-CHAIN, $\{t_j, t_{j+1}, \ldots, t_n\}$ is an FT-Optimal assignment of $\texttt{slack} = \sum_{k=j}^{n} t_k = max\{b_j, \ldots, b_n\}$ to the subchain $T_j, \ldots, T_n$.*

Proof:

We prove the FT-Optimality by induction. The statement is clearly true after T_n is processed. Assume that it holds up to and including the m^{th} task. We denote by T_j the task considered by Phase 1 at the $m + 1^{th}$ iteration (i.e., $j = n - m$).

First, observe that the slack available for *every subchain* $T_v, \ldots, T_n \quad (v = j + 1, j + 2, \ldots, n)$ is the *minimum* that can yield an FT schedule (by Theorem 5), hence the total slack available to any of the involved subchains can not be reduced.

We distinguish two cases. If $b_j \leq \texttt{slack}$, then $max\{b_j, \ldots, b_n\} = max\{b_{j+1}, \ldots, b_n\}$ and the total slack is not increased. Notice that $Slack(j)$ is sufficient to tolerate an error detected in M_j. In this case, we are not allowed to increase t_j which was set to 0, since this would imply decreasing some $t_k (k > j)$ creating a Non-FT schedule for the subchain $T_k, \ldots, T_n$. This shows that t_j should remain 0. Furthermore, since we do not increase the total slack *and* we already have an FT-Optimal schedule for $T_{j+1}, \ldots, T_n$ by the induction assumption, $t_j = 0, t_{j+1}, \ldots, t_n$ is an FT-Optimal distribution of $\texttt{slack}$ to $T_j, \ldots, T_n$.

If $b_j > \texttt{slack}$ we may consider increasing t_j. Note that, after processing every task, the algorithm commits to the current assignments $\{t_i\}$ as the lower bounds $\{l_i\}$ (line 9). All further re-assignments done by ALG-MAX adhere to these lower bounds. Hence, in order to exclude the possibility of having a *better* distribution outside of the rectangular region of these constraints, we need to show that the optimal distribution of b_j to $T_j, \ldots, T_n$ may never result in a decrease in the current assignments.

Now, let $\{t_{j+1}, \ldots, t_n\}$ be the FT-Optimal distribution of $max\{b_{j+1}, \ldots, b_n\}$ to $T_{j+1}, \ldots, T_n$ (observe that $t_j = 0$). Also, let $\{t^\circ_j, t^\circ_{j+1}, \ldots, t^\circ_n\}$ be the FT-Optimal solution for distributing the slack $b_j > max\{b_{j+1}, \ldots, b_n\}$ to $T_j, \ldots, T_n$.

Note that $\sum_{i=j+1}^{n} t_j^\circ \geq \sum_{i=j+1}^{n} t_j = max\{b_{j+1}, \ldots, b_n\}$, otherwise the subchain T_{j+1}, $\ldots, T_n$ would not be FT by Theorem 5. In other words, the total slack available for $T_{j+1}, \ldots, T_n$ may never decrease. Remember that, the problem CHAIN provides FT-Optimal solution for any chain with a given slack. Hence, $\{t_{j+1}, \ldots, t_n\}$ is also a solution to CHAIN when invoked for the subchain $T_{j+1}, \ldots, T_n$ and with the slack $max\{b_{j+1}, \ldots, b_n\} = E$. Because $\sum_{i=j+1}^{n} t_j^\circ \geq \sum_{i=j+1}^{n} t_j$, the proof will be complete if we show that, there is always a solution set $\{t_{j+1}^\circ, \ldots, t_n^\circ\}$ to CHAIN where $t_k^\circ \geq t_k \;\; k = j+1 \ldots n$, when invoked for the same subchain $T_{j+1}, \ldots, T_n$, but with the slack $E + e$ where $e \geq 0$.

The full proof of this last statement is based on Kuhn-Tucker optimality conditions for nonlinear optimization. It is omitted here, but can be found in [2]. Informally, this holds in view of concave and non-decreasing properties of reward functions, where all the derivatives (marginal returns) are non-increasing: If there exists an index i such that $i > j$ and $t_i^\circ < t_i$, then there should exist an index k such that $k > i$ and $t_k^\circ > t_k$ (otherwise, the subchain $T_i, \ldots, T_n$ would not be FT). This could happen only when the optimization algorithm preferred increasing t_k by decreasing t_i *even if the total available slack increased.* Now, according to the best marginal rate principle, CHAIN would have assigned t_i° to T_i even for the first invocation, leaving a larger slack for other tasks.

□

We are now ready to prove Proposition 13: Lemma 3 and 4 show that Phase 1 solves the problem P1. Similarly, it can be seen that the second phase of the ALG-CHAIN solves P2 (lines 15-16 of Figure 6.4). With the help of Proposition 13, we can finally establish the optimality of the least commitment strategy as follows.

Theorem 6 ALG-CHAIN solves optimally the CHAIN problem.

Proof: Let $\{t_i^*\}$ be the solution set of Problem CHAIN. If we consider Proposition 13, all we need to show is that the t_i'' values obtained from the invocation of P2 are equal to $t_i^*, \;\; i = 1, \ldots, n$. First, notice that slack constraints are already satisfied by P1. Also, the solution set $\{t_i'\}$ of P1 serves as lower bounds in the constraint set (6.13) of P2. Thus, if $\forall i \; t_i^* \geq t_i'$ then it will be within the search space (feasible region) of Problem P2. Consequently an algorithm that solves P2 will return a solution set $\{t_i''\}$ with the same total reward as the one yielded by $\{t_i^*\}$. Hence, if we prove that there is always a solution $\{t_i^*\}$ to CHAIN such that $\forall i \; t_i^* \geq t_i'$ the proof will be completed.

In fact, since $d \geq max\{b_1, \ldots, b_n\}$ a reasoning completely analogous to that exposed in the proof of Lemma 4 can establish that choosing an i such that $t_i' > t_i^*$ would violate "best marginal rate" principle and/or produce a non-FT schedule.

□

We emphasize the fact that the optimality of the least commitment strategy is based on concave and nondecreasing properties of reward functions. The worst-case time complexity of ALG-MAX is $O(n^2 \log n)$ (however, an $O(n \log n)$ solution exists for linear functions, see Sections 6.1.3 for the details of algorithm). Hence, the complexity of ALG-CHAIN turns out to be $O(n^3 \log n)$, if the successive calls to ALG-MAX are accounted for. A simple example which illustrates the algorithm ALG-CHAIN is presented in Section 6.1.4.

6.1.3 ALG-MAX: Case of trivial solutions

MAX is characterized by 3-tuple (R, L, d) where R is the set of the reward functions, L is the set of lower bounds that any solution must adhere to and d is the total slack. MAX consists of finding $t_1, \ldots, t_n$ so as to:

$$\begin{array}{ll} \text{maximize} & \sum_{i=1}^{n} R_i(t_i) \\ \text{subject to} & \sum_{i=1}^{n} t_i = d \\ & l_i \leq t_i \quad i = 1 \ldots n \end{array}$$

The reward functions $R_i(t_i)$ in the above formulation are as in Equation (3.1) and hence are non-differentiable at $t_i = o_i \, \forall i$. In the following discussion, we denote by "entire optional part of task T_i" the quantity $max\{l_i, o_i\}$. Note that a lower bound l_i may be larger than the corresponding upper bound o_i in the problem specification, which is usually a result of fault tolerance requirements.

As a preprocessing phase, we check a few conditions which immediately yield trivial solutions. Specifically,

- if the available slack d is not large enough to accommodate all lower bounds $l_1, l_2, \ldots, l_n$ (in other words if $\sum_{i=1}^{n} l_i > d$), then the constraint set is inconsistent and no solution exists.
- if $\sum_{i=1}^{n} l_i = d$, then the unique solution is $t_i = l_i$, $1 \leq i \leq n$.
- if the available slack d is large enough to accommodate all optional parts entirely (i.e., if $d \geq \sum_{i=1}^{n} max\{l_i, o_i\}$) then we first set $t_i = max\{l_i, o_i\} \, \forall i$. That is, we use every optional part fully. Any remaining slack $d - \sum_{i=1}^{n} max\{l_i, o_i\}$ can be arbitrarily distributed among optional parts of tasks $T_1, \ldots, T_n$. For simplicity, we choose to increase t_n.

If none of the above conditions holds, then for every task T_j such that $l_j \geq o_j$ we set $t_j = l_j$ and we reduce the available slack d accordingly. Note that t_j does not need to be set to a value larger than l_j since the reward of T_j does not increase beyond o_j. Finally, the preprocessing procedure completes by obtaining another optimization problem, which involves *only* the tasks such that $l_i < o_i$. For every remaining task T_i, we first reserve l_i units of slack, reduce d and o_i by l_i,

id	m_i	o_i	$F_i(t)$
1	8	10	$5t$
2	8	8	$10t$
3	4	8	$8t$
4	6	5	$6t$

Table 6.3: Example task set for CHAIN

define $f_i(t_i)$ as $F_i(t_i + l_i)$, and get an instance of of the concave optimization problem OPT-LU, first presented in Section 4.5, in the context of periodic reward-based scheduling problem:

$$\text{maximize} \quad \sum_{i=1}^{n} f_i(t_i) \tag{6.14}$$

$$\text{subject to} \quad \sum_{i=1}^{n} t_i = d \tag{6.15}$$

$$t_i \leq o_i \quad i = 1, 2, ...n \tag{6.16}$$

$$0 \leq t_i \quad i = 1, 2, ...n \tag{6.17}$$

Note that after the preprocessing phase (i.e., for problem OPT-LU), $0 < d < \sum_{i=1}^{n} o_i$ and $o_i > 0\, \forall i$. As shown in Section 4.5, this problem admits a solution of time complexity $O(n^2 \log n)$ in the worst-case.

6.1.4 An example for CHAIN

Consider the **chain** of tasks composed of T_1, T_2, T_3 and T_4 with the deadline $D = 40$. The characteristics of the tasks are given in Table 6.3 . We assume that $b_i = m_i \quad \forall i$.

The first phase (bottom-up processing of the chain) starts by considering the last task of the chain, namely T_4. Notice that, even if the optional part O_4 is scheduled entirely (i.e., t_4 is set to o_4), M_4 can not recover from a possible error detected at the end of its execution, since $b_4 > o_4$. Hence, the algorithm has no choice but to set t_4 to b_4 even though the portion of O_4 allocated beyond o_4 will produce no extra reward. At this point, the total slack of the system (denoted by `slack`) is equal to $b_4 = 6$. Furthermore, the lower bound l_4 is raised to b_4: for fault tolerance, the algorithm can not decrease t_4 below b_4 under any condition during next iterations.

Next, T_3 is considered. Since $b_3 <$ `slack`, there is already enough optional allotment in the system that can be used in case that a fault occurs during the execution of M_3. Adhering to the least commitment strategy, the algorithm does not make any changes in the current assignments. Observe that, this decision helps indeed to reserve as much optional execution time as possible to the task with the largest marginal reward, T_2. When T_2 is processed, it turns out that `slack` $< b_2$. Consequently, the algorithm performs the optimal distribution of $b_2 = 8$ time units among T_2, T_3

and T_4 by invoking ALG-MAX, yielding $t_2 = 2$ and $t_4 = 6$. `slack` is set to 8 and l_2 (lower bound for t_2) is set to 2. For the last task T_1, current assignments are not modified since `slack` $\geq b_1$. At the end of the first phase, we have the schedule shown in Figure 6.5. Observe that at this point, the sum of optional service times is equal to $8 = max\{b_i\}$, which is, in turn, the minimum slack necessary to obtain an FT schedule (by Theorem 5).

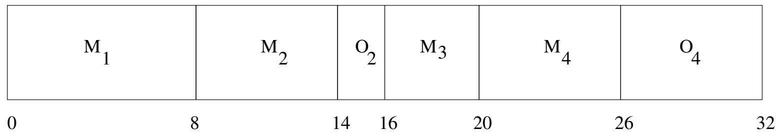


Figure 6.5: The schedule after bottom-up processing

The second phase is relatively easier. The total slack of $D - \sum m_i = 16$ units is subsequently distributed in an optimal way. Note that this distribution takes into account the lower bounds obtained after the execution of the first phase. Hence, the algorithm is not allowed to reduce current assignments, since such a decrease would produce a Non-FT or Non-Optimal schedule as shown in the proof of the least commitment strategy. ALG-MAX indicates that O_2 should be used in its entirety and the remaining 2 time units assigned to O_3. The final schedule is given in Figure 6.6.

One can easily see that the Non-FT optimal solution for the same task set is $t_2 = t_3 = 8$ and $t_1 = t_4 = 0$, which would accrue a total reward of 144. The FT-Optimal schedule of Figure 6.6 produces a total reward of 126, which is 87.5% of the Non-FT Optimal solution.

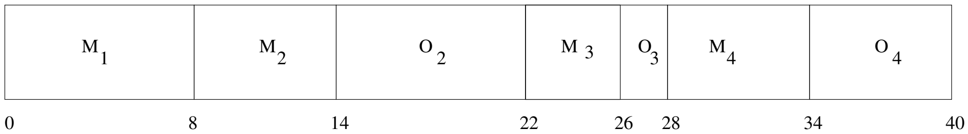


Figure 6.6: The final FT-Optimal schedule

6.1.5 The Preservation of FT-Optimality in the absence of errors

To avoid excessive run-time overhead, it would be preferable not to have to update the schedule as long as no errors are encountered during the execution of mandatory parts, unlike in [12].

Proposition 14 *The FT-Optimality of the schedules generated by ALG-MAX and ALG-CHAIN is preserved as long as no errors are detected on mandatory parts.*

Proof: Suppose there have been no errors during executions of $M_1, M_2, \ldots, M_i (1 \le i \le n)$ and we are about to execute O_i. Assume the contrary, that is, it is possible to have a higher reward by executing O_i during $t_i' \neq t_i$, where t_i is the service time returned by the optimal algorithms (MAX for independent tasks, CHAIN for dependent tasks). But this contradicts the optimality of MAX and CHAIN, since the distribution of remaining slack $d - t_1 - t_2 - \ldots - t_{i-1}$ among $t_i, t_{i+1}, \ldots, t_n$ should also yield the largest reward possible, by the optimality of subproblems. In other words, if this were the case, MAX (CHAIN) would have returned t_i' instead of t_i at the first place.

□

6.1.6 Refinements and Extensions

6.1.6.1 Adaptive Scheme

The algorithms presented in Section 6.1 provide the optimal *static* schedules, in the sense that they simultaneously maximize the total reward while guaranteeing timely completion of recovery operations. However, they do not (and should not) make any assumptions about the mandatory part during which a fault occurs. Furthermore, it is assumed that *once* a fault has occurred in some M_i and recovery is completed, no optional part is allowed to run due to the possibility of interfering with other mandatory parts.

Optimal redistribution of the remaining slack among optional parts yet to execute can provide an **adaptive FT-Optimal** schedule. For example, consider the FT-Optimal schedule given in Figure 6.3b. If an error is detected at the end of M_1, the recovery operation will be initiated. At the end of the recovery (at $t = 6$), the system has still a total slack of 3 time units for the remaining optional parts. Furthermore, we *can* assume that no more faults will occur, since $k = 1$ and the fault has been already encountered. By re-distributing optimally the slack available, we could execute O_3 for 3 time units, thereby obtaining a total reward of 27 units. The resulting adaptive schedule is shown in Figure 6.7.

Figure 6.7: Adaptive adjustment of FT-Optimal schedule

The foregoing discussion shows that, after recovering from an error, we can dynamically invoke ALG-MAX to re-distribute optimally the available slack. Alternatively, instead of invoking ALG-MAX after a fault, one may produce and store n alternative schedules a priori, corresponding to the n possible fault scenarios. In this case, the run-time system should be responsible for "switching" to the appropriate schedule when a fault occurs; in other words, we provide optimality dynamically in a way similar to that of scenario or mode changes [73, 80].

6.1.6.2 Extension to k Faults

Suppose that k faults are to be tolerated during the execution of the task set **T**, instead of a single fault. In this case, we might have more than one recovery block associated with subsequent errors of a task. We assume that the execution time of all recovery blocks of task T_i is b_i. Clearly, the worst-case happens if all k errors occur during the execution and subsequent recovery (or recoveries, as the case may be) of the task that has the largest recovery time. Hence, it is possible to generalize Theorem 5, with an analogous proof (omitted here):

Theorem 7 *A set of RB tasks* **T** *can be scheduled while tolerating* k *faults if and only if* $d \geq k \cdot max\{b_i\}$.

If **T** comprises only independent tasks, then it is not difficult to check that the *form* of the schedule shown in Figure 6.1 is again optimal. Once and when the condition expressed in Theorem 7 is satisfied, we can invoke ALG-MAX and compute optimal t_i values.

If **T** is composed of a chain of tasks, then the "slack constraints" have to be modified (tightened) accordingly. It should be clear that, in this case, the necessary and sufficient condition to recover from k errors is to have the sum of the *remaining* optional parts (i.e., $Slack(i)$) at least equal to $k \cdot \max_{a=i,\ldots,n} \{b_a\}$. Then the slack constraints can be restated as:

$$k \cdot \max_{a=i,\ldots,n} \{b_a\} \leq \sum_{a=i}^{n} t_a \quad i = 1, 2, \ldots, n \tag{6.18}$$

In other words, it is sufficient to scale the lower bounds up in order to tolerate any k faults and Algorithm ALG-CHAIN may be used as it is. Indeed, it is easy to see that when $k = 1$, Equation (6.18) is equivalent to the set of slack constraints given in Equation (6.8).

6.2 Task Sets with Multiple Deadlines

In this section, we extend our FT-Optimality framework the more general case where tasks have potentially different deadlines. Again, we address independent tasks (Section 6.2.1) and chain tasks (Section 6.2.2) separately.

6.2.1 Independent Tasks

We note that the definition of FT(-Optimal) schedule given at the beginning of the chapter becomes incomplete when we consider independent tasks with multiple deadlines, in the sense that one needs to specify *how and when the recovery block is executed*. Since an additional workload is introduced to the system because of the recovery block, *the subsequent optional parts may not have a chance to execute*. We can distinguish two recovery schemes that can be readily applied in practical real-time applications without considerable overhead:

- **Immediate Recovery (IR):** The recovery block is executed immediately when the error is detected at the end of the faulty mandatory task. The mandatory parts yet to execute may have to be delayed/postponed due to the recovery operation.
- **Delayed Recovery (DR):** The recovery block is allowed to execute *only* in time slots which are not dedicated to any mandatory task (i.e., those which are empty or assigned to an optional part), but still before the deadline. As a result, mandatory parts are not shifted in the timeline.

As an example, consider the schedule in Fig. 6.8a. We assume that $b_1 = m_1$. If a fault is detected at the end of M_1 at $t = 4$, the recovery operation is initiated. If the mechanism chosen is the immediate recovery then the recovery proceeds without preemption and until full completion, resulting in a 'shift' of M_2 (Figure 6.8.b). On the other hand, if the delayed recovery is applied then B_1 runs only during the optional slots while before d_1. Clearly, M_2 does not need to be shifted in this case (see Figure 6.8.c). Note that, in both cases M_2 and the recovery block B_1 complete successfully before their deadlines.

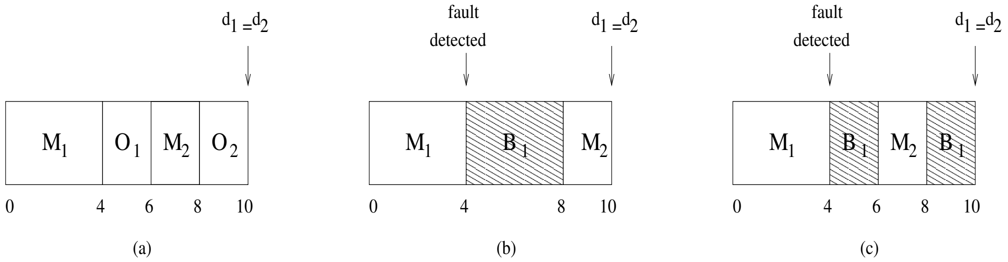


Figure 6.8: (a) A feasible schedule (b) Recovery of M_1 with IR (c)Recovery of M_1 with DR

It should be emphasized that the two schemes above do *not* lead to identical FT schedule spaces. To see this, consider the schedule in Figure 6.9.a. Assuming that $m_1 = m_2 = b_1 = b_2 = 4$, the schedule above is clearly FT according to the immediate recovery scheme, but not according to the delayed recovery: There is simply no optional part scheduled before d_1, during which one would like to run the recovery block B_1. Similarly, the schedule shown in Figure 6.9.b is FT according to the delayed recovery scheme, while the immediate recovery of a fault in M_3 would cause M_2 to miss its deadline (assume again that $b_i = m_i$ for all the tasks, that is, recovery by re-execution).

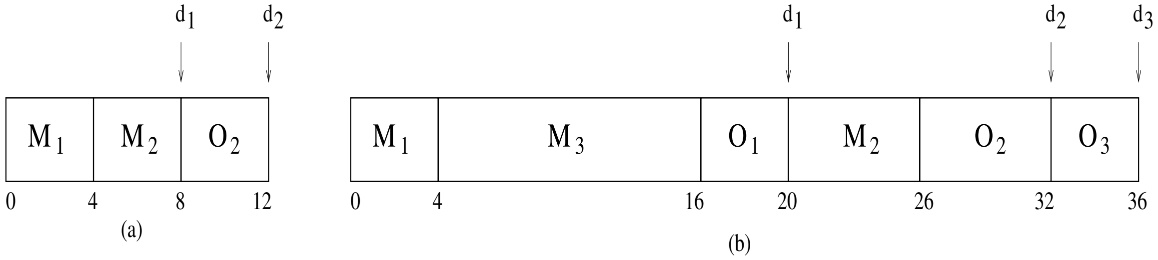


Figure 6.9: (a) An FT schedule according to IR but not DR (b) An FT schedule according to DR but not IR

Given a schedule and a recovery mechanism, checking its tolerance to fault(s) is not a difficult process and can be accomplished in polynomial time. On the other hand, our aim is to construct FT-Optimal schedules in a computationally efficient manner, which turns out to be a non-trivial task. Given a feasible schedule S, we define:

$End_S(M_i)$: The time by which M_i completes in S.

$Opt_S(t_1, t_2)$:The sum of optional parts and empty schedule segments in time interval $[t_1, t_2]$ in S.

We will drop the subscript S, whenever the schedule involved is clear from the context. As an example, in the schedule of Figure 6.9.b, $End(M_2) = 26$, while $Opt(16, 32) = 10$. Clearly, in case of preemptions, the End function should consider the completion of the last portion of the task.

Observe that in any of the schemes we can not impose an *a priori* order among the different tasks, in the absence of any formal result, since our aim is to produce the FT schedule with the highest reward.

Reward Functions: While investigating the FT-Optimality problem for independent tasks with multiple deadlines, we will restrict our analysis to identical linear functions. That is, we will assume that:

$$R_i(t_i) = \begin{cases} F_i(t_i) & if \quad 0 \leq t_i \leq o_i \\ F_i(o_i) & if \quad t_i > o_i \end{cases}$$

where $F_i(t_i) = t_i \quad \forall i$. As it will be seen, the problem remains rather challenging even with this assumption.

6.2.1.1 Immediate Recovery (IR)

We start by defining necessary and sufficient FT conditions according to the IR scheme, considering a single fault. The execution of the recovery block B_i may affect *only* the mandatory parts which are scheduled *after* M_i. We say that a task T_j is *safe* in a given schedule S, if and only if M_j can be completed in a timely manner even in the presence of any faults in tasks preceding or occurring during M_j; or equivalently, if and only if:

$$End(M_j) + b_i - Opt(End(M_i), End(M_j)) \leq d_j \;\forall i \text{ such that } End(M_i) \leq End(M_j) \quad (6.19)$$

For a given i, the left-hand side of the inequality (6.19) indicates the time by which M_j *would* complete, if a fault *were* detected in M_i; or equivalently the (new) $End(M_j)$ value after the fault and recovery of M_i. More precisely, it is derived by taking into account that the workload is increased by an amount of b_i, but decreased by (if necessary) removing all the subsequent optional parts, in case an error is detected at M_i. A schedule S is said to be *Fault Tolerant* according to IR scheme, if and only if all the tasks are safe; that is, if and only if Equation (6.19) holds for all mandatory tasks $M_j \quad j = 1, \ldots, n$.

Given an FT schedule S, consider the effects of moving a portion τ_x of a task T_x to a later in point in the schedule as shown in Figure 6.10. We call such a move *the forward (FW) move* of τ_x. Note that τ_x may originate from mandatory or optional parts of T_x.

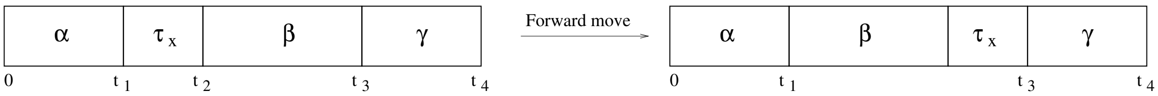


Figure 6.10: The forward move of τ_x

Theorem 8 (Forward-moving theorem) *If a schedule S is FT, then the schedule S' which results from the forward-move of a task portion τ_x belonging to T_x is still FT if and only if, in S':*
i. T_x is safe, and
ii. A fault of T_x does not cause a deadline miss in any mandatory part located in γ.

Proof: We will prove the theorem by showing that in the transformed schedule S',
(i.) Every task T_j such that $End_S(M_j)$ is in α are still safe,
(ii.) Every task T_j such that $End_S(M_j)$ is in β are still safe and,
(iii.) No fault detected in α or β can cause a deadline miss (of mandatory parts) in γ.

To establish (i.), it is sufficient to observe that the segment α remains unchanged in S', hence any task T_j such that $End(M_j) \leq t_1$ should be still safe in the transformed schedule.

To see that (ii.) holds, consider any task T_j such that $t_2 < End_S(M_j) \leq t_3$ (M_j in β). Since T_j was safe in S by definition, the equation (6.19) gives:

$$End(M_j) + b_i - Opt(End(M_i), End(M_j)) \leq d_j \;\forall\; i \text{ such that } End(M_i) \leq End(M_j) \quad (6.20)$$

Consider the fault of a task M_i preceding M_j in S ($End_S(M_i) \leq End_S(M_j)$). If $End_S(M_i) = t_2$, then the FW-moved portion τ_x belongs to M_x, $M_i = M_x$ and the fault of M_x can no longer cause a problem for M_j in S' where now it succeeds M_j. Otherwise, it is either in α or β (preceding M_j). Note that $End_S(M_j) - End_{S'}(M_j) = t_2 - t_1 > 0$ (M_j has been moved backward in the schedule by an amount of $t_2 - t_1$). Also $Opt(End_S(M_i), End_S(M_j)) - Opt(End'_S(M_i), End'_S(M_j))$ is either $t_2 - t_1$ (in case that M_i is in α and τ_x is a portion of the optional part O_x) or 0 (the remaining cases). Hence, the sum of optional parts between M_i and M_j may have decreased, but *at most* by an amount of $t_2 - t_1$. However, this shows that M_j is still safe in S' since:

$$End_{S'}(M_j) + b_i - Opt(End_{S'}(M_i), End_{S'}(M_j)) \leq$$
$$End_S(M_j) + b_i - Opt(End_S(M_i), End_S(M_j)) \leq d_j$$
$$\forall\ i \ \text{ such that } \quad End_{S'}(M_i) \leq End_{S'}(M_j)$$

Finally, to show (iii.), let $End_S(M_j)$ be in γ. Clearly, $End_S(M_j) = End_{S'}(M_j)$. Consider a fault of any task M_i preceding M_j in S', such that $End_S(M_i)$ in α or β. It is not difficult to see that $Opt(End_S(M_i), End_S(M_j)) \leq Opt(End_{S'}(M_i), End_{S'}(M_j))$, that is, the sum of optional parts between M_i and M_j may have never decreased in S'. But this gives:

$$End_{S'}(M_j) + b_i - Opt(End_{S'}(M_i), End_{S'}(M_j)) \leq$$
$$End_S(M_j) + b_i - Opt(End_S(M_i), End_S(M_j)) \leq d_j$$
$$\forall\ i \ \text{ such that } \quad End_{S'}(M_i) \,\text{is in}\, \alpha \,\text{or}\, \beta$$

completing the proof.

□

With the help of the Forward-moving theorem, we can further obtain important properties of FT schedules, as the next theorem demonstrates.

Theorem 9 *It is always possible to transform an FT schedule S to another FT schedule with the same total reward, where:*
i. Mandatory parts are not preempted,
ii. Optional parts are not preempted,
iii. Optional parts follow EDF order.

Proof: (i). Suppose that M_i is preempted once or more in S. Let M_{ia} be the first 'segment' of M_i in S, M_{ib} the second and so on. Consider forward-moving of M_{ia} close to M_{ib} as illustrated in Fig 6.11. We call the resulting schedule S'. According to Theorem 8, we need to check only the safety of M_i and the effect of M_i's fault in γ. It is easy to see that $Opt(End_S(M_j), End_S(M_k)) = Opt(End_{S'}(M_j), End_{S'}(M_k)) \quad \forall j, k$ and further, there is no task M_j such that $End(M_j)$ has increased in S'. Hence the condition (6.19) implies that M_i and all tasks in γ are still safe. One can apply this transformation repeatedly to obtain a schedule where the mandatory parts are never preempted.

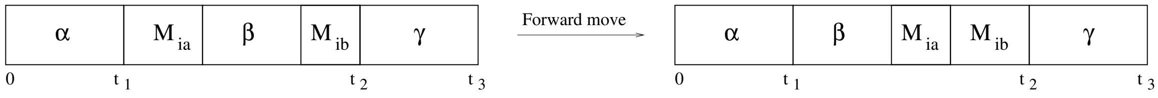


Figure 6.11: Forward move of the preempted mandatory portion

(ii). If O_i is preempted, one can move it forward to obtain the schedule S' (see Figure 6.12). Clearly, since $0 \leq End_S(M_i) = End_{S'}(M_i) \leq t_1$ and M_i is in α, T_i definitely is still safe in S' (where α did not change at all). Also,

$$Opt(End_S(M_i), End_S(M_j)) \leq Opt(End_{S'}(M_i), End_{S'}(M_j)) \quad \forall\, M_j \text{ in } \gamma$$

and the condition (6.19) ensures that all tasks in γ are safe, thus S' is still FT. Hence, it is possible to apply the transformation repeatedly to remove all the preemptions of optional parts.

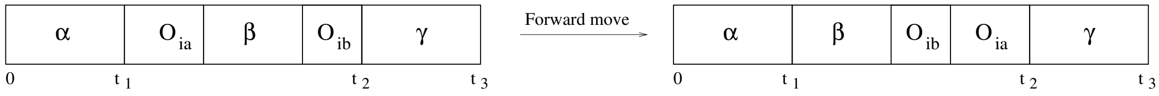


Figure 6.12: Forward move of the preempted optional portion

(iii). Finally, consider a pair of optional parts which do not follow EDF order in S. We can consider that all the preemptions are already removed via the transformations described above. Again, consider moving O_i after O_j (Figure 6.13). Following a reasoning completely analogous to the preceding paragraph, we can establish that M_i and all the tasks in γ are still safe. A finite number of transformations would definitely yield a schedule where optional parts follow EDF order. Note also that, the total reward of the FT schedule remains the same after any forward-moves just described.

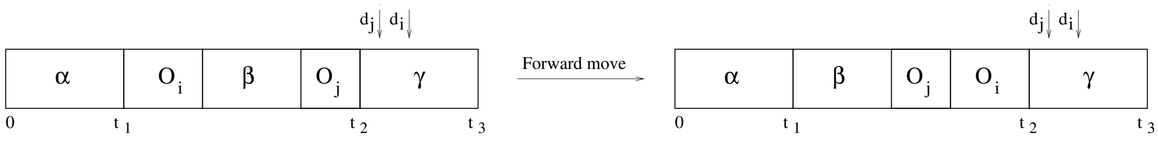


Figure 6.13: Forward move of the optional task with later deadline

□

Corollary 4 *Provided that a task set* **T** *is Fault-Tolerant, there exists an FT-optimal schedule for* **T**, *where no mandatory or optional parts is preempted and optional parts follow EDF order.*

Unfortunately, it may not be always possible to find an FT-optimal schedule where *mandatory* parts follow also EDF order. To see this, consider the tasks T_1, T_2, T_3 with parameters $m_1 = 4, o_1 = 8, d_1 = 20; m_2 = 2, o_2 = 2, d_2 = 24; m_3 = 8, o_3 = 2, d_3 = 26$. We assume that $b_i = m_i \quad i = 1, 2, 3$. In fact, one can schedule the task set in an FT manner without discarding any optional part as shown in Figure 6.14.a, which is consequently an FT-Optimal schedule. On the other hand, if one tries to enforce the EDF rule for mandatory parts also, the best schedule that can be obtained is the one shown in Figure 6.14.b. Observe that O_1 can not be scheduled beyond d_1 and the large recovery block of M_3 contributes to a lower reward.

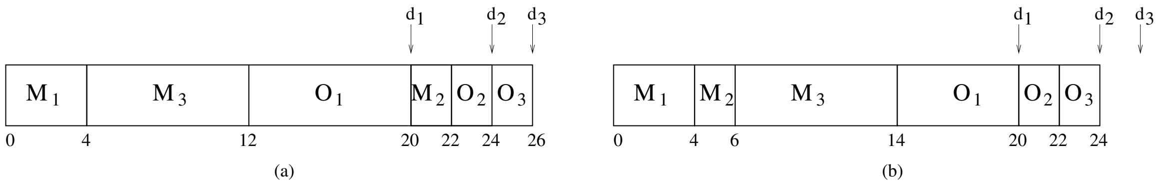


Figure 6.14: (a) The FT-Optimal schedule (b) The sub-optimal schedule with EDF order among mandatory parts

The apparent inability of inferring the order of mandatory parts a priori, is one of the main reasons contributing to the problem complexity. Before presenting our final algorithm, we obtain another useful property of the FT-optimal schedules. Informally, the theorem states that one can always find an FT-Optimal schedule where an optional part with later deadline can always be used instead of an optional part with earlier deadline, as long as the upperbounds on the optional execution times are not violated.

Theorem 10 *If a task set is FT, then there exists always an FT-optimal schedule, where $t_i > 0$ only if $t_j = o_j \quad j = i + 1, \ldots, n$.*

Proof: Without loss of generality, assume that the FT-Optimal schedule is in the 'canonical' form stated in Corollary 4: that is, no preemption exists and optional parts follow EDF order.

Let O_p be the optional part with the *latest* deadline such that $t_p > 0$ and $t_{p+1} < o_{p+1}$ (such a task should definitely exist if the proposition does not hold). Since the schedule is in the canonical form, we can move $\Delta = min(o_{p+1} - t_{p+1}, t_p)$ units of O_p forward and consider it as part of O_{p+1}. Figure 6.15 illustrates the FW-move.

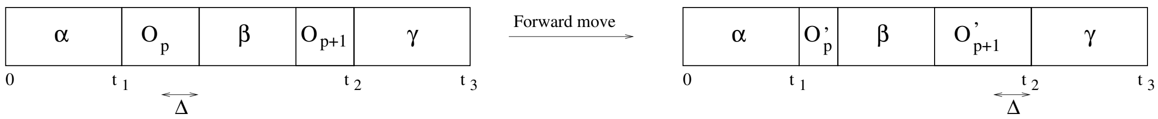


Figure 6.15: Forward move of a portion (Δ) of an optional part and its renaming

Note that from the definition of Δ, $t_{p+1} + t_p = t'_{p+1} + t'_p$, and the total reward remains the same (since all the reward functions are assumed to be identical and linear). Moreover, one can easily verify that the FW-move of Δ units of O_p does not affect the safety of any task. Note that, $t_j = o_j$ for $j = p+2, \ldots, n$ from the assumptions. If the statement of the theorem does not hold after this transformation, then we can apply the technique repeatedly to favor later optional parts without harming the fault tolerance requirement or the optimal reward.

□

Theorem 10 implies that one does not need to schedule an optional part unless *all* the optional parts with the later deadlines are *fully* utilized. Hence, given a set **O** of optional parts $\{O_1, \ldots, O_n\}$ and a positive real number X we can define the function $SUFFIX(O, X)$ which returns the (new) set of optional parts **O'**$= \{0, 0, \ldots, O'_m, O'_{m+1} \ldots, O'_n\}$ such that $o'_i = o_i$ for $i = m+1, \ldots, n$ and $o'_m = X - \sum_{i=m+1}^{n} o'_i$. Note that in case that $X \leq o_n$, **O'**$= \{0, 0, \ldots, 0, O'_n\}$ where $o'_n = X$; or if $X \geq \sum o_i$, then **O'**$=$ **O**.

Corollary 5 *The existence of an FT schedule with total reward larger than or equal to X, implies that there is an FT-Optimal schedule where the optional parts in SUFFIX(O,X) appear fully and in EDF order.*

It may happen that in an FT-optimal schedule $\sum m_i + \sum t_i < d_n$; that is, there are 'empty' slots in the interval $[0, d_n]$. The following establishes that it is possible further to obtain a fully-utilized schedule or timeline without hurting FT, by augmenting t_n.

Theorem 11 *If $\sum m_i + \sum t_i < d_n$ in an FT-optimal schedule S, then we can increase t_n (beyond the upperbound o_n) to obtain an FT-optimal schedule S' where $\sum m_i + \sum t_i = d_n$*

Proof: Since $\sum m_i + \sum t_i < d_n$ in S, there should be empty slots within the timeline interval $[0, d_n]$. By a reasoning similar to the proof of Theorem 10, moving every empty slot to the end of the schedule and renaming it as part of the optional part O_n (beyond the upperbound o_n) keeps the total reward unchanged, the FT property is preserved, and the timeline is now fully utilized.

□

The results above allow us to devise a polynomial-time algorithm (Figure 6.16) to **decide** whether there exists an FT-Optimal schedule with a total reward of **exactly** X.

The algorithm is based on the fact that there exists a schedule S that *fully* utilizes the timeline between 0 and d_n, following Corollary 5 and Theorem 11 (in other words, S is an FT-schedule with total reward X, where t_n may be augmented beyond its upperbound). The algorithm:

- Proceeds backwards from d_n and tries to schedule one (mandatory or optional) task at a time, at the given point of the schedule (represented by the variable $pointer$).

Algorithm DECIDE(M, O, X)

Set $\mathbf{O'} = SUFFIX(O, X)$
Set $\mathit{diff} = d_n - \sum_{i=1}^{n} m_i - \sum_{i=1}^{n} o'_i$
If $\mathit{diff} < 0$ then return FAILURE
        else $o'_n = o'_n + \mathit{diff}$
Set $pointer = shiftlimit = d_n$
Set $index = n$
Set $\mathbf{M_q} = \mathbf{M}$
Repeat
  While $pointer \leq d_{index}$ do {
        /* schedule optional part there */
        Set $pointer = pointer - o'_{index}$
        Set $Timeline[pointer, pointer + o'_{index}] = O_{index}$
        Set $index = index - 1$
                }
  /* Find the FT-forward schedulable mandatory
    part with the latest deadline and
    schedule it /*
  Set $M_a =$ FT-forward-sch$(pointer, shiftlimit)$
  If $M_a =$ nil then return FAILURE
  Set $pointer = pointer - m_a$
  Set $Timeline[pointer, pointer + m_a] = M_a$
  Set $\mathbf{M_q} = \mathbf{M_q} - M_a$
  If $d_a \geq shiftlimit$ then $shiftlimit = shiftlimit - m_a$
            else $shiftlimit = d_a - m_a$
Until $\mathbf{M_q} = \emptyset$
Return(Timeline, SUCCESS)

Figure 6.16: Algorithm to solve FT-Optimality Decision Problem

Function FT-forward-sch($pointer, shiftlimit$)

`Repeat`
  `Set` $counter = n$
  `If` $M_{counter}$ `in` $\mathbf{M_q}$ {
    `Set` $endpoint = pointer + b_{counter}$
    `if` $endpoint \leq min(shiftlimit, d_{counter})$ `return` $M_{counter}$
  }
  `Set` $counter = counter - 1$
`Until` $(counter = 0)$
`Return nil`

Figure 6.17: Function returning FT-Forward schedulable mandatory task

- If unable to find such a task, it exits with a FAILURE message.
- If more than one task is eligible for a given time point, it always favors an optional part (in case of ties, the optional task with the later deadline is chosen). The optional parts are only subject to the deadline constraints.
- If no optional part is eligible, then the mandatory part which is *FT-forward schedulable* at *pointer* is selected. We say that a mandatory part M_i is FT-forward schedulable at t_x, if scheduling M_i between $t_x - m_i$ and t_x does not cause M_i to miss its deadline (i.e., $t_x + b_i \leq d_i$) nor does it cause subsequent tasks to miss their deadlines, in case a fault occurs during the execution of M_i. In order to check how much the subsequent tasks can be pushed later in the timeline, the algorithm keeps a variable named $shiftlimit$. As with optional parts, ties are broken in favor of mandatory task with the later deadline.

Theorem 12 *Algorithm DECIDE solves the decision problem version of FT-Optimality for independent tasks.*

Proof: To begin with, the algoritm's commitment to non-preemptive tasks and to optional parts in SUFFIX(O,X) is immediately justified by Corollaries 4 and 5. Yet, we need to justify the selections of the algorithm whenever two (or more) tasks are eligible at a given point.

Favoring optional parts does not compromise correctness: Let O_x be the optional part (with the later deadline) such that $pointer \leq d_x$. Scheduling O_x at the $pointer$ clearly does not hurt FT-schedulability of the successive tasks. However, suppose that the algorithm's selection is incorrect, in the sense that it fails to find an FT schedule in the following steps, yet an FT-schedule S with the total reward X would have been obtained if a mandatory task M_i had been selected at this point.

The hypothetical schedule S is clearly fully utilized and O_x precedes M_i in S (see Figure 6.18). Consider forward-move of O_x after M_i to obtain the schedule S'. It should be straightforward to observe that two conditions of Theorem 8 still hold in S': The schedule segment up to and including M_x (in α) not being changed, M_x and all the tasks in γ are still safe. Hence the algorithm's selection of O_x in the first place would have also yielded an FT-schedule, and we reach a contradiction.

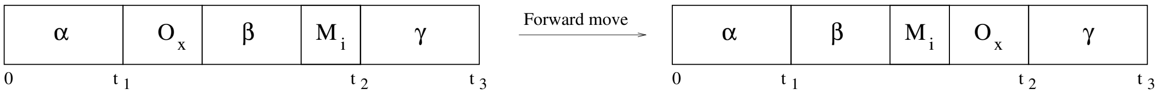


Figure 6.18: Forward move of the optional task

Favoring mandatory part (with the later deadline) does not compromise correctness: Note that the algorithm schedules a mandatory part only when it is FT-forward schedulable; this is clearly a necessary condition in an FT-schedule. As to justify the selection of the mandatory task with the later deadline M_x: suppose again that the algorithm fails after choosing M_x in consecutive steps, while the scheduling of another FT-forward schedulable task M_p would have yielded an FT-schedule S.

Again consider forward-move of M_x after M_p to obtain the schedule S' (See Figure 6.19). The second condition of Theorem 8 is immediately satisfied since M_x is FT-forward schedulable. M_x can not suffer from its own fault because of the same reason.

As the last possibility, suppose M_x misses its deadline in S', due to the fault of a task M_f preceding it. However, S was FT and M_p was safe in S, that is ,

$$End_S(M_p) + b_f - Opt(End_S(M_f), End_S(M_p)) \leq d_p$$

. Clearly, $End_S(M_p) = End_{S'}(M_x)$ and, further:

$$Opt(End_S(M_f), End_S(M_p)) = Opt(End_{S'}(M_f), End_{S'}(M_x))$$

giving $End_{S'}(M_x) + b_f - Opt(End_{S'}(M_f), End_{S'}(M_x)) \leq d_p \leq d_x$. But this shows that M_x is also safe in S' and consequently S' is also FT, yielding a contradiction.

□

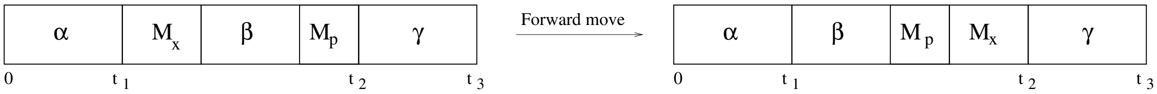


Figure 6.19: Forward move of the mandatory task with later deadline

Having an efficient decision procedure such as DECIDE above, we can easily develop an algorithm to compute the FT-optimal schedule, by adopting a binary-search like technique on the *maximum reward*. The lower and upper bounds for the binary search are initially set to 0 and $d_n - \sum_{i=1}^{n} m_i$, respectively (the latter being the maximum possible reward of the task set, FT or non-FT). The algorithm initially checks the existence of FT schedules with exactly these amount of rewards, and if needed, the 'middle' point as chosen as the average of these values. Lower and upper bounds are adjusted accordingly throughout the iterations.

Complexity: The time complexity of the algorithm DECIDE is clearly $O(n^2)$, since we schedule $2n$ tasks (n optional and n mandatory) and for each of the tasks we execute at most $O(n)$ operations (For mandatory tasks, the function FT-forward-sch and lines 16-20, for optional tasks lines 10-12). The final algorithm has $O(n^2 \log d_n)$ complexity, hence, it is a pseudo-polynomial time algorithm.

6.2.1.2 Extension to k Faults

In this section, we consider the problem of extending the FT-Optimality framework to the case of k faults with IR scheme. We assume that all the recovery blocks for a given task have the same worst-case execution time, and that all k faults *may* also occur during the execution and recoveries of a single task. The following serves as the basis for the approach.

Theorem 13 *A schedule* S *is tolerant to* k *faults if and only if it is tolerant to a single fault for the 'transformed' task set where* $b'_i = k \cdot b_i,\ i = 1 \ldots n$ *and other timing constraints are unchanged.*

Proof:

Note that in the transformed task set with $b'_i = k \cdot b_i$, the safety of a given task T_j is equivalent to:

$$End(M_j) + k\ b_i - Opt(End(M_i), End(M_j)) \leq d_j\ \forall i \text{ such that } End(M_i) \leq End(M_j) \quad (6.21)$$

Now, consider a scenario of k faults. In this case, a task T_j is safe, if and only if under any k-fault scenario preceding or occurring during its execution, the completion time $End(M_j)$ does not exceed d_j. Clearly, after the occurrence of the first fault, all the subsequent optional parts are removed. Let us denote by M_i the task incurring the *first* fault and by M_y the task having the recovery block with the largest execution time and executing between M_i and M_j (i.e. $End(M_i) \leq End(M_y) \leq End(M_j)$ — y may be equal to i or to j). Now, the safety condition for M_j can be expressed as:

$$\begin{gathered} End(M_j) + b_i + (k-1) \cdot b_y - Opt(End(M_i), End(M_j)) \leq d_j \\ \forall\ i \text{ such that } \quad End(M_i) \leq End(M_j) \end{gathered} \quad (6.22)$$

We claim that the equation (6.22) is equivalent to the equation (6.21). First, clearly (6.22) implies (6.21) since $b_y \geq b_i$ by definition.

Moreover, (6.21) also implies (6.22). Suppose the contrary, i.e. there is a k-fault scenario for which (6.21) is satisfied but (6.22) is not. Let M_i be the task which is affected by the *first* fault in this scenario, M_j the first task missing its deadline because of all k-faults occurring in-between M_i and M_j, and M_y the task having the largest recovery block such that $End(M_i) \leq End(M_y) \leq End(M_j)$. Since (6.21) is satisfied by assumption, we can infer that $End(M_j) + k \cdot b_y - Opt(End(M_y), End(M_j)) \leq d_j$. But clearly $Opt(End(M_y), End(M_j)) \leq Opt(End(M_i), End(M_j))$ and $b_y \geq b_i$ by definition, giving:

$$\begin{aligned} & End(M_j) + b_i + (k-1) \cdot b_y - Opt(End(M_i), End(M_j)) \\ & \leq End(M_j) + k \cdot b_y - Opt(End(M_y), End(M_j)) \leq d_j \end{aligned}$$

which implies that T_j was also safe for the given fault-scenario, yielding a contradiction. Repeating the argument for (the safety of) each task, we obtain the theorem.

□

The result underlines that the same FT-Optimality analysis and algorithm presented above can be applied to the *k*-fault case, just by scaling up the b_i values accordingly.

6.2.1.3 Intractability Results (Delayed Recovery and Non-identical Ready Times)

Unfortunately, our efforts to extend the FT-Optimality analysis to the *Delayed Recovery* or to *non-identical ready times* were inconclusive; mainly because the problem becomes provably intractable in these cases, as shown below.

Recall that in the Delayed Recovery (DR) case, the optional parts scheduled before d_i are used as recovery slots to achieve Fault Tolerance. Given a schedule S, we define the function $Slack(M_i)$ to be equal to the sum of optional parts scheduled after M_i but before d_i. We say that a schedule S is fault tolerant according to the DR scheme if $Slack(M_i) = Opt(End(M_i), d_i) \geq b_i \ \forall i$.

Theorem 14 *FT-Optimality with Delayed Recovery scheme is NP-Hard.*

Proof: We will prove the theorem showing that the FT-Optimality problem with DR scheme turns out to be NP-Hard *even for identical ready times and single fault case*. We will define a decision problem version of FT-Optimality, denoted by FT-DR-OPT, and show it is NP-Hard, which implies the intractability of the FT-Optimality problem. To prove NP-hardness of FT-DR-OPT, we will transform the KNAPSACK problem (which is NP-Complete), to an instance of FT-DR-OPT.

FT-DR-OPT: Given a set of Reward-Based tasks $\{T_i\}$, deadlines $\{d_i\}$ and an integer R representing the total reward of the system, is there an FT-Optimal schedule (according to the DR scheme) whose total reward is larger than or equal to R?

KNAPSACK: Let there be a set of items $U = \{u_1, u_2, \ldots, u_n\}$, integers s_i (size) and v_i (value) for each $u_i \in U$, positive integers B and K denoting the size of the knapsack and the value of the items in the knapsack, respectively. Is there a subset $U' \subseteq U$ such that $\sum_{u_i \in U'} s_i \leq B$ and $\sum_{u_i \in U'} v_i \geq K$?

Transformation: Let $S = \sum\limits_{i=1}^{n} s_i$ and $V = \sum\limits_{i=1}^{n} v_i$. Note that the KNAPSACK problem is NP-Complete only if $K \leq V$ and $B < S$ (for if $K > V$ then the answer is definitely 'no'; similarly if $B \geq S$ then the answer is quickly found by checking whether $V \geq K$). Hence, in the following discussion, we will assume that $K + B < S + V$.

If an instance of KNAPSACK is given, we construct a corresponding instance of FT-DR-OPT with $n+1$ *independent* reward based tasks. The first n tasks have the following characteristics:

$$m_i = s_i \quad o_i = v_i \quad b_i = 0 \quad d_i = K + S + V \quad i = 1, \ldots, n$$

We also introduce an extra task T_0 with the following parameters:

$$m_0 = K \quad o_0 = 0 \quad b_0 = K \quad d_0 = 2K + B$$

For convenience, the schedule to fill out is shown in Figure 6.20.

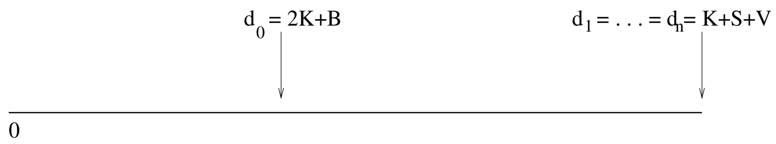


Figure 6.20: The schedule timeline

Now, consider the question: Is there an FT-Optimal schedule of $T_0, T_1, \ldots, T_n$ with the total reward $\geq V = \sum\limits_{i=1}^{n} v_i$? We claim that this instance of FT-DR-OPT is equivalent to that of KNAPSACK.

First, note that $\sum_{i=0}^{n} o_i = V$, hence the inequality above can be readily considered as equality. The total reward is V, implying $t_i \geq o_i = v_i, \quad \forall\, i$.

Moreover, the length of the schedule is $L = d_1 = \ldots = d_n = K + S + V = \sum_{i=0}^{n} m_i + \sum_{i=0}^{n} o_i$ which can be true only when $t_i = o_i \quad \forall\, i$ and when there exists no idle slots in the schedule (otherwise the total reward of $V = \sum_{i=0}^{n} o_i$ can not be reached). Thus the question above can be re-formulated as: "Is there an FT schedule with no idle slots and all optional parts are scheduled entirely?"

We note that $b_0 = K$ is the only recovery block whose execution time is larger than 0, that is, we require fault tolerance only for M_0. Finally, it is easy to see that M_0 and (some) optional parts of total length at least K should be scheduled before d_0 in any FT schedule.

We prove the equivalence of two decision problem instances in two parts:

1. If there is a solution to the KNAPSACK instance, then there is also a solution to the FT-DR-OPT instance:

First note that if U' is a solution set to the KNAPSACK instance, then $\sum_{u_i \in U'} s_i \leq B$ and $\sum_{u_i \in U'} v_i \geq K$. Let $\alpha = \{M_i | u_i \in U'\}$, $\beta = \{O_i | u_i \in U'\}$, $\alpha' = \{M_i | u_i \in U - U'\}$ and $\beta' = \{O_i | u_i \in U - U'\}$. Consider the schedule given in Figure 6.21.

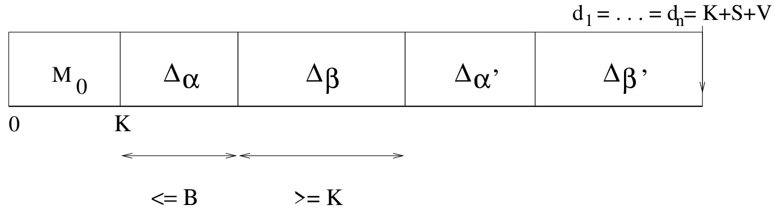


Figure 6.21: The FT-Optimal schedule

In this schedule Δ_X denotes the schedule segment where *only* tasks in the set X are scheduled. In other words, we schedule the tasks in the following order: M_0, the mandatory parts in α, the optional parts in β, the mandatory parts in α', the optional parts in β'. It should be clear that the schedule is feasible, contains no idle slots and $t_i = o_i \quad \forall\, i$.

However, in order to show that it is also Fault-Tolerant, we need to prove that $Slack(M_0) \geq b_0$, that is, the sum of optional parts scheduled before $d_0 = 2K + B$ is at least K. Recall that $m_o = K$. Also, from the solution of the KNAPSACK: $|\Delta_\alpha| = \sum_{M_i \in \alpha} m_i \leq B$ and $|\Delta_\beta| = \sum_{O_i \in \beta} o_i \geq K$. Thus, d_0 is definitely after Δ_α and, $Slack(M_0) \geq d_0 - B - m_0 = d_o - B - K$, which implies $Slack(M_0) \geq 2K + B - B - K$, or $Slack(M_0) \geq K$, which proves the statement.

2. If there is a solution to the FT-DR-OPT instance, then there is also a solution to the KNAPSACK instance:

Suppose that there exists a solution to the FT-DR-OPT instance, hence an FT schedule with total reward V (with no idle slots and $t_i = o_i \quad \forall\, i$). Clearly, $Slack(M_0) \geq K$ in this schedule.

Let t_i^o be the optional service time of T_i in the interval $[0, d_0]$. We remark that $\sum_{i=1}^{n} t_i^o \geq K$, otherwise the slack constraint of M_0 can not be satisfied. Also, observe that if $t_i^o > 0$, then M_i definitely starts *and* completes in the interval $[0, d_0]$, since otherwise O_i can not start executing at all.

Let $\gamma = \{i\, | t_i^o > 0 \text{ and } i > 0\}$. It is easy to see that the following relations hold:

$$\sum_{i \in \gamma} o_i \geq \sum_{i \in \gamma} t_i^o \geq K \tag{6.23}$$

$$\sum_{i\in\gamma} m_i \le d_o - m_o - \sum_{i\in\gamma} t_i^o \le 2K + B - K - K = B \tag{6.24}$$

Recalling that $m_i = s_i$ and $o_i = v_i$ for $i = 1, \ldots, n$, we conclude from (6.23) and (6.24) that $\sum_{i\in\gamma} m_i = \sum_{i\in\gamma} s_i \le B$ and $\sum_{i\in\gamma} o_i = \sum_{i\in\gamma} v_i \ge K$, or equivalently $U' = \{u_i \mid i \in \gamma\}$ is the solution to the KNAPSACK problem, proving the statement and the theorem.

□

Another intractability result concerns the Immediate Recovery (IR) scheme when tasks are allowed to have non-identical ready times.

Theorem 15 *FT-Optimality problem with Immediate Recovery scheme and non-identical ready times is NP-Hard.*

Proof: The reduction is again from the KNAPSACK instance mentioned used in the proof of Theorem 14. The decision problem version we consider is FT-IR-NONID: Given a set of Reward-Based tasks $\{T_i\}$, deadlines $\{d_i\}$, ready times $\{r_i\}$ and an integer R, is there an FT-Optimal schedule with IR whose total reward is larger than or equal to R?

Given an instance of the KNAPSACK (s_i, v_i, B and K values), we construct an instance of FT-IR-NONID this time with $n+2$ Reward-Based tasks. The first n tasks have the following characteristics:

$$m_i = s_i \quad o_i = v_i \quad b_i = 0 \quad r_i = 0 \quad d_i = K + S + V + 1 \quad i = 1, \ldots, n$$

We introduce two additional tasks T_0 and T_{n+1} with the following parameters:

$$m_0 = K \quad o_i = 0 \quad b_0 = K \quad r_i = 0 \quad d_i = 2K + B$$

$$m_{n+1} = 1 \quad o_{n+1} = 0 \quad b_{n+1} = 0 \quad r_{n+1} = 2K + B \quad d_{n+1} = 2K + B + 1$$

where r_{n+1} is the ready time of the task T_{n+1}. Note that again the only task which requires FT is T_0 and the task T_{n+1} is only ready to execute at the deadline of task T_0. Considering the ready time and the deadline of M_{n+1}, we obtain the timeline shown in Figure 6.22.

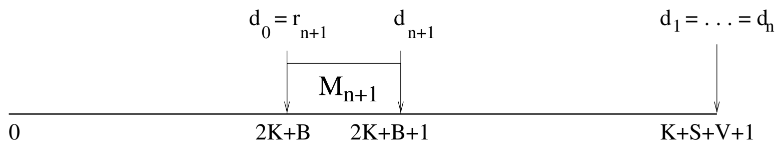


Figure 6.22: The schedule timeline

Now, consider the question: Is there an FT-Optimal schedule of $T_0, T_1, \ldots, T_n, T_{n+1}$ with the total reward larger than or equal to $V = \sum_{i=1}^{n} v_i$?

id	m_i	o_i	d_i	$b_{i,1}$	$b_{i,2}$	$F_i(t)$
T_1	5	25	25	5	3	$5\,t$
T_2	5	10	30	5	1	$4\,t$
T_3	5	20	35	5	1	t

Table 6.4: Example Task Set

The need to provide FT for M_0 before $d_0 = r_{n+1} = 2K + B$ and the strict constraint that M_{n+1} can not be moved, imply that we must have at least K units of optional parts scheduled before d_0. Combining this with the requirement that every optional part should be used entirely to achieve the total reward V and hence, to fully utilize the timeline, we can infer that the interval $[0, 2K + B]$ may be used as a 'template' for the KNAPSACK problem. The details of the proof are analogous to the proof of Theorem 14 and omitted here.

□

6.2.2 Chain Tasks

In this section, we address the FT-Optimality problem for a chain of Reward-based tasks with potentially multiple deadlines. We will allow general concave reward functions, and further assume that the recovery blocks associated with multiple faults of a given task may have different execution times. The worst-case execution time of the recovery block associated with the j^{th} fault of M_i, will be denoted by $b_{i,j}$. In our analysis, we will directly formulate and solve the problem for k faults.

k-Fault Tolerant Optimal Scheduling– A Motivating Example: We present an example with three tasks having linear dependence relations among them. The parameters of the task set are shown in Table 6.4. We assume that only two faults need to be tolerated (i.e., $k = 2$). Further, although non-linear reward functions are more realistic, the tasks in the example have linear reward functions, for the sake of simplicity. Despite its simplicity, the example illustrates many facets of scheduling Reward-Based tasks in the presence of multiple faults.

As it can be seen, each task has two recovery blocks. The first recovery block for each task is merely re-execution ($b_{i,1} = m_i,\ \forall i)$. The worst-case execution times of the second recovery blocks are much smaller than the first ones. Observe that the marginal return of O_1 is the largest among all tasks, hence, the schedule with the highest reward consists in favoring O_1 as much as possible (Figure 6.23); call this schedule S_1.

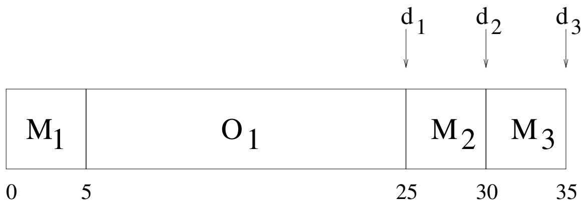


Figure 6.23: An optimal but non-FT Schedule; reward=100

Schedule S_1 is optimal (total reward of 100), but it is not FT: M_2 and M_3 complete just before their deadlines and it is not possible to tolerate even a single fault which occurs in either of them.

Provided that a task set is k-FT, one can always obtain a trivial FT-schedule by completing every mandatory part as soon as possible, and the remaining idle time will be assigned to the *last* optional part. If we adopt this strategy, we end up by getting S_2, shown in Figure 6.24.

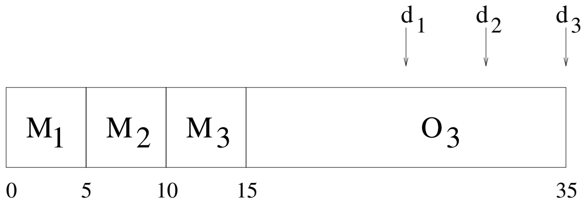


Figure 6.24: A trivial FT Schedule; reward=20

However, the total reward of S_2 is only 20, due to low marginal return of O_3. The approach in Section 6.1 enforced that the sum of optional assignments after a given task (and before its deadline) should be at least equal to the recovery time that may be needed. Further each recovery block was assumed to have the same worst execution time. However, this approach is inadequate for multiple deadlines, in that it excludes some legal k-FT schedules. Indeed, using that strategy, the schedule with the highest reward is S_3 (Figure 6.25, total reward of 55).

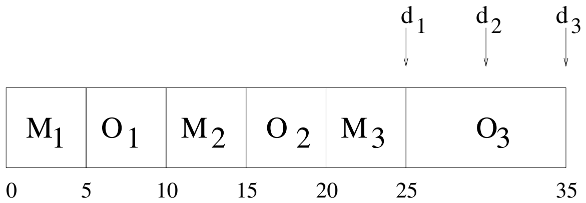


Figure 6.25: A k-FT Schedule with reward=55

Yet, there is another schedule S_4 with considerably higher reward (72), shown in Figure 6.26, which tolerates any pattern of k faults in addition to taking into account non-identical deadlines and non-identical recovery blocks. Yet, it can be easily verified that it is $k-$FT and, in fact, S_4 is a k-FT Optimal schedule for the example task set.

In summary, from this example it is clear that new and effective fault tolerance criteria are needed for the case of task sets with multiple deadlines and different recovery blocks.

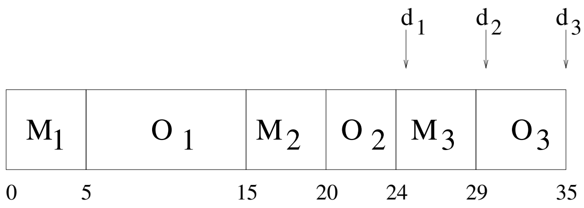


Figure 6.26: The k-FT Optimal Schedule; reward=72

6.2.2.1 Exact characterization of k-FT Schedules

In this section, we consider and solve the following problem: **Given a schedule S of reward-based chain tasks, how can we efficiently check its tolerance to any pattern of *k* faults?** We underline that the characterization developed in this chapter applies also to *any* RB task schedule regardless of the precedence constraints' type, as long as Immediate Recovery is employed.

Naturally, one can solve the problem by generating all the schedules corresponding to the $O(n^k)$ fault patterns and checking the timeliness of each of them in $O(n)$ time, which yields a total time complexity of $O(n^{k+1})$. This is clearly an unacceptable computational cost. The solution that we present stems from a rigorous characterization of FT schedules and it is simultaneously efficient and exact. Besides, such a rigorous characterization of *all* the FT schedules for a given task set gives us valuable insights to achieve our ultimate aim of generating FT-Optimal schedules.

Definition 1 *A* multiple fault pattern *is a set* $P = \{p_1, p_2, \ldots, p_n\}$*, where* p_i *denotes the* actual *number of faults that task* T_i *incurs and such that,* $\sum_{i=1}^{n} p_i \leq k$*.*

Note that, a schedule is k-FT if and only if every mandatory part M_i and any of its executed recovery blocks, can complete by d_i under *any* k-fault pattern P. Clearly, the k-fault tolerance of a schedule implies its feasibility, but the converse is not true.

Definition 2 *A schedule of reward-based tasks* S *is called* k-FT at level i *if it remains feasible after recoveries of any fault pattern* P *where* $p_j = 0, \quad \forall\, j < i$*.*

Informally, level i k-fault tolerance enforces that the schedule remain feasible if *all* the faults occur during or after the execution of M_i (for convenience, we define $p_0 = 0$). The following propositions can be easily verified in view of the above definition.

Proposition 15 *A schedule S is k-FT at level* n *if and only if the schedule remains feasible in case that all the k faults occur in the last task M_n (or its recovery blocks).*

Proposition 16 *A schedule S is k-FT at level $i < n$ if and only if:*

1. *it is k-FT at level $i + 1$, and,*
2. *under any k fault pattern which begins with j faults of M_i, the tasks $M_i, M_{i+1}, \ldots, M_n$ and their recovery blocks complete before their corresponding deadlines.*

Since any k-fault pattern should begin with an error detected at the end of a specific M_i $(1 \leq i \leq n)$, Proposition 16 lets us to deduce the following:

Proposition 17 *A task set* **T** *is k-FT if and only if there exists a schedule which is k-FT at level 1.*

A strict characterization of k-fault tolerance will be obtained iteratively by studying tolerance of consecutive levels. Specifically, we will first obtain necessary and sufficient conditions for a level n k-FT system, then for level $n-1$ and so on, until we reach level 1. Clearly, the first condition is that the schedule should be k-FT at level n, that is, M_n should be scheduled early enough to leave time for recoveries of k faults of M_n. Similarly, k fault tolerance at level $n-1$ requires that M_{n-1} be scheduled early enough to let timely recoveries of *any k faults* which affect M_n and M_{n-1}, and so on. Note that while computing the upper bound on the scheduling time of M_i, we should consider the worst-case scenario which requires maximum recovery time after $End(M_i)$, which corresponds to a k-fault pattern exclusively affecting $M_i, M_{i+1}, \ldots, M_n$. Let us define a function over individual tasks to resume our analysis.

Definition 3 *The function $LCT(M_i)$, denotes the latest completion time of M_i in any schedule for* **T***, which allows the timely completions (and recoveries) of the mandatory parts $M_i, M_{i+1}, \ldots, M_n$ under any k-fault pattern which begins with a fault on M_j $(j \geq i)$, that is, $p_h = 0$ $h = 1, \ldots, i-1$.*

Informally, $LCT(M_i)$ is a measure of the maximum amount M_i can be pushed late in the schedule, without compromising level i k-Fault Tolerance. Using Propositions 16 and 17, we obtain:

Proposition 18 *A schedule S is k-FT at level i if and only if $End_S(M_j) \leq LCT(M_j) \quad j = i, \ldots, n$.*

Corollary 6 *A task set is k-FT if and only if there exists a schedule S where $End_S(M_j) \leq LCT(M_j), \;\; j = 1, \ldots, n$.*

Therefore, once $LCT(M_i)$ is determined for every task, the k-fault tolerance of a given schedule can be decided in $O(n)$ time. Hence, the problem is now reduced to the efficient computation of $LCT()$ values for each task.

Computation of $LCT()$ Values:

We will adopt the dynamic programming approach to compute the LCT values in a bottom-up manner. First, we define an auxiliary function.

Definition 4 *The function $lct(M_i, u, v)$ denotes the latest completion time of M_i in any schedule for* **T***, which allows the timely completions (and recoveries) of the mandatory parts M_i, $M_{i+1}, \ldots,$ M_n under any fault scenario where there are exactly u fault(s) during the execution of $M_1, \ldots, M_{i-1}$ and exactly v fault(s) during the execution of $M_i, \ldots, M_n$, such that $u + v = k$.*

Again, the $lct()$ function puts an upper bound on the amount of shift a task M_i can tolerate in a schedule, in a fault pattern with exactly u faults before it, and exactly v faults on and after it. Note that if M_i itself actually incurs exactly y faults in that specific scenario, it should be scheduled to complete early enough to be able to execute y recovery blocks before d_i, as well as not to push the tasks which follow beyond what is mandated by level i k-Fault Tolerance. Notice that the value of the $LCT(M_i)$ function is (by definition) simply $lct(M_i, 0, k)$. Let us start by evaluating the lct value for the last task M_n. In case that exactly i faults occur in M_n, we should have time to execute all the first i recovery blocks before the deadline d_n, which gives:

$$lct(M_n, k-i, i) = d_n - \sum_{j=1}^{i} b_{n,j} \tag{6.25}$$

Clearly, $LCT(M_n)$ is the minimum among all $lct(M_n, k-y, y)$ values, which correspond to the case where all the faults occur in or after M_n (in this case, all faults will occur *in* M_n):

$$LCT(M_n) = lct(M_n, 0, k) = d_n - \sum_{j=1}^{k} b_{n,j} \tag{6.26}$$

For $i < n$, the lct function will be computed by a dynamic programming approach:

$$lct(M_i, k, 0) = min\{d_i, lct(M_{i+1}, k, 0) - m_{i+1}\} \tag{6.27}$$

The above formula expresses the fact that, assuming that all the faults occur before M_i, $lct(M_i, k, 0)$ is either d_i, or the latest *start* time of M_{i+1} under this scenario (which is $lct(M_{i+1}, k, 0) - m_{i+1}$), whichever is the smallest.

Similarly:

$$lct(M_i, k-1, 1) = min\{X_0, X_1\}$$

$$X_0 = min\{d_i, lct(M_{i+1}, k-1, 1) - m_{i+1}\} \quad X_1 = min\{d_i, lct(M_{i+1}, k, 0) - m_{i+1}\} - b_{i,1}$$

Above, X_0 corresponds to the scenario where the single fault which is supposed to occur during the execution of $M_i, \ldots, M_n$, does *not* occur in M_i, but later; while X_1 captures the case where the single fault occurs in fact in M_i in which case no fault can occur in $M_{i+1}, \ldots, M_n$. We can obtain:

$$lct(M_i, k-v, v) = \min_{j=0}^{v}\{X_j\} \quad v = 0, \ldots, k \text{ where :} \tag{6.28}$$

$$X_j = min\{d_i, lct(M_{i+1}, k-(v-j), v-j) - m_{i+1}\} - \sum_{y=1}^{j} b_{i,y} \tag{6.29}$$

for the general case. Note that the set $\{X_j\}$ above expresses all possible k-fault scenarios where we have exactly $k-v$ faults before M_i and exactly v faults on or after M_i: M_i can incur exactly j faults and the tasks $M_{i+1}, \ldots, M_n$ can incur exactly $v-j$ faults. Once again, $LCT(M_i)$ can be computed by taking the minimum among all $lct(M_i, k-y, y)$ values, which always happens to be $lct(M_i, 0, k)$ —the scenario where all the faults occur during the execution of $M_i, \ldots, M_n$. Observe further that $lct(M_i, k-v, v)$ values for $v < k$ are not used for the computation of $LCT(M_i) = lct(M_i, 0, k)$, instead they contribute to the computation of $LCT(M_j) \ \ (j < i)$.

What is the complexity of computing $lct()$ values for a given task set? First, the computation of $\sum_{y=1}^{j} b_{i,y} \ \ i = 1, \ldots, n; \ \ j = 0, \ldots, k$ can be accomplished in $O(n \cdot k)$ time and we can store them in a look-up table. Further, there are only $n \cdot (k+1)$ function values to be computed. Each of them can be done in $O(k)$ time (we need to perform at most $O(k)$ comparisons and arithmetic operations), which suggests that the overall complexity is $O(n \cdot k^2)$.

Once we compute $LCT(M_i)$ values, we can quickly test the k-fault tolerance of a given schedule S, by checking whether $End_S(M_i) \leq LCT(M_i) \ \ i = 1, \ldots, n$. We conclude the analysis by pointing out that, if during the checking procedure we observe the existence of an $LCT(M_i)$ value such that $LCT(m_i) - m_i < 0$, it immediately implies that there are **no k-FT schedules for the given task set**: there can be no $End_S(M_i)$ values which can satisfy the requirement of Corollary 6, if this is the case. In other words, the test has the additional property of being able to detect task sets for which no k-FT schedules exist.

Example

Let us illustrate the application of the technique on the example task set of Section 6.2.2. We start by the last mandatory task, which is M_3. By applying Equation (6.25), we find

$lct(M_3, 2, 0) = 35$, $lct(M_3, 1, 1) = 30$ and $lct(M_3, 0, 2) = 29$. Hence, the worst-case scenario for level 3 fault tolerance is the occurrence of two faults in M_3; and $LCT(M_3) = lct(M_3, 0, 2) = 29$.

For M_2, we compute three values, namely $lct(M_2, 2, 0)$, $lct(M_2, 1, 1)$, and $lct(M_2, 0, 2)$:
$lct(M_2, 2, 0) = min\{d_2, lct(M_3, 2, 0) - m_3\} = 30$,
$lct(M_2, 1, 1) = min\{X_0, X_1\} = 25$; where
$X_0 = min\{30, lct(M_3, 1, 1) - m_3\} = 25$ and $X_1 = min\{30, lct(M_3, 2, 0) - m_3\} - b_{2,1} = 25$.

Note that X_0 above corresponds to the fault pattern in which the single fault supposed to occur on or after M_2, actually occurs later, not in M_2. In contrast X_1 captures the case where M_2 actually suffers a fault but M_3 does not; we should incorporate the execution of the first recovery block $B_{2,1}$ in the computation. As it turns out, the two scenarios impose the same upper bound on the $lct()$. Lastly, $lct(M_2, 0, 2)$ involves the consideration of three different scenarios:
$lct(M_2, 0, 2) = min\{X_0, X_1, X_2\} = 20$; where
$X_0 = min\{30, lct(M_3, 0, 2) - m_3\} = 24$ (both faults occur in M_3);
$X_1 = min\{30, lct(M_3, 1, 1) - m_3\} - b_{2,1} = 20$ (one fault occurs in M_2 and the other in M_3.)
$X_2 = min\{30, lct(M_3, 2, 0) - m_3\} - b_{2,1} - b_{2,2} = 24$ (both faults occur in M_2).

From these results, $LCT(M_2)$ is evaluated to be equal to X_1, which is 20. Observe that, for level-2 fault tolerance, the worst-case scenario corresponds to the one where both M_2 and M_3 fail once. Finally, we focus on M_1:
$lct(M_1, 2, 0) = min\{d_1, lct(M_2, 2, 0) - m_2\} = 25$,
$lct(M_1, 1, 1) = min\{X_0, X_1\} = 20$; where
$X_0 = min\{25, lct(M_2, 1, 1) - m_2\} = 25$ and $X_1 = min\{25, lct(M_2, 2, 0) - m_2\} - b_{2,1} = 20$.

Computing $lct(M_1, 0, 2)$ also requires a comparison of three values:
$lct(M_1, 0, 2) = min\{X_0, X_1, X_2\} = 15$; where
$X_0 = min\{25, lct(M_2, 0, 2) - m_2\} = 15$ (both faults occur after M_1);
$X_1 = min\{25, lct(M_2, 1, 1) - m_2\} - b_{1,1} = 15$ (one fault occurs in M_1 and the other one after it.)
$X_2 = min\{25, lct(M_2, 2, 0) - m_2\} - b_{1,1} - b_{1,2} = 17$ (both faults occur in M_1).

Level 1 Fault-tolerance requires that M_1 be scheduled no later than $t = 15$, otherwise a single fault of M_1 and a single fault of M_2 would result in a deadline miss. In other words, the $LCT()$ bounds are evaluated as:

$$LCT(M_1) = 15, \quad LCT(M_2) = 20, \quad LCT(M_3) = 29.$$

6.2.2.2 Generation of k-FT optimal schedules for chain tasks

In this section, we address the problem of generating the k-FT schedule of a set of chain tasks with multiple deadlines to yield the maximum reward. Corollary 6 in Section 6.2.2.1 revealed the necessary and sufficient conditions for k-fault tolerance, which involves only pre-computation of $LCT(M_i)$ bounds and checking $End(M_i)$ values against these. Without loss of generality, we consider FT-Optimal schedules with no idle time[1].

To obtain a k-FT optimal schedule S^* for a given task set, we will start creating an initial (template) schedule S_t which contains *only* mandatory parts. Further, in S_t, each mandatory part M_i will be scheduled exactly in time interval $[LCT(M_i) - m_i, LCT(M_i)]$. Figure 6.27 shows the template schedule S_t for the example task set of Table 6.4 (Page 118).

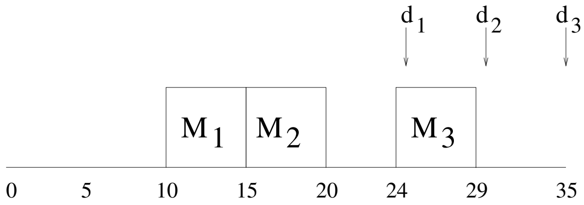


Figure 6.27: The template schedule S_t

Note that mandatory tasks do not overlap in S_t since:

$$\begin{aligned} LCT(M_i) = lct(M_i, 0, k) &\leq min\{d_i, lct(M_{i+1}, 0, k) - m_{i+1}\} \\ &\leq lct(M_{i+1}, 0, k) - m_{i+1} = LCT(M_{i+1}) - m_{i+1} \end{aligned}$$

Thus, S_t will contain mandatory parts scheduled at their latest completion times, with (possibly) gaps among them. However, $Rew_{S_t} = 0$ since we did not schedule any optional part yet. Observe that, in any k-FT schedule, and incidentally in the k-FT Optimal schedule denoted by S^*, **any mandatory part can be moved earlier from its original allocation in S_t to create room for optional parts if needed, but never forward**, since this would result in a non-FT schedule (Corollary 6). For example, Figure 6.28 illustrates three fully utilized schedules corresponding to the template schedule S_t. They all satisfy the k-fault tolerance requirement: yet, the total rewards are different, and any of them could be the k-FT optimal schedule, depending on the specific reward functions. To summarize, S_t represents the boundary conditions that any k-FT schedule must satisfy.

To maximize the total reward, we will try to schedule optional parts into the 'gaps'. Yet, there are several necessary constraints that we should obey. Namely, no optional part O_i can be scheduled beyond deadline d_i. Similarly, precedence constraints and fault tolerance requirements enforce the following:

[1] If there is an idle time in the middle of an optional execution, one can execute that optional task for one more time unit. Idle time which occur during mandatory parts can be removed, effectively decreasing the completion times and not hurting k-fault tolerance. Clearly, the total reward never decreases after such schedule modifications.

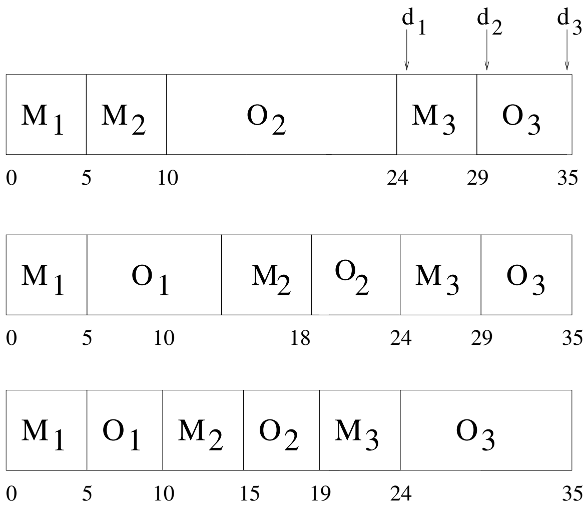


Figure 6.28: Three k-FT schedules

Proposition 19 *In the schedule segment $[LCT(M_i), d_i]$ of any k-FT schedule S, no optional part O_j such that $j < i$ can be scheduled.*

Proof: Suppose there is an optional part O_j scheduled after $LCT(M_i)$ $(i > j)$ in S. Since M_i should be scheduled later than O_j, $End_S(M_i) > LCT(M_i)$, which suggests that S is not k-FT, leading to a contradiction.

□

Hence, we obtain the following **necessary** conditions for k-fault tolerance:

N1: In interval $[d_i, d_{i+1}]$ the only optional parts that can be scheduled are $O_{i+1}, \ldots, O_n$.

N2: In interval $[LCT(M_i), d_n]$ the only optional parts that can be scheduled are $O_i, \ldots, O_n$.

We can reflect the constraints N1 and N2 in the optional assignments as follows. Let us start by noting that $Start_{S_t}(M_i) = End_{S_t}(M_i) - m_i = LCT(M_i) - m_i$ is the latest start time of M_i in any k-FT schedule. Clearly, the total CPU allotment for the first task, namely $m_1 + t_1$, cannot exceed d_1 (constraint N1), or $Start_{S_t}(M_2)$ (constraint N2):

$$m_1 + t_1 \leq min\{d_1, Start_{S_t}(M_2)\}$$

Similarly, the total assignment $m_1 + t_1 + m_2 + t_2$ should satisfy the deadline constraint for T_2 and LCT constraint for M_3:

$$m_1 + t_1 + m_2 + t_2 \leq min\{d_2, Start_{S_t}(M_3)\}$$

And in general:

$$\sum_{i=1}^{j}(m_i + t_i) \leq min\{d_j, Start_{S_t}(M_{j+1})\} \; j < n$$

Finally, the schedule will contain no idle time:

$$\sum_{i=1}^{n}(m_i + t_i) = d_n$$

We continue with the definitions of two other quantities, namely the effective deadline and its corresponding constraint set:

$$\tilde{d}_i = \begin{cases} min\{d_i, Start_{S_t}(M_{i+1})\} & \text{if} \quad i < n \\ \quad d_n & \text{if} \quad i = n \end{cases}$$

$$C_i : \text{the constraint} \begin{cases} \sum_{j=1}^{i}(m_j + t_j) \leq \tilde{d}_i & \text{if} \quad i < n \\ \sum_{j=1}^{i}(m_j + t_j) = \tilde{d}_i & \text{if} \quad i = n \end{cases}$$

Note that $\tilde{d}_i$ represents the *effective* deadline of T_i, that is, the execution of T_i beyond this limit violates either its deadline or k-fault tolerance (for convenience, we define $\tilde{d}_0 = 0$).

The theorem below proves that the constraint set $\{C_i\}$, derived from conditions **N1** and **N2**, is **necessary and sufficient** for k-fault tolerance:

Theorem 16 *A schedule S is k-FT if and only if it satisfies the constraints $C_1, \ldots, C_n$.*

Proof: Suppose that each constraint $C_i, \quad i = 1, \ldots, n$ is satisfied. The schedule is feasible since $\sum_{j=1}^{i}(m_j + t_j) \leq \tilde{d}_i \leq d_i \quad i = 1, \ldots, n$. But also, $\sum_{j=1}^{i-1}(m_j + t_j) \leq \tilde{d}_{i-1} \leq Start_{S_t}(M_i) \quad i = 1, \ldots, n$. Hence, S is also k-FT since each M_i finishes before $LCT(M_i)$:

$$\sum_{j=1}^{i-1}(m_j+t_j)+m_i = End_S(M_i) \leq Start_{S_t}(M_i)+m_i = End_{S_t}(M_i) = LCT(M_i) \quad i = 1, \ldots, n$$

Conversely, suppose that the schedule S is k-FT. This implies:

$$End_S(M_{i+1}) = \sum_{j=1}^{i}(m_j + t_j) + m_{i+1} \leq LCT(M_{i+1}) = End_{S_t}(M_{i+1}) \quad i = 0, \ldots, n-1$$

By subtracting m_{i+1} from both sides, we obtain:

$$\sum_{j=1}^{i}(m_j + t_j) \leq End_{S_t}(M_{i+1}) - m_{i+1} = Start_{S_t}(M_{i+1}) \quad i = 0, \ldots, n-1$$

But the schedule is also feasible and contains no idle slots, hence:

$$\sum_{j=1}^{i}(m_j + t_j) \leq d_i \quad i = 1, \ldots, n-1 \qquad \text{and} \qquad \sum_{j=1}^{n}(m_j + t_j) = d_n$$

Combining last results, we obtain:

$\sum_{j=1}^{i}(m_j+t_j) \leq min\{d_i, Start_{S_t}(M_{i+1})\} \quad i = 1, \ldots, n-1$ and $\sum_{j=1}^{n}(m_j+t_j) = d_n$, yielding:

$$\begin{aligned} \sum_{j=1}^{i}(m_j + t_j) &\leq \tilde{d}_i \quad i = 1, \ldots, n-1 \\ \sum_{j=1}^{n}(m_j + t_j) &= \tilde{d}_n \end{aligned}$$

Hence, all the C_i constraints are satisfied.

□

Having expressed the k-fault tolerance conditions in terms of optional service assignments $\{t_i\}$, we can now formulate the non-linear optimization problem. Note that every t_i assignment should be non-negative, in order to have a physical interpretation. Further, the total reward should be maximized. Hence, we obtain our final theorem:

Theorem 17 *k-FT optimal optional service assignments $\{t_i\}$ are given by the solution of the following optimization problem:*

$$\text{maximize} \qquad \sum_{i=1}^{n} R_i(t_i) \tag{6.30}$$

$$\text{subject to} \qquad \sum_{i=1}^{n} t_i = d_n - \sum_{i=1}^{n} m_i \tag{6.31}$$

$$0 \leq t_i \qquad i = 1, \ldots, n \tag{6.32}$$

$$\tilde{d}_n - \tilde{d}_{i-1} - \sum_{j=i}^{n} m_j \quad \leq \sum_{j=i}^{n} t_j \quad i = 2, \ldots, n \tag{6.33}$$

Proof:

The constraint (6.31) corresponds to C_n. The constraint (6.32) is self-explanatory. The constraint set (6.33) is equivalent to $\{C_i\}$, as proven below. Consider the constraint set:

$$\begin{aligned} m_n + t_n &\geq \tilde{d}_n - \tilde{d}_{n-1} \\ m_{n-1} + t_{n-1} + m_n + t_n &\geq \tilde{d}_n - \tilde{d}_{n-2} \\ &\vdots \end{aligned}$$

$$\sum_{j=n-i+1}^{n}(m_j+t_j) \geq \tilde{d}_n - \tilde{d}_{n-i} \quad i=3,\ldots,n-1$$
$$\vdots$$
$$\sum_{j=1}^{n}(m_j+t_j) = \tilde{d}_n - \tilde{d}_0$$

Now, let us define:

$$\overline{C_i} : \text{the constraint} \begin{cases} \sum_{j=n-i+1}^{n}(m_j+t_j) \geq \tilde{d}_n - \tilde{d}_{n-i} & \text{if} \quad i<n \\ \sum_{j=1}^{n}(m_j+t_j) = \tilde{d}_i - \tilde{d}_0 & \text{if} \quad i=n \end{cases}$$

We claim that the constraint set $\{C_i\}$ is satisfied for a given schedule S if and only if the constraint set $\{\overline{C_i}\}$ is satisfied.

In fact, suppose that the set $\{C_i\}$ is satisfied. Then by subtracting C_{n-i} from C_n, one can obtain $\{\overline{C_i}\}$ (for $i=1,\ldots,n-1$). C_n is identical to $\overline{C_n}$ ($d_n = \tilde{d}_n$ and $\tilde{d}_0 = 0$).

Conversely, suppose that the set $\{\overline{C_i}\}$ is satisfied. One can subtract $\overline{C_{n-i}}$ from $\overline{C_n}$ to get C_i (for $i=1,\ldots,n-1$), and so on. By re-arranging the new constraint set $\{\overline{C_i}\}$, we obtain the formulation needed.

□

The above problem is in fact an instance of the non-linear (concave) optimization problem CHAIN and can be solved by using the algorithm ALG-CHAIN(Section 6.1.2). The complexity of the solution is $O(n^2 \log n)$ for linear, logarithmic or identical concave reward functions; in the general concave case, the complexity is $O(n^3 \log n)$.

To illustrate the derivation of exact constraint set for the CHAIN, we return to the example task set of Table 6.4 (Page 118), of which the template schedule S_t was shown in Figure 6.27. The 'effective' deadline for each task is the minimum of its deadline and the start time of the next task in S_t, thus:

$$\tilde{d}_1 = 15,\ \tilde{d}_2 = 24,\ \tilde{d}_3 = 35$$

.

Similarly, the inequality constraints are obtained as:

$$m_3 + t_3 \geq \tilde{d}_3 - \tilde{d}_2 = 11$$
$$m_2 + t_2 + m_3 + t_3 \geq \tilde{d}_3 - \tilde{d}_1 = 20$$

By substituting the values of $m_i \quad i=1,2,3$, we obtain the instance of CHAIN:

$$
\begin{aligned}
\text{maximize} \sum_{i=1}^{n} \quad & R_i(t_i) & & \\
\text{subject to} \sum_{i=1}^{n} t_i \quad & = & 20 & \\
0 \quad & \leq & t_i & \quad i = 1, 2, 3 \\
6 \quad & \leq & t_3 & \\
10 \quad & \leq & t_2 + t_3 &
\end{aligned}
$$

The algorithm which solves the problem CHAIN (Section 6.1.2) returns the output set $\{t_1 = 10, t_2 = 4, t_3 = 6\}$, which, indeed corresponds to the k-FT Optimal schedule which was previously shown in Figure 6.26.

6.3 Chapter Summary

In this chapter, we addressed FT-Optimal scheduling of aperiodic reward-based tasks. Our algorithms are optimal, in the sense that they simultaneously address the issues of provisioning for timely recoveries of mandatory parts and maximizing the total reward (in the fault-free execution).

We first analyzed the FT-Optimality problem for tasks sharing a single deadline. We showed that for independent tasks, as long as the task set is FT, one can obtain an FT schedule with the same reward as the (non-FT) optimal schedule by postponing the execution of all the optional parts until after all the mandatory parts complete. For chain tasks, we proved the optimality of a least commitment strategy. Both algorithms operate in polynomial-time.

For multiple deadlines case, we proposed two different recovery schemes when analyzing independent tasks. In *Delayed Recovery*, only schedule segments which are originally allocated to optional parts are used for recovery; while in *Immediate Recovery*, the recovery block is executed as soon as the error is detected by postponing the execution of all other tasks and by possibly sacrificing optional parts. We show that the FT-Optimality problem with Delayed Recovery is inherently intractable, while the one with Immediate Recovery admits an optimal solution in pseudo-polynomial time. When considering chain tasks, we first provided an exact characterization of RB schedules that can tolerate up to k faults. Next, based on this characterization, we gave an FT-Optimal algorithm of polynomial-time complexity.

Chapter 7
Conclusion

Reward-based (RB) scheduling is a general framework which unifies hard and soft real-time scheduling theories by addressing feasibility and maximum utility issues simultaneously. Assuring timely completion of mandatory parts guarantees an acceptable overall performance, while optimal scheduling of optional parts maximizes the total user-perceived utility (reward). RB scheduling promotes graceful degradation and resource utilization where worst-case guarantees can not be given due to faults, overload conditions and for applications that admit an intra-task mandatory/optional division. However, simultaneously achieving feasibility and optimality is not a trivial problem, since the reward functions associated with the optional parts bring another dimension to the traditional scheduling framework.

In addition to solving some open problems in the RB scheduling area, this dissertation also justified the use of the framework as a general paradigm in real-time systems research, capable of expressing multiple requirement dimensions at once. In particular, beyond the timeliness and the utility (reward) aspects, our work in power-aware and fault tolerant scheduling hints to the fact that the dichotomy of the CPU capacity as mandatory and optional workloads is effectively a *generic* one: the former is the basis for acceptable performance guarantees, while the latter can be employed to provide fault tolerance, enhanced precision and/or slack for lower CPU speed/power. This is another incentive to look for further improvements in this area, since having requirements on multiple dimensions such as *timeliness, fault tolerance, utility* and *power-efficiency* is a characteristic of emerging applications.

An important contribution of this thesis concerns the *periodic reward-based scheduling* problem. Prior to our research, only heuristics that focus on executing the mandatory parts as soon as possible were proposed for the *instance-based* periodic reward-based scheduling and imprecise computation model. Our main result in this area indicates that, for the most common (i.e., linear and concave) reward functions, it is always possible to find an optimal schedule where the service time of a given task remains constant from instance to instance. This, in turn, implies that well-known scheduling policies such as RMS (with harmonic periods), EDF and LLF can be used to meet all the timing constraints with these optimal assignments. Another important implication of this result is the apparent *convergence* of *instance-based* and *utilization-based* [85] models for concave reward functions.

We also showed that the same methodology readily extends to the multiprocessors case. The computation of optimal service times in polynomial-time involved solving an optimization problem through the careful use of Kuhn-Tucker conditions for non-linear programs. Finally, we investigated the limits of optimality of identical service times by considering different models. A few relatively simple counter-examples showed that this strong result does not generalize to the systems where the deadlines may be different than the period, the dynamic priority scheduling policies are not available or the reward functions are not necessarily concave. We also showed that relaxing the concavity assumption about the reward functions results in an NP-Hard problem.

Another contribution of this thesis is in the *power-aware scheduling* area. Despite the increasing interest in scheduling with adjustable CPU speed and the existence of some sound studies, to the best of our knowledge, the equivalence of power-aware scheduling problem to the reward-based scheduling problem has been established for the first time in this work. Based on this equivalence, we provided a static optimal solution to compute the speed assignments at the task-level, assuming a worst-case workload for the periodic model.

However, tasks may (and often in practice, do) complete earlier than predicted in the worst-case schedule. For this reason, further slowing down the CPU by detecting early completions at run-time is crucial for improving power-efficiency. Yet, in real-time settings, the amount/duration of speed reduction and the beneficiary task(s) should be determined with extreme care. Our *Generic Dynamic Reclaiming Algorithm* provides a sound basis to obtain additional (up to 50%) energy savings while still meeting all the deadlines. Finally, we proposed and evaluated an *aggressive* scheme, which *anticipates early completions in the future* and reduces the speed accordingly. There is surely a risk factor in this speculative approach, in terms of hurting the energy savings in the long-run. Nonetheless, we show that an additional 20% improvement is possible through the use of an appropriate *aggressiveness* level while preserving the feasibility of the schedule.

Finally, the development of *FT-Optimality* framework for the aperiodic RB tasks is another contribution of this research effort. While the RB models allow for greater scheduling flexibility, the mandatory parts have still *hard* deadlines and they should be completed in a timely manner even in the occurrence of faults. Thus, we investigated FT-optimal scheduling algorithms which maximize the total reward while allowing time for recovery from errors of mandatory parts. Focusing first on tasks sharing a single deadline, we showed that the problem admits a polynomial-time solution for both independent and chain tasks, for general concave reward functions.

For tasks having linear precedence constraints (chain tasks), with potentially different deadlines, we showed how to obtain an exact characterization of the schedules which allow timely recovery of k faults in mandatory faults. Then, using this characterization, we provided a polynomial-time scheme to generate the FT-Optimal solution. The problem becomes remarkably harder for independent tasks with multiple deadlines. We proposed two viable recovery schemes (Immediate Recovery and Delayed Recovery) for this case. The FT-Optimality with Delayed Recovery is shown to be intractable even for the simplest assumptions, while the same holds for the Immediate Recovery once we allow different ready times. Our efforts resulted in a provably optimal pseudo-polynomial time algorithm for Immediate Recovery.

Chapter 8

Future Work

Each of the research problems we addressed in this dissertation can be expanded and improved in various ways. In the following, we elaborate on the research avenues that we consider particularly promising for future work in this area.

Despite the optimal solution we obtained in Periodic Reward-Based Scheduling assuming dynamic priority, we believe that investigating effective scheduling strategies for fixed priority policies, such as Rate Monotonic Scheduling, deserves particular attention. As we showed in Chapter 4, the optimality of identical service times no longer holds for fixed priority policies and in general, one has to deal with an exponential number of unknowns (the service times of optional parts) for an exact solution. On the other hand, investigating the average and worst-case performances of schemes which use as the basis identical service times in fixed priority policies would be particularly interesting. Thanks to the connection we established between reward-based and power-aware scheduling frameworks, attempts to provide sub-optimal, yet effective solutions will be also relevant for power-constrained real-time embedded systems where Rate Monotonic Scheduling policy is commonly in use. In particular, the means to prove and improve our conjecture about the performance of schedules constructed using identical service times (page 36) deserve immediate attention.

In the power-aware scheduling area, there are several opportunities (and challenges) for further improvement, especially for dynamic reclaiming and aggressive speed reduction schemes. In dynamic reclaiming, our current scheme is computing the earliness factor by considering only the slack of already completed tasks with an earlier deadline. On the other hand, reclaiming blindly the slack-time of tasks with later deadline can easily compromise feasibility, as we showed with an example in Chapter 5. Devising intelligent schemes which would allow such reclaiming decisions while preserving feasibility is a natural research avenue in this area. In addition, our aggressive speed reduction scheme uses a limited horizon which extends up to the next task arrival. Again, investigating aggressive schemes that would perform speculative moves not constrained by this limitation while maintaining feasibility may be the key to further energy savings.

The power-aware scheduling problem for discrete voltage levels, despite its intractability, has special importance for systems in which there are only a few available speed settings. The adaptation of the dynamic reclaiming and aggressive speed reduction schemes to this model would be particularly useful. In addition, the potential application of the *resource augmentation* technique [46] deserves particular attention, for on-line power-aware scheduling settings. In particular, a characterization of the aggressive approaches using this technique might lead to a better understanding of the worst-case and average-case behavior. For example, one might attempt to determine the initial speed assignments considering the average-case workload on an underutilized processor and then resort to a high CPU speed on a fully utilized processor to meet the deadlines if the prediction turns out to be incorrect.

On the FT-Optimality side, the investigation of efficient approximation schemes for the problems we showed to be intractable, especially for independent tasks model, is particularly important and necessary before the framework can be extended to more general precedence relations.

This dissertation can also be considered as the application of reward-based scheduling to systems with multiple requirement dimensions. Although we effectively addressed timeliness-utility (Chapter 4), timeliness-power efficiency (Chapter 5) and timeliness-utility-fault tolerance (Chapter 6) dimensions throughout the dissertation, there are certainly other combinations worth exploring from both theoretical and practical aspects. As an example, incorporation of power-awareness into the RB task systems, that is, addressing power efficiency, utility and timeliness issues all at once (power-aware reward-based scheduling) is a promising research avenue. Similarly, fault tolerance may be a requirement for some power-aware hard real-time applications: in this case provisioning for timely recoveries may be a constraint that would limit the speed reduction schemes (fault-tolerant power-aware scheduling). These more general models may also involve multiple 'reward' dimensions, one expressing the energy savings, and the other, the precision of the output produced. Depending on user preferences, or availability of resources, the relative importance of these two dimensions may even dynamically change.

Bibliography

Bibliography

[1] A. Avizienis and J.P. Kelly. Fault-tolerance by design diversity. *IEEE Computer*, 17:67-80, 1984.

[2] H. Aydın, R. Melhem and D. Mossé. A Polynomial-time Algorithm to solve Reward-Based Scheduling Problem. *Technical Report 99-10*, Department of Computer Science, University of Pittsburgh, April 1999.

[3] T. P. Baker. Stack-based scheduling of real-time processes. *Journal of Real-Time Systems*, 3, 1991.

[4] S. Baruah, G. Koren, D. Mao, B. Mishra, A. Ragunathan, L. Rosier, D. Shasha, and F. Wang. On the competitiveness of on-line real-time task scheduling. *Journal of Real-Time Systems*, 4: 124-244, 1992.

[5] S. Baruah. Fairness in periodic real-time scheduling. In *Proceedings of 16th IEEE Real-Time Systems Symposium*, December 1995.

[6] S.K. Baruah, N.K. Cohen, C.G. Plaxton and D.A. Varvel. Proportionate Progress: A Notion of Fairness in Resource Allocation. *Algorithmica*, 15:600-625, 1996.

[7] S. K. Baruah and M. E. Hickey. Competitive on-line scheduling of imprecise computations. *IEEE Transactions on Computers,* pp. 1027-1033, September 1998.

[8] M. S. Bazaraa and C. M. Shetty. *Nonlinear Programming,* John Wiley and Sons, New York 1979.

[9] E. Chang and A. Zakhor. Scalable Video Data Placement on Parallel Disk data arrays. In *ISIT/SPIE Symposium on Electronic Imaging Science and Technology*, February 1994.

[10] J. L. Crowley. Vision as Process: Integration and Control of a Real Time Active Vision System. *Swiss Vision '93*, Zurich, September 1993.

[11] A. Bertossi and L. Mancini. Scheduling algorithms for Fault-Tolerance in Hard Real-Time Systems. *Real Time Systems Journal*, 7, 1994.

[12] R. Bettati, N.S. Bowen and J.Y. Chung. Checkpointing Imprecise Computation. *Proceedings of the IEEE Workshop on Imprecise and Approximate Computation*, Dec. 1992.

[13] R. Bettati, N.S. Bowen and J.Y. Chung. On-Line Scheduling for Checkpointing Imprecise Computation. *Proceedings of the Fifth Euromicro Workshop on Real-Time Systems*, June 1993.

[14] M. Boddy and T. Dean. Solving time-dependent planning problems. *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence,* IJCAI-89, pp. 979-984, August 1989.

[15] T. D. Burd, T. A. Pering, A. J. Stratakos and R. W. Brodersen. A Dynamic Voltage Scaled Microprocessor System. *IEEE Journal of Solid-State Circuits*, Vol. 35, No. 11, pp. 1571-1580, November 2000.

[16] A. Burns, R. Davis and S. Punnekkat. Feasibility Analysis of Fault-tolerant Real-Time Task sets. In *Euromicro Workshop on Real-Time Systems*, 1996

[17] E. Chang and A. Zakhor. Scalable Video Coding using 3-D Subband Velocity Coding and Multi-Rate Quantization. In *IEEE Int. Conf. on Acoustics, Speech and Signal processing*, July 1993.

[18] H. Chetto, M. Silly and T. Bouchentouf. Dynamic scheduling of real-time tasks under precedence constraints. *Journal of Real-Time Systems*, 1990.

[19] J.-Y. Chung and J. W.-S. Liu. Algorithms for scheduling periodic jobs to minimize average error. In *Proceedings of 9th IEEE Real-Time Systems Symposium*, December 1988.

[20] J.-Y. Chung, J. W.-S. Liu and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Transactions on Computers,* 19(9): 1156-1173, September 1990.

[21] M.L. Dertouzos. Control robotics: the procedural control of physical processes. *Information Processing*, 1974.

[22] J. K. Dey, J. Kurose, D. Towsley, C.M. Krishna and M. Girkar. Efficient On-Line Processor Scheduling for a Class of IRIS (Increasing Reward with Increasing Service) Real-Time Tasks. *Proceedings of ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems,* May 1993.

[23] J. K. Dey, J. Kurose and D. Towsley. On-Line Scheduling Policies for a class of IRIS (Increasing Reward with Increasing Service) Real-Time Tasks. *IEEE Transactions on Computers* 45(7):802-813, July 1996.

[24] E. W. Dijkstra. Cooperating sequential processes. In *Programming Languages*, Academic Press, New York, 1968.

[25] R. Ernst and W. Ye. Embedded Program Timing Analysis based on Path Clustering and Architecture Classification. In *Computer-Aided Design (ICCAD)'97*. pp. 598-604.

[26] W. Feng and J.W.-S. Liu. Algorithms for Scheduling Real-Time Tasks with Input Error and End-to-End Deadlines. *IEEE Transactions on Software Engineering,* 23(2): 93-106, February 1997.

[27] W. Feng and J. W.-S. Liu. An extended imprecise computation model for time-constrained speech processing and generation. In *Proceedings of the IEEE Workshop on Real-Time Applications*, May 1993.

[28] S. Ghosh, D. Mosse and R. Melhem. Enhancing Real-Time Schedules to Tolerate Transient Faults. In *Proceedings of 16th IEEE Real-Time Systems Symposium*, December 1995.

[29] S. Ghosh, D. Mosse, R. Melhem and J. Sensarma. Fault-Tolerant Rate-Monotonic Scheduling. *Journal of Real-Time Systems*, 1998.

[30] K. Govil, E. Chan and H. Wasserman. Comparing algorithms for dynamic speed-setting of a low-power CPU. In *ACM International Conference on Mobile Computing and Networking*, pp. 13-25, 1995.

[31] J. Grass and S. Zilberstein. Value-Driven Information Gathering. *AAAI Workshop on Building Resource-Bounded Reasoning Systems,* Rhode Island, 1997.

[32] V. Gutnik and A. Chandrakasan. An Efficient Controller for Variable Supply Voltage Low Power Processing. *Symposium on VLSI Circuits*, pp.158-159, 1996.

[33] P. J. M. Havinga and G. J. M. Smith. Design Techniques for Low-power Systems. *Journal of Systems Architecture*. Vol. 46:1, 2000

[34] B. Hayes-Roth, Architectural foundations for real-time performance in intelligent agents. *The Journal of Real-Time Systems*, 2:99-125, 1990.

[35] K.I.J. Ho, J.Y.T. Leung and W.D. Wei. Scheduling Imprecise Computations with 0/1 constraint. *Technical Report UNL-CS-92-13*, Department of Computer Science and Engineering, University of Nebraska, 1992

[36] I. Hong, D. Kirovski, G. Qu, M. Potkonjak and M. Srivastava. Power optimization of variable voltage core-based systems. In *Design Automation Conference*, 1998.

[37] I. Hong, M. Potkonjak and M. B. Srivastava. On-line Scheduling of Hard Real-Time Tasks on Variable Voltage Processor. In *Computer-Aided Design (ICCAD)'98*. pp. 653-656.

[38] I. Hong, G. Qu, M. Potkonjak and M. Srivastava. Synthesis Techniques for Low-Power Hard Real-Time Systems on Variable Voltage Processors. In *Proceedings of 19th IEEE Real-Time Systems Symposium (RTSS'98)*, Madrid, December 1998.

[39] E.J. Horvitz. Reasoning under varying and uncertain resource constraints *Proceedings of the Seventh National Conference on Artificial Intelligence,* AAAI-88, pp. 111-116, August 1988.

[40] C. Hwang and A.C.-H. Wu. A Predictive System Shutdown Method for Energy Saving of Event-Driven computation. In *IEEE/ACM International Conference on Computer-Aided Design*, pp. 28-32, 1997.

[41] Intel, Microsoft, and Toshiba. Advanced Configuration and Power Management Interface (ACPI) Specification, 1999. www.intel.com/ial/WfM/design/pmdt/acpidesc.htm.

[42] R.K. Iyer, D.J. Rossetti and M.C. Hsueh. Measurement and Modeling of Computer Reliability as Affected by System Activity. In *ACM Transactions on Computer Systems*, 4(3): 214-237, Aug 1986.

[43] J.R. Jackson. Scheduling a production line to minimize maximum tardiness. *Management Science Research project No 43*, University of California, LA, 1955.

[44] K. Jeffay, D. F. Stanat, and C. U. Martel. On non-preemptive scheduling of periodic and sporadic tasks with varying execution priority. In *Proceedings of 12th IEEE Real-Time Systems Symposium*, December 1991.

[45] B. W. Johnson and P.M. Julich. Fault-Tolerant Computer System for the A129 Helicopter. *IEEE Transactions on Computers*, 33: 518-528, 1984.

[46] . B. Kalyanasundaram and K. Pruhs. Speed is as powerful as clairvoyance. *IEEE Symposium on Foundations of Computation*, pp. 214 - 221, 1995.

[47] G. Koren and D. Shasha. D^{over}: An Optimal On-line Scheduling Algorithm for Overloaded Real-time Systems. In *Proceedings of 13th IEEE Real-Time Systems Symposium*, December 1992.

[48] R. E. Korf. Real-time heuristic search. *Artificial Intelligence,* 42(2): pp.189 -212, 1990.

[49] C. M. Krishna and Y. H. Lee. Voltage Clock Scaling Adaptive Scheduling Techniques for Low Power in Hard Real-Time Systems. In *Proceedings of the 6th IEEE Real-Time Technology and Applications Symposium (RTAS'00)*, Washington D.C., May 2000.

[50] C. M. Krishna and K. G. Shin. *Real-time Systems.* Mc Graw-Hill, New York 1997.

[51] S. Lauzac. *On Multiprocessor Scheduling of Preemptive Periodic Real-Time Tasks with Error Recovery*, Ph.D. thesis, 2000.

[52] E. L. Lawler. Optimal sequencing of a single machine subject to precedence constraints. *Managements Science*, 1973.

[53] J. Lehoczky, L. Sha and Y. Ding. The Rate-Monotonic Scheduling: Exact Characterization and Average Case Behavior. In *Proceedings of 10th IEEE Real-Time Systems Symposium*, December 1989.

[54] C. Lee, R. Rajkumar, J. P. Lehoczky and D. P. Siewiorek. Practical solutions for QoS-based Resource Allocation Problems. In *Proceedings of 19th IEEE Real-Time Systems Symposium*, December 1998.

[55] C. Lee, R. Rajkumar, J. P. Lehoczky and D. P. Siewiorek. On Quality of Service Optimization with Discrete QoS Options. In *Proceedings of the IEEE Real-Time Technology and Applications Symposium*, June 1998.

[56] C. Lee, J. P. Lehoczky, D. P. Siewiorek, R. Rajkumar and J. Hansen. A Scalable Solution to the Multi-Resource QoS problem. In *Proceedings of 20th IEEE Real-Time Systems Symposium*, December 1999.

[57] Y. H. Lee and C. M. Krishna. Voltage-Clock Scaling for Low Energy Consumption in Real-time Embedded Systems. In *The Sixth International Conference on Real-Time Computing Systems and Applications (RTCSA'99)*, Hong Kong, December 1999.

[58] J. K. Lenstra, A. H. G. Rinnooy Kan, and P. Brucker. Complexity of machine scheduling problems. *Annals of Discrete Mathematics*, (1):343-362, 1977.

[59] J. Y. T. Leung and C.S. Wong. Minimizing the number of late tasks with error constraint. In *Proceedings of 11th IEEE Real-Time Systems Symposium*, December 1990.

[60] J. Y. T. Leung, T. W. Tam, C.S. Wong and G.H. Young. Minimizing mean flow time with error constraints. In *Proceedings of 10th IEEE Real-Time Systems Symposium*, December 1989.

[61] F. Liberato, S. Lauzac, R. Melhem and D. Mosse. Global Fault-Tolerant Real-Time Scheduling on Multiprocessors. In *Euromicro Workshop on Real-time Systems*, 1999.

[62] F. Liberato, R. Melhem and D. Mosse. Tolerance to Multiple Transient Faults for Aperiodic Tasks in Hard Real-time Systems. *IEEE Transactions on Computers*, 49(9), September 2000.

[63] A. L. Liestman and R.H. Campbell. A Fault-Tolerant Scheduling problem. *IEEE Transactions on Software Engineering*, 12(11):1089-1095, November 1988.

[64] K.-J. Lin, S. Natarajan, and J. W. -S. Liu. Imprecise Results: Utilizing partial computations in real-time systems. In *Proceedings of 8th IEEE Real-Time Systems Symposium*, December 1987.

[65] J. W. -S. Liu, K.-J. Lin, and S. Natarajan. Scheduling real-time, periodic jobs using imprecise results. In *Proceedings of 8th IEEE Real-Time Systems Symposium*, December 1987.

[66] Jane W.-S. Liu, K.-J. Lin, W.-K. Shih, A. C.-S. Yu, C. Chung, J. Yao and W. Zhao. Algorithms for scheduling imprecise computations. *IEEE Computer,* 24(5): 58-68, May 1991.

[67] C.L. Liu and J.W.Layland. Scheduling Algorithms for Multiprogramming in Hard Real-time Environment. *Journal of ACM* 20(1), 1973.

[68] C.D. Locke. Best Effort Decision Making for Real-Time Scheduling. *Ph.D. thesis. Carnegie Mellon University, Pittsburgh, PA*, 1986.

[69] D. Luenberger, *Linear and Nonlinear Programming,* Addison-Wesley, Reading Massachusetts, 1984.

[70] H. Mehta, R. M. Owens, M. J. Irwin, R. Chen and D. Ghosh. Techniques for Low Energy Software. In *Proceedings of the 1997 International Symposium on Low Power Electronics*. August 1997, Monterey, CA.

[71] R. Min, T. Furrer and A. Chandrakasan. Dynamic Voltage Scaling Techniques for Distributed Microsensor Networks. In *Proceedings of the IEEE Computer Society Annual Workshop on VLSI (WVLSI '00)*, April 2000.

[72] A.K. Mok, *Fundamental Design Problems of Distributed systems for the Hard Real-Time Environment.* Ph.D. Dissertation, MIT, 1983.

[73] D Mossé. Mechanisms for System-Level Fault Tolerance in Real-Time Systems. In *Int'l Conf on Robotics, Vision, and Parallel Processing for Industrial Automation*, June 1994.

[74] W. Namgoong, M. Yu and T. Meg. A High Efficiency Variable-Voltage CMOS Dynamic DC-DC Switching regulator. *IEEE International Solid-State Circuits Conference*, pp.380-391

[75] L. D. Nguyen and A. M. K. Cheng. An Imprecise Real-Time Image Magnification Algorithm. In *International Conference on Multimedia Systems*, 1996.

[76] Y. Oh and S. H. Son. An Algorithm for Real-Time Fault-Tolerant Scheduling in a Multiprocessor System. In *4th Euromicro Workshop on Real-Time Systems*, June 1992.

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

[78] M. J. Panik. *Classical Optimization: Foundations and Extensions*, North-Holland, Amsterdam, 1976.

[79] M. Pedram. Power Minimization in IC Design: Principles and Applications. *ACM Transactions on Design Automation of Electronics Systems*. 1:1 - pp. 3-56, January 1996.

[80] P. Pedro and A. Burns. Schedulability Analysis for Mode Changes in Flexible Real-Time Systems. In *Proceedings of the 10th Euromicro Workshop on Real-Time Systems*, June 1998.

[81] T. Pering, T. Burd and R. Brodersen. Voltage Scaling in the IpARM Microprocessor System. In *Proceedings of the 2000 International Symposium on Low power Electronics and design*.

[82] C. A. Phillips, C. Stein, E. Torng and J. Wein. Optimal Time-Critical Scheduling via Resource Augmentation. In *ACM Symposium on Theory of Computing*, pp. 140-149, 1997.

[83] J. Pouwelse, K. Langendoen and H. Sips. Dynamic Voltage Scaling on a Low-Power Microprocessor. In *Proceedings of Mobile Computing Conference (MOBICOM)*, 2001.

[84] D. K. Pradhan. *Fault-Tolerant Computer System Design* Prentice-Hall, New Jersey, 1996.

[85] R. Rajkumar, C. Lee, J. P. Lehoczky and D. P. Siewiorek. A Resource Allocation Model for QoS Management. In *Proceedings of 18th IEEE Real-Time Systems Symposium*, December 1997.

[86] B. Randell. System Structure for Software Fault Tolerance. *IEEE Transactions on Software Engineering,* 1(2):220–232, June 1975.

[87] L. Sha, R. Rajkumar and J.P. Lehoczky. Priority inheritance protocols: An approach to real-time synchronization. *IEEE Transactions on Computers*, 39(9), September 1990.

[88] W.-K. Shih, J. W.-S. Liu, J.-Y. Chung and D.W. Gillies. Scheduling tasks with ready times and deadlines to minimize average error. *ACM Operating Systems Review*, July 1989.

[89] W.-K. Shih, J. W.-S. Liu, and J.-Y. Chung. Algorithms for scheduling imprecise computations to minimize total error. *SIAM Journal on Computing,* 20(3), July 1991.

[90] W.-K. Shih and J. W.-S. Liu. Algorithms for scheduling imprecise computations with timing constraints to minimize maximum error. *IEEE Transactions on Computers,* 44(3):466-471, March 1995.

[91] W.-K. Shih and J. W.-S. Liu. On-line scheduling of Imprecise Computations to Minimize Error. *SIAM Journal on Computing,* October 1996.

[92] Y. Shin and K. Choi. Power Conscious Fixed Priority Scheduling for Hard Real-Time Systems. In *Proceedings of the 36th Design Automation Conference, DAC'99*, pp. 134-139.

[93] M. Srivastava, A. P. Chandrakasan and R. W. Brodersen. Predictive System Shutdown and other Architectural Techniques for Energy Efficient Programmable Computation. *IEEE Transactions on VLSI Systems*, 4(1): 42-55, 1996.

[94] J. Stankovic, M. Spuri, K. Ramamritham and G. Buttazzo (eds). *Deadline Scheduling for Real-Time Systems : EDF and Related Algorithms*. Kluwer Academic Publishers, 1998.

[95] G. E. Tellez, A. Farrahi and M. Sarrafzadeh. Activity-driven clock design for low power circuits. In *Proceedings of the IEEE International Conference on Computer Aided Design*, 1995.

[96] S. Ramos-Thuel and J.K. Strosnider. Scheduling Fault Recovery Operations for Time-Critical Applications. In *4th IFIP Conference on Dependable Computing for Critical Applications*, 1995.

[97] http://www.transmeta.com

[98] C. J. Turner and L. L. Peterson. Image Transfer: An end-to-end design. In *SIGCOMM Symposium on Communications Architectures and Protocols*, August 1992.

[99] S. V. Vrbsky and J. W. S. Liu. Producing monotonically improving approximate answers to relational algebra queries. In *Proceedings of IEEE Workshop on Imprecise and Approximate Computation,* December 1992.

[100] M. Weiser, B. Welch, A. Demers and S. Shenker. Scheduling for reduced CPU energy. In *USENIX Symposium on Operating Systems Design and Implementation*, pp. 13-23, 1994.

[101] F. Yao, A. Demers and S. Shankar. A Scheduling Model for Reduced CPU Energy. *IEEE Annual Foundations of Computer Science*, pp. 374 - 382, 1995.

[102] A. Yu. Scheduling parallel real-time tasks that allow imprecise computations. Ph.D. thesis. *Technical Report UIUCDCS-R-92-1738*, Univ. of illinois, 1992

[103] A. Yu and K.J. Lin. A scheduling algorithm for replicated real-time tasks. *Proc. Phoenix Conference on Computers and Communications*, pp.395-402, April 1992.

[104] S. Zilberstein and S.J. Russell. Anytime Sensing, Planning and Action: A practical model for Robot Control. In *IJCAI 13*, France, 1993.