Utility Accrual Real-Time Scheduling: Models and Algorithms

Peng Li

Dissertation submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Doctor of Philosophy
in
Computer Engineering

Binoy Ravindran, Chair
E. Douglas Jensen
Pushkin Kachroo
Scott F. Midkiff
Amitabh Mishra
Richard E. Nance

July 20, 2004
Blacksburg, Virginia

Keywords: Time/utility functions, utility accrual scheduling, real-time scheduling, performance assurance, resource management, overload scheduling

Utility Accrual Real-Time Scheduling: Models and Algorithms

Peng Li

(ABSTRACT)

This dissertation first presents an uniprocessor real-time scheduling algorithm called the Generic Benefit Scheduling algorithm (or GBS). GBS solves a previously open real-time scheduling problem: scheduling activities subject to arbitrarily shaped, time/utility function (TUF) time constraints and mutual exclusion resource constraints. A TUF specifies the utility of completing an application activity as an application- or situation-specific function of when that activity completes. GBS considers the scheduling objective of maximizing system-wide, total accrued utility, while respecting mutual exclusion constraints. Since this problem is $\mathcal{NP}$-hard, GBS heuristically computes schedules in polynomial-time.

The performance of the GBS algorithm is evaluated through simulation and through an implementation on a Portable Operating System Interface (POSIX)-compliant real-time operating system. The simulation studies and implementation measurements reveal that GBS performs close to, if not better than existing algorithms for the cases that they apply. Further, the results verify the effectiveness of GBS for its unique model. We also analytically establish timeliness and non-timeliness properties of GBS including bounds on activity utilities and mutual exclusion.

GBS targets real-time systems that are subject to significant non-determinism inherent in their operating environments e.g., completely unknown activity arrivals. When system uncertainties can be stochastically characterized (e.g., stochastic activity arrivals and execution times), it is possible to provide stochastic assurances on timeliness behavior.

The dissertation also presents algorithmic solutions to fundamental assurance problems in TUF-driven real-time systems, including stochastically satisfying individual, activity utility lower bounds and system-wide, total utility lower bounds. The algorithmic solutions include algorithms for processor bandwidth allocation and TUF scheduling. While bandwidth allocation algorithms allocate processor bandwidth share to activities to satisfy utility

lower bounds, TUF scheduling algorithms schedule activities to maximize accrued utility. The algorithmic solutions and analysis are extended with a class of lock-free and lock-based resource access protocols to satisfy mutual exclusion constraints. We show that satisfying utility lower bounds with lock-based resource access protocols does not imply doing so with the lock-free scheme, and vice versa. Finally, the dissertation presents a rule-based framework for trading off assurance requirements on utility lower bound satisfaction.

This work is partially supported by the U.S. Office of Naval Research (ONR) under Grant N00014-99-1-0158 and N00014-00-1-0549, the MITRE Corporation under Grant 52917, and the QNX Software Systems Ltd. (SSL) through a software grant. The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of ONR, MITRE, or QNX SSL.

Acknowledgments

Many people deserve thanks for helping this dissertation happen. First, I would like to thank my advisor, Dr. Binoy Ravindran. Dr. Ravindran has been very supportive in every aspect of my four-year stay at Virginia Tech. He always believes in my ability to solve problems, even when myself have doubts. Without his support, encouragement, and mentoring, little can I achieve in the past four years. Most often, it is fun to chat with him of research. Some times, I just get too bored to even think about anything of research and feel desperate. Under those situations, it is certainly not fun to talk about research. But he can always pass on his passions about research to me and ignite my desires to move forward.

I am also indebted to Dr. E. Douglas Jensen for his tremendous help and support. He is always willing to listen and to make encouraging comments. In fact, he has suggested the problem of providing performance assurance for UA scheduling that is presented in this dissertation. During the last one and half year, he has spent significant amount of time discussing issues involved in the problem and possible approaches, through e-mails, phone calls, and face-to-face meetings. When I look back, I feel that I am truly honored to have the opportunity of working with Dr. Jensen, a person who has pioneered the whole space of time/utility functions and utility accrual scheduling thirty years ago. His support even goes beyond technical advising. Part of this dissertation research is directly funded through his research project at MITRE.

My other committee members provide many helpful and encouraging comments as well. Besides my committee members, I would like to thank Dr. Ray Clark at MITRE, Dr. Gérard Le Lann at INRIA, and Dr. Doug Locke at Locke Consulting for their support and many helpful discussions. Particularly, Dr. Clark has participated many of our discussions with Dr. Jensen and provided insightful comments. This dissertation, in part, follows and borrows some of the fine ideas developed by Dr. Clark and Dr. Locke. In addition, the solution to the performance assurance problem is inspired and influenced by Dr. Lann.

I am also fortunate to have the opportunity to work with a group of energetic people at Dr. Ravindran's Real-Time Systems Laboratory. I have enjoyed every moment that we have worked together including many sleepless nights. I want especially to thank Tamir Hegazy, Gene Wang, Haisang Wu, Umut Balli, Hyeonjoong

Cho, and Shahrooz Feizabadi, for their direct contributions to components of my dissertation research. Furthermore, Michelle Gong deserves special thanks and appreciations for all the moments we have shared together.

Last, but not least, I would like to thank my family members. When I was born some thirty years ago, my parents never dreamed that I could one day earn a PhD degree. But they are patient and encouraging. Without their constant love and support, I will not be where I am now. My grandmother brought me up and always wishes me the best with her unconditional love. My brother, Dr. Yong Li, has served as my mentor since I was a teenager. He has helped me a lot on almost every single aspect of my journey. Most importantly, through him, I had the vision of the future.

This dissertation is to all people who have helped and are helping me. I am very grateful for all their help.

Contents

List of Algorithms

List of Figures

List of Tables

Chapter 1

Background and Introduction

Real-time computing is fundamentally concerned with satisfying application timing constraints. The most widely studied timing constraint is the deadline. A deadline timing constraint for an application activity essentially implies that completing the activity before its deadline accrues some "utility" to the system and that utility remains the same if the activity were to complete *anytime* before the deadline. Furthermore, completing the activity after the deadline is unacceptable. With deadline timing constraints, one can specify the hard timeliness optimality criterion of always satisfying all deadlines and use hard real-time scheduling algorithms, such as Rate Monotonic Scheduling and Earliest Deadline First [42] to achieve the criterion.

This dissertation focuses on dynamic real-time systems that occur at any level(s) of an enterprise—examples in the defense domain include those at the device level (multi-mode phased array radars [22]), platform level (surveillance aircrafts [23, 21, 16]), and mission level (netted sensor-to-shooter control loops in network-centric warfare [20, 66]). Such systems include time constraints which are "soft" (besides those that are hard) in the sense that completing an activity at any time will result in some (positive or negative) utility to the system, and that utility depends on the activity's completion time. Moreover, they often desire a soft timeliness optimality criterion such as completing all activities as close as possible to their *optimal* completion times—so as to yield maximal collective utility—is the objective.

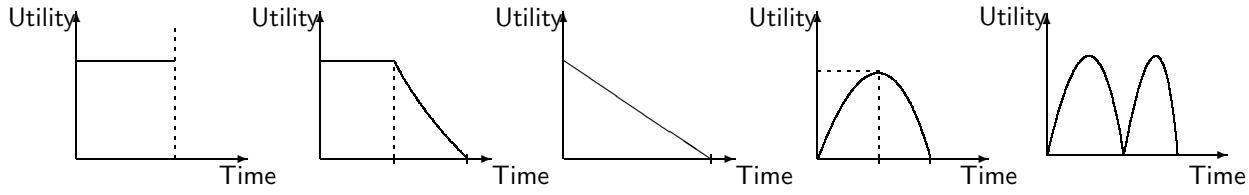
Many enterprise-level real-time systems are subject to significant non-determinism that is inherent in their operating environments, causing great run-time uncertainties in application and system behavior. Typically, in such systems, application workloads, which determine execution and communication latencies, and event and failure occurrences are non-deterministically distributed. For example, US DoD combat systems such as Navy's Aegis [73] and Air Force's AWACS [16] include radar-based tracking subsystems that

associate sensor reports to airborne object tracks. The number of sensor reports generated, and the number of hostile "threats" detected and weapon-launch commands activated per radar sweep significantly change at run-time, as they vary from one sweep to another.

A highly desirable feature of such dynamic real-time systems is their ability to *adapt* to run-time uncertainties. In systems such as [16], when significantly large number of sensor reports arrives, it exceeds the system capacity, causing overloads, resulting in important tracks to go undetected. A desirable behavior during such situations is to adapt by gracefully degrading performance—e.g., process more important reports and discard less important ones. Such adaptive behavior is inconsistent with the hard real-time timeliness optimality.

1.1 TUFs and UA Scheduling

Jensen's time/utility functions [33] (or TUFs) allow the semantics of soft time constraints to be precisely specified. A TUF specifies the utility to the system of completing an application activity as an application- or situation-specific function of when that activity completes. Figure 1.1 shows the conventional deadline (downward step) and several soft time constraints specified using TUFs.



(a) Step (b) Soft-Step (c) Linear (d) Parabolic (e) Multimodal

Figure 1.1: Deadline and Example Soft Timing Constraints Specified Using TUFs

When time constraints are expressed with TUFs, the scheduling optimality criteria are based on maximizing the accrued utility from those activities—e.g., maximizing the sum, or the expected sum, of the activities' attained utilities. Such criteria are called Utility Accrual (UA) criteria, and sequencing (scheduling, dispatching) algorithms that consider UA criteria are called UA sequencing algorithms. In general, other factors may also be included in the optimality criteria, such as resource dependencies and precedence constraints.

UA scheduling inherently facilitates adaptivity and graceful performance degradation during resource-constrained situations. For example, a desirable behavior for many radar-based tracking subsystems such as [16] is to process all tracks when resources are sufficient, and "drop" less important tracks before dropping

more important tracks when resources are insufficient and thereby gracefully degrade performance during overloads.

A UA algorithm that maximizes the sum of activities' attained utilities will seek to meet all activity time constraints (e.g., deadlines) when sufficient processor cycles are available for doing so. Further, when overloads occur, such an algorithm will naturally tend to favor activities that are more important (from whom greater utility can be accrued) than those which are more urgent. (Example such UA algorithms include [44, 17].)

The mapping of utility to application-specific metrics such as track quality, track accuracy, and track importance is intuitive and natural — the work in [16] demonstrates how that can be done.

1.2 Scope and Contributions

This dissertation considers two fundamental problems in UA scheduling:

1. Scheduling activities subject to arbitrarily shaped TUF time constraints and mutual exclusion resource constraints, to maximize system-wide, total attained utility; and

2. Providing stochastic assurances on timeliness behavior, including, stochastically satisfied individual, activity utility lower bounds and system-wide, total utility lower bounds.

1.2.1 Scheduling Arbitrary TUFs Under Resource Dependencies

Scheduling tasks with non step TUFs has been studied in the past, most notably in [44], [15] and [72]. However, to the best of the author's knowledge, Locke's algorithm [44], called LBESA, is the only algorithm that considers arbitrarily shaped TUFs. Another closely related work is that of Strayer's [63], where a framework for scheduling using "importance functions" is presented. An importance function can be arbitrarily shaped and has a similar meaning as that of a TUF.

Besides arbitrarily shaped TUFs, dependencies often arise between tasks due to the exclusive use of shared resources or precedence relationships. Sharing of resources that have mutual exclusion constraints between deadline-constrained tasks has received significant attention in the past. Research on such task models has resulted in a number of protocols including the Priority Inheritance Protocol [57], Priority Ceiling Protocol [57], and the Stack Resource Policy [7].

However, little work has been done for sharing resources (that have mutual exclusion constraints) be-

tween tasks that have TUF time constraints. In [17], Clark considers mutual exclusion resource dependencies, but for tasks with only step TUFs. Further, none of the prior research on step TUFs [36, 48] and non step TUFs [15, 72, 44, 63] consider resource dependencies.

This dissertation combines these two task models. We consider the problem of scheduling tasks subject to arbitrarily shaped TUF time constraints and mutual exclusion resource constraints, to maximize system-wide, total attained utility. This scheduling problem can be shown to be $\mathcal{NP}$-hard. Thus, we present a heuristic algorithm for this problem, called the Generic Benefit Scheduling algorithm (or GBS). [1] GBS computes schedules with a polynomial-time cost of $O(n^3)$, given n tasks in the ready queue.

We study the performance of GBS through simulation and through an implementation on a Portable Operating System Interface (POSIX)-compliant real-time operating system (RTOS). To implement GBS on POSIX systems, which do not support TUFs and UA scheduling, we develop a middleware-level scheduling framework called *meta-scheduler*. Meta-scheduler's middleware-level architecture eliminates the need for modifying the OS kernel for conducting UA scheduling. This facilitates scheduler portability and usage of commercial off-the-shelf (COTS) RTOSes. We are not aware of any other such framework that allows UA scheduling on COTS POSIX RTOSes.

Our experimental results—from simulation studies and from implementation measurements—reveal that GBS performs close to, if not better than existing algorithms for the cases that they apply. Examples of such task models include independent tasks with non step TUFs and tasks with step TUFs and mutual exclusion resource dependencies. Further, our experimental results verify the effectiveness of GBS for its unique model of arbitrary TUFs and resource constraints.

We also analytically establish timeliness and non-timeliness properties of GBS including bounds on activity utilities, mutual exclusion, and freedom from deadlocks.

1.2.2 UA Scheduling with Stochastic Assurance

Most existing UA scheduling algorithms provide assurances on timeliness behavior for some special cases, such as optimal timeliness during under-load situations for step TUFs [44, 17, 74] or assured lower bounds on accrued utilities for deterministic task arrival models [75].

Another form of assurance considered in the past is the notion of competitive ratio. A scheduling algorithm $\mathcal{A}$ has a competitive ratio of φ_A if and only if it can guarantee a cumulative utility $\Gamma_A \geq \varphi_A \Gamma^*$, where Γ^* is the cumulative utility achieved by the optimal clairvoyant scheduler.

[1] GBS is later renamed as Generic Utility Scheduling algorithm (GUS).

In [8], the authors show that when the processor load $\rho > 2$ (see [14] for a definition of processor load), the best competitive ratio an on-line scheduling algorithm can guarantee is $1\Big/(1+\sqrt{k})^2$, where k is the important ratio between the highest and the lowest utility density. Notice that this upper bound on competitive ratio is valid only for tasks with step TUFs, which seriously limits its applicability. Furthermore, the competitive ratio provides a measure of how well an on-line algorithm performs with respect to the optimal clairvoyant scheduler. Though this is an important metric, it is not related to the absolute performance of an algorithm that is measurable by an end-user e.g., the amount of accrued utility.

The problem of timeliness assurance is complicated by the fact that the systems of interest are dynamic and they cannot be described using deterministic task models. For example, task execution times in [16, 46] are highly context dependent, and therefore worst-case execution time (WCET) analysis is infeasible, or is too pessimistic to be useful. Furthermore, task arrivals do not have known minimum inter-arrival times.

The aforementioned non-determinism naturally lends itself to a stochastic task model, where task execution times and inter-arrival times can be stochastically described using random variables. We consider such a stochastic model. Further, we allow tasks to share resources, which are subject to mutual exclusion constraints. Furthermore, we consider application-specified lower bounds on individual task utilities that must be probabilistically satisfied i.e., a tuple $\langle AP_i, AU_i \rangle$ is specified for each task T_i, which implies that task T_i needs to accrue at least AU_i utility with the probability AP_i. AP is called Assurance Probability, and AU is called Assurance Utility. We also consider a lower bound on system-wide, total utility that must be asymptotically satisfied.

For such a model, we consider the problem of scheduling tasks to satisfy the individual and collective utility lower bounds and maximize system-wide, total attained utilities.

Note that satisfying individual task utility lower bounds may contradict with satisfying the system-wide utility lower bound and vice versa. To resolve this contradiction, precedence rules may be specified by the application e.g., satisfying individual utility is more important than satisfying collective utility. Such rules allow tradeoffs between contradicting assurance requirements.

We present algorithmic solutions to the problem of providing performance assurance for TUF-driven, dynamic real-time systems. The algorithmic solutions include algorithms for processor bandwidth allocation and UA scheduling. While bandwidth allocation algorithms allocate processor bandwidth share to activities to probabilistically satisfy individual and collective, utility lower bounds, UA scheduling algorithms schedule activities to maximize accrued utility.

We extend our algorithmic solutions with a class of lock-free and lock-based resource access protocols to satisfy mutual exclusion constraints on shared resources. We prove that satisfying utility lower bounds

with lock-based protocols does not imply doing so with the lock-free scheme, and vice versa. Further, we establish the conditions under which lock-based protocols are "stronger" than the lock-free scheme, and vice versa.

Finally, we present a rule-based framework that allows tradeoff and negotiation between utility lower bound satisfaction and utility maximization, and that between individual and collective utility lower bound satisfaction.

To the best of our knowledge, our algorithm solutions for the UA assurance problem are the first of its kind. We are not aware of any other efforts that address this problem.

1.2.3 Summary of Contributions

Thus, the contributions of the dissertation include:

1. The GBS scheduling algorithm that schedules activities subject to arbitrarily shaped TUF time constraints and mutual exclusion resource constraints, to maximize system-wide, total attained utility;

2. The meta-scheduler scheduling framework [40] that facilitates UA scheduling on COTS POSIX RTOSes; and

3. The class of algorithmic solutions that provide stochastic assurances on timeliness behavior on activities subject to TUF time constraints and mutual exclusion resource constraints, including, stochastically satisfied individual and collective utility lower bounds.

1.3 Organization of Dissertation

The rest of the dissertation is organized as follows. To provide motivation for the TUF/UA model, we summarize two significant applications that were successfully implemented using that model in Chapter 2. Chapter 3 surveys existing scheduling algorithms, focusing on overload scheduling, UA scheduling algorithms, and single machine scheduling theories that are related to UA scheduling. In Chapter 4, we introduce our task and resource models considered by the GBS algorithm. We also describe the scheduling objective of GBS in this chapter.

Chapter 5 present the GBS algorithm. Timeliness and non timeliness properties of the GBS algorithm are established in Chapter 6. We evaluate performance of GBS through simulation in Chapter 7 and through implementation in Chapter 9. We present an overview of the meta-scheduler scheduling framework

in Chapter 8.

Chapter 10 leads the presentation for stochastic assurance problem. Chapters 11, 12 and 13 discuss our solutions for the task-level and system-wide performance assurance problems, respectively.

Finally, this dissertation concludes by describing its contributions and proposed work in Chapter 14.

Chapter 2

Motivating Application Examples

As example real-time systems requiring the expressiveness and adaptability of TUF time constraints, we summarize TUFs of two applications. These include: (1) AWACS (Airborne WArning and Control System) surveillance mode tracker system [16] built by The MITRE Corporation and The Open Group (TOG); and (2) a coastal air defense system [46] built by General Dynamics (GD) and Carnegie Mellon University (CMU). We only summarize some of the application time constraints here; other details can be found in [16, 46], respectively.

2.1 TUFs in AWACS

The AWACS is an airborne radar system with many missions, including air surveillance. Surveillance missions generate aircraft tracks for command and control (C2) and battle management (BM). The surveillance tracker consists of several different activities. Its most demanding computation, called *association*, associates sensor reports to aircraft tracks. The tracker employs two sensors that sweep 180 degrees out of phase with a ten second period. Thus, association has a "critical time" at the period length. If the computation can process a sensor report for a track in under five seconds (half the sweep), that will provide better data for the corresponding report from the out-of-phase sensor. Thus, prior to critical time, utility of association decreases as critical time nears.

After the critical time, the utility of association is zero, because newer sensor data has probably arrived. Thus, if the processing load in one sensor sweep period is so heavy that it cannot be completed, probably the load will be about the same in the next period. So there will not be any resources to also process sensor

data from the previous sweep.

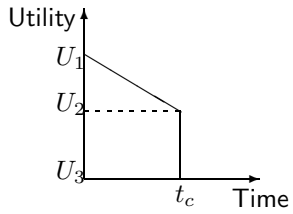
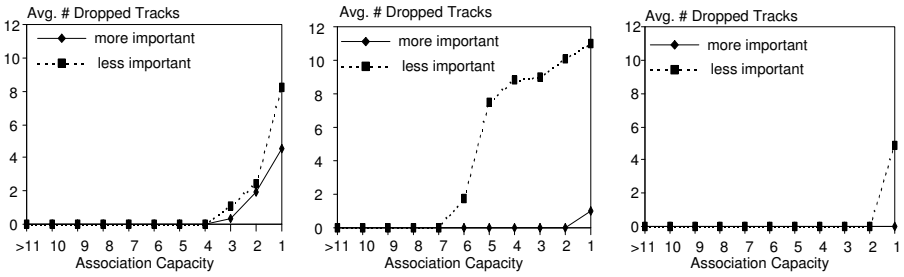


Figure 2.1: AWACS Track Association TUF

This timeliness behavior, which requires the expressiveness and adaptability of soft yet mission-critical time constraints, would be difficult to describe using priorities. An effective solution is to describe it using TUFs.

The described semantics establish association's TUF shape: a critical time t_c at the sweep period; utility that decreases from a value U_1 to a value U_2 until t_c; and an utility value U_3 after t_c. U_1, U_2, and U_3 are determined using Application QoS (AQoS) metrics such as: (1) track quality, which is a measure of the amount of sensor data incorporated in a track record; (2) track accuracy, which is a measure of the uncertainty in the estimate of a track's position and velocity; and (3) track importance, which is measure of track attributes such as its threat. Figure 2.1 shows the association thread's TUF.

The tracker creates threads for each airborne object that it tracks. The threads perform a sequence of activities, including association. The TUFs of all threads have the same basic shape shown in Figure 2.1, but use different values for U_1, U_2, and U_3. The system's UA scheduling algorithm resolves the resource contention among all the association (and other) threads and schedules system resources to maximize the total summed utility.



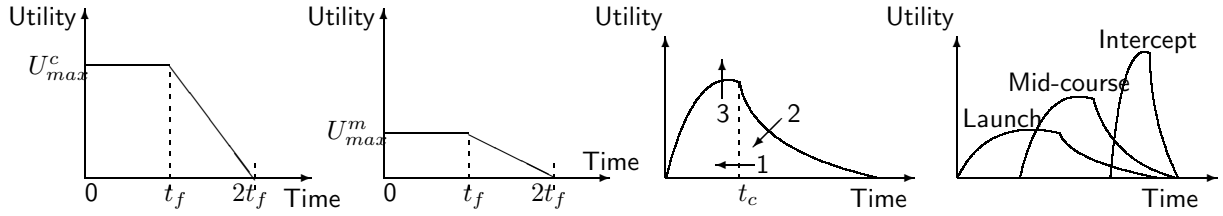
(a) FIFO Priority (b) Fixed Priority (c) UA Algorithm

Figure 2.2: Average Number of Dropped Tracks Under Decreasing Association Capacity

The AWACS surveillance tracker implementation was done using TOG's MK7 operating system [64]. MK7 contains the UA scheduling algorithm described in [44]. To understand how well MK7's UA algorithm is able to schedule system resources in a mission-oriented way, significant performance measurements were made. Different scheduling algorithms, including FIFO and fixed priority, were compared with [44]. Figure 2.2 shows the average number of dropped tracks for the three scheduling policies under decreasing association capacity. The figure illustrates that the UA algorithm minimizes the number of dropped tracks, thereby illustrating the adaptivity of the TUF/UA paradigm.

2.2 TUFs in Coastal Air Defense System

The coastal air defense system defends the coastline from incoming cruise missiles and bombers, using a variety of assets including guided interceptor missiles. Time constraints of three activities in the GD/CMU coastal air defense system, called *plot correlation*, *track maintenance*, and *missile control* are shown in Figures 2.3(a), 2.3(b), and 2.3(c), respectively.



(a) Plot Correlation (b) Track Maintenance (c) Missile Control (d) Missile Ctrl. Shapes

Figure 2.3: TUFs of Three Activities in GD/CMU Coastal Air Defense

AQoS metrics such as track quality and weapon spherical error probable are used to define how each service's timeliness contributes to its utility to the current state of the mission. Note that the TUF for sending guidance updates to interceptor missiles have shapes that evolve during the course of each missile's engagement with its incoming target. This adaptive effect is extremely difficult to achieve with priorities. Performance evaluation of the system proves the effectiveness of TUF/UA. For brevity, here we skip the details of TUFs, application implementation, and adaptive timeliness measurements; these can be found in [46].

Chapter 3

Literature Review

Uniprocessor real-time scheduling algorithms can be broadly classified into two categories: (1) deadline scheduling and (2) overload scheduling. Algorithms in the first category generally seek to satisfy all task deadlines, if possible. Furthermore, all scheduling decisions are solely based on task deadlines. In contrast, algorithms in the second category deal with deadlines as well as non deadline timing constraints such as non rectangular TUFs, wherever proper. For example, if it is impossible to satisfy all task deadlines, the second category algorithms seek to maximize accrued task utilities.

Besides the existing real-time scheduling theories, there is a rich set of single machine scheduling theories that originate from the Operational Research (OR) community. Some of the optimization objectives are directly related to job completion times. Thus, algorithms for these optimization objectives are naturally of interests for UA scheduling.

3.1 Deadline Scheduling

The problem of optimal scheduling for satisfying all task deadlines has been extensively studied in the literature. The Rate Monotonic scheduling algorithm [42] and the Earliest Deadline First (EDF) scheduling algorithm [26] have been established to be optimal for fixed and dynamic priority scheduling, respectively. Furthermore, deadline scheduling has been extended to variants of the hard timeliness optimality criterion of satisfying all deadlines. For instance, in [54], the author presents an overload management scheme that can satisfy deadlines of at least m instances of a periodic task within k consecutive releases, which is called the (m, k) *firm guarantee*.

The aforementioned scheduling algorithms only deal with sharing CPU cycles among real-time tasks that are subject to deadlines. The same or similar algorithms can be used for other shared resources individually, such as semaphores or disk I/O. When real-time tasks need both CPU cycles and other shared resources simultaneously (which is a fairly common scenario), interferences due to resource dependencies must be bounded by using real-time resource access protocols.

In the context of lock-based access, examples of real-time resource access protocols include the Priority Inheritance Protocol [57], the Priority Ceiling Protocol [57], and the Stack Resource Policy [7]. Furthermore, shared resources may be accessed in a lock-free manner, where a task is not blocked upon a failed access. Rather, the task continuously attempts to access the resource. The lock-free access scheme is a viable alternative to the lock-based access scheme, as shown in [6].

3.2 Overload Scheduling

Many existing algorithms for overload scheduling consider rectangular TUFs. In the event of no overload, EDF is optimal in the sense that it can satisfy all deadlines. Thus, these algorithms aim to mimic the behavior of EDF during under-loaded situations as closely as possible. Furthermore, they seek to optimize other performance metrics during overloaded situations, since all deadlines cannot be satisfied during overloads.

One important performance metric that is considered by many algorithms during overload situations is the sum of utility (or "value") that is accrued by all tasks. In [8], the authors show that the upper bound on the competitive factor of any on-line scheduling algorithm is $1/(1+\sqrt{k})^2$, where k is the *importance ratio* of the task set. Note that k is defined as the maximum task value density divided by the minimum task value density of the task set.

The $1/(1+\sqrt{k})^2$ upper bound on the competitive factor is achieved by the D^{over} algorithm presented in [36]. However, the optimal competitive ratio does not imply the best performance for D^{over}. This is because, D^{over} may overly reject tasks to cope with the worst-case task sequences so that it can achieve the highest competitive factor, no matter how strong the "adversary" is. In fact, in Chapter 7, we observe that D^{over} performs the worst among all algorithms for random tasks.

Besides the optimal D^{over} algorithm, heuristic algorithms have also been developed for effective scheduling during overloaded situations. In [44], Locke presents the LBESA algorithm that uses the notion of value density, which is complemented with feasibility tests. Locke's work is extended by several others including [48, 4]. These variant algorithms are also similar to LBESA in that they reject tasks, by ascending order of task value density, or a variant metric of value density, to resolve any overload situation. Performance of

the algorithms presented in [48] is better than LBESA's, but in general, it is very close.

Apart from the rectangular time-utility model with one segment of execution per task, the concept of "imprecise computations" has also been proposed in the literature as an effective technique to handle overloads [43]. The imprecise computation model assumes that the task execution consists of mandatory and optional parts. The scheduling objective is then to satisfy all deadlines of the mandatory parts, with the option of rejecting optional parts. In essence, the imprecise computation model assumes two versions for each task: execution of the mandatory part is necessary to yield a result, but execution of both mandatory and optional parts can produce higher quality results.

As an example, the RED (Robust Earliest Deadline first) algorithm presented in [12] combines many features including graceful performance degradation during overload, deadline tolerance, and resource reclaiming.

Our work fundamentally differs from the RED model in that there is no deadline tolerance specifications. Rather, a task may yield any utility for completing anytime before the deadline time. Furthermore, the RED algorithm does not specify any particular policy to reject tasks when overload occurs. In the case of rejecting tasks in ascending order of task value densities, the RED scheduler behaves like the LBESA scheduler. Thus, we do not directly compare the performance of GBS with the RED algorithm (but indirectly through LBESA).

The work on feedback control theory scheduling further extends the imprecise computation model and assumes the presence of N versions of the same task ($N \geq 2$) [45]. Each version of a task consumes different system resources such as CPU cycles and yields different levels of quality of service (QoS).

Our work fundamentally differs from all the aforementioned algorithms in that we consider arbitrarily-shaped TUFs *and* mutual exclusion resource constraints. All the previously mentioned algorithms, except for the LBESA algorithm, only consider rectangular TUFs. Furthermore, the LBESA algorithm does not consider mutual exclusion constraints.

In [3], the authors present the concept of "timeliness-functions." Unlike a rectangular time-utility function, which drops to a zero utility at the task deadline, a timeliness-function linearly decreases to zero when the laxity of a task reaches zero. In [4], the same authors show that scheduling the task with the highest Dynamic Timeliness-Density (DTD) is more effective than scheduling the highest value density task. The DTD heuristic is echoed in a special case of GBS, where tasks do not share resources. For that scenario, the GBS algorithm selects a task only if the task value density is positive, which essentially requires positive laxity as in DTD.

Non-increasing TUFs have been explored in the context of non-preemptive scheduling of independent

activities. The BPA [72] and CMA [15] algorithms precisely address such a model. We show the comparison of GBS with BPA and CMA in Chapter 7 and Chapter 9. Again, note that GBS allows preemption, arbitrary TUFs (including non-increasing functions), and mutual exclusion resource dependencies.

In [63], Strayer presents a framework for scheduling using "importance functions." An importance function can take arbitrary shapes, and has the same meaning as a time-utility function. Furthermore, Strayer shows that several algorithms such as First-In-First-Out, RMA and EDF can be mapped to the importance-function framework. However, no new scheduling algorithms are presented in [63]. Furthermore, the task model considered in [63] do not consider resource dependencies.

In the context of overload scheduling, few work consider shared resources that have mutual exclusion constraints. The DASA algorithm [17] considers shared resources with mutual exclusion constraints, but only for rectangular TUFs. The GBS algorithm is similar to DASA in the sense that GBS considers the potential utility density—called "potential value density" in DASA—as a metric in constructing the schedule. However, GBS allows arbitrarily shaped TUFs, whereas DASA is restricted to rectangular functions. To the best of our knowledge, DASA and GBS are the only two algorithms that schedule both CPU cycles and other shared resources while allowing timing constraints to be expressed using TUFs.

There are however, a significant number of algorithms that can simultaneously manage multiple shared resources (either multiple units of the same resource or multiple resources). These algorithms usually combine resource reservation and admission control mechanisms to guarantee certain levels of application QoS.

As an example, the Q-RAM model [53] allocates portions of shared resources to real-time tasks, such that the overall system utility is maximized. However, the Q-RAM model assumes that each task can be executed in a number of ways, where different executions require different amount of shared resources, but yield different utilities. Furthermore, task timing constraints in Q-RAM are limited to deadlines. Thus, the Q-RAM task model fundamentally differs from that of GBS.

3.3 Single Machine Scheduling Theories

Single machine scheduling theories arise from the need of allocating scarce resources, mostly machines in shops, to jobs (also called "tasks") over time. There is a rich set of literatures on single machine scheduling theories that consider different job models and different optimization objectives. Many of the optimization objectives are dependent on job completion times and thus have direct counterparts in terms of maximizing the sum of accrued utilities. These objectives and their corresponding TUFs are summarized as follows:

1. Minimizing the total weighted completion times, $1||\sum w_i C_i$, where C_i is the completion time of job

J_i.

Consider that each job J_i has a linearly decreasing TUF, i.e., $U_i(t) = U_i^m - w_i t, \forall t \geq 0$. Thus, the problem of maximizing $\sum_{i=1}^{n} U_i(C_i)$ becomes the $1||\sum w_i C_i$ problem.

This problem has an optimal solution by using the Weighted Shortest Processing Time first (WSPT) rule, i.e., tasks are executed in the decreasing order of w_i/p_i, where p_i is the processing time (execution time) of job J_i [58].

2. Minimizing the (weighted) number of tardy tasks, $1||\sum w_i U_i$, where U_i is one if job J_i is tardy, and is zero otherwise.

   Let each job J_i has a step TUF, i.e., $U_i(t) = w_i, t \leq d_i$ and $U_i(t) = 0, t > d_i$. Then, the objective $\max \sum_{i=1}^{n} U_i(t)$ is equivalent to minimizing the weighted number of tardy jobs.

   This optimization problem is $\mathcal{NP}$-hard. In addition, observe that a special case of is the so-called knap-sack problem if all task deadlines are equal. To solve this problem, one popular heuristic is to use the WSPT rule. Furthermore, a pseudo-polynomial time solution based on dynamic programming is due to Lawler and Moore [37] with a complexity of $O(n \sum_{i=1}^{n} p_i)$. Fully polynomial-time approximation algorithms are also presented in [56] and [19].

3. Minimizing the total (weighted) tardiness, $1||\sum w_i T_i$, where $T_i = \max(C_i - d_i, 0)$ is the tardiness of job J_i.

   This optimization problem corresponds to soft-step TUFs:

$$U_i(t) = \begin{cases} U_i^m \\ U_i^m - w_i (t - d_i) \end{cases} . \tag{3.1}$$

   If all weights are equal, the problem is $\mathcal{NP}$-hard in the ordinary sense. Thus, pseudo-polynomial solution similar to the Lawer and Moore algorithm [37] can apply. However, if the weights are different, the problem is $\mathcal{NP}$-hard in the strong sense. A variety of heuristics have been proposed. Not surprisingly, some heuristics come directly from the WSPT rule or the EDF rule, as they are optimal in some sense. Furthermore, a class of heuristics, called Composite Dispatching Rules, that seek to combine the benefits of several rules have been proposed. An example rule is the Apparent Tardiness Cost (ATC) heuristic [68] that combines WSPT and *Minimal Slack first* (MS) rule.

4. Minimizing the total (weighted) sum of earliness and tardiness, $1||\sum w_i E_i + \sum w_i T_i$, where $E_i = max(d_i - C_i, 0)$ is the earliness of a job.

   Observe that minimizing the weighted sum of earliness implies that completing a job too early may actually accrue less utility than a job's optimal utility. Thus, this optimization objective corresponds

to an increasing TUF. Combining with minimizing the total weighted sum of tardiness, TUF of each job becomes V-shaped:

$$U_i(t) = \begin{cases} U_i^{\min} + w_i t, t \leq d_i \\ U_i^{\max} - w_i (t - d_i), t > d_i \end{cases} . \tag{3.2}$$

This problem is obviously $\mathcal{NP}$-hard, as it subsumes the $1||\sum w_i T_i$ problem. Thus, heuristics are popular ways to solve this problem. However, for some special cases, such as all jobs have the same deadlines, a number of properties of the optimal schedule can be derived and utilized [51].

5. Minimizing the expected sum of weighted completion times.

   This problem is the counterpart problem of $1||w_i Ci$ when job execution times are stochastically specified. Naturally, the WSPT rule is optimal for some scenarios. In [51], the WSPT rule is proven to be optimal in the class of nonpreemptive static list policies as well as in the class of nonpreemptive dynamic list policies. It is also optimal in the class of preemptive dynamic policies when job processing time distributions are Increasing Completion Rates (ICRs).

Besides the above optimization problems in single machine scheduling theories, interested readers are referred to Pinedo's introductory book [51] for a more complete descriptions. Furthermore, in [76], the authors survey the applications of single machine scheduling theories for transmitting time-sensitive web objects. In addition to step and linear functions, the authors also discuss several other functions and the cases of dependencies.

It is worth noting that most of the single machine scheduling theories only consider some special TUFs, such as step, linear, and exponential. Arbitrary TUFs, as considered by the GBS algorithm, clearly lead to $\mathcal{NP}$-hard problems in general. Thus, heuristics have to be employed to solve the optimization problem. Most of the conventional heuristics used for scheduling systems such as Tabu search, local search, and simulated annealing are computationally intensive (see the book by Morton and Pentico [47] for a collection of heuristic algorithms used in scheduling systems). Therefore, they may not be directly used for solving the problem considered by GBS. On the other hand, notice that some approximation algorithms provide bounded performance distance from the optimal algorithms. These approximation algorithms, however, do not provide performance assurance in the form that can be measured by an end-user, e.g., the amount of utility an algorithm can accrue.

Chapter 4

GBS Models and Objectives

This chapter describes the task and resource models, and the optimization objectives of GBS. In describing the models, we outline the scope of the research.

4.1 Task Model

We consider the "thread" abstraction—a single flow of execution—as the basic scheduling entity in the system. In this dissertation, a "thread" is equivalent to a "task" or a "job" in the literature. We denote a thread as T_i.

Note that some activities are not implemented as threads and thus not sequenced as the same way as threads. For example, interrupt service routines are not implemented as threads. Consequently, their timing constraints are usually expressed as upper bound on completion times [34]. The proposed GBS algorithm does not apply to those activities.

A thread can be subject to certain timing constraints. As Jensen points out in [32], a timing constraint usually has a "scope"—a segment of the thread control flow that is associated with a timing constraint. We call such a scope as a "scheduling segment." Following [32], we call a thread a "real-time thread" while it is inside a scheduling segment. Otherwise, it is called a "non real-time thread," because no timing constraint is imposed here.

Similar to Real-Time CORBA 2.0 [50], GBS allows disjointed and nested scheduling segments. Thus, it is possible that a thread executes inside multiple scheduling segments. If that is the case, GBS uses the

timing constraint associated with the inner most scheduling segment for scheduling, because the inner most scheduling segment carries the latest timing constraint. Therefore, for a real-time thread, the scheduler only uses one timing constraint for scheduling purpose at any given time. From the perspective of the scheduler, a scheduling segment corresponds to a real-time thread. Thus, the terms "scheduling segment," "thread," and "task" are used interchangeably in the rest of the dissertation, unless otherwise specified.

The objective of the GBS scheduler is to schedule the real-time threads such that their timeliness is maximized. The non real-time threads are scheduled only if no real-time threads are competing for the CPU time or shared resources. Thus, the real-time threads always have privileges over the non real-time threads. Hence, the activities of non real-time threads can be considered as "background noise," as they will not affect the behavior of the scheduler and the real-time threads.

4.2 Resource Model

To model resources and resource requests, we make the following assumptions:

A.1 Resources are reusable and can be shared, but have mutual exclusion constraints. Thus, only one thread can be using a resource at any given time;

A.2 Only a single instance of a resource is present in the system;

A.3 A resource request (from a thread) can only request a single instance of a resource.

Assumption A.1 applies to both physical and logical resources. Examples of physical resources include disks and network interfaces (for performing disk I/O and network I/O, respectively). In fact, a group of POSIX functions, namely `flockfile(), ftrylockfile()` and `funlockfile()` allows explicit application-level locking for `FILE *` objects [27]. An application thread can use these functions to delineate a sequence of I/O operations that are executed as a unit. Other physical resources include application resources such as a valve or a robot arm in a control system.

Assumption A.1 also applies to logical resources such as critical code sections that are guarded by mutexes or other synchronizers. Note that physical resources can be mapped onto logical resources and thus be examined in a consistent manner.

Assumption A.2 implies that systems with multiple identical resources or multiple instances of the same resource are not allowed. However, we can model such systems by considering each identical instance of a resource as a distinct resource. For example, two physically identical network interface cards (NICs) attached

to the processor are two different resources in our model. Furthermore, Assumption A.2 implies that a thread must explicitly specify which resource it wants to access. Thus, in the aforementioned example, a thread must identify the specific NIC that it wants to access with a resource request. Hence, the thread can only request one specific NIC with each request. With Assumption A.2, deadlocks can be elegantly handled with little effort.

Without loss of generality, we use Assumption A.3 mainly for practical reasons. If multiple resources are needed for a thread to make progress, the thread must acquire all the resources through a set of consecutive resource requests.

4.3 Resource Request and Release Model

During the life time of a thread, it may request one or more shared resources. In general, the requested time intervals of holding resources may be overlapped.

In Figure 4.1 and Figure 4.2, we show an example thread with two scheduling segments. During scheduling segment 1, the thread first requests resource R_2, then requests resource R_1 before it releases resource R_2. Thus, the time intervals during which the thread holds resources R_1 and R_2 are nested. Similarly, during scheduling segment 2, the requested time intervals are overlapped. Note that a pair of `req_CPU()` and `rls_CPU()` functions delimit a scheduling segment, while `req_resource()` and `rls_resource()` functions are used to request and release a resource, respectively. Furthermore, when a thread requests a resource, it is still subject to timing constraints associated with the inner most scheduling segment—timing constraint is the essential characteristic of a scheduling segment.

We assume that a thread can explicitly release resources before the end of its execution. Thus, it is necessary for a thread that is requesting a resource to specify the time to hold the requested resource. We refer to this time as $HoldTime$. The scheduler uses the $HoldTime$ information at run time to make scheduling decisions.

4.4 Abortion of Threads

There are several reasons to abort a thread. First a thread may have to be aborted to resolve a deadlock. Secondly, in case of resource dependencies, the scheduler may decide to abort the current owner thread and grant the resource to the requesting thread. The motivation for doing so is that executing the latter thread (that became eligible to execute with the granting of the resource) may lead to greater total timeliness utility

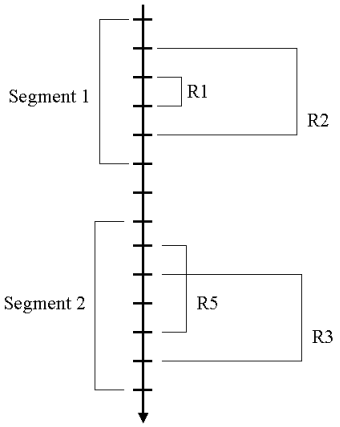


Figure 4.1: A Thread and its Resource Requests

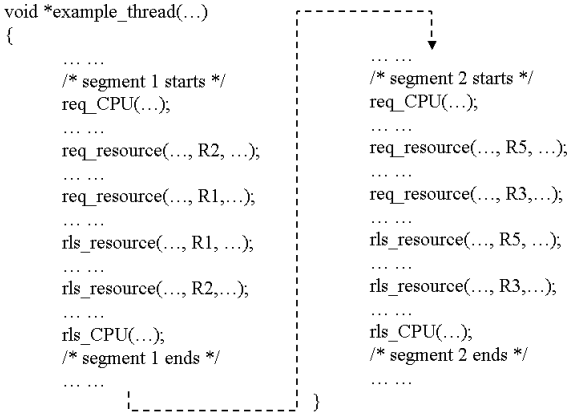


Figure 4.2: Code Skeleton of an Example Thread

than executing the former owner thread, in spite of the overhead associated with doing so.

We say that a thread is executing in $ABORT$ mode if it is being aborted; otherwise, the thread is said to be executing in $NORMAL$ mode. An abortion usually involves necessary cleanup operations, which consumes a certain CPU time. We refer to this time as $AbortTime$.

In general, an $AbortTime$ is always associated with a shared resource. $AbortTime$ is needed because of the physical and logical constraints that may necessitate the state of the affected resource to be changed to a safe and consistent state, when the owner thread of the resource is aborted. Examples of such constraints include resetting a valve controlled by an aborted thread to a safe position, and setting a database record to a value that is consistent with others.

Note that term "shared resource" should be understood in a general sense: any system or application resource that requires non negligible cleanup time (if the owner thread is aborted) should be viewed as a shared resource, even if only one thread uses the resource. For example, consider a thread T that is controlling a valve and assume that T is the only one that will manipulate the valve. In case T is aborted, it needs to move the valve back to a safe position, which consumes certain amount of CPU time. Thus, the valve should be modelled as a shared resource associated with some abortion time.

Furthermore, it is important to note that not all threads can be aborted due to application-specific constraints, such as unacceptable loss of life as a consequence of thread abortion. In the meanwhile, threads may not allow abortion at arbitrary points. We refer to this aspect of a thread as its "abortability."

We assume that the abortability of a thread is application-specific. An application may or may not allow abortion of a thread. For those threads that can be aborted, an application can specify the allowable abortion points. These abortion points could be any time during the execution of the thread, or some well-defined times in the thread control flow.

As an example, we consider the POSIX abortion (or "cancellation" as in POSIX) mechanism [27]. The POSIX specification uses `pthread_cancel()` to force a thread terminate. Once a cancellation is acted on a thread, cancellation handlers (similar to exception handlers) of the target thread should be invoked before the thread terminates. Moreover, POSIX uses `pthread_setcanceltype()` to control the points at which a thread may be asynchronously cancelled. Cancellation type `PTHREAD_CANCEL_ASYNCHRONOUS` allows cancellations be enacted whenever a `pthread_cancel()` is issued to the target thread; cancellation type `PTHREAD_CANCEL_DEFERRED` delays the action of cancellation, if any, until the next cancellation point in the control flow of the target thread. POSIX also denotes a set of functions, such as the standard C function `getc()` as cancellation points. Furthermore, POSIX call `pthread_testcancel()` allows a user to explicitly create cancellation points in the thread code.

If a thread can be asynchronously cancelled (or aborted), execution time of its cleanup handler(s) is measured as $AbortTime$ in our model. In case that abortions can only happen at well-defined cancellation points, $AbortTime$ consists of the execution time of the thread cleanup handler(s) and the execution time from acquiring the resource until the nearest cancellation point. [1] The exception is for the case where the nearest cancellation point happens *after* the resource is released, i.e., after a `rls_resource()` call is made. For this case, $AbortTime$ should be set to ∞, to indicate that the thread cannot be aborted while it is holding the resource. Likewise, the infinite $AbortTime$ can be used for other cases where it simply means that a thread is not abortable, holding a resource or not.

If a thread T_i requests a resource R that is currently held by thread T_j, we say that the thread T_i is dependent on the thread T_j. Thus, T_j is the immediate predecessor of T_i.

4.5 TUFs and A Soft Timeliness Optimality Criterion

We use Jensen's TUFs to specify the timing constraint of a thread. A time-utility function describes a thread's contribution to the system as a function of its completion time. We denote the time-utility function

[1] This time interval only measures the *upper bound* on the time needed to reach the nearest cancellation point. At run-time, a thread may need less time to reach the nearest cancellation point, because the thread may have held the resource for some amount of time.

of a thread T_i as $U_i(.)$. Thus, the completion of a thread T_i (since the release of the thread) at a time t will yield a utility $U_i(t)$.

A TUFs $U_i, i \in [1, n]$ has an initial time I_i and a termination time TM_i. Initial time is the earliest time for which the function is defined and termination time is the latest time for which the function is defined. That is, $U_i(.)$ is defined in the time interval of $[I_i, TM_i]$. Beyond that, $U_i(.)$ is undefined. If the termination time of U_i is reached and the thread has not finished execution (of the scheduling segment), an exception is raised. Usually, the exception causes abortion of the thread. We discuss details of how GBS handles this exception in Chapter 5.

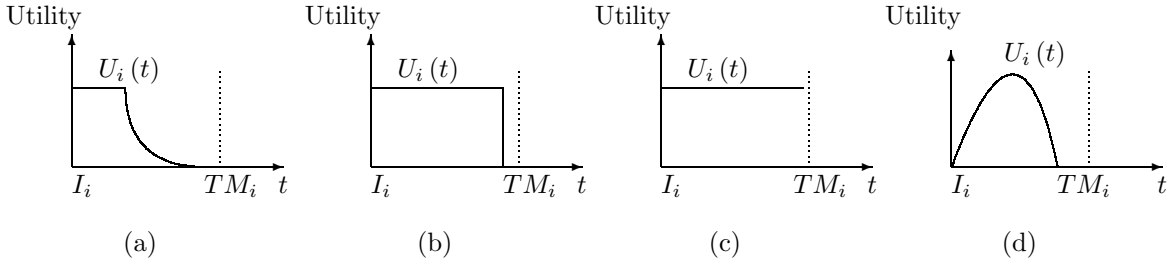


Figure 4.3: Example TUFs

Furthermore, a TUF is allowed to take arbitrary shapes, as shown in Figure 4.3. For $t \in [I_i, TM_i]$, $U_i(t)$ could be positive, zero, or negative. However, U_i does not need to have zero or negative values i.e., it may never touch the time axis. This kind of TUF's implies that completion of an activity can always yield some utility to the system no matter when the activity finishes, which is particularly useful for describing non real-time activities. For example, a constant TUF (see Figure 4.3(c)) can be used for representing a non-time constrained activity: the height of the constant TUF could be used as a way for expressing the activity's relative importance.

Note that our model does not use the "deadline" notation as in hard real-time computing. However, a deadline timing constraint can be specified as a rectangular time-utility function (see Figure 4.3(b)). That is, completing the activity before the deadline accrues some uniform utility and accrues zero utility otherwise.

Given TUFs to describe the timing constraints of dependent threads, we consider the soft timeliness optimality criterion of maximizing the total timeliness utility that is accrued by the completion of all threads i.e., *maximize* $\sum_{i=1}^{n} U_i(f_i)$, where f_i is the finishing time of thread T_i.

This scheduling problem is $\mathcal{NP}$-hard, as it subsumes the problem of: (1) scheduling dependent tasks with rectangular-shaped TUFs; and (2) scheduling independent tasks with non rectangular-shaped, but non-increasing TUFs. Both these scheduling problems have been shown to be $\mathcal{NP}$-hard in [17], and in [15], respectively. Thus, the scheduling problem that we consider is $\mathcal{NP}$-hard. The GBS algorithm presented here

is therefore a heuristic algorithm that seeks to maximize the total accrued utility while respecting all thread dependencies.

Chapter 5

The GBS Algorithm

This chapter presents details of GBS algorithm. Before the GBS scheduling algorithm is invoked upon the arrival of a scheduling event, deadlock needs to be detected and resolved, if any, in a way consistent with the scheduling objective, i.e., maximizing the sum of accrued utilities. Therefore, we discuss GBS deadlock handling mechanism in Section 5.3.

Rationale of the algorithm is discussed in Section 5.1. This discussion is followed by description of the GBS algorithm at high-level in Section 5.4. Sections 5.5 and 5.6 further discuss several key algorithms used in GBS. Recall that each time-utility function has a termination time and an exception may be raised if the termination time is reached. We show how GBS handles this exception in Section 5.7. Finally, we analyzes the complexity of GBS in Section 5.8.

5.1 Algorithm Rationale

The key concept of GBS is the metric called *Potential Utility Density* (or PUD), which was originally developed in [17]. [1] The PUD of a thread simply measures the amount of value (or utility) that can be accrued per unit time by executing the thread and the thread(s) that it depends upon; it essentially measures the "return on investment" for the thread. Furthermore, by considering the dependent threads in computing the PUD, we explicitly account for the dependency relationships among the threads.

Since we cannot predict the future, the scheduling events that may happen later such as new thread

[1] In [17], this metric was called *Potential Value Density* (or PVD).

arrivals, new resource requests, cannot be considered at the time when the scheduler is invoked. Thus, a reasonable heuristic is to use a "greedy" strategy, which means selecting a thread and the threads that it is dependent on (i.e., its predecessors), whose execution will yield the maximum PUD over others.

To deal with an arbitrarily shaped TUF, our philosophy is to regard it as a user-specified "black box" in the following sense: The black box (or the function) simply accepts a thread completion time and returns a numerical utility value. Thus, we ignore the information regarding the specific shape of TUFs in constructing schedules.

Therefore, to compute the PUD of a task T_i at time t, the algorithm considers the expected completion time of T_i (denoted as t_f), and the expected finishing times of T_i's predecessors as well. For each task T_j in T_i's dependency chain that needs to be completed before executing T_i, its expected finishing time is denoted as t_j. PUD of task T_i is calculated as $U_{total}/(t_f - t)$, where the expected utility $U_{total} = U_i(t_f) + \sum_{T_j \in T_i.Dep} U_j(t_j)$.

GBS does not mimic a deadline-based scheduling algorithm such as EDF, unlike many overload scheduling algorithms such as Dependent Activity Scheduling Algorithm (referred to as DASA here) [17], who mimics EDF to reap its optimality during under-loads. This is because, for a task model with arbitrarily shaped TUFs, the deadline of a thread (with an associated TUF) may neither specify its timing urgency nor its relative importance with respect to other threads. Thus, an "optimal" schedule—one that accrues the maximal possible utility—may not be directly related to the thread deadlines. Furthermore, for non-step TUFs, the notion of an under-load situation in terms of timeliness feasibility does not make sense, as threads can yield different timeliness utility depending upon their completion times.

5.2 State Components and Auxiliary Functions

State components are used to facilitate the algorithm description and are described as follows:

1. Resource requests and assignments

Each resource in the system is associated with an integer number, denoted as $ResourceId$. This integer serves as the identifier of the resource and is used by the scheduler and by the application threads. For each resource R, $R.Owner$ denotes the identifier of the thread that is currently holding the resource R. If resource R is not held by any thread (i.e., is free), $R.Owner$ is set to ϕ to indicate this status. We use $Owner(R)$ to denote the task that is currently holding the resource R.

A request for a resource is a triple $ResourceElement = \langle ResourceId, HoldTime, AbortTime \rangle$, where $ResourceId$ refers to the identifier of the requested resource, $HoldTime$ is the time for holding the resource, and

AbortTime is the time for releasing the resource by abortion. The *ResourceElement* triple can also apply to the resource that is currently held by a thread. In that case, *ResourceId* is the identifier of the resource that is being held and *HoldTime* is the *r*emaining holding time for the resource.

Let function $holdTime\,(T, R)$ return the holding time that is desired for a resource R by a thread T. Similarly, function $abortTime\,(T, R)$ returns the time that is needed to release the resource R by aborting the thread T (which is holding R).

2. State components of threads

Recall that abortion of a thread essentially transfers the thread's control flow to its cleanup handlers, if any (see Chapter 4). Thus, a running thread can either follow its normal control flow, or execute its cleanup handlers. We denote this aspect of a thread as its "mode" of execution. The current execution mode of a thread is denoted by $Mode \in \{\texttt{NORMAL}, \texttt{ABORT}\}$. A thread has the `ABORT` mode if it is executing its cleanup handlers; a thread has the `NORMAL` mode otherwise.

ExecTime denotes the *currently remaining* execution time of a thread. Recall that we assume that a thread will release all resources it acquires before it ends. Thus, it follows that for any resource R held by a thread T, $holdTime\,(T, R) \leq T.ExecTime$.

AbortTime denotes the *currently remaining* time to abort a thread. As discussed previously, *AbortTime* is always associated with shared resources. Thus, whenever a thread acquires a shared resource, which is requested as $\langle R, HoldTime, AbortTime\rangle$, the thread's *AbortTime* is increased. Furthermore, we assume that resources are released in the reverse order that they are acquired if the owner thread is aborted. [2]

ReqResource is a *ResourceElement* triple that describes the resource requested by a thread. Note that our resource request model does not allow multiple resources to be requested as part of a single resource request. Thus, for any thread, there is only one *ReqResource* component. A thread not requesting any resource is described as $ReqResource = \langle\phi, \phi, \phi\rangle$. We use the function $reqResource\,(T)$ to denote the identifier of the resource that is currently requested by a thread T.

$HeldResource = \{\langle R_i, HoldTime_i, AbortTime_i\rangle\}$ denotes the set of resources that is currently held by a thread, meaning zero or more resources are held by the thread.

3. The schedule

The output of the scheduling algorithm is an ordered sequence of triples, called a "schedule." A schedule

[2] POSIX specification requires maintaining a stack of cleanup handlers for each thread. These cleanup handlers are pushed into the stack by invoking `pthread_cleanup_push()` and can be popped out by using `pthread_cleanup_pop()`.

consists of zero or more triples of $SchedElement = \langle ThreadId, Mode, Time \rangle$, where $ThreadId$ is the identifier of a thread, $Mode$ is the execution mode of the thread (either `NORMAL` or `ABORT`), and $Time$ is the CPU time allocated to the thread for the current execution.

It is possible that the same thread may appear at several positions within a schedule. This situation may happen because the scheduler may decide to execute a thread just long enough (either in `NORMAL` mode or in `ABORT` mode) so that the thread releases the resource requested by other threads. The remaining portion of that thread may be scheduled to execute later.

5.3 Deadlock Handling

The deadlock handling mechanism is invoked upon a scheduling event and before the GBS algorithm is executed. We consider a deadlock detection and resolution strategy, instead of a deadlock prevention or avoidance strategy. Our rationale for this is that deadlock prevention or avoidance strategies normally pose extra requirements e.g., resources are always requested in ascending order of their identifiers. Furthermore, some resource access protocols make assumptions on the resource requirements. For example, the Priority Ceiling Protocol [57] assumes the highest priority of the threads that will access a resource, called "ceiling" of the resource, is known. Likewise, the Stack Resource Policy [7] assumes "preemptive levels" of threads *a priori*. Such requirements or assumptions, in general, are not practical, due to the dynamic nature of the real-time applications that we are focusing in this paper.

There can be different strategies for deadlock detection and resolution. We present one such mechanism in Algorithm 1, which considers the loss of utility.

For a single-unit resource request model, the presence of a cycle in the resource graph is the necessary *and* sufficient condition for a deadlock. Thus, the complexity of detecting a deadlock can be mitigated by a straightforward cycle-detection algorithm.

The deadlock handling mechanism is therefore invoked by the scheduler whenever a thread requests a resource. Initially, there is no deadlock in the system. By induction, it can be shown that a deadlock can occur if and only if the edge that arises in the resource graph due to the new resource request lies on a cycle. Thus, it is sufficient to check if the new edge produces a cycle in the resource graph.

However, the main difficulty here is to determine a thread to abort such that the loss of utility resulting from the abortion is minimized. Our strategy for this follows: For any thread T_j that lies on a cycle in the resource graph, we compute the utility that the thread can potentially accrue by itself if it were to continue its execution. If the thread T_j were to be aborted, then that amount of utility is lost.

```
1:  input : requesting task T_i; the current time t;
    /* deadlock detection */
2:  Deadlock := false;
3:  T_j := Owner(reqResource(T_i));
4:  while T_j ≠ ∅ do
5:      if T_j.Mode = NORMAL then
6:          T_j.LossPUD := U_j(t + T_j.ExecTime)/T_j.ExecTime;
7:      else
8:          T_j.LossPUD := 0;
9:      if T_j = T_i then
10:         Deadlock := true;
11:         break;
12:     else
13:         T_j := Owner(reqResource(T_j));
    /* deadlock resolution if any */
14: if Deadlock = true then
15:     abort(The minimal LossPUD task T_k in the cycle);
```

Algorithm 1: Deadlock Detection and Resolution in GBS

Thus, the loss of utility per unit time by aborting a thread T_j called $LossPUD$, can be determined as $U_j\left(t + T_j.ExecTime\right)/T_j.ExecTime$. Once the $LossPUD$'s of all threads that lie on the cycle are computed, the algorithm then aborts the thread whose abortion will result in the smallest loss of utility. Note that here $LossPUD$ cannot be calculated by taking into account T_j's predecessors in the graph, since T_j lies on a cycle.

5.4 High Level Description of The GBS Algorithm

The scheduling events of GBS include the arrival of a thread, the completion of a thread, a resource request, a resource release, and the expiration of a time constraint such as the arrival of the termination time of a TUF.

A description of GBS at a high level of abstraction is shown in Algorithm 2. The algorithm accepts an unordered task list and produces a schedule. The format of the GBS schedule differs a simple ordered list of thread (or task) identifiers in the following two ways: (1) any given thread must execute in a certain mode, either `NORMAL` or `ABORT`; and (2) a thread may be split into several segments within the same schedule, where each segment executes for some designated time $Time$.

When the GBS scheduler is invoked at time t_{cur}, it first builds the chain of dependencies for each task (line 5-6). Then, each task's PUD is computed (line 7-8) by considering the task and all tasks in its

dependency chain, called a *PartialSchedule*. Note that $T_i.TotalTime$ and $T_i.TotalUtility$ (line 7) are the total execution time and the utility of T_i's partial schedule, respectively. Finally, the maximum PUD task and its dependencies are added into the current schedule (line 9-12), if they can produce a positive utility.

```
1:  input  : An unordered task list UT;
2:  output: An ordered schedule Sched;
3:  Initialization: t := t_cur, Sched := ∅;
4:  while UT ≠ ∅ do
5:      for ∀T_i ∈ UT do
6:          Build dependency list of T_i: T_i.Dep := buildDep(T_i);
7:          ⟨T_i.PartialSched, T_i.TotalTime, T_i.TotalUtility⟩ := createPartialSched(T_i, T_i.Dep);
8:          T_i.PUD := T_i.TotalUtility / T_i.TotalTime;
9:      Pick the largest PUD task T_k among all tasks left in UT;
10:     if T_k.PUD > 0 then
11:         Sched := Sched · T_k.PartialSched;
12:         UT := delPartialSched(UT, T_k.PartialSched);
13:         t := t + T_k.TotalTime;
14:     else
15:         break;
16: return Sched;
```

Algorithm 2: A High-level Description of the GBS Algorithm

Note that a partial schedule is appended to the existing schedule (Algorithm 2, line 11), instead of being inserted. This is because of the way we compute the PUD for each task, where we assume that the tasks are executed at the current position in the schedule. If the selected partial schedule is inserted into the existing schedule, the previously computed PUDs become void. Furthermore, the algorithm does not consider the deadline order, due to the reasons we discussed in Section 5.1.

Once a partial schedule is appended to the schedule, GBS updates the time t, which is the starting time of the next partial schedule if there exists one. We call this time variable t as *virtual time* in the rest of the paper, because it denotes time in the future. GBS repeats the procedure until either it exhausts the unordered list, or no tasks can produce any positive utility.

We discuss details of creating and deleting partial schedules in Section 5.5. A sub-problem of creating a partial schedule is to determine the execution mode of tasks in the dependency chain, and it is addressed in Section 5.6.

5.5 Manipulating Partial Schedules

A *p*artial schedule is part of the complete schedule, containing a sequence of *SchedElement*'s. We use $T_i.PartialSched$ to denote the partial schedule that is computed for task T_i. $T_i.PartialSched$ consists of task T_i, and all of, or portions of T_i's predecessors. We show how GBS computes the partial schedule for T_i in Algorithm 4.

Before GBS computes task partial schedules, the dependency chain of each task must be determined. This procedure is shown in Algorithm 3. The algorithm simply follows the chain of resource request/ownership. Each task T_j in the dependency list has a successor task that needs a resource that is currently held by T_j. For convenience, the input task T_i is also included in its own dependency list, so that all other tasks in the list have a successor task. The `buildDep` algorithm stops either because a predecessor task does not need any resource, or the requested resource is free.

Note that we use the operator "**.**" to denote an append operation. Thus, the dependency list starts with the farthest predecessor of T_i (which can be retrieved by the function `Head(`$T_i.Dep$`)` and ends with T_i itself.

```
1: input  : task T_i;
2: output: dependency list of T_i: T_i.Dep;
3: Initialization : T_i.Dep := T_i;
4: PrevT := T_i;
5: while reqResource(PrevT) ≠ ∅ ⋀ Owner(reqResource(PrevT)) ≠ ∅ do
6:     T_i.Dep :=Owner(reqResource(PrevT)) .T_i.Dep;
7:     PrevT := Owner(reqResource(PrevT));
```

Algorithm 3: `buildDep()`: Build Dependency List

The `createPartialSched()` algorithm accepts a task T_i, its dependency list $T_i.Dep$, and a virtual time t. The virtual time t is the time to execute the partial schedule to be created. On completion, the `createPartialSched()` algorithm produces a partial schedule for T_i, the total execution time of the partial schedule called $TotalTime$, and the aggregate utility that can be obtained by executing the partial schedule at time t, called $TotalUtility$. The algorithm computes the partial schedule by assuming that tasks in $T_i.Dep$ are executed from the current position (at time t) in the schedule while following the dependencies.

The total execution time of task T_i and its predecessors consists of two parts: (1) the time needed to release the resources that are needed to execute T_i; and (2) the remaining execution time of T_i itself. The order of executing the corresponding tasks or portions of tasks, in their particular modes, together becomes the partial schedule.

Lines 4-14 of Algorithm 4 compute the first part and lines 15-21 compute the second part. Note that, to release a resource R, a task T_j can either execute in `NORMAL` mode for `holdTime(`T_j, R`)` or in `ABORT` mode

for `abortTime`(T_j, R). These two alternatives are accounted for in lines 6-14 of the algorithm by calling the algorithm `determineMode()`.

Since our application model requires explicit release of resources before the end of a thread, it is possible that a task is selected to execute for only a portion of its remaining execution time, after which one or more of the resources that it holds are released. The remaining portion of that task may be scheduled to execute later.

```
1:  input  : task T_i and its dependency list T_i.Dep; t: the time to start executing the partial
             schedule;
2:  output: a partial schedule PartialSched; the total execution time of PartialSched, called
             TotalTime; the total utility accrued by executing PartialSched, called TotalUtility;
3:  Initialization: PartialSched := ∅; TotalTime := 0; TotalUtility := 0;
    /* consider tasks in T_i's dependency chain */
4:  for ∀T_j ∈ T_i.Dep ⋀ T_j ≠ T_i, from head to tail do
5:      R := reqResource(T_j → Next);
6:      Mode :=determineMode(T_j, TotalUtility, TotalTime, t);
7:      if Mode = NORMAL then
8:          PartialSched := PartialSched · ⟨T_j, NORMAL, holdTime(T_j, R)⟩;
9:          TotalTime := TotalTime+ holdTime(T_j, R);
10:         if holdTime(T_j, R) = T_j.ExecTime then
11:             TotalUtility := TotalUtility + U_j (t + TotalTime);
12:     else
13:         PartialSched := PartialSched · ⟨T_j, ABORT, abortTime(T_j, R)⟩;
14:         TotalTime := TotalTime+ abortTime(T_j, R);
    /* consider T_i itself */
15: if T_i.Mode = NORMAL then
16:     PartialSched := PartialSched · ⟨T_i, NORMAL, T_i.ExecTime⟩;
17:     TotalTime := TotalTime + T_i.ExecTime;
18:     TotalUtility := TotalUtility + U_i (t + TotalTime);
19: else
20:     PartialSched := PartialSched · ⟨T_i, ABORT, T_i.AbortTime⟩;
21:     TotalTime := TotalTime + T_i.AbortTime;
22: return ⟨PartialSched, TotalTime, TotalUtility⟩
```

Algorithm 4: The `createPartialSched()` Algorithm

If a task T_j is scheduled to complete it's execution after it releases resources that are needed by its successor, then T_j may accrue some utility. This is accounted for in lines 10-11. Finally, task T_i itself may be executed in either `NORMAL` or `ABORT` mode, which has been determined before the current scheduling event. If T_i is executing in `NORMAL` mode, naturally, it may accrue some positive utility (line 18). Otherwise, no utility can be accrued from the execution of T_i.

If the selected partial schedule contains the remaining portion of a task T, either in `NORMAL` mode or in `ABORT` mode, task T needs to be removed from the unordered task list UT. Consequently, if the selected

partial schedule only contains a portion of task T's remaining part, state components of T need to be updated to reflect this.

The GBS algorithm uses another algorithm called `delPartialSched` to delete a partial schedule from an unordered task list, as shown in Algorithm 5. The `delPartialSched` algorithm examines the partial schedule, from the head to the tail. If a task T has been determined to execute in `NORMAL` mode, then its remaining execution time is updated (line 11). Moreover, T may release one or more resources during the allocated time. Therefore, the $HoldTime$'s of the resources that are currently held by T are also updated (lines 6-10). In the event that T is selected to complete its execution such that it can release the resources, then T is completely removed from RUT (lines 12-13). Furthermore, we consider the *abortion* of a task as the execution of a different piece of code segment for the task. Thus, the same procedure applies to those tasks that have been determined to execute in the `ABORT` mode (lines 15-22).

If a task T is not the tail of a partial schedule, it must release at least one resource. This is because, the only reason for executing the task T, either in `NORMAL` mode or in `ABORT` mode, is to release the requested resource, so that the successor of task T is able to execute. However, task T_i (recall that the partial schedule is due to task T_i)must complete in the partial schedule. Therefore, it is completely removed from RUT (line 23).

5.6 Determining Task Execution Mode

Besides resolving deadlocks, abortions can also be used to improve the aggregate utility. The intuition for doing so is that the time to abort a task may be different from the normal resource hold time, which in turn may affect the timeliness of the tasks that depend upon it. Thus, determining the execution mode of the tasks in a dependency list is necessary to achieve better performance.

Algorithm 6 determines the execution mode of a task T_j in the dependency list of task T_i. In case that task T_j is running in `ABORT` mode, the `determineMode()` algorithm immediately returns `ABORT` mode (lines 3). If a thread cannot be aborted (i.e., $AbortTime$ is infinity), the algorithm immediately returns `NORMAL` mode (line 4).

In other cases, to determine which execution mode is better, the algorithm compares the PUD's of T_i, when T_j is executed in the two modes—`NORMAL` mode (lines 5-6) and `ABORT` mode (lines 7-8). The required time and accrued utility are computed under the two execution modes (called $NormalTime$ and $NormalUtility$ for the first scenario; $AbortTime$ and $AbortUtility$ for the second scenario). The algorithm then follows the dependency chain to examine T_j's successors in $T_i.Dep$ except T_i, assuming that all of them

```
1:  input  : a partial schedule T_i.PartialSched and an unordered task list UT;
2:  output: a reduced task list RUT;
3:  Copy UT into RUT: RUT := UT;
4:  for ∀⟨T, Mode, Time⟩ ∈ PartialSched ⋀ T ≠ T_i from head to tail do
5:      if Mode = NORMAL then
6:          for ∀⟨R, HoldTime, AbortTime⟩ ∈ T.HeldResources do
7:              Update HoldTime: HoldTime := HoldTime − Time;
8:              if HoldTime := 0 then
9:                  T.HeldResource := T.HeldResource − ⟨R, HoldTime, AbortTime⟩;
10:                 R.Owner := ∅;
11:         Update T.ExecTime, T ∈ RUT: T.ExecTime := T.ExecTime − Time;
12:         if T.ExecTime = 0 then
13:             Remove T from RUT : RUT := RUT − T;
14:     else
15:         for ∀⟨R, HoldTime, AbortTime⟩ ∈ T.HeldResources do
16:             Update AbortTime : AbortTime := AbortTime − Time;
17:             if AbortTime = 0 then
18:                 T.HeldResource := T.HeldResource − ⟨R, HoldTime, AbortTime⟩;
19:                 R.Owner := ∅;
20:         Update T.AbortTime, T ∈ RUT: T.AbortTime := T.AbortTime − Time;
21:         if T.AbortTime := 0 then
22:             Remove T from RUT : RUT = RUT − T;
23: Remove T_i from RUT;
24: return RUT;
```

Algorithm 5: Removing a Partial Schedule from a Task List `delPartialSched()`

execute normally (lines 5-20) if they are not currently in `ABORT` mode. This assumption is reasonable even though the execution modes of those tasks have not yet been determined when T_j is examined, because a task's initial mode is `NORMAL`, unless changed by the scheduler.

Finally, the algorithm considers task T_i itself (lines 21-28), whose mode has been determined before the current scheduling event. If T_i is in `NORMAL` mode, it requires $T_i.ExecTime$ to finish the execution of the task. This may or may not produce some positive utility. On the other hand, if T_i is being aborted, it needs $T_i.AbortTime$ to complete the abortion. However, this will not produce any utility.

Once total execution times and total utilities under the two scenarios are computed, the algorithm computes the two different PUD's. If executing T_j in `NORMAL` mode will yield a higher PUD for T_i, then the algorithm decides to execute T_j normally (lines 29-30). Otherwise, T_j is aborted (lines 31-32).

Note that our approach to determine a task execution mode is different from that of the DASA algorithm [17]. DASA seeks to acquire the requested resource as soon as possible. Thus, if the abort time of a task is shorter than its execution time, the task is aborted. Otherwise, the task executes normally. This simple criterion works well for step time/utility functions, because the timeliness of a task will not be negatively affected if the task finishes earlier than its deadline. However, this is not true for arbitrarily shaped time/utility functions. In fact, for non-increasing time/utility functions, the GBS `determineMode()` algorithm can simply return the `NORMAL` mode if a task's abort time is longer than its execution time; otherwise it can return the `ABORT` mode.

5.7 Handling Termination Time Exceptions

Each TUF U_i has a termination time TM_i (see Chapter 4). If the termination time is reached and thread T_i has not finished execution or even not yet started execution, an exception should be raised. Normally, this exception causes abortion of T_i, which implies execution of the thread's cleanup handlers, if any.

In this paper, we assume that an handler for a termination time exception is associated with an application-specific time constraint—i.e., a time/utility function. Thus, the termination handler is scheduled in the same way as other threads. In fact, execution of the exception handler is simply part of the thread itself.

```
1:  input  : task T_j ∈ T_i.Dep, the current accumulative utility of T_i, TotalUtility; the current
             accumulative execution time of T_i, TotalTime, the current virtual time t;
2:  output: the execution mode of T_j;
    /* T_j is currently running in ABORT mode */
3:  if T_j.Mode = ABORT then          return ABORT;
     /* T_j is not abortable */
4:  if T_j.AbortTime = ∞ then         return NORMAL;
     /* scenario I: assuming T_j executes normally */
5:  NormalTime := TotalTime+ holdTime(T_j, reqResource(T_j → Next));
6:  NormalUtility := TotalUtility + U_j(t + NormalTime);
     /* scenario II: assuming T_j aborts execution */
7:  AbortTime := TotalTime+ abortTime(T_j, reqResource(T_j → Next));
8:  AbortUtility := TotalUtility;
9:  NextT := T_j → Next;
10: while NextT ≠ ∅ do
        /* consider tasks in T_i's dependencies */
11:     if NextT.Mode = NORMAL then
12:         NormalTime := NormalTime+ holdTime(NextT, reqResource(NextT → Next));
13:         NormalUtility := NormalUtility + U_NextT(t + NormalTime);
14:         AbortTime := AbortTime+ holdTime(NextT, reqResource(NextT → Next));
15:         AbortUtility := AbortUtility + U_NextT(t + AbortTime);
16:         NextT := NextT → Next;
17:     else
18:         NormalTime := NormalTime+ abortTime(NextT, reqResource(NextT → Next));
19:         AbortTime := AbortTime+ abortTime(NextT, reqResource(NextT → Next));
20:         NextT := NextT → Next;
    /* consider T_i itself */
21: if T_i.Mode = NORMAL then
22:     NormalTime := NormalTime + T_i.ExecTime;
23:     NormalUtility := NormalUtility + U_i(t + NormalTime);
24:     AbortTime := AbortTime + T_i.ExecTime;
25:     AbortUtility := AbortTime + U_i(t + AbortTime);
26: else
27:     NormalTime := NormalTime + T_i.AbortTime;
28:     AbortTime := AbortTime + T_i.AbortTime;
    /* determine the execution mode of T_j */
29: if NormalUtility/NormalTime ≥ AbortUtility/AbortTime then
30:     Mode := NORMAL;
31: else
32:     Mode := ABORT;
33: return Mode;
```

Algorithm 6: `determineMode()`: Determining Task Execution Mode

5.8 Complexity of GBS

To analyze the complexity of the GBS algorithm (Algorithm 2), we consider n tasks and a maximum of r resources in the system. Observe that, in the worst case, each task may hold up to r resources and may be split into $(2r+1)$ partial schedules. Thus, the *while*-loop starting at line 4 in Algorithm 2 may be repeated $O(nr)$ times. Each execution of the loop body examines up to n tasks (or portions of the tasks) that remain in the unordered task list. Clearly, complexity of the *while*-loop body is dominated by the complexity of creating a partial schedule (Algorithm 2, line 7), which in turn is dominated by the cost of determining execution modes of up to n tasks in the dependency chain. Since Algorithm 6 costs $O(n)$ for each task, the worst-case complexity of Algorithm 4 is $n \times O(n) = O(n^2)$. Therefore, the worst-case complexity of the GBS algorithm is $nr \times \left(n \times O\left(n^2\right)\right) = O\left(n^4 r\right)$.

Note that dispatching using the GBS algorithm is sufficient, unlike most other scheduling algorithms. This is because, a new partial schedule is appended by GBS only at the tail of the existing schedule, and cannot affect the partial schedule at the head (of the existing schedule) by any means. Thus, for GBS to work as a *dispatching* algorithm instead of a *scheduling* algorithm, the *while*-loop starting at line 4 in Algorithm 2 can be eliminated. In this case, the complexity of GBS as a dispatching algorithm can be reduced to $O(n^3)$.

Chapter 6

Analytical Results: Timeliness and Non-Timeliness Properties

This chapter presents two classes of properties for GBS. Fist, we consider the timeliness properties of GBS, as a real-time system fundamentally differs a non real-time system in that correctness of a real-time system requires timeliness properties. We then discuss several non timeliness properties of GBS, such as safety.

6.1 Timeliness Feasibility Condition

Timeliness feasibility conditions are conditions under which a real-time system will satisfy its desired timeliness properties [38]. In the context of our task model, the feasibility condition of primary interest is the condition under which the system will accrue a desired lower bound on aggregate utility. Timeliness feasibility conditions lie at the heart of constructing real-time systems with predictable timeliness behavior.

To derive the feasibility condition of GBS, we consider the sporadic task arrival model. That is, each task T_i has a minimal inter-arrival time of δ_i. Note that the periodic task model is a special case of the sporadic task model, where the task inter-arrival times are always the same as its minimal inter-arrival time.

We first derive the upper bound of task response times under GBS (the lower bound is trivia i.e, is always the task execution time). To do that (for task T_i), we consider interferences from all other tasks, which is sufficient but may not be necessary. This upper bound of task response times are then substituted into task TUF's, to compute the bound on accrued utilities. The analysis of GBS feasibility condition is

inspired by the work presented in [39].

To determine the upper bound on the response time of a task T_i under the GBS scheduling algorithm, denoted as $R(T_i)$, we restrict our analysis to independent task sets, because the analysis for dependent task sets under GBS is intractable.

To illustrate the intractability of analyzing response time bounds for dependent tasks, consider a task T_i that holds a resource R. Assume that T_i has reached its termination time and thus is kept in *DeadQ* queue (see Chapter 5). If another task T_j were to arrive and to request the same resource R, GBS will schedule T_i first to release R. Thus, the execution of T_j is interfered by T_i that can arrive arbitrarily long ago. This phenomena implies that a task may suffer unbounded inferences from other tasks, if shared resources are present in the system. This unbounded interference is partly because of the dynamic nature of our task model; tasks can arrive at arbitrary times and can request arbitrary number of resources. Techniques similar to the blocking time analysis in Priority Ceiling Protocol [57] do not apply here, because they require *a prio* known information of all resource requests from all tasks.

For systems without shared resources, a thread that has reached its termination can be safely aborted. Let $A(T_i)$ denote the arrival time of a task T_i and let $TM(T_i)$ denote its termination time. We also denote the length of time interval $[A(T_i), TM(T_i)]$ as $d(T_i)$.

To compute the upper bound on the response time, we consider the strongest adversarial situation, where all other tasks in the task set T are unfortunately scheduled before task T_i. Thus, we need to compute an upper bound on the number of tasks that can interfere with the execution of task T_i, denoted as $r(T_i)$, and their total execution time, denoted as $u(T_i)$.

Observe that a task T_j may interfere with the execution of another task T_i only if T_j arrives during the time interval $[A(T_i) - d(T_j), A(T_i) + d(T_i)]$, which has the length: $(A(T_i) + d(T_i)) - (A(T_i) - d(T_j)) = d(T_i) + d(T_j)$. This is because, if T_j were to arrive before $(A(T_i) - d(T_j))$, then T_i would have either finished its execution or has been aborted by GBS. Thus, T_j will not be able to interfere with the execution of T_i. Similarly, GBS will not execute T_i after $A(T_i) + d(T_i)$. Therefore, the upper bound on $r(T_i)$ can be established as:

$$r(T_i) = \sum_{T_j \in T} \left(\left\lceil \frac{d(T_i) + d(T_j)}{\delta(T_j)} \right\rceil \right). \tag{6.1}$$

Let $C(T_j)$ be the execution time of T_j. It follows that

$$u(T_i) = \sum_{T_j \in T} \left(\left\lceil \frac{d(T_i) + d(T_j)}{\delta(T_j)} \right\rceil \times C(T_j) \right). \tag{6.2}$$

Therefore, the upper bound on the task response time $R(T_i)$ is computed as the sum of the maximum interference time and the execution time of T_i itself, which becomes $R(T_i) = u(T_i) + C(T_i)$.

6.2 Non-Timeliness Properties of GBS

This section presents a class of non-timeliness properties of GBS, including resource safety and task execution mode assurance.

Lemma 6.1. *A task within a partial schedule is either ready to execute, or becomes ready after its predecessor task within the same partial schedule executes. We call a partial schedule "self-contained."*

Proof. This lemma can be shown to be true by examining the `createPartialSched` algorithm (Algorithm 4). Consider a task T_i within a partial schedule PS. If T_i lies at the head of PS, then it must also lie at the head of the dependency chain. Thus, it is ready to execute and the resource that it needs is available.

If T_i does not lie at the head of PS, then let T_j be the immediate predecessor task of T_i in PS. Let R be the resource requested by T_i. If T_j is added into the partial schedule in `NORMAL` mode, then it must execute for `holdTime`(T_j, R) time units (line 8). On the other hand, if T_j is selected to execute in `ABORT` mode, then T_j is aborted after `abortTime`(T_j, R) (line 13) time units, releasing resource R. Thus, resource R must be free when T_i is scheduled to execute. □

Lemma 6.2. *A complete schedule is self-contained if every partial schedule within it is self-contained.*

Proof. We consider a task T within a complete schedule. Since a complete schedule is a concatenation of a set of partial schedules, T must also belong to a partial schedule PS. By Lemma 6.1, this lemma is therefore true. □

Theorem 6.3. *When a task T_i that requests a resource R is selected for execution by GBS, the resource R will be free at the time of execution of T_i.*

Proof. The theorem can be proved using Lemma 6.1 and Lemma 6.2. □

Theorem 6.4. *If a task T_i is executing in `ABORT` mode, GBS will not later schedule T_i to execute in `NORMAL` mode.*

Proof. This theorem is a natural requirement, because our task model assumes that an aborted task cannot execute in `NORMAL` mode in the future, although the abort operation may be split into several stages. The

correctness of this theorem can be seen from lines 3-3 of the `determineMode()` algorithm (Algorithm 6). This mode information is further used in the `createPartialSched()` algorithm. If a task is executing in `ABORT` mode, the algorithm will append the task in `ABORT` mode at the tail of the partial schedule (lines 12-14 and lines 19-21). □

Recall that a non-abortable thread is associated with infinite abortion time. Thus, by similar argument (see line 4-4 of Algorithm 6), we can prove the following theorem.

Theorem 6.5. *If a task T_i is not abortable, GBS will not schedule it to execute in `ABORT` mode.*

Chapter 7

Simulation Results

We conducted simulation studies to evaluate the performance of GBS. Two sets of experiments were performed. The first set of experiments are "static" simulations in the sense that each experiment examines a task ready queue at a particular scheduling event and produces a schedule. That is, these simulation experiments do not involve time variables.[1] The major advantage of conducting such static simulations is that we can compare the schedule produced by GBS with the optimal schedule that is obtained by exhaustive search.

The second set of experiments are "dynamic" simulations, as we randomly generated streams of tasks at run time and measured the aggregate utility produced by GBS. The simulation system is developed using the OMNet++ discrete event simulation toolkit [67].

For performance comparison, we considered the DASA [17], LBESA [44], BPA [72], CMA [15], and D^{over} [36] algorithms. The DASA algorithm is the only algorithm that we are aware of that schedules dependent tasks with rectangular TUFs. LBESA, BPA, and CMA algorithms consider independent tasks with non-rectangular TUFs. However, all these algorithms are heuristic algorithms. On the other hand, the D^{over} algorithm is optimal for independent tasks with rectangular TUFs. D^{over} is optimal in the sense that it has the optimal competitive factor.

[1] A static simulation is also called a Monte Carlo simulation (see pp.403-404 in [28]).

7.1 Static Simulation Results

To reduce the impact of random factors, we considered three probability distributions, including uniform distribution, normal distribution, and exponential distribution. Each experiment input is a randomly generated 9-task set with certain distributions. Furthermore, we compared the performance of the algorithms with the schedule acquired by exhaustive search. Thus, we normalize the accrued utility with that accrued by the exhaustive search. We call such a performance metric as *normalized accrued utility ratio* or "normalized AUR" for short. Furthermore, a single data point was obtained as the average of 500 independent experiments.

Table 7.1: Simulation Parameters

Distribution	ExecTime	Static Deadline	InterArr Time	Dynamic Laxity
uniform	U[0.05, $2C_{avg}$]	U[0.01, $2D_{avg}$]	U[0.01, $2C_{avg}/\rho$]	N[0.25, 0.25]
normal	N[C_{avg}, C_{avg}]	N[D_{avg}, D_{avg}]	N[C_{avg}/ρ, 0.5]	N[0.25, 0.25]
exponential	E[C_{avg}]	E[D_{avg}]	E[C_{avg}/ρ]	N[0.25, 0.25]

Let C_{avg} be the average task execution time and let D_{avg} be the average task relative deadline. Since static simulations evaluate the scheduler output at particular scheduling events, task relative deadline is equivalent to task absolute deadline. Given a 9-task set, the average load ρ can be determined as $\rho = 9C_{Avg}/D_{avg}$. This load calculation is based on the observation that no matter what the deadline order is for the nine tasks, the average total execution time demand of the task set is $9C_{avg}$. Given the average deadline D_{avg}, the load factor for the longest deadline task is $9C_{Avg}/D_{avg}$, which is also the maximum load factors among all others calculated for other shorter deadline tasks.

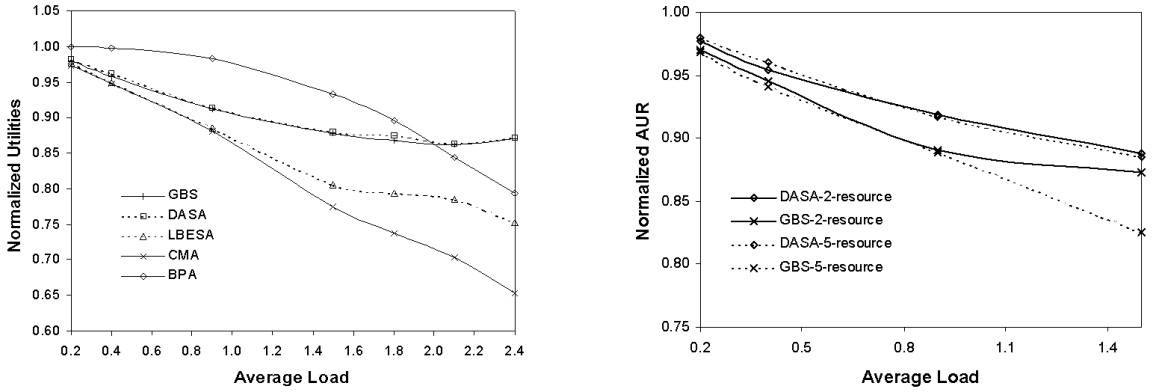
Following [13], the load factor for the entire task set is the maximum load factor among all others. Therefore, one can calculate D_{avg} from ρ and C_{avg}. In all the simulation experiments, we chose C_{avg} as 0.5 sec.

In Table 7.1, we show the simulation parameters. Parameters for the static simulations are shown in columns 2 and 3 of the table. [2]

Our first static simulation experiments involve five scheduling algorithms for independent tasks with rectangular TUFs: DASA, LBESA, GBS, BPA, and CMA. The D^{over} algorithm requires a timer for *Latest*

[2]We use U[a, b] to denote a random variable that is uniformly distributed between a and b. N[a, b] specifies a normal distribution with mean value of a and variance of b. Similarly, E[a] is an exponential distribution with mean value of a.

Start Time so that it can achieve the optimal competitive factor [36]. Thus, it can only be dynamically simulated.



(a) Without Dependencies (b) With Dependencies

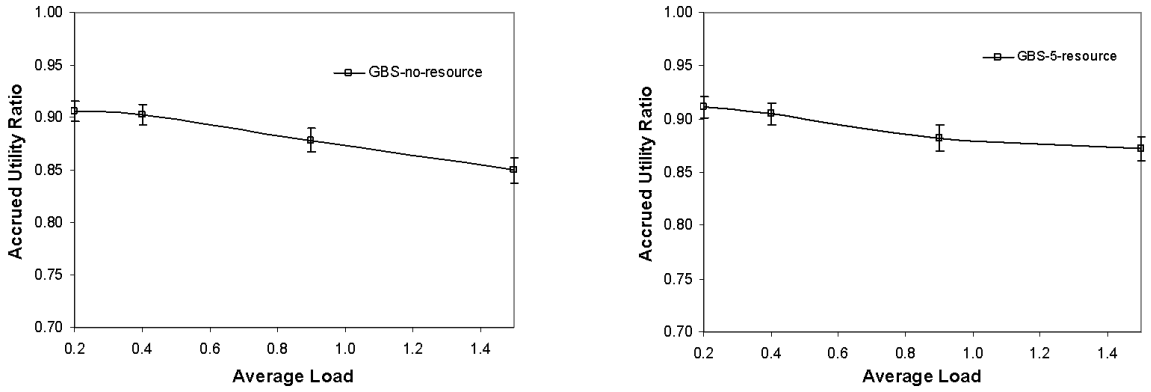
Figure 7.1: Performance of Algorithms Under Rectangular TUFs and Uniform Distributions

In Figure 7.1(a), we show the performance of the algorithms under uniform distributions. As shown in the figure, BPA has the best performance during light and medium load situations. However, during heavily loaded situations, DASA and GBS outperforms BPA. Also, observe that DASA and GBS have very close performance for the entire load range, and DASA occasionally performs slightly better than GBS. On the other hand, CMA and LBESA exhibits the worst performance among all algorithms.

Our experiments with normal and exponential distributions showed consistent results. These are shown in Figures A.1(a) and A.1(b) in Appendix A. Thus, we conclude that GBS, in general, has close performance to DASA for independent task sets with rectangular TUFs.

We also considered dependent tasks that share resources, which can only be scheduled by DASA and GBS. As an example, the performance of DASA and GBS schedulers for dependent tasks is shown in Figure 7.1(b). The experiments were performed for tasks that have rectangular TUFs and uniformly distributed task parameters. Furthermore, resource request parameters such as the number of resources to be requested, resource request times, and resource hold times are all uniformly distributed.

Observe that in Figure 7.1(b), GBS performs worse than DASA for dependent task sets. Furthermore, the performance gap increases when more resources are present in the system. We attribute this worse performance of GBS to the fact that GBS ignores tasks deadlines in making scheduling decisions. Though deadline order may not be appropriate for arbitrarily shaped TUFs (see Chapter 5), it can be beneficial for tasks with rectangular TUFs. On the other hand, DASA only outperforms GBS for no more than 10% in terms of normalized AUR, even if the system is overloaded and has five shared resources.



(a) Without Dependencies (b) With Dependencies

Figure 7.2: Performance of GBS Under Arbitrarily Shaped TUFs and Uniform Distributions

In Figures 7.2(a) and 7.2(b), we show the performance of GBS for tasks with arbitrarily shaped TUFs. We characterize an arbitrarily shaped time-utility function by a polynomial, a deadline d_i, and a maximum utility $maxU_i$. In general, a time-utility function represented as a continuous and derivable curve may be approximated by a polynomial. Without loss of generality, we choose 3^{rd} order polynomials $y_i(t) = a_0 + a_1 \times t + a_2 \times t^2 + a_3 \times t^3$. Furthermore, each coefficient abides the normal distribution with equal mean and variance. These means (or variances) are 9, 11, 7, and 8 for a_0, a_1, a_2, and a_3, respectively. Thus, given $y_i(t)$, d_i, and $maxU_i$, the time-utility function of task T_i is determined as

$$U_i(t) = \begin{cases} 0 & \text{if } t \geq d_i, \\ maxU_i & \text{if } y_i(t) > maxU_i \ , \\ y_i(t) & \text{otherwise.} \end{cases} \tag{7.1}$$

Note that the error bar around each data point in the plots represents the 90% confidence of that data point, since each single data point is an average of 500 independent experimental results. As shown in the two figures, GBS does not suffer abrupt performance degradation with or without shared resources. Furthermore, if the system is not heavily overloaded, GBS can accrue most of the utility (over 80%) that can be accrued by an optimal schedule. Simulation results for normal distribution and exponential distribution are shown in Appendix A.

7.2 Dynamic Simulation Results

Our main motivation in conducting dynamic simulations is to compare the performance of GBS with the D^{over} algorithm, which cannot be investigated in static simulations. Each dynamic simulation experiment generates a stream of 1000 tasks. Given the task average execution time C_{avg} and a load factor ρ, the average task inter arrival time can be calculated as C_{avg}/ρ. This average task inter arrival is applied to different distributions, as shown in column 4 of Table 7.1. Furthermore, the task laxity is modelled as a random variable with normal distribution N[0.25, 0.25] (Table 7.1, column 5). To compare the algorithm performance, we consider independent tasks with rectangular TUFs. Again, the maximum values of the TUFs abide normal distribution N[10, 10].

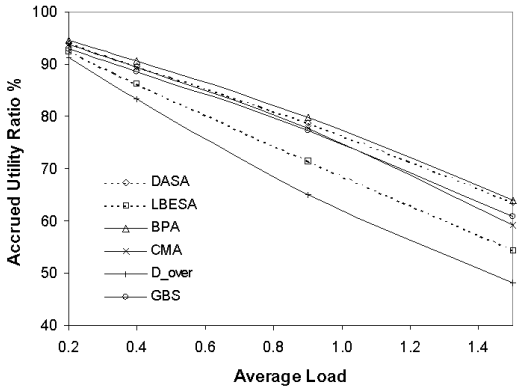


Figure 7.3: Performance of Algorithms Under Rectangular TUFs Without Dependencies (Uniform Distributions)

We show the performance of the six scheduling algorithms in terms of Accrued Utility Ratio (AUR) in Figure 7.3. Note that AUR is defined as the accrued utility divided by the maximal utility that can possibly be accrued from the tasks. As shown in the figure, the D^{over} algorithm performs the worst among all six algorithms. This result is consistent with that presented in Section 8.5 of [14]. The poor performance of D^{over} is because the algorithm rejects more high utility tasks than it should, to guarantee the worst-case performance [14]. Furthermore, the competitive factor only measures the *worst-case* performance of a scheduling algorithm, whereas we measure the *average* performance for large number of tasks here.

Interestingly, BPA, DASA, CMA, and GBS all have close performance. On the other hand, LBESA and D^{over} are among the worst two algorithms. Recall that in static simulation, we show that BPA, DASA, and GBS have superior performance over LBESA and CMA, and LBESA performs better than CMA. We attribute this result to the different operational environment. In static simulation, the input task queue for each experiment consists of nine tasks, whereas the average queue length for all schedulers in dynamic

simulation is less than 3.25.

Chapter 8

The Meta-Scheduler Implementation Platform

Implementation of arbitrary schedulers in research environments has typically involved custom-designed operating systems, or kernel-level modifications to existing commercial-off-the-self (COTS) operating systems, notably Linux [77, 59].

The meta-scheduler is an application-level framework for implementing arbitrary scheduling algorithms on POSIX-compliant operating systems [27], without modifying the operating system. Using a model-checking tool, we have formally specified and verified the behavior of the meta-scheduler, establishing several critical properties including its safety, correctness, mutual exclusion, deadlock-/livelock-freedom in [40].

Our major motivation to consider the meta-scheduler framework for implementing GBS and other scheduling algorithms (as opposed to using an OS kernel environment) is simple: It is significantly easier to implement and debug scheduling algorithms, and instrument algorithm performance in the meta-scheduler framework, as good tools are available that can aid that process. Development, debugging, and instrumentation in kernel environments, on the other hand, are difficult, as few tools are available. Furthermore, we believe that, it is important to demonstrate the effectiveness of GBS "on top of" POSIX-compliant operating systems, as POSIX is a major OS standard that is supported by a number of vendors and enjoys a wide installation base. In the meanwhile, the meta scheduler maintains reasonably low overhead (from 10 usec to a few hundred usec in a common hardware platform) and enjoys small footprint (approximately 10KB binary code).

The meta scheduler framework is first presented in [41], and then in [40]. This chapter provides with

an overview of the meta scheduler framework.

8.1 The Event-Driven Computation Model

Traditionally, a real-time system consists of a set of timer-driven periodic tasks. Additional aperiodic tasks are only allowed to use the spare processor capacity. Hence the schedulability of the periodic task set is preserved. This time-driven model fits well into uniprocessor real-time systems that are simple and low-level, such as direct monitor and control of physical devices. However, in real-time distributed systems that are large and complex in nature, a task execution may span several physical processing nodes and have end-to-end timing constraints. Furthermore, most of the task executions are triggered by external events, such as a data message arrival from the network. Due to the distributed nature of the system, even if the initial data are sampled periodically, the release jitters in the later stages of processing may be too large to apply the periodic task model. Thus, the task interarrival time at a particular processor may not have known lower bound.

For these reasons, an event-driven model is more suitable for those real-time distributed systems that are large and complex. Here, an "event" refers to an occurrence that causes an execution of a task. Examples of a event include the arrival of a data message to be processed by a task, timer expiry, etc.

In our event-driven model, each application program has a *main* thread, which handles external events and may create child threads in response to these triggering events. Normally the *main* thread creates a child thread for every triggering event. For example, the *main* thread may be blocked on a `receive()` function call, waiting for the arrival of new data messages. Once a data message arrives, the *main* thread creates a new child thread to process the data. Our reason for choosing multi-threading as the basic concurrency mechanism is due to its low overhead. However, the meta-scheduler can schedule threads within different application processes in a system-wide manner. Furthermore, since the most significant part of the computation task is executed by the child threads, they become the entities to be scheduled by the meta-scheduler.

8.2 The Approach

The scheduling problem is, in its basic nature, a synchronization problem. All child threads need to synchronize their executions with the meta-scheduler, such that at any time there is only one child thread from an application process is executing. Furthermore, that child thread must be the one selected by the scheduling algorithm. Since a POSIX-compliant OS always schedules threads based on their priorities, it is possible to

synchronize threads executions through priority operations.

Thus, the key idea of our meta-scheduler architecture is dynamic priority mapping and adjustment according to the user-specified scheduling algorithm. The child threads from all application processes are provided with two distinct priorities e.g., P_1 and P_2 where P_2 is higher than P_1. The meta-scheduler dynamically adjusts the priorities of all application child threads in response to a scheduling event. Moreover, the thread selected by the scheduling algorithm is assigned the higher priority P_2, while all other application child threads are assigned the lower priority P_1. Therefore, once the scheduling decision is made, the underlying priority-driven POSIX scheduler will execute the selected child thread and preempt all other child threads if necessary. The priority adjustment can be implemented by using the POSIX system call `pthread_setschedparam()`.

Another possible scheduling decision is to abort a child thread. For example, a child thread may be aborted due to its timing failure. We view abortion as a separate execution path leading to the thread termination. The abortion operation can be implemented through POSIX `pthread_cancel()`.

8.3 The Meta-Scheduler Architecture

Inspired by the General Scheduling Automaton Framework (GSAF) proposed in [17], we developed a similar architecture that consists of one meta-scheduler process, and zero or more application processes. Each application process may contain one or more threads, as discussed previously. The application threads interact with the meta-scheduler through a set of *scheduling primitives*.

8.3.1 Architecture and Code Skeleton

Besides the *main* thread, each application process also contains another thread, called *sched_stub*, to facilitate the dynamic priority adjustment on the application child threads. This is because some implementations of the POSIX specification may not allow one thread to adjust the priority of another thread residing in a different process. The *sched_stub* thread is thus responsible for accepting the meta-scheduler commands and dynamically adjusting the priorities of the threads. The *main* thread of an application process creates the *sched_stub* thread by calling `init_sched()`. Figure 8.1 shows the code skeleton of an example application *main* thread. After the *sched_stub* is created, the *main* thread enters an infinite loop, waiting for triggering events such as new data messages. Once triggered, the *main* thread creates a new child thread to handle the triggering event.

```
main()
{
  ... ...
  /* initialize the sched_stub thread */
  init_sched();

  while(1)
  {
     /* blocked here */
     receive(....);
     ... ...
     pthread_create(..,foo,..);
  }
  ... ...
}
```

Figure 8.1: The *main* Thread Skeleton

```
void *foo(void *arg)
{
   /* initialization code */
   ... ...

   /* blocked until this thread is selected */
   request_cpu(...);

   /* process the data */
   ... ...

  /* notify the meta-scheduler it is done */
  release_cpu(...);
}
```

Figure 8.2: The *child* Thread Skeleton

Figure 8.2 shows the skeleton of an example child thread. At the beginning of its execution, the child thread issues the `request_cpu()` primitive that transfers the control flow to the underlying meta-scheduler. The `request_cpu()` primitive may be blocked indefinitely until the calling thread is selected to execute by the meta-scheduler. Similarly, at the end of an application thread execution, it issues the `release_cpu()` primitive, notifying the meta-scheduler that the currently running thread has finished execution.

8.3.2 Priority Assignment Policy

As discussed in Section 8.2, the application child threads use two priorities, P_2 and P_1. To allow a new thread to be created at arbitrary time, it is necessary to assign a higher priority to the *main* threads than those of the child threads. We denote the priority of the *main* thread as P_3. Similarly, the *sched_stub* thread should also have higher priority than P_2. However, it is desired that the executions of the *main* and the *sched_stub* threads are "atomic," if they are not interrupted by higher priority threads. That is, the *main* thread will not execute until the currently running *sched_stub* thread finishes, and vice versa. Note that since the body of both the *main* thread and the *sched_stub* thread are infinite loops, an execution of these threads should be understood as one execution of the loop body. This "atomic" property can be achieved by assigning the same priority (P_3) to both threads *and* by using the FIFO scheduling policy (SCHED_FIFO in POSIX).

Furthermore, a newly created child thread simply inherits the priority of its creating *main* thread (P_3), such that it can preempt any other child threads in the system, and successfully issues the `req_cpu()` primitive. Finally, the meta-scheduler has the highest priority over all other threads. This highest priority is denoted as P_4. Thus, the meta-scheduler architecture only uses four distinct priorities — this requirement is satisfied by all nowadays POSIX-compliant OS's. An interesting aspect of the priority assignment is that the

OS may support more priorities than needed by the meta-scheduler architecture. Furthermore, other threads in the system may have already been assigned higher priority than P_4. Thus, the activities of those threads imposes interferences on the real-time application threads. This is acceptable as we are not targeting the hard real-time applications with timing constraint approaching the hardware limitations. In the meanwhile, selecting four high priorities will help to reduce the interferences.

Chapter 9

Implementation Results

In this chapter, we discuss an actual implementation of GBS and measurements that were obtained from the implementation. We implemented GBS and all other scheduling algorithms (used in the simulation studies) in the meta-scheduler framework.

Our experimental uses an implementation of the meta-scheduler on the POSIX-compliant, QNX Neutrino 6.2.1 real-time operating system. In our experiments, the meta-scheduler and all the scheduling algorithms are implemented as a user process.[1] The experimental hardware platform is a Toshiba Satellite 1805-S254 laptop that has common processing power for nowadays desktop computers. We summarize the specifications of the experimental hardware platform in Table 9.1.

Table 9.1: Hardware Specifications of The Experimental Platform

Processor family	Intel Pentium III
Processor speed	1GHz
L2 Cache size	256KB
SDRAM size	256MB
Motherboard/Data bus speed	100MHz

[1]Since the QNX Neutrino OS is a micro kernel architecture, there is essentially no difference between user processes and system processes.

9.1 Experimental Setup

We used synthetic workload to evaluate the performance of the different scheduling algorithms. During each experiment, 100 tasks are generated with randomly distributed parameters such as task execution times and deadlines. In addition, each experiment was repeated three times with independent random seeds, and the average performance was taken as the final result. We now discuss the details of the task parameters.

1. Worst-Case Execution Times, Inter-arrival Times, Laxities, and Deadlines

In our experiments, the worst-case execution times (WCETs) of tasks [2] are exponentially distributed with a mean of 500 msec. To be practical, we artificially set 5 sec as the upper bound on the task execution times. Otherwise, a single experiment can generate tasks with execution times that are in the order of hours. Given a workload ρ, we calculate the mean inter-arrival time as the mean execution time divided by ρ. In addition, the laxity of a task is uniformly distributed between 50 msec and 1 sec. The task relative deadline is the sum of the task execution time and laxity.

2. TUFs

We use TUFs with six different shapes in our experimental study. These are shown in Figure 9.1.

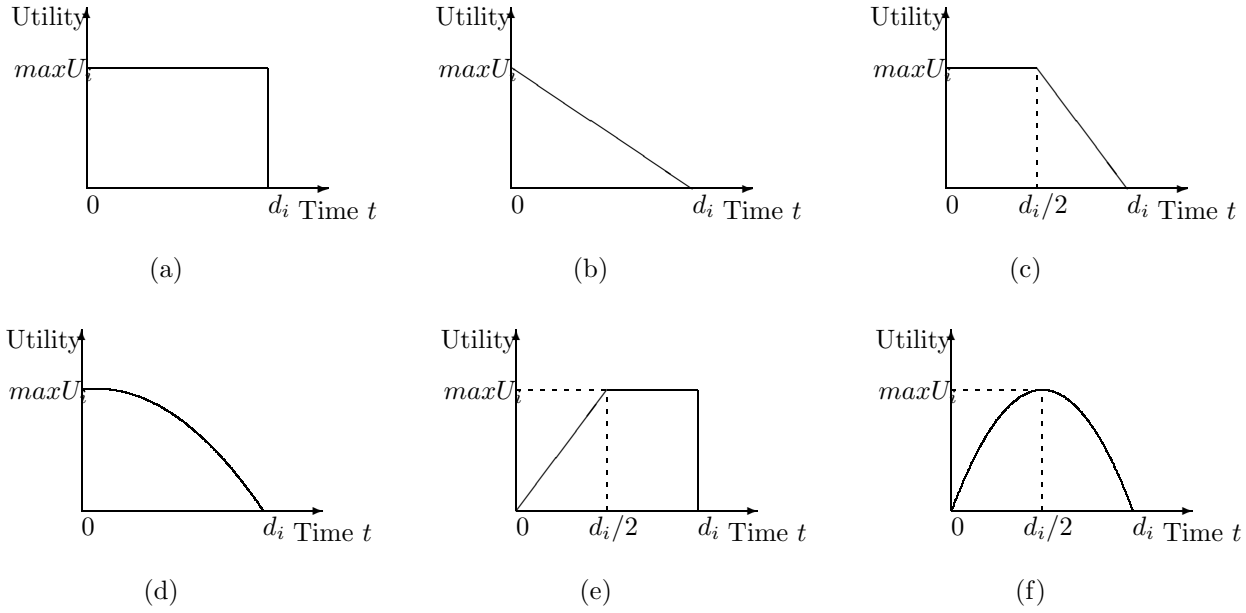


Figure 9.1: TUFs Considered in Experimental Study

Note that the time-utility function of a task depends not only on its shape (i.e., one of the six shapes), but

[2] WCETs are equal to exact execution times in our experiments.

also on task parameters such as relative deadline and maximal utility. Furthermore, observe that TUFs I to IV are non-increasing functions, while V and VI are not. In our experiments, the maximal utility of tasks are uniformly distributed between 10 and 500.

3. Resource Requests

We use uniform distributions to define resource request parameters in our experimental study. The number of resources that is requested by a task is uniformly distributed between 0 and the maximal number of resources in the system. The resource hold times and abort times are also uniformly distributed. Without loss of generality, we always generate abort times that are less than or equal to the corresponding hold times. Furthermore, the requested resource hold time segments are arbitrary in the experiments — they could be isolated, overlapped, or nested.

9.2 Performance Measurements

We implemented five scheduling algorithms including DASA, LBESA, BPA, D^{over}, and GBS as part of our experimental study. The CMA algorithm that we had considered in the simulation studies was not implemented because it requires significant amount of memory space and CPU time for median number of ready tasks, and therefore is impractical.

Furthermore, we considered all six TUFs shown in Figure 9.1. We measured the performance of the algorithms in terms of cumulative deadline-satisfaction ratio (DSR) and accrued utility ratio (AUR).

Table 9.2: Experiment Grouping

Experiments	DASA	LBESA	BPA	D^{over}	GBS
R-N	Y	Y	Y	Y	Y
R-D	Y				Y
NINC-N		Y	Y		Y
NINC-D					Y
ARB-N		Y			Y
ARB-D					Y

Observe that all algorithms with the exception of GBS, have certain restrictions. For example, DASA can deal with dependency relationships but only for rectangular TUFs. Therefore, we partitioned the experiments into six groups according to different shapes of TUFs and the presence of dependency relationships.

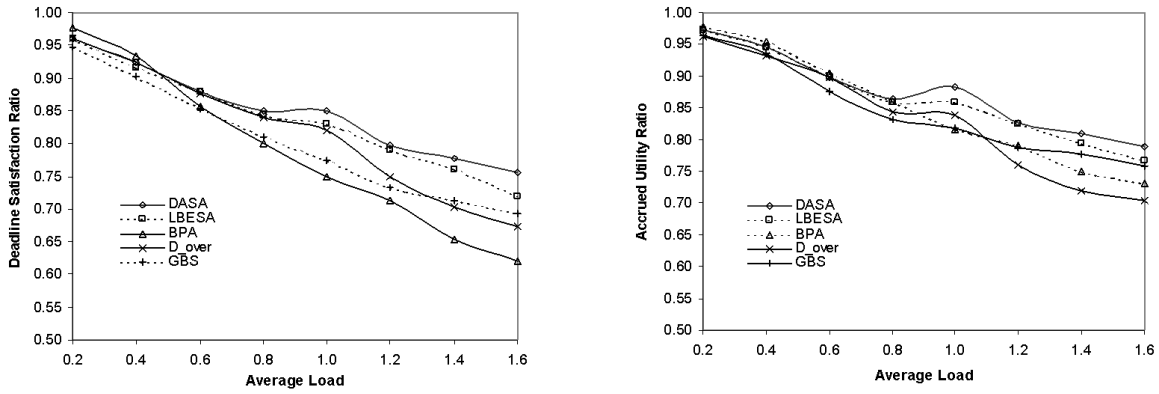
In Table 9.2, we show this experiment grouping, where 'Y' means that the experiment applies for the

corresponding algorithm.

We now discuss the results for each experiment group in the subsections that follow.

1. Rectangular TUF's Without Dependencies (R-N)

Experiments in this class is interesting because all five algorithms can deal with the task model. The results are shown in Figures 9.2(a) and 9.2(b).



(a) Deadline Satisfaction Ratio (b) Accrued Utility Ratio

Figure 9.2: Performance of Algorithms Under Rectangular TUFs and No Dependencies

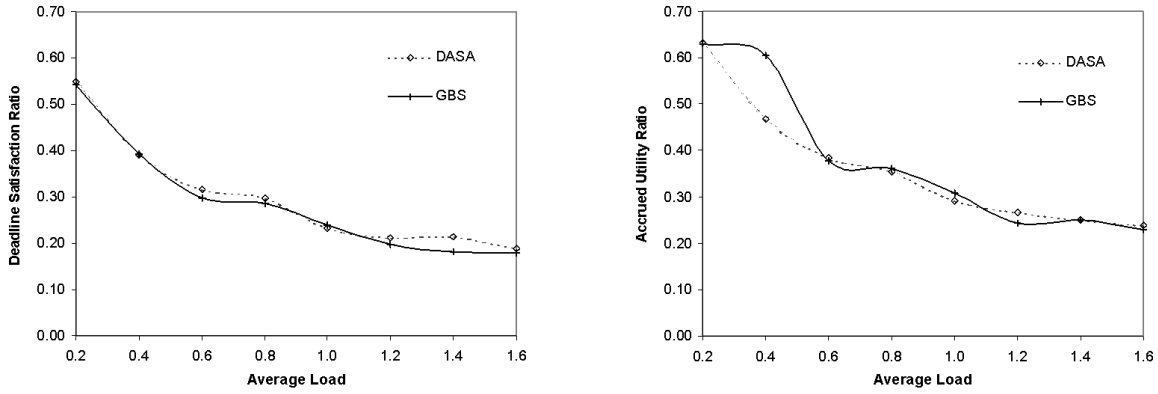
From the figures, we observe that the performance of the five algorithms do not significantly differ for light load and medium load conditions (workload is less than 0.8). However, DASA and LBESA show superior performance for heavy and overloaded situations. Furthermore, we observe that GBS performs worse than DASA and LBESA, but better than BPA and D^{over}.

It also worth to observe that D^{over} does not perform the best in our experiments, although it is an optimal algorithm. This experimental result is consistent with our simulation results and has been explained previously.

2. Rectangular TUF's With Dependencies (R-D)

This task model applies only to DASA and GBS. Our experiments in this class used three shared resources as an example scenario. The results of the experiments are shown in Figures 9.3(a) and 9.3(b).

We would expect DASA to outperform GBS for this class of experiments, because GBS neither conducts a feasibility test, nor considers the deadline order for scheduling the feasible task subset. The figures, however, show that GBS actually performs better than DASA during light workload situations. Performance of the two schedulers are very close during overloaded situations.



(a) Deadline Satisfaction Ratio (b) Accrued Utility Ratio

Figure 9.3: Performance of DASA and GBS Under Rectangular TUFs and Dependencies

We conjecture that the better performance of GBS is because of its lower CPU time cost in making scheduling decisions. Another aspect of interest is the significant performance degradation of the algorithms compared with no shared resource scenarios. In our simulation studies (Chapter 7), we observed that a small number of shared resources (such as three resources) generally do not cause severe performance degradation, which is in contrast to the experimental results from our implementation. This inconsistency suggests that the schedulers may incur significant overhead for dealing with dependency relationships. In Section 9.3, we provide further evidence for our conjecture by measuring scheduling overheads.

3. Non-increasing TUF's Without Dependencies (NINC-N)

This task model is applicable to LBESA, BPA and GBS schedulers. We show the results of the experiments in Figures 9.4(a) and 9.4(b).

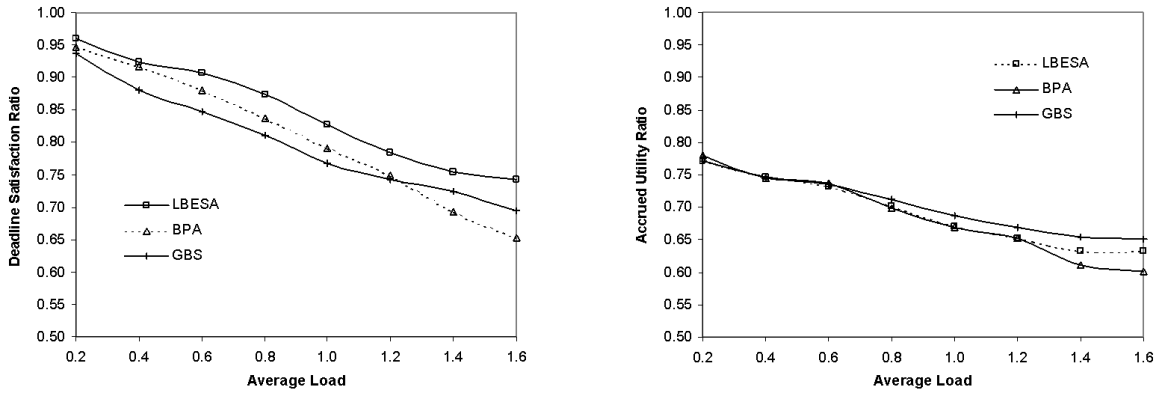
From the figures, we observe that LBESA performs the best in terms of DSR, but GBS performs slightly better than others in terms of AUR. Overall, the three algorithms have close performance for non-increasing TUFs. This is because all of them use heuristics similar to the "value density" that was first developed in LBESA.

4. Arbitrarily Shaped TUF's Without Dependencies (ARB-N)

This task model is applicable for LBESA and GBS. The results are shown in Figures 9.5(a) and 9.5(b).

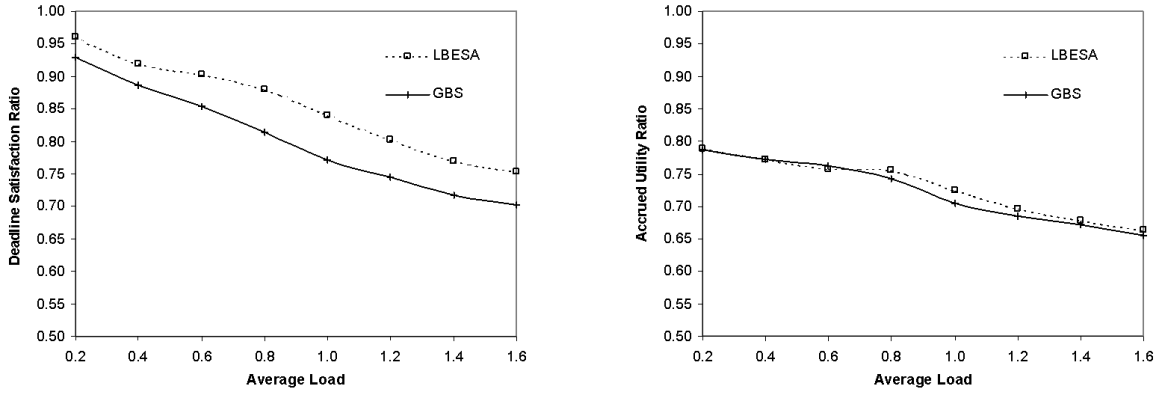
From Figure 9.5(b), we observe that LBESA and GBS exhibit close performance in terms of AUR (similar to the results for the class of non-increasing TUFs without dependencies).

On the other hand, from Figure 9.5(a), we observe that LBESA satisfies more task deadlines than GBS does. This is not surprising, because LBESA seeks to avoid overload situations by testing for task feasibility and



(a) Deadline Satisfaction Ratio (b) Accrued Utility Ratio

Figure 9.4: Performance of LBESA, BPA, and GBS Under Non-Increasing TUFs and No Dependencies



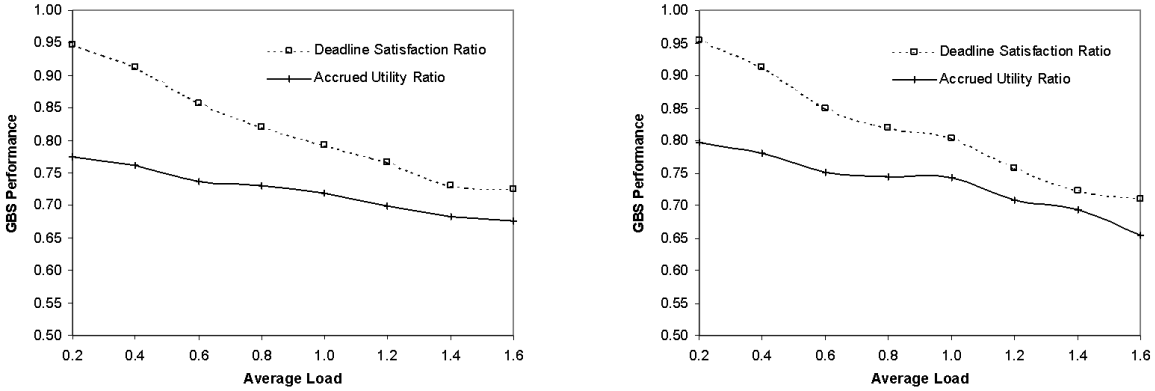
(a) Deadline Satisfaction Ratio (b) Accrued Utility Ratio

Figure 9.5: Performance of LBESA and GBS Under Arbitrary TUFs and No Dependencies

rejecting infeasible tasks. However, as discussed in Chapter 5, deadline order or deadline-based feasibility test is not appropriate for non-rectangular TUFs.

5. Non-increasing and Arbitrarily Shaped TUF's With Dependencies (NINC-D and ARB-D)

This task model considers dependencies for non-rectangular TUFs, which can only be handled by GBS. We show the experimental results in Figures 9.6(a) and 9.6(b).



(a) Under Non-Increasing TUFs

(b) Under Arbitrary TUFs

Figure 9.6: Performance of GBS With Dependencies

Although the performance of the algorithm cannot be interpreted from the figures in an absolute sense, it indicates graceful performance degradation during increasing workload situations.

Furthermore, observe that the DSR is much higher than the AUR. This is due to the way we calculated the AUR, which is defined as the accrued utility divided by the maximal possible utility. In the case of non-rectangular TUFs, a task may not be able to accrue its maximal possible utility ($maxU_i$ in the figures), even if there is no interference.

9.3 Overhead of Scheduling Algorithms

Our objective in measuring the scheduling overhead is twofold: (1) determine whether the overhead of GBS is reasonable for general-purpose processor systems; and (2) determine the reason for the inconsistency between the simulation performance and the implementation performance of GBS.

We show the average scheduling overhead of all five scheduling algorithms in Figure 9.7. In Table 9.3, we show the 90% confidence intervals of the scheduling overhead.

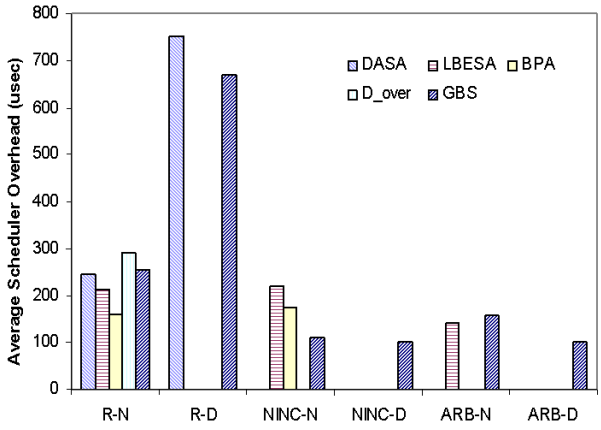


Figure 9.7: Average Scheduling Overhead

Table 9.3: 90% Confidence Intervals of Scheduler Overhead in Microseconds

Exp.	DASA	LBESA	BPA	D^{over}	GBS
R-N	246.98 ± 14.83	214.72 ± 53.62	159.25 ± 9.58	289.51 ± 81.69	254.49 ± 44.54
R-D	753.38 ± 240.83	n/a	n/a	n/a	668.72 ± 354.64
NINC-N	n/a	218.81 ± 15.13	175.55 ± 10.89	n/a	110.55 ± 4.78
NINC-D	n/a	n/a	n/a	n/a	100.76 ± 6.01
ARB-N	n/a	141.12 ± 9.66	n/a	n/a	156.04 ± 7.79
ARB-D	n/a	n/a	n/a	n/a	101.09 ± 5.70

As can be seen from the table, the scheduling overhead may vary in a large range. For instance, the overhead of the GBS scheduler is as low as 101.09 usec for the ARB-D class of experiments, but as high as 668.72 usec for the R-D class of experiments. Furthermore, observe that the GBS overhead is roughly 100 usec less than that of the DASA scheduler for the R-D class of experiments. This less overhead of the GBS scheduler justifies our conjecture that the reason why GBS performs better than DASA for the R-D class of experiments is due to its less overhead.

Chapter 10

Overview of Performance Assurance

10.1 Problem Statement

We consider the problem of providing performance assurance to the end-user and maximizing total accrued utility. Examples of performance assurance include bounds on accrued utilities for individual tasks and bound on system-wide accrued utility. More specifically, we consider two types of assurance requirements:

- Probabilistic bounds on individual tasks' accrued utility; and
- Bound on system wide Accrued Utility Ratio.

A probabilistic bound on a task T_i's accrued utility is specified as a tuple $\langle AU_i, AP_i \rangle$. This specification indicates that task T_i is expected to accrue the utility of AU_i with the probability of AP_i. Note that this specification only expresses the end-user's minimal requirement. A stronger performance assurance, such as higher accrued utility or higher assurance probability is desired.

The bound on system accrued utility is specified as a lower bound on system wide Accrued Utility Ratio (AUR). AUR measures the amount of utilities that can be accrued with respect to the maximal possible utility accrual.

We discusses several important issues involved in performance assurance. The solutions are at the abstract level. Detailed algorithms for solving a specific assurance problem are presented in the chapters that follow.

10.2 Design Tradeoff

Recall that an end-user is allowed to specify several assurance requirements. Given n tasks, there are up to n task-level assurance requirements. In addition, the end-user can specify a lower bound on system wide AUR. Therefore, a system can be subject to $n + 1$ assurance requirements. Ideally, all requirements are satisfied. However, in case that some of them cannot be satisfied, a negotiation and tradeoff mechanism shall be available. This mechanism must be user-oriented and allow user-specified tradeoff.

We assume the existence of a set of user-specified *rules*. These rules specify the order of assurance requirements that are considered and satisfied. Particularly, they indicate that in case not all requirements can be satisfied, which requirement(s) need to be sacrificed. The rules can be static and predefined, or situation specific.

		Task requirements		
		None	Partial	All
System requirement	Yes	Rule 1	Rule 2	N/A
	No	Rule 3	Rule 4	Rule 5

Figure 10.1: Rules of Tradeoff

Figure 10.1 shows the matrix of rules. In an ideal case, both system requirement and task requirements can be satisfied—no need for any rule. However, under other cases, rules are needed to resolve conflicting requirements. For example, Rule 1 is used when system assurance requirement can be satisfied while none of the task assurance requirements are satisfied. If Rule 1 prefers system requirement, resource allocation for satisfying system requirement is finalized. Otherwise, resources will be allocated in a manner to satisfy task requirements as much as possible.

Note that the "Partial" column in Figure 10.1 means that some, but not all of the task requirements are satisfied. This result implies an order of allocating resources to individual tasks. Assume that the order is T_i followed by T_{i+1}. Then, the first $k < n$ tasks' assurance requirements are satisfied under the "Partial" result.

Independent of the tradeoff rules, there are three basic assurance problems:

1. Satisfying task requirements without considering system requirement;
2. Satisfying system requirement without considering task requirements; and
3. Satisfying both task requirements and system requirement.

We present our solutions to the first basic problem in Chapters 11 and 12. Solutions to the second and third basic problems are discussed in Chapter 13.

10.3 Handling Unexpected Tasks

Achieving assurance for a task T_i requires properties of the task is known at design time. This assumption is justified for many applications. However, for some highly dynamic systems, "unexpected" tasks may be created and demand performance assurance. These tasks are "unexpected" in the sense that they are not known or even do not exist at design time.

For example, consider a satellite that has a software problem while it is flying in deep space. After some research, the ground support scientists decide to write a piece of software to correct the problem. Thus, they need to upload the new code into the satellite computer, which means a set of "unexpected" tasks are initiated and executed in the satellite computer system.

Unexpected tasks may have two consequences. First, since the existing performance assurance is based on the assumption of no such unexpected tasks, it may be violated. Second, performance assurance for the unexpected tasks may not be possible, if the system does not have spare processing power, and degrading the existing performance assurance is not allowed.

One way to handle unexpected tasks is to ignore the assurance requirement of the unexpected task and does not allow it to interfere existing performance assurance. This approach is similar to how a hard real-time system handles overloads. The second approach is so called "best-effort" manner. Since assumptions of the system are violated when unexpected tasks arrive, an algorithm should seek to maximize some user-specified performance metrics, such as the sum of accrued utilities. We believe how unexpected tasks are handled are highly application- and scenario-dependent. Therefore, we leave details open.

Chapter 11

Achieving Task-Level Probabilistic Assurance

The problem solved in this chapter can be informally stated as follows. Suppose an end-user is allowed to specify a tuple $\langle AU_i, AP_i \rangle$ for each task T_i in the system. The tuple indicates that each task T_i needs to accrue at least the utility of AU_i with the probability of AP_i. In the meanwhile, the end-user also desires to maximize the system wide accrued utility. Then, the central problem is how to satisfy the per task assurance requirement as well as maximizing system wide utility.

In Section 11.1, we first present the problem in details. An approach of using processor bandwidth allocation and proportional share is then discussed in Section 11.2. Section 11.3 shows a sufficient feasibility condition for computing the required processor bandwidth of a task. Solutions to the feasibility condition are explored in Section 11.4. Finally, we present a utility accrual, job scheduling algorithm that seeks to maximize the accrued utility.

11.1 Models and Objectives

11.1.1 Recurring Task Models

The problem considered in this chapter concerns with the probability of satisfying task assurance requirements. A probability, in general, is a measure of frequency of occurrences [60]. In that context, a task that

is released only once has inherent difficulties of obtaining probabilistic assurance. For example, consider a one-shot task with an unknown execution time and an unknown arrival time. In the extreme case, the processor is *always* idle whenever the task arrives. Also suppose the task of interest has a deadline. Then, calculating the probability of meeting the task's deadline is meaningless unless it may arrive again in some future times. Otherwise, for the single occurrence of task arrival, the task's deadline can either be satisfied or not. Probability of satisfying the task's deadline is thus meaningless.

Thus, for achieving task-level probabilistic assurance, a recurring task arrival model is assumed. We elaborate the task arrival model in Section 11.1.2. Throughout the discussion of task assurance, we assume that a task T_i may be released repeatedly. Each instance of task T_i is called a *job* of task T_i, denoted as $J_{i,j}$, $j \geq 1$. In case that we are only interested in generic jobs without task-specific, the first subscript may be dropped.

Observe that this recurring task model is in contrast to the model assumed in the GBS work. The GBS scheduling algorithm examines the task ready queue at every scheduling instant. Scheduling decisions are made solely based on information of the existing tasks and no information of future task arrivals are assumed. This is because the GBS algorithm considers a highly dynamic system where arrival patterns of tasks are impossible to derive, or are unknown. Therefore, the GBS algorithm falls into the class of "best-effort" scheduling algorithm in the sense that it seeks to accrue as much utility as possible. However, no performance assurance similar to that discussed in this chapter is possible.

Considering recurring task models has been an important aspect that differs real-time scheduling from machine scheduling [51]. In the domain of machine scheduling theories, majority of the algorithms assume that "jobs" (scheduling entities) are ready at the event of scheduling, or will be ready in some known, future time instants. Scheduling decisions, therefore, are based on the properties of tasks and optimization objectives, e.g., minimizing maximal lateness. Furthermore, because the time scale in canonical machine scheduling theories are usually in the magnitude of days or even longer, there seems no compelling reason for considering recurring job arrivals.

On the other hand, recurring task models, particularly periodic task models have been the dominant models in real-time scheduling theories. The periodic task model naturally arises from device-level, monitor and control real-time tasks which periodically poll devices, or send control commands to devices. With its regularity, the periodic task model enjoys solid analytic results, such as necessary and sufficient feasibility conditions under a variety of scenarios. From the perspective of how task arrivals are seperated, the periodic task model is one extreme point of the spectrum of recurring task models. The aperiodic task model where no information of task inter-arrival time is assumed lies on another end of the spectrum. In the middle of the spectrum, there exist models which are neither completely regular, nor completely irregular. The spectrum

of regularity with example models is depicted in Figure 11.1.

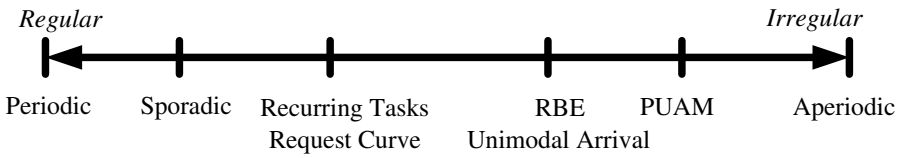


Figure 11.1: The Spectrum of Regularity

The sporadic model assumes a minimal separation between two consecutive arrivals. The "recurring tasks" model [9] generalizing the sporadic model by introducing the structure of a task, which is represented as a Directed Acyclic Graph (DAG). Note that the "recurring tasks" in Figure 11.1 refers to the specific model in [9]. The "request curve" abstraction [65], borrowed from the networking research community, uses an envelope curve to bound the processor demand during any time interval. More formally, the request curve α_r of a request function $R(t)$ satisfies $R(t) - R(s) \leq \alpha_r(t-s), \forall s \leq t$, where $R(t)$ is the processor demand of a task up to time t. In [65], the authors propose a polynomial time algorithm to derive the request curve for a task represented by a DAG similar to that in [9]. Therefore, we group the "request curve" abstraction with the "recurring task" model.

The Rate-Based Execution (RBE) model [30] is inspired by task arrival patterns in real-time distributed systems, such as a video-conferencing system. In such a real-time distributed system, packets and corresponding processing tasks may not have minimal inter-arrival times due to network communication jitters. In fact, tasks can arrive simultaneously, provided that the application only requires x events be processed every y time units. The unimoal arrival model [25] associates a tuple $\langle a, W \rangle$ with a task, meaning the maximal number of task arrivals during any sliding time window of W is a. Again, a tasks may arrive simultaneously. Finally, our Probabilistic Unimodal Arrival Model (PUAM) model provides high degree of flexibility by allowing the probability of some number of arrivals be specified. The PUAM model is presented in Section 11.1.2.

11.1.2 The Probabilistic Unimodal Arrival Model

Observe that unimodal arrival model only specifies the worst case scenario, i.e., the upper bound on the number of arrivals during an interval of length w. Therefore, it is intrinsically suitable for hard real-time analysis, where feasibility is analyzed in a deterministic sense by assuming the worst case scenarios. This deterministic feasibility analysis in in contrast to our goal: a probabilistic feasibility analysis framework for

utility accrual. Therefore, we augment the unimodal arrival model with the probabilities of the number of arrivals in a time interval of w. We call this augmented task arrival model Probabilistic Unimodal Arrival Model (PUAM).

A PUAM specification is a tuple of $\langle p(k), W\rangle, \forall k \geq 0$, where $p(k)$ is the probability of k arrivals during any time interval W. Note that we have $\sum_{k=0}^{\infty} p(k) = 1$.

We present some examples of the PUAM specifications as follows. By doing so, we provide intuitive and concrete instances of the PUAM model. Furthermore, we show that the standard unimodal arrival model, periodic arrival model, and sporadic arrival model are special cases of the PUAM model.

Binomial PUAM: $p(k) = \begin{pmatrix} n \\ k \end{pmatrix} \theta^k (1-\theta)^{n-k}$, $k = 0, ..., n$ and $p(k) = 0, k > n$. This Binomial PUAM specification is useful for modeling the number of occurrences during n Bernoulli trials. For example, consider a task that is responsible for monitoring n communication channels. Each channel has a probability of θ to generate a triggering message during any time window of W. Thus, the number of task arrivals during a window of W can be modeled using a Binomial PUAM specification.

Poisson PUAM: $p(k) = e^{-\lambda}\lambda^k \Big/ k!$, $k \geq 0$. The Poisson distribution is another widely used discrete distribution, particularly for queueing systems. For example, the number of requests received by a server, e.g., a web server thread, during a time window is usually modeled as a random variable following a Poisson distribution.

Worst Case PUAM: $p(k) > 0, k = 0, ..., m$ and $p(k) = 0, k > m$. This worst case PUAM specification implies that there are *at most* m arrivals during any time interval of W. Furthermore, the probability for a particular number of arrival $p(k), k \leq m$ is unknown. It can be seen that a Worst Case PUAM is equivalent to the standard unimodal arrival model.

Periodic PUAM: $p(1) = 1$ and $p(k) = 0, \forall k \neq 1$. This specification indicates that there is *always* one task arrival during a time window of W, i.e., a periodic task arrival model.

Sporadic PUAM: $p(0) = \lambda, p(1) = 1 - \lambda$ and $p(k) = 0, \forall k > 1$. This specification is a special case of the Worst Case PUAM with $m = 1$. It means that for every time interval of W, there is *at most* one task arrival. It is also possible that no task arrives during a particular time window. Therefore, any two consecutive arrivals are separated by at least W.

Remarks: Notice that majority of the existing task arrival models assume a minimal separation between two consecutive arrivals. This is done to bound processor demand. The RBE model is one of the exceptions, for which the processor demand is bounded by properly postponing some task deadlines. However, in our case, processor demand only needs to be *probabilistically* bounded. This need motivates our design of the

PUAM model.

11.1.3 Task Execution Time and Time Constraints

The execution time of a task T_i is modeled as a non-negative random variable, denoted as c_i. The execution time of a job $J_{i,j}$, represented as $c_{i,j}$, is an instance of the random variable c_i.

One example of non-negative random distributions is the gamma distribution. The gamma distribution rises naturally in modeling the waiting time between Poisson distributed events. A gamma distribution $\Gamma(\alpha, \theta)$ has two free parameters, namely α and θ. The standard exponential distribution is a special case of a gamma distribution with $\alpha = 1$ and $\theta = 1/\lambda$, where $1/\lambda$ is the mean of the exponential distribution. Figure 11.2 shows probability density functions of two example gamma distributions.

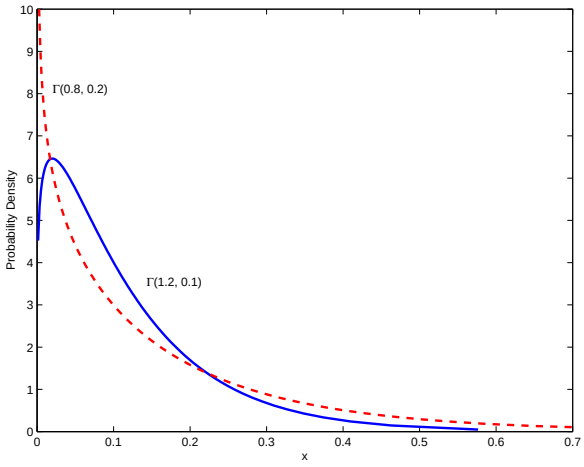


Figure 11.2: Probability Density Functions of Gamma Distributions

Notice that a normal distribution is usually not a good model for task execution time, because of its probability of generating negative values. This negative-probability can be negligible if the normal distribution is close to some positive mean value.

We focus on unimodal TUFs, as they encompass the majority of time constraints in many applications. The TUF for task T_i is denoted as $U_i(t)$. For each task, a user also specifies the assurance requirement as a tuple $\langle AU_i, AP_i \rangle$. AU_i is the desired utility that T_i needs to accrue and AP_i is the probability that T_i accrues at lease the utility of AU_i.

One difficulty of scheduling tasks with unimodal TUFs lies on the fact that a TUF could be increasing. This increasing TUF semantics is in contrast with many existing time constraints, e.g., the earlier the better.

To deal with increasing TUFs or increasing segment of a unimodal TUF, we propose the "early execution" technique. The idea is to postpone the completion of a task with an increasing TUF until its optimal time, if possible. In that sense, the proposed scheduling algorithm may allow CPU idleness even if task ready is queue is not empty, i.e., it is a non work-conserving scheduling algorithm.

If a TUF of task T_i, i.e., $U_i(t)$ contains an increasing segment we can transform it into a non-increasing TUF $U_i'(t)$ by the following procedure.

1. Find the latest time t_{opt} such that $\forall t > t_{opt}$, $U_i(t) < U_i(t_{opt})$;
2. $$U_i'(t) = \begin{cases} U_i(t_{opt}) & \forall t \leq t_{opt} \\ U_i(t) & otherwise \end{cases}, \tag{11.1}$$

Let $U_i'(t)$ be the transformed TUF for task T_i. In the case of a non-increasing TUF, $U_i'(t)$ is simply the same as $U_i(t)$. Once we complete the TUF transformation, all transformed TUFs, all $U_i'(t)$ should now be non-increasing functions.

11.1.4 Objective

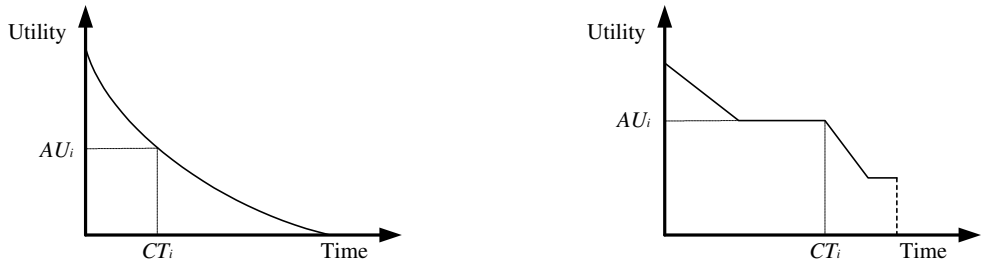
Given a set of tasks $\mathcal{T} = \{T_i | i = 1, ..., n\}$ and their attributes, including arrival patterns, execution time profiles, TUFs, and assurance requirements, our objective is two-fold:

1. Satisfy all tasks' assurance requirements, if possible; and
2. Maximize the accrued utility from all tasks.

The first requirement is the constraint that our algorithm aims to satisfy. The second requirement is merely an optimization objective. Notice that the first requirement shares conceptual similarities with the canonical deadline scheduling problems, where the objective is to schedule a set of tasks to satisfy their deadlines, if possible. In fact, we later show that the problem of satisfying task-level assurance requirements can be solved by probabilistic feasibility analysis and resource allocation techniques in Section 11.2. However, the second requirement distinguishes our assurance problem from the problem of simply meeting task deadlines—scheduling and resource allocations must be performed in a manner of maximizing accrued utility.

11.2 Approach

The problem of satisfying $\langle AU_i, AP_i \rangle$ is a binary optimization problem in the sense that the constraint can either be satisfied or not. Recall that a deadline constraint is also a binary optimization objective. Naturally, there is a relationship between these two problems. For non-increasing TUFs, satisfying a designated AU_i requires that the task's sojourn time is upper bounded by a "critical time" (CT_i). Given a desired assurance utility AU_i, CT_i satisfies that $\forall t_1 \leq CT_i, u_i(t_1) \geq AU_i$ and $\forall t_2 > CT_i, u_i(t_2) < AU_i$. In other words, the critical time of a task is the latest time instant after which the accrued utility is below AU_i. Figure 11.3 shows two examples TUFs and their critical times. Once the requirement of accruing AU_i is converted to bounding task sojourn time by CT_i, a deadline-based probabilistic feasibility analysis can be conducted.



(a) A decreasing TUF (b) A non-increasing TUF

Figure 11.3: Example TUFs and their critical times

The basic approach we take to tackle this assurance problem is to allocate a certain portion of processor time to each task. Our rationale is that without dedicated resource for each task *to some extent*, performance assurance is not possible.

In our context, resource allocation does not necessarily imply resource reservations as in hard real-time systems [52]. In hard real-time systems, the goal is to reserve a portion of system resources to a task so that the task never misses deadlines. Probabilistic deadline satisfaction is not an objective in general. Another major difference is that a resource reservation technique such as [52] is not aware of time utilities of tasks. It also worth to note that the canonical deadline-driven scheduling of periodic tasks [42] guarantees a portion of processor time for each periodic task. Given a periodic task with period D and worst case execution time C, in a feasible system, the EDF scheduling policy essentially allocates C units of processor time to the task every D time units. The other extreme case is the server approaches for handling aperiodic tasks (See Chapters 5 and 6 in [14]). The server approaches cannot accommodate aperiodic tasks subject to time

constraints—the objectives of the algorithms are to satisfy deadlines of periodic tasks and to minimize the *average* response time of aperiodic tasks. Therefore, *all* aperiodic tasks compete for the processor time left by periodic tasks.

In this work, we use *processor bandwidth* as the first class abstraction for resource allocation. The *processor bandwidth* is defined as follows:

Definition 11.1. If a task is actively competing for processor time in a time interval $[t, t + L]$ and has a processor bandwidth ρ, then it receives at least ρL processor time in that time interval $\forall t, L > 0$.

Since the granularity of assurance requirement considered in this chapter is on a per task basis, processor bandwidth is allocated to each task. This strategy can be easily extended to deal with assurance requirements for task groups and subsystems.

Once tasks are allocated processor bandwidth, jobs compete for the processor bandwidth allocated to their parent task. With known, probabilistically bounded processor demand of a task, i.e., assuming the PUAM model and random execution times, it is possible to calculate the probability of satisfying the task's critical time. If the calculated probability is lower than AP_i, a higher processor bandwidth is desired.

Notice that competition among jobs of the same task is resolved by a job scheduler. In that sense, our approach shares similarities with *hierarchical scheduling* (See [55] for example). In practice, similar approach has been used in scheduling user-level threads by many thread packages, such as the FSU PThreads library [49] and the GNU Portable Threads [18].

There are three problems yet to be answered:

1. How to realize and enforce the bandwidth allocated to a task?
2. How much bandwidth is needed for a given task to achieve its assurance requirement?
3. How are competition among jobs of the same task resolved?

We present our solutions to these problems in the sections that follow.

11.2.1 Realizations of Bandwidth Allocation

The definition of processor bandwidth naturally lends it to a class of techniques called *proportional share* (PS) [62, 24, 69]. In a proportional share system, each client (an entity competing for shared resources, e.g., a process) has a weight w_i. If a client i is active in time interval $[t_1, t_2]$, then it receives a service time of

$$s_i(t_1, t_2) = w_i \int_{t_1}^{t_2} \frac{1}{\sum_{j \in \mathcal{A}(t)} w_j} dt. \tag{11.2}$$

$\mathcal{A}(t)$ is the set of active clients at time t. If each task T_i is assigned a weight of $w_i = \rho_i$, then the processor bandwidth of T_i is in deed at least ρ_i, if $\sum_i \rho_i \leq 1$.

Equation 11.2 represents the service time for a client in an ideal "fluid" model, where the time quantum can be infinitely small. However, an infinitely small quantum is impractical. Therefore, one major performance metric of proportional share algorithms is so-called *lag*. For a scheduling entity i, its lag at time t, denoted as $lag_i(t)$ is defined as the difference between the service time it should receive in an ideal fluid model and the service it actually receives. Let t_0 be the time when a scheduling entity becomes active for competing shared resources. And let $s_i(t_0, t)$ be the service time it actually receives until time t, $S_i(t_0, t)$ be the service time it should receive in an ideal fluid model. Then, $lag_i(t)$ is

$$lag_i(t) = S_i(t_0, t) - s_i(t_0, t). \tag{11.3}$$

The earliest-eligible-virtual-deadline-first (EEVDF) algorithm [62] is proven to be an optimal proportional share algorithm in the sense that it has the minimal, optimal lag. Given a quantum size Q, the lag of a client at any time in an EEVDF-scheduled system is bounded by Q

$$-Q < lag(t) < Q. \tag{11.4}$$

Other proportional share algorithms [70, 71, 24] cannot achieve this optimal lag. Therefore, we focus our discussion on the EEVDF algorithm hereafter.

Besides proportional share, resource reservation is another widely used paradigm. In [55], the authors classify resource reservations into four types:

- basic, hard: RESBH x, y
- basic, soft: RESBS x, y
- continuous, hard: RESCH x, y
- continuous, soft: RESCS x, y

The "basic" reservation guarantees that there exists a time t such that for every integer i, a task of interest will receive x units of processor time during the time interval of $[t + iy, t + (i + 1)y]$. The standard RMA and EDF algorithms provide basic reservations. The "continuous" reservation relaxes the requirement on the time t. It indicates that for *any* time t, a task will receive x units of processor time during the interval of $[t, t + y]$. The Rialto/NT real-time kernel [35] is an example of continuous reservation. A "hard" reservation does not allow extra processor time to be used by a task, while a "soft" reservation does.

In [61], the authors point out the duality between resource reservation and proportional share. Later, in [55], the authors show that converting the guarantee provided by a resource reservation to that of a proportional share is possible, and vice versa.

Other bandwidth management techniques include several server approaches, such as the Constant Bandwidth Server (CBS) algorithm [1]. Again, a duality exists between the CBS approach and proportional share, and each has advantages as well disadvantages under certain conditions [2].

11.2.2 Motivations for Considering Proportional Share

In essence, proportional share provides each task with a "virtual processor" that executes in a fraction of the original processor speed. This property facilitates and simplifies the design and analysis of job scheduling algorithms.

Recall that our assurance problem has a two-fold objective, i.e., to probabilistically satisfy task utility bounds and to maximize the accrued utility. The second requirement needs a UA scheduling algorithm for resolving competition among jobs of the same task. In general, UA scheduling algorithms are dynamic and complicated. This is particularly true during overloads, while overloads (at least transient overloads) are inevitable for systems with stochastic parameters. Therefore, designing and analyzing UA job scheduling algorithms on a "virtual processor", rather than considering competitions from other tasks, is highly desired.

Another advantage of using proportional share is to have the capability of *temporal isolation*, i.e., the behavior of one task does not affect the bandwidth received by another task. Consider a complex system that is built by groups of develops across several enterprizes. Each group is responsible for developing a component of the system or a subsystem. Due to a variety of reasons, e.g., different security levels and/or management, proprietary techniques, it is very difficult, if not impossible, for any one to have a global knowledge of the system. The lack of global knowledge prevents system designers from conducting system wide, holistic analysis. Therefore, the provision of "virtual processors" can greatly aid system design, analysis, and testing.

Due to the duality between proportional share and resource reservation, the latter can also provide a "virtual processor" to each task. However, resource reservation algorithms often concern with providing processor time in period-like intervals, i.e., the y parameters in the four types of reservations. Proportional share, on the other hand, provides processors time in *any* time interval of *any* length, which is a more natural model for realizing bandwidth allocations. The same argument also applies to the server approaches. Therefore, we assume a proportional share scheduling algorithm, particularly the EEVDF algorithm, is used for realizing bandwidth allocation.

11.3 A Sufficient Feasibility Condition

This section presents our answer to the question: how much bandwidth is needed for a given task to achieve its assurance requirement? Our solution first establishes the relationship between processor bandwidth and achievable assurance. Given a task-level assurance requirement, the required processor bandwidth is then solved from this relationship.

To analyze the feasibility of tasks with stochastic parameters, we consider the "processor demand analysis" approach [10]. The processor demand analysis analyzes and compares the available processor time with the processor demand on any time interval. Let $C_p(t, t+L)$ be the processor demand of a task on a time interval $[t, t+L]$ and let $S_p(t, t+L)$ be the available processor time for the task on the same time interval. A task is schedulable if and only if

$$\forall t, L > 0, S_p(t, t+L) \geq C_p(t, t+L). \tag{11.5}$$

By "schedulable", we mean that a task's deadline or critical time can be satisfied.

The processor demand approach has been used to analyze the schedulability of real-time tasks under a variety of conditions. In [10], the authors develop the necessary and sufficient schedulability condition for a set of periodic tasks with relative deadlines shorter than their periods. Later, Jeffay and Stone [31] use the processor demand approach to account for the overhead of interrupt handlers. A more recent work using the processor demand approach is performed in the context of Rate-Based Executions [30].

Theorem 11.1. *Suppose a task T has a unimodal model arrival pattern $\langle k, W \rangle$. Also assume all jobs of T have identical relative deadline D. Then, all job deadlines can be satisfied if T has at least a processor bandwidth of $\rho = \max\{C/D, C/W\}$, where C is the total execution time of k jobs in a time window of W.*

Proof. The necessary and sufficient condition for satisfying job deadlines is

$$S_p(0, L) \geq C_p(0, L), \forall L > 0 \tag{11.6}$$

Let ρ be the processor bandwidth allocated to T. Thus, $S_p(0, L) = \rho L$. Furthermore, the total amount of processor time demand on $[0, L]$ is $C_p(0, L) = \left(\left\lfloor (L - D) \Big/ W \right\rfloor + 1 \right) C$. Therefore, condition 11.6 can be rewritten as

$$\rho L \geq \left(\left\lfloor \frac{L - D}{W} \right\rfloor + 1 \right) C, \forall L > 0 \tag{11.7}$$

Since $\left(\lfloor \frac{L-D}{W} \rfloor + 1 \right) \leq \left(\frac{L-D}{W} + 1 \right)$, it is sufficient to have $\rho L \geq \left(\frac{L-D}{W} + 1 \right) C, \forall L > 0$. This leads to

$$\rho \geq C \frac{L - D + W}{LW} = \frac{C}{W} \left(1 + \frac{W - D}{L} \right), \forall L > 0 \tag{11.8}$$

Case 1: $W > D$

It is easy to see that ρ monotonically decreases with L. Furthermore, notice that if $L \leq D$, $C_p(0, L) = 0$ because no job has a deadline earlier than D. Thus, it is sufficient to consider the case of $L = D$, where $\rho \geq C/D$.

Case 2: $W < D$

For such a case, ρ monotonically increases with L. Therefore, the maximal ρ is reached when $L \to \infty$. This leads to $\rho \geq C/W$. Since we know $W < D$, $\rho \geq C/W > C/D$.

Case 3: $W = D$

ρ is independent of L under such a case. It can be seen that $\rho \geq C/W = C/D$.

Combining the above three cases, we have a sufficient condition $\rho \geq \max\{C/D, C/W\}$. □

Observe that in Theorem 11.1, the processor bandwidth is allocated to accommodate the maximal processor load. This maximal load occurs only if the k jobs arrive simultaneously. Since the unimoal arrival model does not require a minimal separation between consecutive arrivals, simultaneous arrivals are possible.

Theorem 11.1 assumes that an arbitrary processor bandwidth of ρ can be perfectly realized. Recall that a bandwidth allocation realized by a proportional share algorithm has an optimal lag of Q, the size of one time quantum. The following theorem gives a practical bandwidth requirement under proportional share.

Theorem 11.2. *Under the condition of Theorem 11.1, if bandwidth allocation is realized using a proportional share algorithm with a maximal lag Q, then the required processor bandwidth is $\rho = \max\{(C+Q)/D, C/W\}$.*

Proof. With a maximal lag of Q, Equation 11.7 becomes

$$\rho L - Q \geq \left(\left\lfloor \frac{L - D}{W} \right\rfloor + 1 \right) C, \forall L > 0 \tag{11.9}$$

By the same argument as in Theorem 11.1, ρ satisfies

$$\rho \geq \frac{C}{W} + \frac{1}{L} \left[(C + Q) - \frac{DC}{W} \right] \tag{11.10}$$

Case 1: $C + Q > \frac{DC}{W}$. The maximal ρ occurs when $L = D$, which leads to $\rho \geq (C + Q)/D$.

Case 2: $C + Q < \frac{DC}{W}$. The maximal ρ is reached when $L \to \infty$. That is, $\rho \geq C/W > (C + Q)/D$.

Case 3: $C + Q = \frac{DC}{W}$. Obviously, $\rho \geq C/W$ is sufficient.

Combining these three cases, it is sufficient to have $\rho \geq \max\{(C + Q)/D, C/W\}$. □

In Theorems 11.1, solution $\rho \geq C/D$ implies $D < W$, i.e., the relative deadline of a job is shorter than the specification time window, which is often the case. For example, in a periodic task model, this case corresponds to a task with a relative deadline shorter than its period. For brevity in presentation, we focus on solution $\rho \geq C/D$ in Theorem 11.1 and solution $\rho \geq (C+Q)/D$ in Theorem 11.2 hereafter. Discussions and techniques assuming the aforementioned solutions can be easily extended to other solutions.

An important property of proportional share is that its time quantum is at least the maximal size of all task critical sections that are measured on a dedicated processor. Otherwise, mutual exclusion on shared resources may be failed. In [5], the authors use a lock-free scheme to demonstrate that even in the presence of shared resource, the maximal lag of a proportional share system is still bounded by Q, which is now the maximal size of all task critical sections. By doing so, Theorem 11.2 can be applied in a proportional share system with exclusively accessed shared resources.

The approach of using a large time quantum for resource access works well when all critical sections are relatively short. However, it is not efficient for handling long critical sections. One direct consequence of a large time quantum is over-allocating bandwidth. For each task T_i, Q/D_i bandwidth may be overly allocated compared with the ideal case. If one critical section in a system is considerably large, then *all* tasks are forced to suffer the same bandwidth over-allocation. We develop several lock-based, time/utility driven resource access protocols for large critical sections in Chapter 12.

Corollary 11.3. *Under the condition of Theorem 11.1 and $D_i < W_i, \forall i$, a set of tasks are schedulable if $\sum_{i=1}^{n} C_i/D_i \leq 1$. Under the condition of Theorem 11.2 and $(C_i+Q)/D_i > C_i/W_i, \forall i$, a set of tasks are schedulable if $\sum_{i=1}^{n} (C_i+Q)/D_i \leq 1$.*

Proof. This corollary can be directly derived by observing that a set of tasks are schedulable if and only if $\sum_{i=1}^{n} \rho_i \leq 1$. □

Corollary 11.3 naturally serves as a criterion for schedulability test. Next, we show that our feasibility conditions are equivalent to the Liu and Layland condition for periodic tasks [42], and are stronger sufficient conditions than that in [10] for periodic tasks with deadlines shorter than their periods. In addition, our bandwidth allocation and feasibility condition are applicable for a broader range of task models, e.g., aperiodic tasks, while the the existing sufficient and necessary conditions are true only for periodic task models.

Theorem 11.4. *Consider a set of periodic tasks with deadlines equal to periods, $\mathcal{T} = \{T_i | i = 1, ..., n\}$. If $\mathcal{T}$ is schedulable under the Liu and Layland condition [42], then it is also schedulable under the condition in Corollary 11.3. The reverse statement is also true.*

Proof. Observe that a periodic task has a arrival pattern of one arrival during any time interval of length w, which is called "period" and equals to task relative deadline in this special case. Therefore, C_i becomes T_i's job execution time. Furthermore, recall that the Liu and Layland condition requires $\sum_{i=1}^{n} C_i/D_i \leq 1$. This requirement essentially is the same as that of Corollary 11.3. □

Theorem 11.5. *Consider a set of periodic tasks with deadlines shorter than periods (i.e., $D_i < W_i$), $\mathcal{T} = \{T_i | i = 1, ..., n\}$. If $\mathcal{T}$ is schedulable by Corollary 11.3, then it is also schedulable under the condition of Baruah, Rosier, and Howell [10]. The reverse statement is not true.*

Proof. Let w_i and D_i be the period and relative deadline of a task T_i, respectively. And let $\mathcal{D} = \{d_{i,k} | d_{i,k} = kw_i + D_i, d_{i,k} \leq \min(B_p, H), i = 1, ..., n, k \geq 0\}$, where B_p is the busy period of the task set and H is its hyper period.

The condition of Baruah, Rosier, and Howell [10] states that $\mathcal{T}$ is schedulable if and only if

$$\forall L \in \mathcal{D}, L \geq \sum_{i=1}^{n} \left(\left\lfloor \frac{L - D_i}{w_i} \right\rfloor + 1 \right) C_i \tag{11.11}$$

To prove this theorem, we need to prove two statements are true.

1. Statement 1: If $\sum_{i=1}^{n} C_i/D_i \leq 1$ (Corollary 11.3), then Equation 11.11 also holds.

   Since $D_i < W_i$, $\sum_{i=1}^{n} \left(\left\lfloor \frac{L-D_i}{W_i} \right\rfloor + 1 \right) C_i < \sum_{i=1}^{n} \left(\left\lfloor \frac{L-D_i}{D_i} \right\rfloor + 1 \right) C_i$. Furthermore, $\left\lfloor \frac{L-D_i}{D_i} \right\rfloor + 1 \leq \left(\frac{L-D_i}{D_i} \right) + 1 = \frac{L}{D_i}$. Therefore

   $$\begin{aligned} &\sum_{i=1}^{n} \left(\left\lfloor \frac{L-D_i}{W_i} \right\rfloor + 1 \right) C_i < \sum_{i=1}^{n} \left(\left\lfloor \frac{L-D_i}{D_i} \right\rfloor + 1 \right) C_i \\ &\leq L \sum_{i=1}^{n} \frac{C_i}{D_i} \leq L \end{aligned} \tag{11.12}$$

2. Statement 2: If Equation 11.11 holds, $\sum_{i=1}^{n} C_i/D_i$ may be greater than 1, i.e., the reverse statement is not true.

   Observe $\sum_{i=1}^{n} C_i/D_i \leq 1$ requires that $L \sum_{i=1}^{n} C_i/D_i \leq L, \forall L > 0$. Therefore, to demonstrate the reverse statement is not true, we need to prove that for some C_i, D_i and L,

   $$\begin{aligned} &\sum_{i=1}^{n} \left(\left\lfloor \frac{L-D_i}{w_i} \right\rfloor + 1 \right) C_i < L \sum_{i=1}^{n} \frac{C_i}{D_i} \\ &\Rightarrow \sum_{i=1}^{n} \left(\left\lfloor \frac{L-D_i}{w_i} \right\rfloor + 1 - \frac{L}{D_i} \right) C_i < 0 \end{aligned} \tag{11.13}$$

   Notice that $x - 1 \leq \lfloor x \rfloor \leq x, \forall x \geq 0$. Therefore, it is possible that

   $$\begin{aligned} &\sum_{i=1}^{n} \left(\left\lfloor \frac{L-D_i}{w_i} \right\rfloor + 1 - \frac{L}{D_i} \right) C_i = \sum_{i=1}^{n} \left(\frac{L-D_i}{w_i} - \frac{L}{D_i} \right) C_i \\ &< \sum_{i=1}^{n} \left(\frac{L-D_i}{D_i} - \frac{L}{D_i} \right) C_i = -\sum_{i=1}^{n} C_i < 0 \end{aligned} \tag{11.14}$$

□

Having examined the bandwidth requirement and feasibility condition for unimodal arrival models, we now proceed with tasks specified by PUAM arrival models.

Let N_i be the random variable for the number of arrivals during a time window of W_i. The, the processor demand of task T_i during a time window of W_i is $C_i = \sum_{j=1}^{N_i} c_{i,j}$. By Theorem 11.2, $\rho_i \geq (C_i + Q)/CT_i$, where CT_i is the critical time assigned to task T_i. To satisfy the assurance probability, we require that the probability of satisfying Theorem 11.2 is at least AP_i.

$$\Pr\left[\forall j, s_{i,j} \leq CT_i\right] = \Pr\left[\sum_{j=1}^{N_i} c_{i,j} \leq (\rho_i CT_i - Q)\right] \geq AP_i \tag{11.15}$$

The above condition is the fundamental bandwidth requirement for satisfying a task's critical time. In case that $N_i = k$, the total processor time demand during a time window becomes $\sum_{j=1}^{k} c_{i,j}$. Therefore, Equation 11.15 can be rewritten as a sum of a set of conditional probabilities:

$$\Pr\left[\forall j, s_{i,j} \leq CT_i\right] = \sum_{k=1}^{\infty}\left(p_i(k) \times \Pr\left[\sum_{j=1}^{k} c_{i,j} \leq (\rho_i CT_i - Q)\right]\right) \geq AP_i \tag{11.16}$$

11.4 Solutions to The Feasibility Condition

11.4.1 A General Solution

This section presents a general solution to the problem of computing the minimal bandwidth for a task (Equation 11.15). The only assumed information is the average number of task arrivals in a specification time window and the average execution time of a task. In general, distributions of task arrival pattern and execution times are difficult to obtain. However, the average values may be obtainable through measurements for many applications.

Let N_i be the number of arrivals during a time window of W and let $C_p = \sum_{j=1}^{N_i} c_{i,j}$. The feasibility condition can be rewritten as:

$$\Pr\left[C_p \leq \rho_i CT_i - Q\right] = 1 - \Pr\left[C_p \geq \rho_i CT_i - Q\right] \geq AP_i \tag{11.17}$$

Note that N_i is a random variable and follows a distribution specified by $p_i(a)$. Thus, C_p is also a random variable. By Markov's Inequality, $\Pr[X \geq t] \leq E(X)/t$ for any non-negative random variable X.

Therefore,

$$\Pr[C_p \leq \rho_i CT_i - Q] = 1 - \Pr[C_p \geq \rho_i CT_i - Q] \geq 1 - \frac{E(C_p)}{\rho_i CT_i - Q} \tag{11.18}$$

If we can determine a ρ_i so that $1 - \frac{E(C_p)}{\rho_i CT_i - Q} \geq AP_i$, $\Pr[C_p \leq \rho_i CT_i - Q] \geq AP_i$ is also satisfied. This leads to

$$\rho_i \geq \frac{E(C_p)}{CT_i\,(1 - AP_i)} + \frac{Q}{CT_i} \tag{11.19}$$

Furthermore, by Wald's Equation, $E(C_p) = E\left(\sum_{j=1}^{N_i} c_{i,j}\right) = E(c_i)E(N_i)$. Thus,

$$\rho_i \geq \frac{E(c_i)E(N_i)}{CT_i\,(1 - AP_i)} + \frac{Q}{CT_i} \tag{11.20}$$

This solution is applicable for any distributions of c_i and N_i. Naturally, it is pessimistic for some distributions. For example, consider a periodic task ($N_i = 1$) with a constant execution time C_i. Equation 11.20 yields a solution of $\rho_i = C_i / \left(CT_i(1 - AP_i)\right) + Q/CT_i$. However, the required bandwidth for a periodic task is only C_i/CT_i. This phenomena suggests that high performance assurance may yield a highly pessimistic bandwidth allocation, particularly for tasks with regular patterns of arrivals and of execution time.

In addition, the second term in Equation 11.19 is due to the lag in proportional share. Ideally, $Q \ll CT_i$. Otherwise, the bandwidth allocation will be too pessimistic to be useful.

11.4.2 A Binary Search Strategy

The previous section assumes the minimal information regarding task arrivals and execution times, e.g., only average values are assumed. Therefore, the solution in Equation 11.20 is applicable for any distribution, but may be pessimistic for some distributions. This section presents a binary search strategy that demands and utilizes the information of full distributions of task arrivals and execution times. Thus, in general, this binary search strategy can yield a tighter solution that requires smaller bandwidth than that resulting from Equation 11.20.

To avoid wordy presentation, we introduce a shorthand notation for the left hand side of Equation 11.16. Let $feasibleProb_i(\rho_i) = \sum_{k=0}^{\infty} \left(p_i(k) \times \Pr\left[\sum_{j=1}^{k} c_{i,j} \leq \rho_i CT_i - Q \right] \right)$. This function calculates the probability of satisfying all job critical deadlines if task T_i were assigned a processor bandwidth of ρ_i.

Then, Equation 11.16 can be restated as solving the minimal ρ_i that satisfies $feasibleProb_i(\rho_i) \geq AP_i$. Notice that function $feasibleProb_i(\rho_i)$ is a monotone in terms of ρ_i. Therefore, a binary search is applicable

on the set of $\rho_i \in [0, 1]$.

The Binary Search strategy, presented in Algorithm 7 works as the following. The algorithm accepts a range to search the minimal bandwidth, denoted as $[a, b]$, and the required assurance probability AP_i. In our case, invoking $minBW(0, 1, AP_i)$ returns either the minimal required ρ_i, or **failure** if even the maximal processor bandwidth, i.e., $\rho_i = 1$, cannot satisfy AP_i. The algorithm recursively search half of the input search range until the range is smaller than a predefined allocation error ϵ.

It is obvious to see that in the worst case, function $minBW(a, b, AP_i)$ performs $\log_2((b-a)/\epsilon)$ searches. If $a = 0$ and $b = 1$, the worst case complexity of $minBW(0, 1, AP_i)$ becomes $\log_2(1/\epsilon)$.

```
1: Parameters ε: the maximal acceptable allocation error;
2: Input  : a, b, AP_i
3: Output : the minimal ρ_i that satisfies feasibleProb(ρ_i) ≥ AP_i for task T_i; failure if
            the maximal bandwidth b cannot satisfy AP_i
4: if feasibleProb_i(b) < AP_i then
5:     return failure;
6: if b − a ≤ ε then
7:     return a;
8: if feasibleProb_i((a + b)/2) ≥ AP_i then
9:     return minBW (ε, a, (a + b)/2, AP_i);
10: else
11:    return minBW (ε, (a + b)/2, b, AP_i);
```

Algorithm 7: `minBW`(a, b, AP_i) Function

Given a task arrival pattern $\langle p_i(a), W_i \rangle$, Algorithm 7 requires calculating $\Pr\left[\sum_{j=1}^{k} c_{i,j} \leq \rho_i CT_i - Q\right]$ so that $feasibleProb_i(\rho_i)$ can be computed. Recall that a task specification only states the distribution of job execution times. Computing the sum distribution of a set of independent random variables, i.e., the sum of k job execution times, requires convolutions. In practice, convolutions are performed for small k. When k is large, the sum distribution can be approximated by the Central Limit Theorem (CLT), regardless of the original distribution of job execution times.

Let $S_k = \sum_{j=1}^{k} c_{i,j}$. And let the mean and variance of task T_i's execution time be μ_i and σ_i^2, respectively. The Central Limit Theorem states that when k is large, S_k converges to a normal distribution $\mathcal{N}(k\mu_i, \sqrt{k}\sigma_i)$. Let $\Phi(t)$ be the cumulative distribution function of the standard normal distribution $X \sim \mathcal{N}(0, 1)$, i.e., $\Phi(t) = \Pr[X \leq t]$. The probability that the sum of k job execution times is less than or equal to t can be approximated as

$$\Pr[S_k \leq t] \approx \Phi\left(\frac{t - k\mu_i}{\sqrt{k}\sigma_i}\right) \tag{11.21}$$

The CLT approximation is accurate when k is large. The problem of how large a k should be remains unsolved. That is, the error between an approximation using CLT and its actual value needs to be bounded. The most widely used bound for such an approximation is the Berry-Esséen bound, if the third absolute central moment of a task execution time is finite.

If $E\left(|c_i - \mu_i|^3\right) < \infty$, the Berry-Esséen Theorem states that

$$\left|\Pr\left[\frac{S_k - k\mu_i}{\sqrt{k}\sigma_i} \leq t\right] - \Phi(t)\right| \leq \frac{33}{4}\frac{E\left(|c_i - \mu_i|^3\right)}{\sqrt{k}\sigma_i^3} \tag{11.22}$$

Or:

$$\left|\Pr\left[S_k \leq t\right] - \Phi\left(\frac{t - k\mu_i}{\sqrt{k}\sigma_i}\right)\right| \leq \frac{33}{4}\frac{E\left(|c_i - \mu_i|^3\right)}{\sqrt{k}\sigma_i^3} \tag{11.23}$$

This theorem indicates that the rate of converging to a normal distribution is on the order of $1/\sqrt{k}$. Therefore, the probability of $\Pr[S_k \leq t]$ is lower bounded by

$$\Pr\left[S_k \leq t\right] \geq \Phi\left(\frac{t - k\mu_i}{\sqrt{k}\sigma_i}\right) - \frac{33}{4}\frac{E\left(|c_i - \mu_i|^3\right)}{\sqrt{k}\sigma_i^3}. \tag{11.24}$$

The problem of computing the minimal bandwidth becomes determining a minimal ρ_i so that

$$\Pr\left[\sum_{k=1}^{\infty}\left(p_i(k) \times \left(\Phi\left(\frac{\rho_i CT_i - Q - k\mu_i}{\sqrt{k}\sigma_i}\right) - \frac{33}{4}\frac{E\left(|c_i - \mu_i|^3\right)}{\sqrt{k}\sigma_i^3}\right)\right)\right] \geq AP_i \tag{11.25}$$

The Central Limit Theorem and Berry-Esséen Theorem may provide a better approximation for ρ_i than that can be computed from Section 11.4.1. However, for some special cases, we can compute the exact values of $\Pr[S_k \leq t]$, rather than by approximation. These special cases include constant job execution times, and gamma distributed execution times.

1. **Constant execution time** Assume task T_i's execution is a constant, denoted as C_i. Then, $S_k = kC_i$. $\Pr[S_k \leq t]$ can either be zero or one.

$$\Pr\left[S_k \leq t\right] = \begin{cases} 1, k \leq \left\lfloor \frac{t}{C_i} \right\rfloor \\ 0, otherwise \end{cases}$$

   Therefore, $feasibleProb_i(\rho_i) = \sum_{k=0}^{k_0} p_i(k)$, where $k_0 = \lfloor \rho_i CT_i/C_i \rfloor$.

2. **Gamma distributed execution time** If the execution time of a task T_i, i.e., c_i is gamma distributed with parameters α and θ, then $E(c_i) = \alpha\theta$ and $Var(c_i) = \alpha\theta^2$. Furthermore, $S_k = \sum_{j=1}^{k} c_{i,j}$ is also gamma distributed with parameters of $k\alpha$ and θ. Therefore,

$$\Pr[S_k \leq t] = P\left(k\alpha, \frac{t}{\theta}\right) \tag{11.26}$$

   where $P(a, z)$ is a regularized gamma function.

Table 11.1: An Example Task Set

Task	Arrival Pattern	Execution Time (s)	AP_i	CT_i
T_1	$\mathcal{P}(6)$, $W_1 = 2$	constant, 0.1	0.7	1.8
T_2	$\mathcal{P}(3)$, $W_2 = 2$	$\Gamma(1.2, 0.1)$	0.6	1.6
T_3	$\mathcal{B}(10, 0.2)$, $W_3 = 3$	constant, 0.05	0.85	2.1
T_4	$\mathcal{B}(5, 0.3)$, $W_4 = 2.2$	$\Gamma(0.8, 0.2)$	0.8	1.7

Table 11.2: Bandwidth Allocation for The Example Task Set ($\epsilon = 0.05$, $Q = 1ms$)

Task	Upper Bound	CLT Approximation	Exact Solution
T_1	1.1111	0.3750	0.3750
T_2	0.5625	**failure**	0.2500
T_3	0.3175	0.0938	0.0938
T_4	0.7059	**failure**	0.5313

11.4.3 Numerical Examples

In this section, we present an example task set with four tasks. We calculate and compare bandwidth allocation using the pessimistic estimation in Section 11.4.1 (abbreviated as "Upper Bound"), Central Limit Theorem approximation in Section 11.4.2 (abbreviated as "CLT Approximation"), and exact solution (abbreviated as "Exact Solution").

The arrival patterns of the tasks follow either Poisson distributions, or Binomial distributions. For task execution time distributions, we consider constant execution times and gamma distributed execution times. Notice that exact solutions for these two execution time profiles can be easily computed because the sum distribution $\Pr[S_k \leq t]$ is known. Therefore, they serve as illustration examples. Exact solutions of other execution time distributions may be computed using convolutions.

The arrival patterns, execution time distributions, and timeliness requirements of the task set are summarized in Table 11.1. The bandwidth allocation results using the aforementioned approaches are listed in Table 11.2.

Observe that the "Upper Bound" values are loose, compared with the exact solutions. However, they indicate the worst case bandwidth requirements. If a task set is schedulable using the "Upper Bound" bandwidth allocation, a feasible bandwidth allocation must exist.

On the other hand, the CLT approximation works well for constant execution times, but is significantly inaccurate for gamma distributed execution times. This is because the Berry-Esséen bound is large in this particular task set. For example, in Table 11.1, task T_3's execution time follows $\Gamma(0.8, 0.2)$. Even when $k = 5$,

the Berry-Esséen bound is computed as $11.6783 \gg 1 \geq \Phi(t)$. This large bound yields $S_k(t) \geq -10.6783$, which is certainly useless. This phenomena suggests that a combination of convolution (for a small number of jobs) and CLT approximation (for a large number of jobs) may result in a more accurate bandwidth calculation. Particularly, a Binomial distribution has a limited number of task arrivals, for which convolution is more appropriate.

11.5 A Utility Accrual Job Scheduling Algorithm

EDF is known to be optimal in terms of satisfying all deadlines if they can be satisfied. In our context, all jobs of the same task have identical relative critical time. Thus, EDF is equivalent to FIFO for scheduling jobs of the same task. However, EDF is not directly related to maximizing accrued utility under two cases:

- The processor is overloaded. The EDF algorithm is known to suffer "domino" effects during overloads [44] and cannot selectly miss job deadlines. Essentially, all job deadlines will be missed once the processor becomes overloaded.
- A task has a non-step TUF. If the task has a downward step TUF and the processor is not overloaded, satisfying all deadlines implies maximizing accrued utilities, because there is no benefit of completing a job earlier that its deadline time. However, if a job has a non-step TUF, e.g., an exponentially decreasing TUF, an early completion can actually accrue higher utility than that at the deadline time. Furthermore, a late completion can also be beneficial as it may accrue some utility.

Therefore, we aim to develop a utility accrual job scheduling algorithm that possesses several properties:

1. If all job deadlines can be satisfied by EDF, then UJS should be able to do so and accrue at least the same utility as EDF does; and
2. In case that not all jobs deadlines can be satisfied, UJS seeks to accrue as much utility as possible.

In the meanwhile, a fast job scheduling algorithm is desired. This is particularly true in the context of proportional share, where the proportional share mechanism itself may be implemented by another scheduling algorithm, such as EEVDF [62]. Thus, we adopt the Highest Utility Density First (HUDF) heuristic as a way to improve accrued utility. Our rational for doing so is because HUDF is easy to implement, incurs small overhead, and has exhibit high performance both during underloads and overloads, which has been partly reflected by GBS' performance.

Let t_0 be the time instant when a scheduling event occurs. On a dedicated processor, the Utility Density (UD) of a job J is defined as the ratio of its utility at predicted completion time over its remaining

execution time. That is, $UD = U\left(t_0 + c(t_0)\right)/c(t_0)$, where $U(t)$ is the TUF of job J and $c(t_0)$ is J's remaining execution time at instant t_0. On a virtual processor provided by the underlying proportional share algorithm, the expected completion of J is prolonged to $t_0 + (c(t_0) + Q)/\rho$. Therefore, the UD of J on a virtual processor is $U(m)/m$, where $m = (c(t_0) + Q)/\rho$.

```
1:  Input  : a queue of ready jobs, denoted as RQ
2:  Output : job J_s to be executed next; or NULL
3:  if RQ is empty then
4:      Select NULL job;
5:  Let U_d be the accrued utilities of all jobs in RQ under EDF;
6:  Let U_h be the accrued utilities of all jobs in RQ under Highest Utility Density First
    (HUDF);
7:  if RQ is feasible under EDF then
8:      if RQ is feasible under HUDF and U_h > U_d then
9:          Select the highest utility density job;
10:     else
11:         Select the earliest deadline job;
12: else
13:     Select the highest utility density job;
```

Algorithm 8: `UJSsched` Algorithm

The design of the `UJSshced` algorithm closely follows its desired properties. As shown in Algorithm 8, the algorithm first examines if the set of ready jobs are schedulable under both EDF and HUDF, assuming jobs are executed on a virtual processor with an execution rate of ρ. If they are, the highest utility density job is selected. If the jobs are only schedulable under EDF, then the earliest critical time job should be executed. In case that the jobs are not schedulable, the next job is selected by HUDF.

Without loss of generality, we assume that at time instant t_0, k jobs of a task T_i compete for the processor bandwidth allocated to T_i. Furthermore, jobs are in the order of non-decreasing absolute deadlines, i.e., $d_i \leq d_{i+1}$. Then, the set of jobs are schedulable under EDF if and only if

$$\forall i = 1, ..., k, \frac{\sum_{j=1}^{i} c_j(t_0) + Q}{d_i} \leq \rho \tag{11.27}$$

This condition is a direct extension to that for a set of generic tasks on a dedicated processor [11].

Lemma 11.6. *If all job deadlines can be satisfied by EDF, then UJSsched algorithm should be able to do so and accrue at least the same utility as EDF does.*

Proof. The ability to satisfy all job deadlines is obvious, because `UJSshed()` is invoked only if it can produce a feasible schedule. □

Chapter 12

Resource Access Protocols

This chapter is motivated by the fact that proportional share uses large time quanta to ensure mutual exclusion. Using large time quanta works well for short critical sections. However, we conjecture that for some cases, a small time quantum combined with lock-based, resource access protocols may yield lower bandwidth requirement.

12.1 Problem Statement and Terminologies

When time quanta are smaller than the length of critical sections, preemptions of a task while it is inside a critical section may happen. Therefore, we use a lock-based approach to ensure mutual exclusion. That is, a job is required to acquire the lock guarding a shared resource R before it can start using R.

Due to the use of locks, three types of blocking may occur:

1. **Direct Blocking** If a job $J_{i,m}$ requests a resource R that is currently held by another job $J_{j,k}$, we say that job $J_{i,m}$ is *directly blocked* by job $J_{j,k}$. Job $J_{j,k}$ is called the blocking job. Because processor bandwidth is allocated on a per task basis, we also say that task T_i is blocked by task T_j.

2. **Transitive Blocking** If a job J_a is blocked by J_b which in turn is blocked by job J_c, we say that job J_a is *transitively blocked* by J_c.

3. **Queue Blocking** Suppose a set of tasks $\mathcal{TB} = \{T_{b1}, T_{b2}, ..., T_{bk}\}$ are simultaneously blocked on a resource R, held by task T_o. When T_o releases R, one of the blocked tasks, e.g., task T_{bm} will acquire R and continue execution. Thus, another task T_{bn} will suffer additional blocking time due to T_{bm},

besides the blocking time due to T_o. We call such an additional blocking *queue blocking* as it is caused by a queue of blocked tasks. This definition can be expanded to the case of multiple tasks in $\mathcal{TB}$ being granted R before T_{bn}.

The objective of resource access protocols is to effectively bound or reduce the blocking time suffered by a task. In this chapter, we present three protocols. The first protocol is called Bandwidth Inheritance (BWI). The BWI protocol speeds up the execution of a blocking task and thus reduces direct blocking times. The second protocol, Resource Level Policy (RLP), can bound the queue blocking time suffered by a task. However, neither BWI nor RLP can solve the problem of transitive blocking. Furthermore, without additional mechanism, deadlock may occur. Therefore, a third protocol, called Early Blocking Protocol (EBP) is proposed. The EBP protocol can avoid deadlock as well as bounding the duration of transitive blocking.

Recall that we use the `UJSsched` algorithm (Section 11.5) to resolve competition among jobs of the same task. Therefore, resource blocking can occur among jobs, which complicates the analysis of the job scheduling algorithm. Furthermore, notice that assurance requirements are at the task level. Therefore, we simply disallow preemptions while a job holds a resource. From the perspective of the virtual processor, the `UJSsched` algorithm is invoked when a new job arrives and when the currently executing job completes.

Observe that both transitive blocking and deadlock can occur only in the presence of nested critical sections. This observation is formulated in Lemma 12.1. Therefore, both BWI and RLP protocols only consider the case of no nested critical sections.

Lemma 12.1. *Transitive blocking can occur only in the presence of nested critical sections. That is, if a job J_a is transitively blocked by another job J_c, there must be a job J_b that is currently inside nested critical sections.*

Proof. By the definition of transitive blocking, there exists a job J_b that blocks J_a and is blocked by J_c at the same time. Since J_a is blocked by J_b, J_b must hold a resource, e.g., R_1. In addition, the fact that J_b is blocked by J_c implies that J_b requests another resource, e.g., R_2 which is currently held by J_c. Therefore, J_b must be inside nested critical sections. □

Besides the property of no transitive blocking, lacking nested critical sections also prevents deadlocks, because one of the necessary condition for deadlock, namely *hold-and-wait* is not possible. Therefore, the BWI and RLP protocols are suitable for the cases of no nested critical sections.

Before we proceed with our resource access protocols, we discuss notations and assumptions as follows.

- $z_{i,j}$ denotes the j^{th} critical section of task T_i.

- $d_{i,j}$ denotes the duration of critical section $z_{i,j}$ on a dedicated processor without processor contention.
- $R_{i,j}$ is the resource associated with critical section $z_{i,j}$.
- d_i^j denotes the duration of task T_i's critical section that accesses resource R_i
- $z_{i,k} \subset z_{i,m}$ means that $z_{i,k}$ is entirely contained in $z_{i,m}$.
- All critical sections are "properly" nested, i.e., for any pair of $z_{i,k}$ and $z_{i,m}$, only one of the following conditions is true:
  - $z_{i,k} \subset z_{i,m}$
  - $z_{i,m} \subset z_{i,k}$
  - $z_{i,k} \bigcap z_{i,m} = \emptyset$.
- All critical sections are guarded by binary semaphores. Thus, only one job can be inside a critical section at any given time.

12.2 Bandwidth Inheritance

Bandwidth Inheritance (BWI) is inspired by Priority Inheritance (PIP) [57]. However, in the context of PIP, the primary objective is to avoid so called "priority inversion" and thus to bound task blocking times. In our context, priority may not play a role. Rather, our objective is to reduce direct blocking times.

The basic idea of BWI protocol is to speed up the execution time of a blocking task, e.g., T by transferring all bandwidth of tasks that are blocked by T. Consequently, these blocked tasks lose their bandwidth and thus are stalled.

12.2.1 Protocol Definition

The Bandwidth Inheritance Protocol is defined as a set of rules:

1. If a task T_i is blocked on resource R that is currently held by T_j, the processor bandwidth of task T_i is inherited by task T_j. That is, the processor bandwidth of task T_j is temporarily increased to $\rho_i + \rho_j$ until T_j releases resource R. In the meanwhile, the bandwidth of task T_i becomes zero. Thus, T_i is stalled even if some jobs of T_i are eligible for execution.
2. Bandwidth inheritance is transitive. That is, if a task T_a is blocked by T_b which in turn is blocked by task T_c, then the bandwidth of T_a is also transferred to T_c.
3. Bandwidth inheritance is additive. Suppose a task T_a holds a resource R, and a set of tasks $\mathcal{TB} = \{T_i, \forall i = 1, ..., k\}$ are all blocked on R. Then, the bandwidth of T_a is increased to $\rho_a + \sum_{i=1}^{k} \rho_i$.

Recall that we have identified three types of blocking, i.e., direct blocking, transitive blocking, and queue blocking. The three rules of our BWI protocol explicitly indicate how the bandwidth of blocked tasks can be transferred to the blocking task for the three types of blocking. Thereby, we can reduce the duration of the blocking task's critical section.

Furthermore, transferring of task bandwidth can be implemented as dynamical task *join* and *leave*. The ability to allow tasks joining and leaving a system while maintaining a constant lag is one of the unique features of the EEVDF algorithm.

12.2.2 Blocking Time under BWI

Having defined the BWI protocol, one problem remains unsolved: what is the upper bound on the blocking task's duration of critical section? Given a duration of d_i on a dedicated processor, the answer obviously depends on how much bandwidth the blocking task has.

Assume that the blocking task has a total bandwidth of ρ, possibly through bandwidth inheritance. Then, the duration of the critical section is d_i/ρ. Therefore, the key to bound the duration is to lower bound the processor bandwidth allocated to a blocking task. An arbitrarily small bandwidth essentially yields an unbounded blocking time.

In Chapter 11, we have developed methods to calculate the minimal bandwidth to satisfy a task's timeliness requirements, assuming no resource blocking. In the following theorem, we establish the relationship between the bandwidth requirements with and without resource blocking.

Theorem 12.2. *For the same task model assumed in Theorem 11.2, if a task may be blocked on resource access, then the minimal required bandwidth is* $\rho = (B+C+Q)/D$*, where* B *is the total blocking time suffered by jobs of the task during a specification time window of* W*.*

Proof. The proof is similar to that of Theorem 11.2. To satisfy job deadlines, the available processor time during any time interval $[0, L]$, excluding the blocking time, should be greater than or equal to job processor demand.

$$S_p(0,L) - Q - \left(\left\lfloor \frac{L-D}{W} \right\rfloor + 1\right) B \geq \left(\left\lfloor \frac{L-D}{W} \right\rfloor + 1\right) C, \forall L > 0 \tag{12.1}$$

This leads to

$$\rho L \geq \left(\left\lfloor \frac{L-D}{W} \right\rfloor + 1\right) (B+C) - Q, \forall L > 0 \tag{12.2}$$

By the same argument as in the proof of Theorem 11.2, we have $\rho \geq (B+C+Q)/D$. □

Therefore, if $\rho_i^{min} = (C_i + Q)/D_i$ is T_i's processor bandwidth by assuming no resource blocking, it is safe to use ρ_i^{min} as the lower bound on T_i bandwidth even in the presence of resource blocking. Furthermore, observe that if T_i is a blocking task, it must inherit the bandwidth of at least one blocked task. Let $\mathcal{TR}$ be the set of tasks that may be blocked by T_i. T_i's total bandwidth while it is inside the critical section (of using resource R) is at least $\rho_i^{min} + \min\{\rho_j^{min} | j \neq i \bigwedge T_j \in \mathcal{TR}\}$. The direct blocking time caused by T_i is upper bounded by $(d_i + Q) \Big/ \left(\rho_i^{min} + \min\{\rho_j^{min} | j \neq i, T_j \in \mathcal{TR}\}\right)$, where d_i is the duration of T_i critical section of using resource R. This blocking time calculation is repeated for all critical sections of a task and for all jobs of a task in a specification window.

12.2.3 Bandwidth Allocation under BWI

Assume that each task T_i may access n_i shared resources, numbered as $R_{i,j}, j = 1, ..., n_i$. We also introduce the following notations:

- $d_{R_{i,j}}$ is the maximal length of the critical section that a task accesses resource $R_{i,j}$
- $\rho_{R_{i,j}}^{min}$ is the smallest ρ^{min} among all tasks that may access resource $R_{i,j}$

The direct blocking time that T_i may suffer when accessing $R_{i,j}$ is

$$B_{R_{i,j}} = \frac{d_{R_{i,j}}}{\rho_{R_{i,j}}^{min} + \rho_i^{min}} \tag{12.3}$$

The direct blocking time that a job of T_i may suffer is

$$B_D = \sum_{j=1}^{n_i} B_{R_{i,j}} = \sum_{j=1}^{n_i} \frac{d_{R_{i,j}} + Q}{\rho_{R_{i,j}}^{min} + \rho_i^{min}}, \tag{12.4}$$

where n_i is the number of critical sections that T_i may use.

According to Theorem 12.2, we require that the probability of satisfying task critical time is at least AP_i. This leads to

$$\begin{aligned} &\sum_{k=0}^{\infty} p_i(k) \Pr[B + C + Q \leq \rho_i CT_i] \geq AP_i \\ &\Rightarrow \sum_{k=0}^{\infty} p_i(k) \Pr\left[k \sum_{j=1}^{n_i} \frac{d_{R_{i,j}} + Q}{\rho_{R_{i,j}}^{min} + \rho_i^{min}} + \sum_{j=1}^{k} c_{i,j} + Q \leq \rho_i CT_i\right] \geq AP_i \end{aligned} \tag{12.5}$$

For all tasks, we first calculate the minimal bandwidth requirements without resource blocking, i.e., ρ_i^{min}, using the techniques presented in Chapter 11. The direct blocking time for each job of T_i, namely B_D is then calculated. Observe that the net effect of resource blocking is to increase the task execution time. In the case of direct blocking, the execution time of a job is increased by B_D, which has been calculated. Once the blocking time is calculated, the bandwidth requirement under BWI can be computed from Equation 12.5. Solutions in Chapter 11 can be applied to solve Equation 12.5 for ρ_i.

12.3 Resource Level Policy

12.3.1 Protocol Definition

The Resource Level Policy (RLP) protocol is specifically developed to bound queue blocking times. The idea is to associate a static numeric value with each task, called Resource Level (RL) of the task. The RL of a task is static is the sense that it is assigned when a task is created, maintains intact during the life time of the task, and is the same for all jobs of the same task. By using static RLs, we aim to produce predictable order of accessing a shared resource in case a queue of tasks are blocked on the same resource. Therefore, queue blocking times may be bounded. Similar technique has been used in the Stack Resource Policy [7], where each task is assigned a static Preemption Level.

If there are n tasks in a system, the RLs of tasks are integers from 1 to n. We assume that a larger numeric value means higher resource level. However, there are different ways of assigning static RLs. In general, the way of assigning static RLs should reflect our goal of utility accrual, i.e., maximizing the accrued utilities. Therefore, we propose several criteria for assigning static RLs:

1. **Maximal Heights of TUFs** For any pair of tasks, if $maxU_i > maxU_j$, then $RL_i > RL_j$. $maxU_i$ is the maximal height of a TUF, i.e., $maxU = \{U_i(t)|I_i < t < X_i\}$. The approach is easy to implement and works well for step TUFs. However, it ignores the task execution time information. Furthermore, for non-step TUFs, the maximal height of a TUF may be much higher than that can be accrued by a task.

2. **Pseudo Slopes** The pseudo slope of a task is defined as follows: $pSlope_i = U_i(I_i)/(X_i - I_i)$. The use of pseudo slope seeks to capture the shape information of a TUF. Still, task execution time is ignored from the calculation.

3. **Pseudo Utility Densities** The pseudo utility density of a task measures the amount of utility it can accrue, by average, per unit time of execution. $pUD_i = \frac{U_i(\rho_i^{min} E(c_i))}{\rho_i^{min} E(c_i)}$.

Using static RLs, the task with the highest RL shall be granted a resource R if there is a queue of tasks blocked on R. Therefore, when calculating the queue blocking time for task T_i, we only need to consider tasks with RLs higher than that of T_i. For example, if $RL_i = i$, then task T_i only suffers queue blocking due to tasks $T_j, j = i + 1, ..., n$.

Unfortunately, this scheme of using static RLs may yield unbounded queue blocking times for low RL tasks. Figure 12.1 shows an example of why static resource levels are not adequate. In Figure 12.1, task T_2 is blocked on a resource request and is later starved.

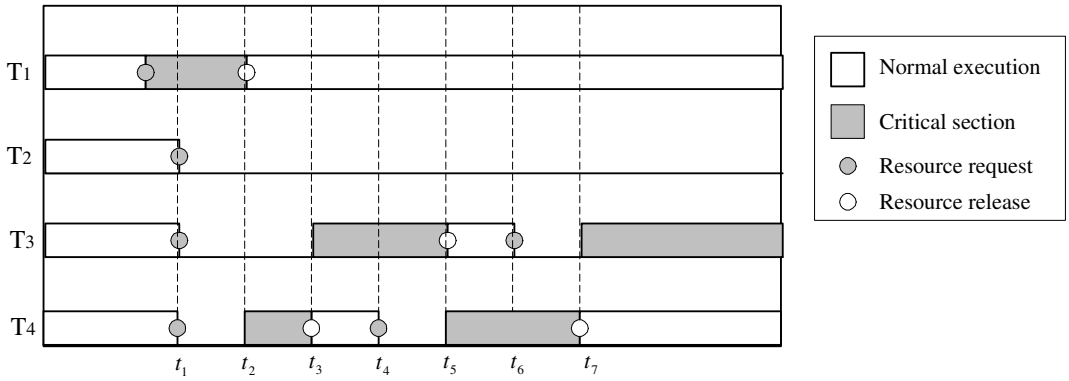


Figure 12.1: An Example of Using Static Resource Levels

To overcome the difficulty with static resource levels, we introduce the concept of Effective Resource Level (ERL). Besides RL, each task is associated with an ERL, which may increase over time. The idea is to use ERL to prevent a few high RL tasks from dominating the usage of shared resources.

Now, the revised resource level policy with effective resource levels works as follows.

1. If a task is not blocked on any resource, its ERL is the same as its static RL.
2. Whenever a resource R is released, the ERL's of all tasks that are currently blocked on R are increased by n, where n is the number of tasks in the system.
3. When a resource R becomes free, one of the blocked tasks with the highest ERL is granted resource access. If a tie among the highest ERL tasks occurs, the task with the longest blocking time wins.
4. When a task acquires the resource on which it was blocked, its ERL returns to its static RL.

Theorem 12.3. *Under resource level policy, a task T_k can be queue blocked on a resource R for at most the duration of $(m-2)$ critical sections, where m is the number of tasks that may access R.*

Proof. Consider a set of tasks $\mathcal{TB}$, including task T_k, are blocked on a resource R. Obviously, $|\mathcal{TB}| \leq m-1$, because one task must be holding the resource. Assume at time instant t_0, resource R is released by the current blocking task. Thus T_k's ERL is increased to $RL_k + n$, which is higher than $RL_i, \forall i$. This high ERL effectively ensures that no tasks that are blocked on R after t_0 can queue block T_k. Therefore, T_k can only suffer additional queue blocking from existing blocked tasks, which are at most $(m-3)$ critical sections. Note that at t_0, one of the tasks from $\mathcal{TB}$, i.e., T_m is granted resource R. Therefore, the number of the remaining blocked tasks, excluding T_k, is $|\mathcal{TB} - T_k| - 1 \leq (m-3)$. This theorem follows by summing up queue blocking times before and after instant t_0, i.e., $1 + (m-3) = (m-2)$.

The above proof also holds in the presence of multiple resources. Observe that a task, i.e., T_m that is blocked on R after t_0 must not be blocked on any other resources when it requests R. Therefore, by the RLP protocol, its ERL must be the same as its RL, which is lower than that T_k. This argument is true even if T_m holds other resources. □

Corollary 12.4. *The ERL of a task T_i is within the range of $[RL_i, (m-1)n + RL_i]$, where m is defined in Theorem 12.3 and n is the number of tasks in the system.*

Proof. According to Theorem 12.3, a task can suffer a queue blocking time of at most $(m-2)$ critical sections. In addition, it suffers one direct blocking. Upon releasing a shared resource, these blocking tasks increase the ERL of a task $(m-2)+1 = m-1$ times. Since each increase is n, the ERL of T_i is bounded by $(m-1)n + RL_i$. □

Theorem 12.5. *Let $\mathcal{T}_R$ be the set of tasks that may access resource R. The bound on queue blocking time in Theorem 12.3 is tight for any task $T_i \in \mathcal{T}_R$, except the highest RL task in $\mathcal{T}_R$.*

Proof. Without loss of generality, let $\mathcal{T}_R = \{T_1, T_2, ..., T_m\}$ and $RL_i = i$. We prove this theorem by showing that there *always* exists a resource access pattern so that any task $T_i \in \mathcal{T}_R, i < m$ suffers a queue blocking time of $(m-2)$ critical sections. The resource access pattern can be constructed as follows. Note that t_i is a time stamp and satisfies $t_{i+1} > t_i$.

- t_0: Task T_{i+1} is holding resource R and tasks $\mathcal{TB} = \{T_k | T_k \in \mathcal{T}_R, k \neq i \bigwedge k \neq i+1\}$ are blocked on R. $|\mathcal{TB}| = (m-2)$.
- t_1: Task T_{i+1} releases R. A task in $\mathcal{TB}$, say T_r is granted resource R. ERL's of remaining tasks in $\mathcal{TB}$ are increased by n.
- t_2: Task T_{i+1} requests R and is blocked on R.
- t_3: Task T_i requests R and is blocked R.

Now, at time t_3, the ERL of task T_i is lower than those of all other tasks in the blocked task queue, which includes $(m-2)$ tasks. Therefore, T_i will suffer a queue blocking time of $(m-2)$ critical sections. □

We can now revisit the example in Figure 12.1. In Figure 12.2, we show the behavior of tasks by using the dynamic resource level adjustment rules. Note that the numbers on each timeline of a task indicates the ERL of that task. In this case, $m = 4$. Therefore, task queue blocking times should be bounded by $m-2 = 2$ critical sections, which is consistent with Figure 12.2. Observe that task T_2 is queue blocked for exactly two critical sections (of T_3 and T_4, respectively). On the other hand, task T_3 suffers one critical section of queue blocking for its resource requests; task T_4 only incurs one critical section of queue blocking during its second resource request.

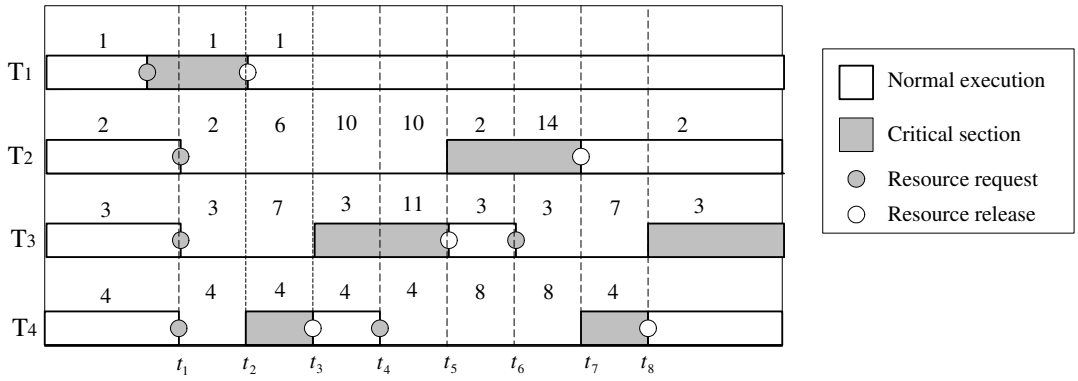


Figure 12.2: Dynamic Resource Levels

12.3.2 Queue Blocking Times under RLP

We consider a task T_b, along with a queue of k tasks, are blocked by a task T_a. This scenario is illustrated in Figure 12.3.

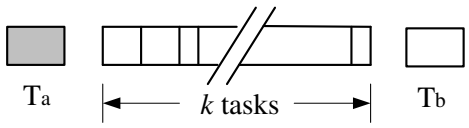


Figure 12.3: An Example of Queueing Blocking

To calculate the queue blocking time for T_b, we examine the blocking time due to each task in the $k-task$ queue. Observe that the qi^{th} task in the $k-task$ queue executes with a processor bandwidth of at least $\rho_{qi}^{min} + \sum_{j=i+1}^{k} \rho_{qj}^{min} + \rho_b^{min} = \sum_{j=i}^{k} \rho_{qj}^{min} + \rho_b^{min}$ due to bandwidth inheritance. Thus, the total queue blocking time resulting from the k tasks is

$$B_Q(k) = \sum_{i=1}^{k} \frac{d_{qi} + Q}{\sum_{j=i}^{k} \rho_{qi}^{min} + \rho_b^{min}} \tag{12.6}$$

Let $d_q = \max\{d_{qi} | i = 1, ..., m-2\}$ and $\rho_q^{min} = \min\{\rho_{qi}^{min} | i = 1, ..., m-2\}$. Then, $B_Q(k)$ is bounded by

$$B_Q^m(k) = \sum_{i=1}^{k} \frac{d_q + Q}{\sum_{j=i}^{k} \rho_q^{min} + \rho_b^{min}} = \sum_{i=1}^{k} \frac{d_q + Q}{(k-i+1)\rho_q^{min} + \rho_b^{min}} = \sum_{i=1}^{k} \frac{d_q + Q}{i\rho_q^{min} + \rho_b^{min}} \tag{12.7}$$

Now, we need to determine a k such that $B_Q^m(k)$ achieves its maximal value and therefor to bound task T_b's queue blocking time. We show that the maximal queue blocking time occurs when there are the maximal number of tasks in the queue, i.e., $k = (m-2)$.

Lemma 12.6. *The $B_Q^m(k)$ function defined in Equation 12.7 monotonically increases with k.*

Proof. We define two auxiliary functions $B_Q^-(k)$ and $B_Q^+(k)$. $B_Q^-(k)$ is the amount of blocking time that may be reduced if a $(k+1)^{th}$ blocked task is added into the existing $k-task$ queue. $B_Q^+(k)$ is the additional queue blocking time due to the $(k+1)^{th}$ blocked task. That is, $B_Q^-(k) = \sum_{i=1}^{k} \frac{d_q+Q}{i\rho_q^{\min}+\rho_b^{\min}} - \sum_{i=1}^{k} \frac{d_q+Q}{(i+1)\rho_q^{\min}+\rho_b^{\min}}$ and $B_Q^+(k) = \frac{d_q+Q}{\rho_q^{\min}+\rho_b^{\min}} = B_Q^+$.

By using $B_Q^-(k)$ and B_Q^+, one can derive the relationship between $B_Q^m(k+1)$ and $B_Q^m(k)$

$$B_Q^m(k+1) = B_Q^m(k) + B_Q^+ - B_Q^-(k) \tag{12.8}$$

Furthermore, observe that

$$\begin{aligned}
&B_Q^-(k)\,/(d_q+Q) = \sum_{i=1}^{k} \frac{1}{i\rho_q^{\min}+\rho_b^{\min}} - \sum_{i=1}^{k} \frac{1}{(i+1)\rho_q^{\min}+\rho_b^{\min}} = \sum_{i=1}^{k} \left(\frac{1}{i\rho_q^{\min}+\rho_b^{\min}} - \frac{1}{(i+1)\rho_q^{\min}+\rho_b^{\min}} \right) \\
&= \frac{1}{\rho_q^{\min}+\rho_b^{\min}} - \frac{1}{2\rho_q^{\min}+\rho_b^{\min}} + \frac{1}{2\rho_q^{\min}+\rho_b^{\min}} - \frac{1}{3\rho_q^{\min}+\rho_b^{\min}} + ... + \frac{1}{k\rho_q^{\min}+\rho_b^{\min}} - \frac{1}{(k+1)\rho_q^{\min}+\rho_b^{\min}} \\
&= \frac{1}{\rho_q^{\min}+\rho_b^{\min}} - \frac{1}{(k+1)\rho_q^{\min}+\rho_b^{\min}} \\
&= \frac{1}{\rho_q^{\min}+\rho_b^{\min}} \frac{k\rho_q^{\min}}{(k+1)\rho_q^{\min}+\rho_b^{\min}} \\
&= \frac{1}{\rho_q^{\min}+\rho_b^{\min}} \frac{k\rho_q^{\min}}{k\rho_q^{\min}+\rho_q^{\min}+\rho_b^{\min}} \\
&< \frac{1}{\rho_q^{\min}+\rho_b^{\min}} \\
&= B_Q^+/(d_q+Q)
\end{aligned}$$

Therefore, $B_Q^m(k+1) = B_Q^m(k) + B_Q^+ - B_Q^-(k) > B_Q^m(k)$. □

By Lemma 12.6, the queue blocking time a task T_i can suffer is

$$B_Q = \sum_{j=1}^{n_i} B_{Q_j}^m(m_j - 2) \tag{12.9}$$

where $B_{Q_j}^m(m_j-2)$ is the maximal queue blocking time for accessing resource $R_{i,j}$.

$$B_{Q_j}^m(m_j-2) = \sum_{l=1}^{m_j-2} (d_{qj}+Q) \Big/ \left(l\rho_{qj}^{min} + \rho_i^{min}\right) \tag{12.10}$$

Using a technique similar to that in Equation 12.5, the bandwidth requirement under RLP is

$$\begin{aligned}
&\sum_{k=0}^{\infty} p_i(k) \Pr[B_D + B_Q + C + Q \le \rho_i CT_i] \ge AP_i \\
&\Rightarrow \sum_{k=0}^{\infty} p_i(k) \Pr\left[k \sum_{j=1}^{n_i} \frac{d_{R_{i,j}}+Q}{\rho_{R_{i,j}}^{min}+\rho_i^{min}} + k \sum_{j=1}^{n_i} B_{Q_j}^m(m_j-2) + \sum_{j=1}^{k} c_{i,j} + Q \le \rho_i CT_i \right] \ge AP_i
\end{aligned} \tag{12.11}$$

12.4 The Early Blocking Protocol

The Early Blocking Protocol (EBP) is specifically designed for dealing with nested critical sections. Nested critical sections create two problems, namely potential deadlocks and transitive blocking.

One common approach for handling nested critical sections is to disallow them. If critical sections $z_{i,j} \subseteq z_{i,k}$, then disallowing nested critical sections requires that resources $R_{i,j}$ and $R_{i,k}$ are requested and held simultaneously over the complete duration of $z_{i,k}$. The advantages are easy implementation and deadlock-free. The disadvantage is reduced concurrency. If $z_{i,k}$ is considerably longer than $z_{i,j}$, concurrency will be significantly reduced. Thus, in general, disallowing nested critical sections assumes that critical sections are short and preemptions are rare. If that is the case, little can be gained by allowing nested critical sections. This assumption has been the basis for the Earliest Deadline First/Dynamic Deadline Modification (EDF/DDM) algorithm [29].

Our research is motivated by the fact that not all critical sections are short. For example, the time to access some physical devices, e.g., a valve, can be on the magnitude of milliseconds or even longer. In this section, we present the EBP protocol. We show that the EBP protocol avoids deadlock as well as bounding the duration of transitive blocking.

12.4.1 Protocol Definition

The basic idea is to block a "unsafe" resource request even if the requested resource is free. An "unsafe" resource request has the danger of causing deadlocks. Hence, the name "early blocking" reflects this design approach. In the meanwhile, a safe resource request is granted. Similar approach has been explored in the Priority Ceiling Protocol (PCP) [57] and Stack Resource Policy (SRP) [7].

Assume that a task T invokes $nest\_req\_resource(R', RV)$ for requesting to enter nested critical sections. In their order of access, RV is a list of resources that T may access while it is inside nested critical sections. We call RV a "resource vector." R' is the first element in RV.

For single-unit resources, as we assume in this work, the necessary and sufficient condition for deadlock is the existence of a cycle in the resource graph. A cycle can only be formed by at least two tasks inside nested critical sections. Furthermore, there is at least one resource R that is requested by one task T_i and is held by another task T_j, both of which are inside nested critical sections. In other words, the resource vectors of T_i and T_j overlap. Therefore, the EBP protocol compares the resource vector of a requesting task with those of the existing tasks. If any overlap of resource vectors exists, there is a possibility of deadlock. Thus, the requesting task is blocked.

Let T be the requesting task that invokes $nested\_req\_resource(R', RV)$. We formulate the EBP protocol as follows:

1. If R' is currently held by another task, T is blocked.
2. If R' is free, $nest\_req\_resource(R', RV)$ may or may not be granted:
   (a) Let $\mathcal{T}_{nest}$ be the set of tasks that are currently inside nested critical sections. For any task $T_i \in \mathcal{T}_{nest}$, let RV_i be T_i's current resource vector.
   (b) If for any $T_i \in \mathcal{T}_{nest}$, $RV \bigcap RV_i = \emptyset$, $nest\_req\_resource(R', RV)$ is granted; the request is blocked otherwise.
3. Whenever a task exits a nested critical section, the protocol is invoked to check if granting any pending $nest\_req\_resource(R', RV)$ is safe. If more than one pending $nest\_req\_resource(R', RV)$ become safe, the Resource Level Policy shall be invoked.

12.4.2 Properties and Transitive Blocking Times under EBP

Next, we establish that the EBP protocol is deadlock-free and can bound transitive blocking times. This is done through a set of lemmas and theorems.

Lemma 12.7. *Under EBP, for any pair of tasks that are currently inside nested critical sections, their resource vectors do not have comment elements.*

Proof. Suppose two tasks T_1 and T_2 enter nested critical section at instants $t_1 < t_2$, respectively. If $RV_1 \bigcap RV_2 \neq \emptyset$, T_2 cannot enter its nested critical sections. Therefore, the resource vectors of T_1 and T_2 do not have comment elements.

It seems that a race condition may arise if $t_1 = t_2$. However, because we only consider uni-processor systems, T_1 and T_2 cannot make resource requests at the same time, though simultaneous resource requests may occur on virtual processors. □

Theorem 12.8. *The EBP protocol avoids deadlock.*

Proof. We prove this theorem by contradiction. Suppose deadlock happens under EBP. Then, a cycle exists in the resource graph, as shown in Figure 12.4. Notice that *all* tasks on the cycle are inside nested critical sections. Furthermore, any pair of adjacent tasks on the cycle have a common element in their resource vectors, which contradicts Lemma 12.7. □

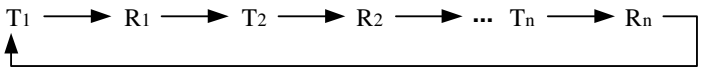


Figure 12.4: A Cycle in A Resource Graph

Corollary 12.9. *Under EBP, if a task T_1 is blocked by T_2 while T_1 is inside nested critical sections, then T_2 is not inside nested critical sections.*

Proof. Suppose T_2 is inside nested critical sections. If T_1 is blocked by T_2, then T_1 needs a resource R that is currently held by T_2. Thus, R is a common element in T_1 and T_2's resource vectors. This phenomena violates Lemma 12.7. □

Theorem 12.10. *Under EBP, a chain of transitive blocking includes three tasks.*

Proof. We use $T_i \rightarrow R_i$ to denote that task T_i needs resource R_i. Similarly, $R_i \rightarrow T_i$ means that resource R_i is currently held by task T_i. Thus, a chain of transitive blocking has the form of $T_1 \rightarrow R_1 \rightarrow T_2 \rightarrow R_2 \rightarrow T_3 \rightarrow ... \rightarrow T_n$. It is easy to see that any task $T_i, i \neq 1 \bigwedge i \neq n$ must be inside nested critical sections. By Corollary 12.9, if T_2 is inside nested critical sections, T_3 cannot be inside nested critical sections. Therefore, T_3 must be at the end of the chain. Thus, $n = 3$. □

Theorem 12.11. *Suppose a task T requests for resource R_i. Let $\mathcal{T}_{i,j}$ be the set of tasks that have a resource vector in the form of $RV = \{..., R_i, ..., R_j, ...\}$ and let $\mathcal{T}_j$ be the set of tasks that may access resource R_j. The transitive blocking time of T's request for R_i is bounded by $(d_{max} + Q) \Big/ (\rho^{min} + \rho^{min}_{R_{i,j}} + \rho^{min}_{R_j})$. ρ^{min} is the minimal bandwidth for task T, $d_{max} = \max\{d^j_k | T_k \in \mathcal{T}_j\}$, $\rho^{min}_{R_{i,j}} = \min\{\rho^{min}_k | T_k \in \mathcal{T}_{i,j}\}$, and $\rho^{min}_{R_j} = \min\{\rho^{min}_k | T_k \in \mathcal{T}_j\}$.*

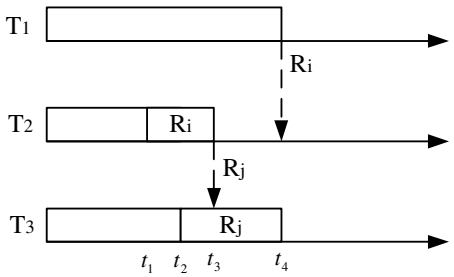


Figure 12.5: Illustration of Transitive Blocking

Proof. Consider a chain of transitive blocking as in Figure 12.5. Task T_1 is transitively blocked by task T_3 when it requests resource R_i. By Theorem 12.10, the scenario illustrated in Figure 12.5 is the only possible scenario.

Furthermore, task T_3 has a bandwidth of at least $\rho_1^{min} + \rho_2^{min} + \rho_3^{min}$ due to bandwidth inheritance. We further consider the worst case where the most pessimistic bounds are assumed. That is, $\rho_2^{min} = \rho_{R_{i,j}}^{min} = \min\{\rho_k^{min} | T_k \in \mathcal{T}_{i,j}\}$ and $\rho_3^{min} = \rho_{R_j}^{min} = \min\{\rho_k^{min} | T_k \in \mathcal{T}_j\}$. These pessimistic bounds lead to the theorem. □

12.5 Formal Comparison with The Lock-free Scheme

As discussed earlier, our conjecture is that for some cases, our lock-based, resource access protocols may work well. For other cases, the lock-free scheme, i.e., setting quantum size as the longest critical section in the system [5], may perform better. This section explores the conditions under which resource access protocols may be beneficial, and the reverse conditions as well.

The discussion focuses on two aspects: (1) bandwidth requirement for a given task; and (2) feasibility of a task set. Given a set of n tasks and their allocated bandwidth, if $\sum_{i=1}^{n} \rho_i \leq 1$, we say that the task set is feasible for the particular bandwidth allocation. Otherwise, the task set is said infeasible for the particular allocation.

We first introduce some notations:

- ρ_i^p: bandwidth requirement of task T_i under lock-based resource access protocols
- ρ_i^{np}: bandwidth requirement of task T_i under the lock-free scheme (also called *non-preemptive* scheme as there will be at most one preemption while a task tries to access a resource [5])
- Q_p: quantum size under the lock-based resource access protocols
- Q_{np}: quantum size under the lock-free scheme

Lemma 12.12. *Suppose Q_{np} equals to the length of a critical section of task T_m (accessing resource R_m). If a task T_i may be blocked on R_m, then $\rho_i^p > \rho_i^{np}$.*

Proof. Let $d_R = Q_{np}$ be the length of the critical section. If task T_i may be blocked on R, it suffers at least one direct blocking due to access to R. The direct blocking time is calculated as

$$B_D = k \sum_{j=1}^{n_i} \frac{d_{R_{i,j}} + Q_p}{\rho_{R_{i,j}}^{min} + \rho_i^{min}} \geq \frac{d_R + Q_p}{\rho_R^{min} + \rho_i^{min}} \geq d_R + Q_p > d_R \tag{12.12}$$

The total blocking time is $B = B_D + B_Q + B_T \geq B_D > d_R$. Given the total execution time of C during a specification time window, we have

$$B + C + Q_P > d_R + C + Q_p = Q_{np} + C + Q_p > Q_{np} + C \tag{12.13}$$

Recall that the fundamental bandwidth requirement under resource access protocols is

$$\sum_{k=0}^{\infty} p_i(k) \Pr\left[B_k + C_k + Q_p \leq \rho_i^p C T_i\right] \geq AP_i \tag{12.14}$$

and under the lock-free scheme is

$$\sum_{k=0}^{\infty} p_i(k) \Pr\left[C_k + Q_{np} \leq \rho_i^{np} C T_i\right] \geq AP_i \tag{12.15}$$

where C_k is the sum of k job execution times, B_k is the total blocking time of k jobs. Since $C_k + Q_{np} < B_k + C_k + Q_p, \forall k$, $\rho_i^{np} < \rho_i^p$. □

Lemma 12.13. *Suppose Q_{np} equals to the length of a critical section of task T_m (accessing resource R_m). If a task T_i may not be blocked on R_m, then ρ_i^p can be smaller than ρ_i^{np}.*

Proof. We prove this lemma by considering an extreme case where resource R_m is only accessed by task T_m and another task T_k. All other tasks in the system do not use any shared resources. For any task that does not use any shared resource, its blocking time is zero. Furthermore, Q_p can be smaller than Q_{np}. Therefore,

$$B + C + Q_p = C + Q_p < C + Q_{np} \tag{12.16}$$

If that is the case, ρ_i^p is smaller than ρ_i^{np}. □

Theorem 12.14. *If a task set is feasible under the lock-free scheme, it can be infeasible under resource access protocols, and vice versa.*

Proof. We prove this theorem by examples.

1. A task set is feasible under the lock-free scheme, but infeasible using resource access protocols.

   Suppose all tasks access a single resource R in a system. By Lemma 12.12, $\rho_i^{np} < \rho_i^p, \forall i = 1, ..., n$. Thus,

   $$\sum_{i=1}^{n} \rho_i^{np} < \sum_{i=1}^{n} \rho_i^p \tag{12.17}$$

   Also assume $\sum_{i=1}^{n} \rho_i^{np} = 1$ for this particular task set. Then, $\sum_{i=1}^{n} \rho_i^p > 1$, and hence the task set if infeasible under resource access protocols.

2. A task set is feasible under resource access protocols, but infeasible under the lock-free scheme.

Consider a system where only two tasks, T_1 and T_2 need to access a resource R. Other tasks do not need to access any shared resources. Let

$$\begin{aligned} U_p &= \sum_{i=1}^{n} \rho_i^p = (\rho_1^p + \rho_2^p) + \sum_{i=3}^{n} \rho_i^p \\ U_{np} &= \sum_{i=1}^{n} \rho_i^{np} = (\rho_1^{np} + \rho_2^{np}) + \sum_{i=3}^{n} \rho_i^{np} \end{aligned} \tag{12.18}$$

By Lemma 12.12, $\rho_i^{np} < \rho_i^p, i = 1, 2$. However, if $\rho_1^p + \rho_2^p$ is small enough, we have

$$\begin{aligned} U_p &\approx \sum_{i=3}^{n} \rho_i^p \\ U_{np} &\approx \sum_{i=3}^{n} \rho_i^{np} \end{aligned} \tag{12.19}$$

By Lemma 12.13, $\rho_i^p < \rho_i^{np}, i = 3, ..., n$. Therefore, $U_p < U_{np}$. If $U_p = 1$ for this particular task set, then the task set if infeasible under the lock-free scheme. □

Through the proofs of the above lemmas and theorem, we demonstrate that neither the lock-free scheme, nor the resource access protocols are *always* better than the other. Specifically, if only a small number of tasks share a few resources, then using resource access protocols is beneficial. If resources are shared by most of the tasks in the system, then the lock-free scheme is more suitable in terms of bandwidth requirement.

Another hybrid case is that tasks can be partitioned into logical groups. Tasks in each logic group closely interact with each other and share resources. In addition, resource sharing across group boundaries is rare. For example, in a networked computer, device drivers may share the protocol input/output queues with the network protocol stack. On the contrary, a word processor is very unlikely to access the protocol queues. For this hybrid case, if the critical sections in a logic group are considerably longer than those in other groups, resource access protocols may still help to reduce bandwidth requirement. If all critical sections are on the same magnitude, little can be gained by using resource access protocols. Resource access protocols may even adversely affect system performance, because smaller time quanta result in higher overhead.

Chapter 13

Achieving System Wide Performance Assurance

This chapter presents our solutions to the last basic assurance problems in Section 10.2.

1. Satisfying system requirement without considering task requirements; and
2. Satisfying both task requirements and system requirement.

13.1 Problem Statement

Let $u_{i,j}$ be the utility accrued by job $J_{i,j}$. The system wide, asymptotical accrued utility ratio is calculated as

$$AUR = \lim_{m \to \infty} \left[\sum_{i=1}^{n} \sum_{j=1}^{m} u_{i,j} \Big/ m \sum_{i=1}^{n} U_i^{\max} \right], \tag{13.1}$$

where U_i^{max} is the maximal possible utility accrued by task T_i.

Our goal is to maximize AUR. If the maximized AUR is greater than the user-specified system assurance requirement, then the requirement is met. Thus, the fundamental problem is to maximize system accrued utility ratio.

Since the system is random in nature, the sojourn time of a job $s_{i,j}$ and the corresponding utility $u_{i,j}$ are also drawn from random variables. Let u_i be the random variable for the accrued utility of task T_i. The

optimization criterion of AUR can be re-written as:

$$AUR = \sum_{i=1}^{n} \left[\lim_{m \to \infty} \left(\frac{1}{m} \sum_{j=1}^{m} u_{i,j} \right) \right] \Big/ \sum_{i=1}^{n} U_i^{\max} = \sum_{i=1}^{n} E\left(u_i\right) \Big/ \sum_{i=1}^{n} U_i^{\max} \tag{13.2}$$

Thus, the optimization problem essentially requires maximizing the sum of the expected utilities of all tasks.

The optimization problem may be subject to task-level assurance constraints. As in Chapter 11, our basic mechanism for satisfying task-level assurance is bandwidth allocation and control. Therefore, this constrained optimization problem can be solved by first allocating bandwidth to task-level requirements. The remaining bandwidth, if any, then is used to improve system wide accrued utility. In other words, the unconstrained and constrained optimization problems can be solved in a similar manner.

13.2 Solutions

We first establish the relationship between task-level assurance and system AUR. Recall that our bandwidth allocation is performed on a per task basis. By establishing the relationship, we can understand how bandwidth allocation affects system AUR.

Lemma 13.1. *If a task T_i has a non-negative TUF and can accrue utility AU_i with the probability of at least AP_i, then the mean of its accrued utility, i.e., $E(u_i)$ is at least $AU_i \times AP_i$.*

Proof. Let $p_i(u)$ be the probability density of task T_i's accrued utility. By the definition of $E(u_i)$ and the assumption of non-negative TUFs, we have:

$$\begin{aligned} E(u_i) &= \int_0^{+\infty} up_i(u)du = \int_0^{AU_i} up_i(u)du + \int_{AU_i}^{+\infty} up_i(u)du \\ &\geq \int_{AU_i}^{+\infty} up_i(u)du \geq AU_i \times \int_{AU_i}^{+\infty} p_i(u)du \end{aligned} \tag{13.3}$$

Since $\int_{AU_i}^{+\infty} p_i(u)du$ is the probability that u_i is greater than AU_i, we have $\int_{AU_i}^{+\infty} p_i(u)du \geq AP_i$. The lemma thus follows. □

Theorem 13.2. *The system wide accrued utility ratio (AUR) is lower bounded by $\frac{\sum_{i=1}^{n}(AU_i \times AP_i)}{\sum_{i=1}^{n} U_i^{max}}$.*

Proof. This theorem directly follows by combining $AUR = \sum_{i=1}^{n} E(u_i) \Big/ \sum_{i=1}^{n} U_i^{max}$ and Lemma 13.1. □

By Theorem 13.2, a reasonable approach to maximize system AUR is to maximize its lower bound, i.e., $\sum_{i=1}^{n}(AU_i \times AP_i)$. The term $AU_i \times AP_i$ represents task T_i's contribution to system AUR. For convenience, we define $y_i = AU_i \times AP_i$.

Recall that the general solution presented in Section 11.4.1 establishes a relationship between processor bandwidth ρ_i and the resulting CT_i and AP_i.

$$\rho_i = \frac{E(c_i)E(N_i)}{CT_i(1 - AP_i)} + \frac{Q}{CT_i}. \tag{13.4}$$

Combining Equation 13.4 with $AU_i = U_i(CT_i)$, we have

$$y_i(CT_i, \rho_i) = U_i(CT_i) \times \left(1 - \frac{E(c_i)E(N_i)}{\rho_i CT_i - Q}\right) \tag{13.5}$$

We then define $f_i(\rho_i) = \max_{CT_i \geq 0} y_i(CT_i, \rho_i)$. Given a ρ_i, $f_i(.)$ is the maximal value that task T_i can contribute to system AUR.

Lemma 13.3. *Function $f_i(\rho_i)$ monotonically increases with ρ_i.*

Proof. Observe that function $y_i(.)$ monotonically increases with ρ_i. Given a ρ_i, assume CT_i^m yields the maximal $y_i(.)$, i.e., $f_i(\rho_i) = y_i(CT_i^m, \rho_i) = \max_{CT_i \geq 0} y_i(CT_i, \rho_i)$. Then, $y_i(CT_i^m, \rho_i)$ is also a monotone of ρ_i. This lemma thus follows. □

Once the $f_i(.)$ function of all tasks are derived, the optimization problem becomes

$$\begin{aligned} &\text{Objective: } \max \sum_{i=1}^{n} f_i(\rho_i) \\ &\text{Subject to: } \sum_{i=1}^{n} \rho_i \leq 1 \end{aligned} \tag{13.6}$$

Though $f_i(.)$ has been shown to be an increasing function of ρ_i, it may not be continuous. This is particularly true if T_i's TUF is piece-wise continuous. Therefore, an analytic solution, such as Lagrange relaxation, may not be appropriate. We show a greedy search procedure in Algorithm 9.

Next, we consider the constrained optimization problem. Similar to the unconstrained optimization problem, the key is to determine the relationship between ρ_i and y_i. To ensure that task-level requirements are satisfied, we keep AP_i intact. Therefore, given a ρ_i, CT_i can be solved as

$$CT_i = \frac{1}{\rho_i}\left(\frac{E(c_i)E(N_i)}{1 - AP_i} + Q\right) \tag{13.7}$$

Then, y_i becomes a function of ρ_i. We use $g_i(\rho_i)$ to represent this function, i.e., $y_i = U_i(CT_i) \times AP_i = g_i(\rho_i)$. Assume the bandwidth required to satisfy task T_i's assurance requirement is ρ_i^0. An search algorithm similar to the unconstrained case is presented in Algorithm 10.

1: Initial bandwidth allocation: $\rho_i = 0, \forall i = 1, ..., n$;
2: $\rho = 1$;
3: **repeat**
4: | Allocate $\Delta\rho$ bandwidth to the task T_k that satisfies
| $f_k(\rho_k + \Delta\rho) - f_k(\rho_k) = \max\{f_i(\rho_i + \Delta\rho) - f_i(\rho_i) | i = 1, ..., n\}$;
5: | $\rho \leftarrow \rho - \Delta\rho$;
6: | $\rho_k \leftarrow \rho_k + \Delta\rho$;
7: **until** $\rho < \Delta\rho$;

Algorithm 9: Allocating Bandwidth to Maximize System AUR

1: Initial bandwidth allocation: $\rho_i = \rho_i^0, \forall i = 1, ..., n$;
2: $\rho = 1 - \sum_{i=1}^{n} \rho_i^0$;
3: **repeat**
4: | Allocate $\Delta\rho$ bandwidth to the task T_k that satisfies
| $g_k(\rho_k + \Delta\rho) - g_k(\rho_k) = \max\{g_i(\rho_i + \Delta\rho) - g_i(\rho_i) | i = 1, ..., n\}$;
5: | $\rho \leftarrow \rho - \Delta\rho$;
6: | $\rho_k \leftarrow \rho_k + \Delta\rho$;
7: **until** $\rho < \Delta\rho$;

Algorithm 10: Allocating Bandwidth to Satisfy Task-Level Requirements and to Maximize System AUR

Chapter 14

Conclusions and Contributions

Our simulation results have shown that the GBS algorithm has comparable performance with algorithms such as DASA, LBESA, and D^{over} for all the application scenarios where they apply. In addition, GBS can handle task sets with arbitrarily shaped TUFs and mutual exclusion resource constraints; none of the existing algorithms can schedule such a task set. Furthermore, we establish several fundamental properties of GBS.

Measurements from our implementation of GBS (using the meta-scheduler scheduling framework) on a real-time POSIX-compliant operating system also reveal the strong effectiveness of the algorithm.

It is important to note that the measured performance of GBS from the implementation is not an "absolute" measure. The measurements only demonstrate the effectiveness of GBS, despite its $O(n^3)$ worst-case complexity. In fact, the meta-scheduler implementation will incur additional overhead compared with a kernel implementation, as the scheduler itself is scheduled by the underlying kernel scheduler. This obviously can negatively impact the scheduler performance. Therefore, we conjecture that the performance of GBS will be better when implemented inside an operating system kernel.

GBS targets real-time systems that are subject to significant non-determinism that is inherent in their operating environments e.g., completely unknown activity arrivals. When system uncertainties can be stochastically characterized (e.g., stochastic activity arrivals and execution times), it is possible to provide stochastic assurances on timeliness behavior.

The dissertation also presents algorithmic solutions to fundamental assurance problems in TUF real-time systems, including, stochastically satisfying individual, activity utility lower bounds and system-wide, total utility lower bounds. The algorithmic solutions include algorithms for processor bandwidth allocation

and UA scheduling. While bandwidth allocation algorithms allocate processor bandwidth share to activities to probabilistically satisfy individual and collective, utility lower bounds, UA scheduling algorithms schedule activities to maximize accrued utility.

We extend our algorithmic solutions with a class of lock-based and lock-free resource access protocols to satisfy mutual exclusion constraints on shared resources. We prove that satisfying utility lower bounds with lock-based protocols does not imply doing so with the lock-free scheme, and vice versa. Further, we establish the conditions under which lock-based protocols are "stronger" than the lock-free scheme, and vice versa.

Finally, we present a rule-based framework that allows tradeoff and negotiation between utility lower bound satisfaction and utility maximization, and that between individual and collective utility lower bound satisfaction.

Thus, the contributions of the dissertation include:

1. The GBS scheduling algorithm that schedules activities subject to arbitrarily shaped TUF time constraints and mutual exclusion resource constraints, to maximize system-wide, total attained utility;

2. The meta-scheduler scheduling framework [40] that facilitates UA scheduling on COTS POSIX RTOSes; and

3. The class of algorithmic solutions that provide stochastic assurances on timeliness behavior on activities subject to TUF time constraints and mutual exclusion resource constraints, including, stochastically satisfied individual and collective utility lower bounds.

Bibliography

[1] L. Abeni and G. Buttazzo, "Integrating multimedia applications in hard real-time systems," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1998, pp. 3–13.

[2] L. Abeni, G. Lipari, and G. Buttazzo, "Constant bandwidth vs. proportional share resource allocation," in *Proceedings of IEEE International Conference on Multimedia Computing and Systems*, vol. 2, June 1999.

[3] S. A. Aldarmi and A. Burns, "Time-cognizant value functions for dynamic real-time scheduling," Department of Computer Science, The University of York, U.K., Tech. Rep., 1998, YCS-306.

[4] ——, "Dynamic value-density for scheduling real-time systems," in *Proc. of Euromicro Conference on Real-Time Systems*, June 1999, pp. 270–277.

[5] J. H. Anderson, R. Jain, and K. Jeffay, "Efficient object sharing in quantum-based real-time systems," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1998.

[6] J. H. Anderson, S. Ramamurthy, and K. Jeffay, "Real-time computing with lock-free shared objects," *ACM Transactions on Computer Systems*, vol. 15, no. 2, pp. 134–165, May 1997.

[7] T. P. Baker, "Stack-based scheduling of real-time processes," *Journal of Real-Time Systems*, vol. 3, no. 1, pp. 67–99, March 1991.

[8] S. Baruah, G. Koren, B. Mishra, A. Raghunathan, L. Rosier, and D. Shasha, "On-line scheduling in the presence of overload," in *Proc. IEEE Annual Symposium on Foundations of Computer Science*, October 1991, pp. 101–110.

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

[10] S. K. Baruah, L. E. Rosier, and R. R. Howell, "Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor," *Real-Time Systems*, vol. 2, no. 4, pp. 301–324, November 1990.

[11] G. Buttazzo, M. Spuri, and F. Sensini, "Value vs. deadline scheduling in overload conditions," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1995, pp. 90–95.

[12] G. Buttazzo and J. Stankovic, "RED: Robust earliest deadline scheduling," in *Proceedings of International Workshop on Responsive Computing Systems*, September 1993, pp. 100–111.

[13] ——, "Adding robustness in dynamic preemptive scheduling," in *Responsive Computer Systems: Steps Toward Fault-Tolerant Real-Time Systems*, D. Fussel and M. Malek, Eds. Boston: Kluwer Academic Publishers, October 1995, pp. 67–88.

[14] G. C. Buttazzo, *Hard Real-Time Computing Systems: Predictable Scheduling Algorithms and Applications*. Boston: Kluwer Academic Publishers, 1997.

[15] K. Chen and P. Muhlethaler, "A scheduling algorithm for tasks described by time value function," *Journal of Real-Time Systems*, vol. 10, no. 3, pp. 293–312, May 1996.

[16] R. Clark, E. D. Jensen, A. Kanevsky, J. Maurer, P. Wallace, T. Wheeler, Y. Zhang, D. Wells, T. Lawrence, and P. Hurley, "An adaptive, distributed airborne tracking system," in *Proceedings of IEEE International Workshop on Parallel and Distributed Real-Time Systems*, ser. Lecture Notes in Computer Science, vol. 1586. Springer-Verlag, April 1999, pp. 353–362.

[17] R. K. Clark, "Scheduling dependent real-time activities," Ph.D. dissertation, Carnegie Mellon University, 1990, CMU-CS-90-155.

[18] R. S. Engelschall, "The GNU portable threads," http://www.gnu.org/software/pth, 2003.

[19] G. V. Gens and E. V. Levner, "Fast approximation algorithm for job sequencing with deadlines," *Discrete Applied Mathematics*, vol. 3, pp. 313–318, 1981.

[20] GlobalSecurity.org, "Bmc3i battle management, command, control, communications and intelligence," http://www.globalsecurity.org.

[21] ——, "E-3 sentry (awacs)," http://www.globalsecurity.org.

[22] ——, "Multi-platform radar technology insertion program," http://www.globalsecurity.org.

[23] ——, "Multi-sensor command and control aircraft," http://www.globalsecurity.org.

[24] P. Goyal, H. M. Vin, and H. Cheng, "Start-time fair queueing: A scheduling algorithm for integrated services packet switching networks," in *Proceedings of ACM SIGCOMM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication*, Stanford, CA, 1996, pp. 157–168.

[25] J.-F. Hermant and G. L. Lann, "A protocol and correctness proofs for real-time high-performance broadcast networks," in *Proceedings of the The 18th International Conference on Distributed Computing Systems*, 1998, pp. 360–369.

[26] W. Horn, "Some simple scheduling algorithms," *Naval Research Logistics Quaterly*, vol. 21, pp. 177–185, 1974.

[27] IEEE and OpenGroup, "The open group base specifications issue 6 (IEEE Std. 1003.1)," http://www.opengroup.org/onlinepubs/007904975/nframe.html, 2003.

[28] R. Jain, *The Art of Computer Systems Performance Analysis*. New York: John Wiley & Sons, 1992.

[29] K. Jeffay, "Scheduling sporadic tasks with shared resources in hard-real-time systems," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1992, pp. 89–99.

[30] K. Jeffay and S. M. Goddard, "A theory of rate-based execution," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1999, pp. 304–314.

[31] K. Jeffay and D. L. Stone, "Accouting for interrupt handling costs in dynamic priotity task systems," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1993, pp. 212–221.

[32] E. D. Jensen, "Asynchronous decentralized real-time computer systems," in *Real-Time Computing*, W. A. Halang and A. D. Stoyenko, Eds. Springer Verlag, October 1992.

[33] E. D. Jensen, C. D. Locke, and H. Tokuda, "A time-driven scheduling model for real-time systems," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1985, pp. 112–122.

[34] E. D. Jensen, "A timeliness paradigm for mesosynchronous real-time systems," http://www.real-time.org/docs/omg%20ertws.pdf, July 2003.

[35] M. B. Jones, D. Roşu, and M.-C. Roşu, "CPU reservations and time constraints: Efficient, predictable scheduling of independent activities," in *Proceedings of ACM Symposium on Operating Systems Principles*, October 1997, pp. 198–211.

[36] G. Koren and D. Shasha, "D-over: An optimal on-line scheduling algorithm for overloaded real-time systems," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1992, pp. 290–299.

[37] E. L. Lawler and J. M. Moore, "A functional equation and its application to resource allocation and sequencing problems," *Management Sciences*, vol. 16, pp. 77–75, 1969.

[38] G. Le Lann, "Predictability in critical systems," in *Symposium on Formal Techniques in Real-Time and Fault-Tolerant Systems (Lecture Notes in Computer Science)*, A. Ravn and H. Rischel, Eds. Springer-Verlag, September 1998, vol. 1486, pp. 315–338.

[39] ——, "Proof-based system engineering and embedded systems," in *Lecture Notes in Computer Science*, G. Rozenberg and F. Vaandrager, Eds. Springer-Verlag, October 1998, vol. 1494, pp. 208–248.

[40] P. Li, B. Ravindran, S. Suhaib, and S. Feizabadi, "A formally verified application-level framework for real-time scheduling on POSIX real-time operating systems," *IEEE Transactions on Software Engineering*, vol. 30, no. 9, September 2004.

[41] P. Li, B. Ravindran, J. Wang, and G. Konowicz, "Choir: A real-time middleware architecture supporting benefit-based proactive resource allocation," in *Proc. of IEEE International Symposium on Object-oriented Real-time distributed Computing*, May 2003, pp. 292–299.

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

[43] J. W. S. Liu, W.-K. Shih, K.-J. Lin, R. Bettati, and J.-Y. Chung, "Imprecise computations," *Proceedings of the IEEE*, vol. 82, no. 1, pp. 83–94, January 1994.

[44] C. D. Locke, "Best-effort decision making for real-time scheduling," Ph.D. dissertation, Carnegie Mellon University, 1986, CMU-CS-86-134.

[45] C. Lu, J. Stakovic, G. Tao, and S. Son, "Feedback control real-time scheduling: Framework, modeling and algorithms," *Journal of Real-Time Systems*, vol. 23, no. 1/2, pp. 85–126, July 2002.

[46] D. P. Maynard, S. E. Shipman, R. K. Clark, J. D. Northcutt, R. B. Kegley, B. A. Zimmerman, and P. J. Keleher, "An example real-time command, control, and battle management application for alpha," Carnegie Mellon University Computer Science Deptartment, Tech. Rep., December 1988, Archons Project Technical Report 88121.

[47] T. E. Morton and D. W. Pentico, *Heuristic Scheduling Systems: With Applications to Production Systems and Project Management.* New York: Wiley, 1993.

[48] D. Mosse, M. E. Pollack, and Y. Ronen, "Value-density algorithm to handle transient overloads in scheduling," in *Proc. Euromicro Conference on Real-Time Systems*, June 1999, pp. 278–286.

[49] F. Mueller, "A library implementation of POSIX Threads under UNIX," in *Proceedings of the USENIX Conference*, January 1993, pp. 29–41, http://moss.csc.ncsu.edu/~mueller/pthreads.

[50] OMG, "Dynamic scheduling real-time corba 2.0, omg final adopted specification," OMG Document orbos/01-08-34, Tech. Rep., 2001.

[51] M. Pinedo, *Scheduling: Theory, Algorithms, and Systems*, 2nd ed. Upper Saddle River, New Jersey: Prentice-Hall, 2001.

[52] R. Rajkumar, K. Juvva, A. Molano, and S. Oikawa, "Resource kernels: A resource-centric approach to real-time systems," in *Proceedings of SPIE/ACM Conference on Multimedia Computing and Networking*, 1998.

[53] R. Rajkumar, C. Lee, J. Lehoczky, and D. Siewiorek, "A resource allocation model for QoS management," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1997, pp. 298–307.

[54] P. Ramanathan, "Overload management in real-time control applications using the (m, k) firm guarantee," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 6, pp. 549–559, June 1999.

[55] J. Regehr and J. A. Stankovic, "HLS: A framework for composing soft real-time schedulers," in *Proceedings of IEEE Real-Time Systems Symposium*, December 2001, pp. 3–14.

[56] S. K. Sahni, "Algorithms for scheduling independent tasks," *Journal of the ACM*, vol. 23, no. 1, pp. 116–127, January 1976.

[57] L. Sha, R. Rajkumar, and J. P. Lehoczky, "Priority inheritance protocols: An approach to real-time synchronization," *IEEE Transactions on Computers*, vol. 39, no. 9, pp. 1175–1185, September 1990.

[58] W. E. Smith, "Various optimizers for single stage production," *Naval Research Logistics Quarterly*, vol. 3, pp. 59–66, 1956.

[59] B. Srinivasan, S. Pather, R. Hill, F. Ansari, and D. Niehaus, "A firm real-time system implementation using commercial off-the-shelf hardware and free software," in *Proceedings of IEEE Real-Time Technology and Application Symposium*, June 1998, pp. 112–119.

[60] H. Stark and J. W. Woods, *Probability, Random Processes, and Estimation Theory for Engineers*, 2nd ed. Upper Saddle River, New Jersey: Prentice-Hall, 1994.

[61] I. Stoica, H. Abdel-Wahab, and K. Jeffay, "On the duality between resource reservation and proportional share resource allocation," in *Proceedings of Multimedia Computing and Networking*, 1997, pp. 207–214.

[62] I. Stoica, H. Abdel-Wahab, K. Jeffay, S. K. Baruah, J. E. Gehrke, and C. G. Plaxton, "A proportional share resource allocation algorithm for real-time, time-shared systems," in *Proceedings of IEEE Real-Time Systems Symposium*, December 1996, pp. 288–299.

[63] W. T. Strayer, "Function-driven scheduling: A general framework for expressing and analysis of scheduling," Ph.D. dissertation, University of Virginia, May 1992, Department of Computer Science.

[64] The Open Group Research Institute's Real-Time Group, *MK7.3a Release Notes*. Cambridge, Massachusetts: The Open Group Research Institute, October 1998.

[65] L. Thiele, S. Chakraborty, and M. Naedele, "Real-time calculus for scheduling hard real-time systems," in *IEEE International Symposium on Circuits and Systems*, March 2000, pp. 101–104.

[66] US Department of Defense, Command and Control Research Program, "Network centric warfare," http://www.dodccrp.org/research/ncw/ncw.htm.

[67] A. Varga, "Omnet++ discrete event simulation system," http://whale.hit.bme.hu/omnetpp.

[68] A. P. J. Vepsalanen and T. E. Morton, "Priority rules for job shops with weighted tardiness costs," *Management Sciences*, vol. 33, no. 8, pp. 1035–1047, 1987.

[69] C. A. Waldspurger, "Lottery and stride scheduling: Flexible proportional-share resource management," Ph.D. dissertation, Massachusettes Institute of Technology, September 1995, MIT/LCS/TR-667.

[70] C. A. Waldspurger and W. E. Weihl, "Lottery scheduling: Flexible proportional-share resource management," in *Proceedings of Symposium on Operating Systems Design and Implementation (OSDI)*, 1994, pp. 1–11.

[71] ——, "Stride scheduling: Deterministic propertional share resource manegement," MIT Laboratory for Computer Science, Tech. Rep. MIT/LCS/TM-528, June 1995.

[72] J. Wang and B. Ravindran, "Time-utility function-driven switched ethernet packet scheduling algorithm, implementation, and feasibility analysis," *IEEE Transactions on Parallel and Distributed Systems*, vol. 15, no. 1, pp. 1–15, January 2004.

[73] L. R. Welch, B. Ravindran, B. Shirazi, and C. Bruggeman, "Specification and modeling of dynamic, distributed real-time systems," in *Proceedings of The IEEE Real-Time Systems Symposium*, December 1998, pp. 72–81.

[74] H. Wu, B. Ravindran, E. D. Jensen, and U. Balli, "Utility accrual scheduling under arbitrary time/utility functions and multiunit resource constraints," in *IEEE Real-Time and Embedded Computing Systems and Applications*, August 2004, to appear. http://www.ee.vt.edu/~realtime/2004.html.

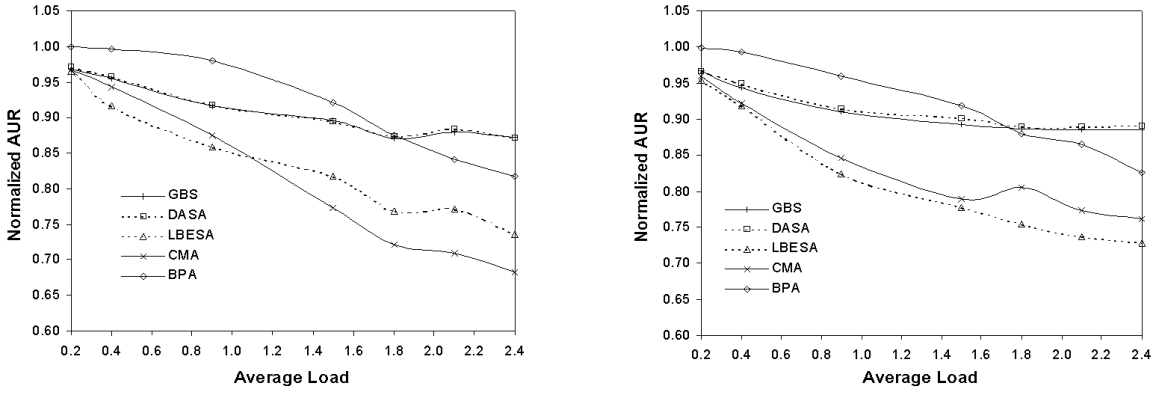
[75] H. Wu, B. Ravindran, E. D. Jensen, and P. Li, "Energy-efficient, utility accrual scheduling under resource constraints for mobile embedded systems," in *ACM International Conference on Embedded Software*, September 2004, to appear. http://www.ee.vt.edu/~realtime/2004.html.

[76] Y. Xia and D. Tse, "Survey of single machine scheduling with application to web object transmission," UC Berkeley, Tech. Rep. UCB/ERL M00/54, July 2000.

[77] V. Yodaiken, "The RTLinux approach to real-time," http://fsmlabs.com.

Appendix A

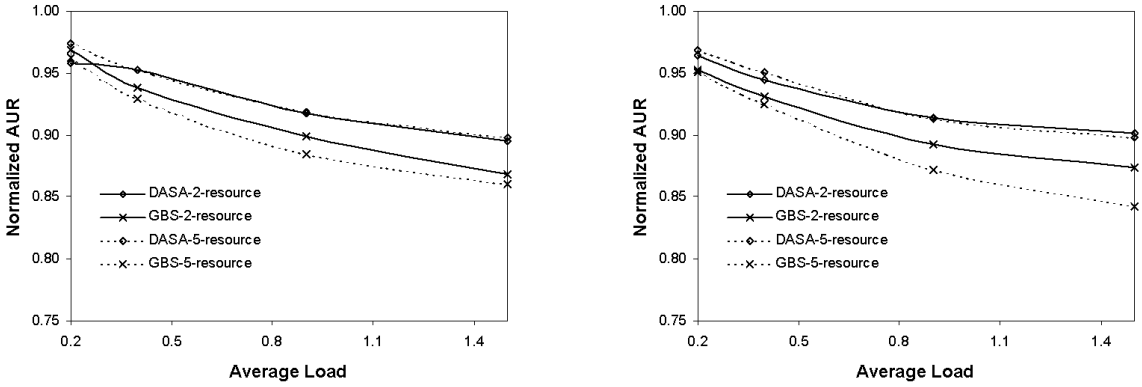
Additional Static Simulation Results for GBS



(a) Under Normal Distribution

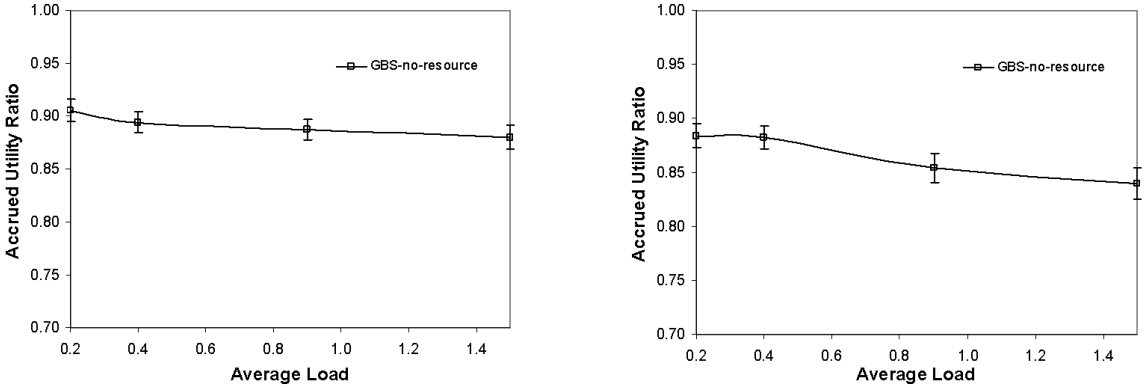
(b) Under Exponential Distribution

Figure A.1: Performance of Algorithms Under Rectangular TUFs and No Dependencies



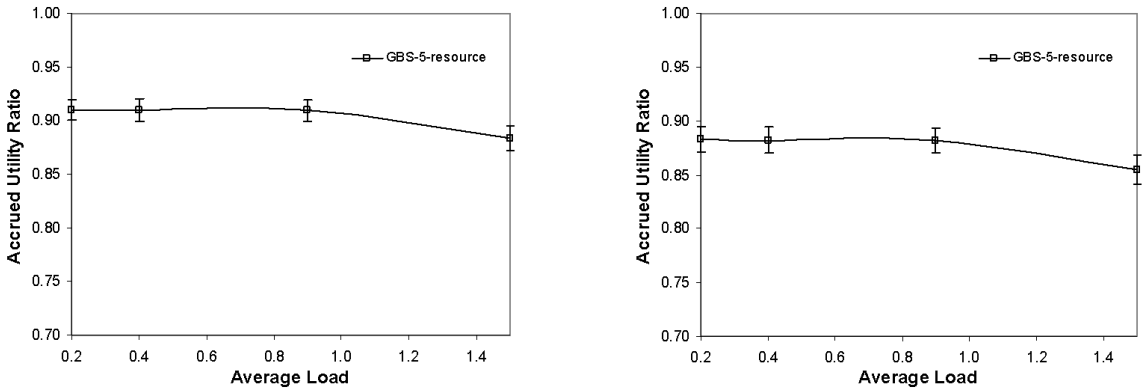
(a) Under Normal Distribution (b) Under Exponential Distribution

Figure A.2: Performance of Algorithms Under Rectangular TUFs and Dependencies



(a) Under Normal Distribution (b) Under Exponential Distribution

Figure A.3: Performance of GBS Under Arbitrary TUFs and No Dependencies



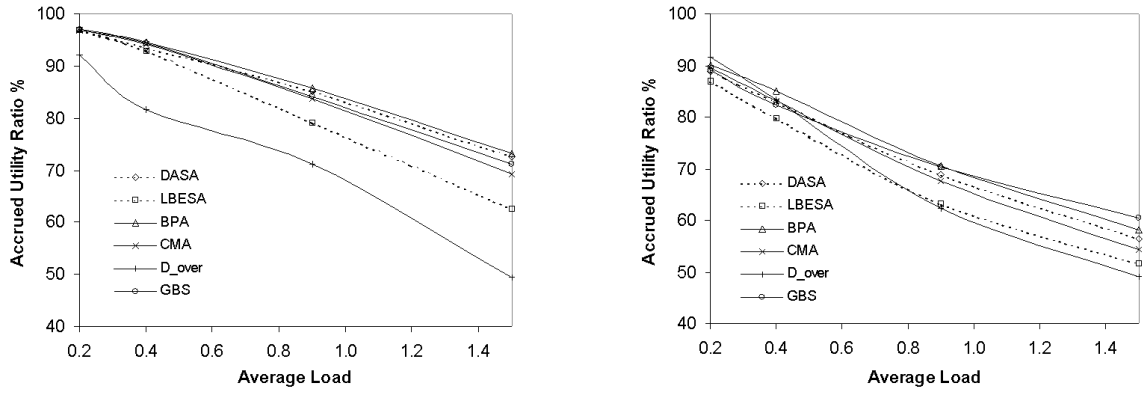
(a) Under Normal Distribution

(b) Under Exponential Distribution

Figure A.4: Performance of GBS Under Arbitrary TUFs and Dependencies

Appendix B

Additional Dynamic Simulation Results for GBS



(a) Under Normal Distribution

(b) Under Exponential Distribution

Figure B.1: Performance of Algorithms Under Rectangular TUFs and No Dependencies

Vita

Peng Li received his BS degree in Automatic Control (1997) and ME degree in Communication and Information Systems (2000), both from the University of Electronic Science and Technology of China (UESTC). His current research interests include real-time scheduling, distributed systems, embedded systems, operating systems, and real-time communication.

Mr. Li has been the recipient of "Excellent Student Award" during all years of his study at UESTC, from 1993 to 2000, for his distinguished academic performance. He is also a recipient of the Graduate Student Support Grants from the 9^{th} IEEE Real-Time and Embedded Technology and Applications Symposium, 2003.