Embedded Systems

Sanjoy Baruah
Marko Bertogna
Giorgio Buttazzo

Multiprocessor Scheduling for Real-Time Systems

Springer

Embedded Systems

Sanjoy Baruah • Marko Bertogna
Giorgio Buttazzo

Multiprocessor Scheduling for Real-Time Systems

Springer

Sanjoy Baruah
Computer Science Sitterson Hall
The University of North Carolina
Chapel Hill
North Carolina
USA

Giorgio Buttazzo
Scuola Superiore Sant'Anna TeCIP Institute
Pisa
Italy

Marko Bertogna
Università di Modena
Modena
Italy

ISSN 2193-0155 ISSN 2193-0163 (electronic)
Embedded Systems
ISBN 978-3-319-08695-8 ISBN 978-3-319-08696-5 (eBook)
DOI 10.1007/978-3-319-08696-5

Library of Congress Control Number: 2014954028

Springer International Publishing Cham Springer

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

To family: gods ("Deutas") and mums; Light and Illusion. And The One Who Left.—SkB

To Lidia, Luca and Diego.—MB

To the ones who like me.—GB

Preface

In the context of computer systems, scheduling theory is concerned with the efficient allocation of computational resources, which may be available in limited amounts, amongst competing demands in order to optimize specified objectives. Real-time scheduling theory deals with resource allocation in real-time computer systems, which are computer systems in which certain computations have a timing correctness requirement in addition to a functional one—the correct value must be computed, at the right time.

Computer systems, real-time and otherwise, are increasingly coming to be implemented upon multiprocessor and multicore platforms. The real-time systems research community has certainly recognized this reality, and has responded with enthusiasm and gusto: A large body of research has been performed addressing the various issues, challenges, and opportunities arising from this move towards multiprocessor platforms. A substantial fraction of this body of work is devoted to the scheduling-theoretic analysis of multiprocessor real-time systems.

We started out with the objective of distilling the most relevant ideas, techniques, methodologies, and results from this body of work into a foundational intellectual core for the discipline of multiprocessor real-time scheduling theory. However, the process of identifying such a core proved to be very challenging and trying. We soon discovered that we could not possibly provide comprehensive coverage of all the important ideas; indeed the sheer volume of excellent research that has been produced by the community meant that we could not even guarantee to at least mention every good idea that we came across. We have therefore instead chosen to select a self-contained collection of topics from the vast body of research literature on multiprocessor real-time scheduling theory, and to provide a cohesive, relatively deep, and complete coverage of this collection of topics. Our choice of topics was governed by (i) their relevance to the discipline; (ii) the depth of knowledge and understanding that has been acquired regarding them; and (iii) their cohesiveness—i.e., that all taken together they should comprise a complete narrative for a substantial and important subset of multiprocessor real-time scheduling theory. With these goals in mind, we chose to focus the coverage in the book upon the real-time scheduling of sporadic[1] task systems upon preemptive identical multiprocessor platforms, generally describing positive results (such as efficient algorithms for analysis) in preference to negative ones (lower bounds, impossibility proofs, etc.)

Organization

Broadly speaking, the chapters of this book are organized into the following four categories.

1. *Background material.* Chapters 1–3 provide background and context, and define many of the terms used during the remainder of the book. They additionally serve to demarcate the scope of this book, and explain the reasoning that guided our choice of this particular scope. Chapter 4 provides a brief review of some uniprocessor scheduling theory results that will be used during our study of multiprocessor scheduling.
2. *Liu and Layland systems.* The Liu and Layland task model (described in detail in Chap. 2) is a simple formal model for representing real-time workloads that is widely used, and has been very widely studied. Chapters 5–9 provide a fairly detailed coverage of a selection of topics dealing with the scheduling and schedulability analysis of Liu and Layland task systems.
3. *Three-parameter sporadic task systems.* This generalization to the Liu and Layland task model (also described in detail in Chap. 2) has been the focus of a lot of the research in multiprocessor real-time scheduling theory. A substantial portion of this book—Chaps. 10–20—is devoted to discussing the multiprocessor scheduling of real-time systems that are represented using the three-parameter sporadic task model.
4. *Emerging topics.* Many additional topics have come to be recognized as being extremely important and relevant to the development of a comprehensive theory of multiprocessor real-time scheduling. Some preliminary research has been conducted upon these topics and a few results obtained, but the state of our knowledge is not as deep or as comprehensive for the topics discussed in Chaps. 1–20. We briefly review what is known about a few such topics towards the end of the book. Chapter 21 discusses the sporadic DAG tasks model. Chapter 22 discusses real-time scheduling upon heterogeneous multiprocessor platforms.

For further information. As stated above, we have chosen to cover only a selection of the wide range of topics that have been studied in multiprocessor real-time scheduling theory. To get a more complete sense of the scope of the discipline, we recommend that the reader browse through the proceedings from recent years of conferences such as the IEEE Real-Time Systems Symposium (RTSS). and the EuroMicro Conference on Real-Time Systems (ECRTS).

[1] We will formally define all these terms—sporadic, preemptive, identical, and efficient—in Chap. 1.

Acknowledgements

Our choice of topics for this book, as well as the style of presentation of the selected topics, has been deeply influenced by our interactions with the many colleagues with whom we have performed collaborative research on multiprocessor real-time scheduling theory; these include Ted Baker, Alan Burns, Vincenzo Bonifaci, Alberto Marchetti-Spaccamela, Nathan Fisher, Shelby Funk, Joel Goossens, Jim Anderson, Sebastian Stiller, Leen Stougie, Martin Niemeier, Andreas Wiese, Greg Plaxton, Kirk Pruhs, Bjorn Andersson, Enrico Bini, Anand Srinivasan, Bjorn Brandenburg, Konstantinos Bletsas, Jeremy Erickson, Nan Guan, Bipasa Chattopadhyay, Peppe Lipari, John Carpenter, Jan Jonsson, Abhilash Thekkilakattil, Don Varvel, Neil Cohen, Johannes Gehrke, Phil Holman, Umamaheswari Devi, Insik Shin, Radu Dobrin, Sasikumar Punnekkat, Michele Cirinei, and Paolo Valente. We are additionally grateful to the many other researchers—too numerous to list here—with whom we have had useful discussions on various aspects of scheduling theory, particularly as it pertains to multiprocessor real-time systems. Two of the authors of this book have participated in Dagstuhl Seminars on Scheduling (Baruah in 2008 and 2010; Bertogna in 2010); the insights, interactions, and collaborations arising from this participation have proved invaluable to framing the perspectives that are reflected in this book.

Baruah's research on multiprocessor real-time scheduling has been funded by multiple grants from the US Government's National Science Foundation. Bertogna was supported by the P-SOCRATES EU FP7 project.

Chapel Hill, NC (USA) and Pisa (Italy)
August 2014

Sanjoy Baruah
Marko Bertogna
Giorgio Buttazzo

Contents

Chapter 1
Introduction: Background, Scope, and Context

Motivated by both vastly increased computational demand of real-time workloads and the trend in hardware toward multicore and multiprocessor CPUs, real-time systems are increasingly coming to be implemented upon multiprocessor platforms. Multiprocessor real-time scheduling theory, the subject of this book, is concerned with the development of techniques and methodologies that enable the correct and resource-efficient implementation of real-time systems upon multiprocessor platforms.

This chapter starts out with a brief overview in Sect. 1.1 of the main causes for the rise of multicore systems, explaining the consequences that such a transition has in terms of software development, performance, and timing analysis.

After presenting this background and context, the remainder of this chapter is devoted to delineating the scope of this book. The discipline of multiprocessor real-time scheduling theory has seen tremendous advances over the past couple of decades, and significant advances continue to occur at a very rapid pace. We cannot hope to cover all of multiprocessor real-time scheduling theory in this book; instead, we have chosen to focus upon a fundamental core of techniques and results concerning the *hard-real-time* (Sect. 1.4) scheduling of systems of *recurrent sporadic tasks* (Sect. 1.3) upon *identical* (Sect. 1.2) multiprocessor platforms. After briefly describing these terms, we will further discuss the organization and scope of this book in Sect. 1.5.

1.1 Background and Motivation

Since the beginning of this century the computer chip market has experienced an unprecedented phenomenon, referred to as the multicore revolution, that is pushing all major chip producers to switch from single core to multicore platforms. In particular, on May 17th, 2004, Intel, the world's largest chip maker, canceled the development of the Tejas processor, Intel's successor to the Pentium4-style Prescott processor, due to extremely high power consumption. After more than three decades profitably producing single core devices, the company decided to switch to multiprocessor chips, following the example of Advanced Micro Devices. The retirement of the Pentium brand was marked by the official release of the first wave of Intel Core Duo processors, on July 27, 2006.

1.1.1 Why the Move to Multicore?

To understand the reasons behind this choice, it is necessary to go back to the sixties, when Gordon Moore, Intel's founder, predicted that the density of transistors on a chip was going to double every 24 months. Since then, this law, referred to Moore's Law, has not been violated. The exponential increase in the number of transistors on a die was made possible by the progressive reduction of the integrating process, from the micrometer resolutions of past decades (with tens of thousands of transistors/chip in the 1980's), until the recent achievement of resolution below a hundred nanometers (with more than a hundred million transistor/chip). Today, hardware producers are selling devices realized with technologies down to 90 and 65 nm, and are exploring even lower resolutions.

The benefit of reducing dimensions lies not only in the higher number of gates that can fit on a chip, but also in the higher working frequency at which these devices can be operated. If the distance between the gates is reduced, signals have a shorter path to cover, and the time for a state transition decreases, allowing a higher clock speed. At the launch of Pentium 4, Intel expected single processor chips to scale up to 10 GHz using process technologies below 90 nm. However, they ultimately hit a frequency ceiling far below expectations, since the fastest retail Pentium 4 never exceeded 4 GHz. Why did that happen?

The main reason is related to power dissipation in CMOS integrated circuits, which is mainly due to two causes: dynamic power and static power.

Dynamic power has two components: a transient power consumption (P_{switch}), consumed when the device changes logic states (from 0 to 1 or vice versa), and a capacitive load power (P_{cap}), consumed to charge the load capacitance. It can be expressed as

$$P_{\text{dyn}} = P_{\text{switch}} + P_{\text{cap}} = (C + C_L)V_{dd}^2 \cdot f \cdot N^3 \tag{1.1}$$

where C is the internal capacitance of the chip, C_L is the load capacitance, f is the frequency of operation, and N is the number of bits that are switching. Note that dynamic power increases with the frequency of the chip and is closely tied to the number of state changes.

On the other hand, static power is due to a quantum phenomenon where mobile charge carriers (electrons or holes) tunnel through an insulating region, creating a leakage current independent of the switching activity. Static power consumption is always present if the circuit is powered on. As devices scaled down in size, gate oxide thicknesses decreased, resulting in larger and larger leakage currents due to tunneling.

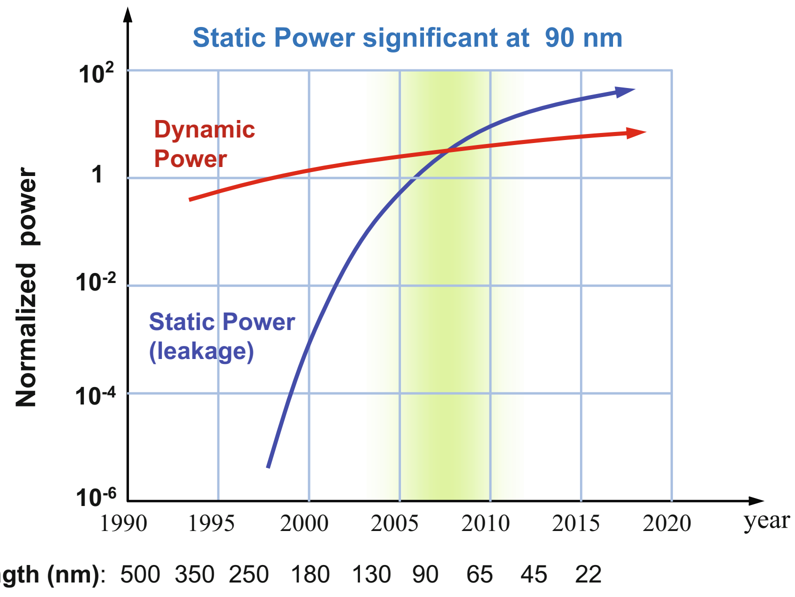


Fig. 1.1 Increase of static and dynamic power in CMOS circuits

Figure 1.1 shows how static and dynamic power components in CMOS circuits increased over the years as the gate length scaled down. Dynamic power dissipation was the dominant factor (about 90 % of total circuit dissipation) in digital CMOS circuits with gate length up to 180 nm [2]. After the year 2005, below the scale of 90 nm, static power became comparable with dynamic power, and today it is predominant.

A side effect of power consumption is heat, which, if not properly dissipated, can damage the chip. If processor performance would have improved by increasing the clock frequency, as suggested by Eq. 1.1, the chip temperature would have reached levels beyond the capability of current cooling systems.

The solution followed by all major hardware producers to keep the Moore's law alive exploited a higher number of slower logic gates, building parallel devices made with denser chips that work at lower clock speeds.

It is interesting to note that the number of transistors continued to increase according to Moore's Law, but the clock speed and the performance experienced a saturation effect.

In summary, since a further increase of the operating frequency of current computing devices would cause serious heating problems and considerable power consumption, chip makers moved to the development of computing platforms consisting of multiple cores operating at lower frequencies.

1.1.2 New Issues in Multicore Systems

The efficient exploitation of multicore platforms poses a number of new problems that only recently started to be addressed in the research community, thus a great amount of investigation is still needed in the future. When porting a real-time application from a single core to a multicore platform, the following key issues have to be addressed in order to better exploit the available computational capabilities:

1. How to split the executable code into parallel segments that can be executed simultaneously?
2. How to allocate such segments to the different cores?

Code parallelization can be done at different levels. Typically, at the instruction level some decisions can be taken automatically by the compiler, but at the high level, code splitting requires more specific information from the programmer. So another problem to be solved is:

How to express parallelism in the source code?

Parallel programming represents a full research topic that is out of the scope of this book. In this context, it suffices to mention two different approaches.

- Parallel programming languages. They are languages specifically designed to program algorithms and applications on parallel computers. There are plenty of languages, based on different paradigms (e.g., Ada, Java, CAL).
- Code annotation. This approach does not require a special programming language to address parallelism, since the information on parallel code segments and their dependencies is inserted in the source code of a sequential language by means of special constructs analyzed by a pre-compiler. An example of this approach is the OpenMP library [1], which adds annotations into C++ programs by means of the *pragma* construct.

Beside these issues, there are other important concerns that need to be taken into account when moving to a multicore system. To better understand these issues, we have to consider that the application software of complex embedded systems is typically structured as a set of concurrent tasks interacting through shared resources (mainly memory and I/O devices). In single core systems, such concurrent tasks are sequentially executed on the processor, although some interleaving (virtual parallelism) is permitted among them in order to be able to give greater priority to incoming tasks with higher importance. The sequential execution implies that the access to physical resources is implicitly serialized, so, for instance, two tasks can never cause a contention for a simultaneous memory access.

The situation is quite different when multiple tasks can run simultaneously on different cores sharing physical resources. Figure 1.2 illustrates the possible resource contentions that may occur in a dual core processor when multiple tasks share the main memory [161]. Once a processing element gains access to the main memory, any other device requesting access is blocked, and hence experiences a delay. As a consequence, resource contention introduces additional blocking times during task execution, limiting the available parallelism offered by the hardware.

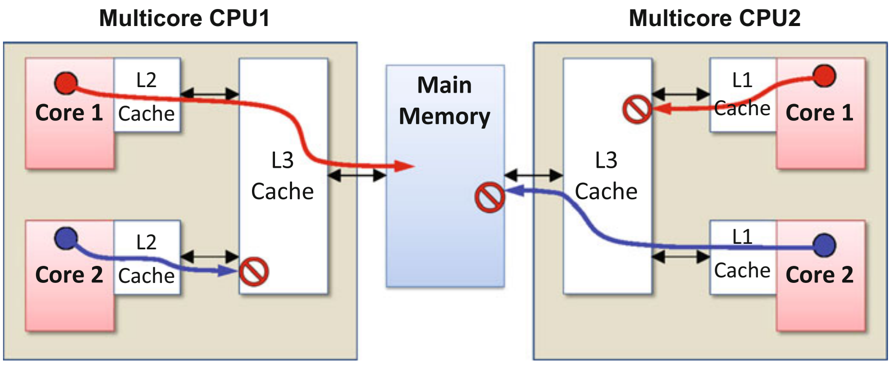


Fig. 1.2 Sources of interference in a multicore platform

Note that the contention on physical resources not only increases the response time of a task by inserting extra blocking delays, but also affects its worst-case execution time (WCET)! An experimental study carried out at Lockheed Martin on an 8-core platform shows that the WCET of a task can increase up to 6 times when the same code is executed on the same platform when an increasing number of cores are active [163].

For the same reasons, the performance of an application can vary significantly depending on how tasks are allocated and scheduled on the various cores. Therefore, the following problems have to be also addressed when dealing with multicore systems:

- How to allocate and schedule concurrent tasks in a multicore platform?
- How to analyze real-time applications to guarantee timing constraints, taking into account communication delays and interference?
- How to optimize resources (e.g., minimizing the number of active cores under a set of constraints)?
- How to reduce interference?
- How to simplify software portability?

Assuming we are able to express the parallel structure of our source code, the next questions are:

- How much performance can we gain by switching from 1 core to m cores?
- How can we measure the performance improvement?

1.1.3 Performance Improvement

The performance improvement that can be achieved on a multicore platform is strictly related to how the application code is parallelized and can be measured by the so called *speedup factor*.

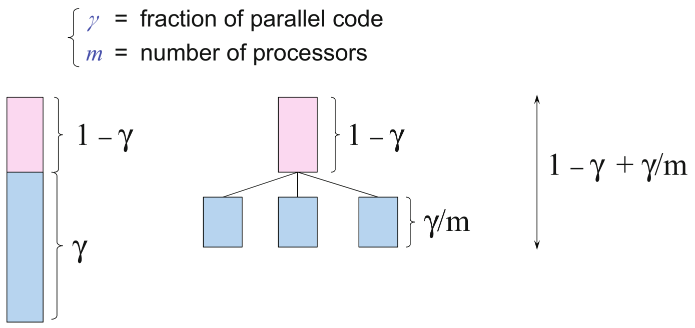


Fig. 1.3 Completion time of a program on a single core and a set of m cores, assuming no interference

Definition 1.1 The *Speedup factor* is a metric that measures the relative performance improvement achieved when executing a task on a new computing platform, with respect to an old one. If R_{old} and R_{new} are the task response times when executing on the old and new platform, respectively, the Speedup factor S is defined as

$$S = \frac{R_{old}}{R_{new}}. \tag{1.2}$$

If the old architecture is a single core platform and the new architecture is a platform with m cores (each having the same speed as the single core one), the *Speedup factor* can be expressed as a function of m, as follows:

$$S(m) = \frac{\text{completion time on 1 processor}}{\text{completion time on } m \text{ processors}} \tag{1.3}$$

To express the speedup factor, consider a program of length L and assume that only a fraction γ of the program can be executed in parallel on m cores, as shown in Fig. 1.3. If s is the speed of each core, it follows that the response time on a single core is $R_1 = L/s$, while the response time on the m-core platform is (ideally) $R_m = L[(1-\gamma) + \gamma/m]/s$.

Hence, the speedup factor results to be not only a function of the number of cores, but also a function of the percentage of parallel code:

$$S(m,\gamma) = \frac{R_1}{R_m} = \frac{1}{1-\gamma+\gamma/m}. \tag{1.4}$$

This formula is known as Amdahl's Law. To understand the implications of this result, consider a program in which only 50 percent of the code can be parallelized ($\gamma = 0.5$). When running this program on a platform with 100 cores ($m = 100$), the theoretical speedup with respect to a single core given by Eq. 1.4 is only $S = 2$. Figure 1.4 shows how the speedup factor varies as a function of γ in a platform with m cores.

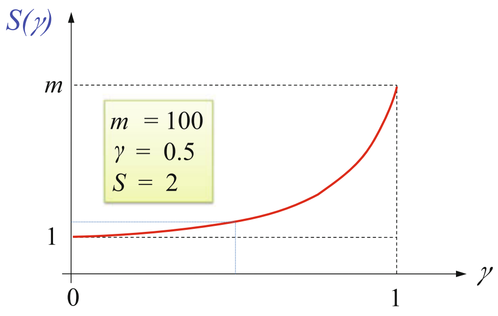


Fig. 1.4 Speedup factor for a platform with m cores as a function of the percentage γ of parallel code

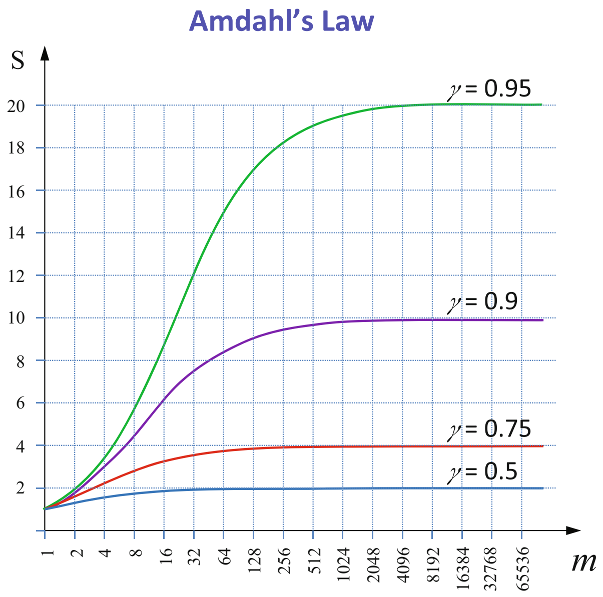


Fig. 1.5 Speedup factor for a platform with m cores as a function of the percentage γ of parallel code

Figure 1.5 shows how the speedup factor varies as a function of m for different values of γ. It is worth observing that every time a processor is added to the platform, the performance gain gets lower. For a given value of γ, the speedup factor tends to saturate to a limit for a large number of cores. This limit is equal to:

$$S_{m\to\infty}(\gamma) = \frac{1}{1-\gamma}. \tag{1.5}$$

For instance, with a program having 95 % of parallel code, the speed up factor cannot be higher than 20, no matter how many cores are used.

The result stated in Eq. 1.5 poses a strong limitation on the performance improvement achievable in a multicore platform. In fact, even assuming that the majority of application software can be parallelized, there will always be portions of code that must be executed sequentially (e.g., I/O operations on specific devices, distribution of input data, collection of output results), hence the factor γ can never equal 1.0. Moreover, considering communications costs, memory and bus conflicts, and I/O bounds, the situation gets worse.

In summary, parallel computing can be effectively exploited for limited numbers of processors and highly parallel applications (high values of γ). On the contrary, applications characterized by intensive I/O operations and including tasks that frequently exchange data and contend for shared resources, or include series of pipeline dependent calculations, are not suited for running on a multicore platform.

1.2 Multiprocessor Platforms

"Traditional" scheduling theory (of the kind studied in the Operations Research community and covered in, for example, the excellent text-books by Baker and Trietsch [16] and Pinedo [155]) has come up with the following classification of multiprocessor platforms:

Identical: These are multiprocessor platforms in which all the processors are identical, in the sense that each processor in the platform has the same computing capabilities as every other processor.

Uniform: By contrast, each processor in a uniform (or *related*) multiprocessor platform is characterized by its own computing capacity, with the interpretation that a job that executes on a processor of computing capacity s for t time units completes $s \times t$ units of execution.

Unrelated: In an unrelated multiprocessor, a different execution rate may be specified for each job upon each processor, with the interpretation that if a job has an execution rate r specified for a processor than executing it on that processor for t time units completes $r \times t$ units of execution.

(Observe that identical multiprocessors are a special case of uniform multiprocessors, and uniform multiprocessors are a special case of unrelated multiprocessors.)

Much of multiprocessor real-time scheduling theory research thus far has focused upon identical multiprocessor platforms, and so will much of this book. (We will see a use of the uniform multiprocessor platform model in Sect. 8.1, where it is used as an abstraction to facilitate the derivation of certain properties regarding scheduling on identical multiprocessors).

However, there is an increasing trend in industry toward heterogeneous multicore CPUs containing specialized processing elements such as digital signal processing cores (DSPs), graphics processing units (GPUs), etc., in addition to general-purpose processing cores. Since such heterogeneous multicore CPUs are better modeled using the unrelated multiprocessors model, there is an increasing recognition in the real-time scheduling community that the unrelated multiprocessors model needs further study, and we are beginning to see some work being done on developing a scheduling-theoretic framework for the analysis of real-time systems implemented upon unrelated multiprocessors. The current state of knowledge is rather primitive here when compared to what we know regarding identical multiprocessor; we will briefly describe some of this knowledge in Chap. 22.

1.3 Sporadic Tasks

One of the main differences between real-time scheduling theory and "traditional" scheduling theory (of the kind studied in the Operations Research community) is that in real-time scheduling theory, the workload is typically characterized as being generated by a finite collection of *recurrent* tasks or processes. Each such recurrent task may model a piece of code that is embedded within an infinite loop, or that is triggered by the occurrence of some external event. Each execution of the piece of code is referred to as a *job*; a task therefore generates a potentially unbounded sequence of jobs.

A recurrent task is said to be *periodic* if successive jobs of the task are required to be generated a constant duration apart, and *sporadic* if a lower bound, but no upper bound, is specified between the generation of successive jobs. For reasons that are discussed in Sect. 2.3.1, we will, for the most part, restrict our attention to sporadic task systems in this book.

Various models have been proposed for representing such sporadic tasks; some of the more widely-used models include the *Liu and Layland model* [137], and the *three-parameter model* [132, 133, 147].

In recent years, the real-time systems community has increasingly come to recognize that the kinds of workloads encountered in actual real-time systems are characterized not merely by the fact that they are typically generated by recurrent tasks, but that they may additionally possess *intra-task parallelism.* Hence, directed acyclic graph (DAG) based models have been proposed [38, 63, 135, 166] for representing the recurrent processes that comprise real-time systems.

Much of this book is devoted to the study of sporadic task systems represented using the Liu and Layland and the three-parameter models. We decided to focus primarily upon these models because this is where most progress has been reported with regards to research on multiprocessor real-time scheduling. [1] There has recently been some progress with regards to the scheduling of systems represented using the DAG-based models; we will briefly discuss some of this work in Chap. 21.

[1] It is (probably apocryphally) claimed that the American bank robber Willie Sutton replied "because that's where the money is," when asked by a reporter, upon his arrest, why he had chosen to rob banks.

1.4 Hard Real-time

Real-time computer systems are required to satisfy a dual notion of correctness: not only must the correct value be determined, this value must be determined at the correct time. In *hard* real-time systems, certain pieces of computation have *deadlines* associated with them, and it is imperative for the correctness of the system that all such pieces of computation complete by their deadlines. (In contrast, *soft* and *firm* real-time systems may allow for an occasional deadline to be missed, or for a deadline to be missed by no more than a certain amount, etc.) This book focuses almost exclusively upon the scheduling of hard real-time systems; the large body of excellent research results on soft- and firm-real-time systems that has been generated over the past decade or so falls outside the scope of this book.

1.5 Scope of This Book

As stated above, this book is primarily focused upon the multiprocessor scheduling of *hard-real-time* systems of *recurrent sporadic tasks* upon *identical* multiprocessor platforms. This is still an enormous topic, and we cannot hope to do justice to all of it. We have therefore chosen to further restrict the scope of coverage, as follows.

Preemptive Scheduling We will, for the most part, consider preemptive scheduling—we will assume that an executing job may be preempted at any instant in time, and its execution resumed later.[2] There has indeed been some interesting work on non-preemptive and limited-preemptive scheduling upon multiprocessors (see, e.g., [52, 74, 101, 122, 146, 169]); however, this body of results is dwarfed by the amount of results concerning preemptive scheduling, and aside from occasional references as appropriate, we will not discuss them further in this book.

Runtime Efficiency Considerations Along with the many useful scheduling and schedulability analysis algorithms that we will cover in this book, the real-time scheduling theory community has generated a vast body of results showing the *in*tractability of several basic multiprocessor scheduling problems. We will, for the most part, not detail the derivation of such results (except to occasionally point them out, perhaps to explain why we are not presenting a solution to some problem).

To be more specific, we will for the most part only describe schedulability analysis algorithms that have a worst-case run-time computational complexity that is *polynomial* or *pseudo-polynomial* in the representation of a problem specification.

[2] While we will not assign a penalty to such preemption, we are cognizant of the fact that preemptions may incur a cost in an implemented system. Hence, we will point out if some scheduling strategies we describe may result in an inordinately large number of preemptions; for others, we indicate upper bounds, if such bounds are known, on the number of preemptions that may occur in a schedule generated by that strategy—see, e.g., Sect. 2.3.3.

Note that the acceptance of pseudo-polynomial run-time implies that we do not necessarily have to consider problems that are merely shown to be NP-hard as intractable, since such problems may have psdeuo-polynomial time solutions. To show that a problem is unlikely to have a pseudo-polynomial time solution, it must be shown that the problem is NP-hard *in the strong sense* (see, e.g., Theorem 4.1 in Sect. 4.4). And all is not necessarily lost even for problems that are shown to be NP-hard in the strong sense: there remains the possibility of polynomial or pseudo-polynomial time algorithms for solving these problems *approximately* — see, e.g., the polynomial-time approximation algorithms in Sect. 6.3 for solving the strongly NP-hard problem of partitioning Liu & Layland task systems.

One consequence of this decision to only look for polynomial or pseudo-polynomial run-time algorithms is that we do not generally exploit the large body of research that exists in the traditional (i.e., Operations Research) scheduling community, on solving scheduling problems based on integer linear programming (ILP). This is because solving ILPs is known to be NP-hard in the strong sense. (We will however see some use of ILPs in Chap. 22, where polynomial-time techniques for solving ILPs *approximately* are used to obtain approximate solutions to some scheduling problems.)

Communication Costs We will discuss both partitioned scheduling (in which individual tasks are restricted to executing only upon a single processor) and global scheduling (in which a task may execute upon different processors at different points in time. However, when considering global scheduling we ignore the cost and delay in communicating between processors, and make the simplifying assumption that interprocessor communication incurs no cost or delay. This is a significant shortcoming of much of the existing body of research into multiprocessor scheduling theory today; although this shortcoming is starting to be addressed in extended platform models that consider, for example, routing issues for networks-on-chip (see, e.g., [65, 108]), this work is not really mature enough for us to be able to cover with any degree of authority.

1.6 Organization of This Book

Chapters 1–4 can be thought of constituting the introductory part of the book. Chapter 2 describes the workload and platform models we will by considering for much of this book, and seeks to justify this choice of models. Chapter 3 discusses some of the considerations that go into determining the quality and effectiveness of strategies for multiprocessor real-time scheduling. Chapter 4 briefly reviews some important concepts and results from uniprocessor real-time scheduling theory.

Chapters 5–9 discuss the multiprocessor real-time scheduling of real-time task systems that are modeled using the particularly simple (yet widely used) Liu & Layland task model. Chapter 5 discusses the model, and introduces two metrics—utilization bound and speedup factor—that are often used to make quantitative comparisons between different scheduling algorithms. Chapter 6 discusses the partitioned[3] scheduling of Liu & Layland task systems; Chapters 7–9 each deal with global scheduling when different restrictions are placed upon the form of the scheduling algorithms that may be used (these restrictions are discussed in Sect. 3.2). Chapter 7 describes an approach to multiprocessor scheduling that is called *pfair* scheduling, Chap. 8 discusses approaches that are beaded upon the well-known earliest deadline first algorithm, and Chap. 9 discusses fixed-task priority algorithms.

Chapters 10–20 discuss the multiprocessor real-time scheduling of real-time task systems that are modeled using the three-parameter sporadic tasks model. This is the model that has been the most thoroughly studied in multiprocessor real-time scheduling theory, and so there is a lot to discuss here. We start out in Chap. 10 with a description of the model, and define some characterizing attributes and properties. We then describe algorithms and analyses for partitioned scheduling in Chap. 11. Chapters 12–20 are devoted to global scheduling; a detailed discussion of the contents of these chapters is postponed to Chap. 10.

Chapters 21–22 provide brief summaries of known results on selected topics of emerging importance to multiprocessor real-time scheduling theory. Chapter 21 discusses the multiprocessor scheduling of real time systems represented using the sporadic DAG tasks model, which is more general than the three-parameter model. Chapter 22 discusses real-time scheduling upon unrelated multiprocessor platforms—these platforms are more general than the identical multiprocessor platforms that are the focus of the remaining chapters of the book.

[3] The terms *partitioned* and *global* are formally defined in Sect. 2.2.1.

Chapter 2
Preliminaries: Workload and Platform Models

Multiprocessor real-time scheduling theory studies the scheduling of real-time workloads upon multiprocessor platforms. This chapter describes some of the models that are currently widely used for representing such workloads and platforms, and provides some explanation and justification for the use of these particular models.

2.1 Workload Models

In choosing a model to represent a hard-real-time computer application system, the application system's designers are faced with two—often contradictory—concerns. On the one hand, they would like the model to be *general*, in order that it may accurately reflect the relevant characteristics of the application system being modeled. On the other hand, it is necessary that the model be *efficiently analyzable*[1], if it is to be helpful in system design and analysis.

Over the years, attempts to balance these two different goals have resulted in various models being proposed for representing real-time workloads. While these models differ considerably from each other in their expressive power and in the computational complexity of their associated analysis problems, most of them share some common characteristics.

The workload is modeled as being comprised of basic units of work known as *jobs*. Each job is characterized by three parameters: an *arrival time* or *release date*; a *worst-case execution time (WCET)*; and a *deadline*, with the interpretation that it may need to execute for an amount up to its WCET within a *scheduling window* that spans the time interval within its release date and its deadline (see Fig. 2.1). Each job is assumed to represent a single thread of computation; hence, a job may execute upon at most one processor at any instant.

Much of the real-time scheduling theory deals with systems in which the jobs are generated by a finite collection of independent recurrent tasks.

[1] Recall our discussion in Sect. 1.5: we seek models for which the analysis problems of interest can be solved in polynomial or pseudo-polynomial time.

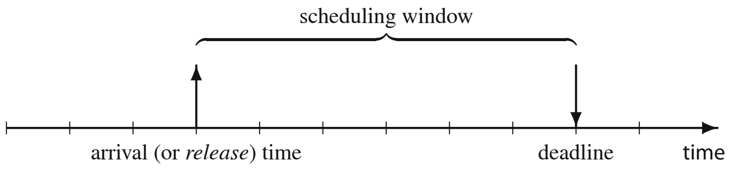


Fig. 2.1 Jobs: terminology. A job is characterized by an arrival (or release) time, a worst-case execution time, and a deadline. The time-interval between its release time and its deadline is called its scheduling window. It needs to execute for an amount up to its worst-case execution time within its scheduling window

The different tasks are *independent*[2] in the sense that the parameters of the jobs generated by each task are completely independent of the other tasks in the system. Different models for hard-real-time tasks place different constraints upon the permissible values for the parameters of the jobs generated by each task. We focus here on what are known as *sporadic* task models. In such models, lower bounds are specified between arrivals of successive jobs of the same task; by contrast, *periodic* and some other models specify exact separations, or upper bounds on these separations. In this book, we will look at different sporadic task models. The *three-parameter model* is the most widely-studied one; it is described in Sect. 2.1.2. A *directed acyclic graph (DAG)-based model* has recently come in for some attention in the context of multiprocessor systems, because it is better able to model parallelism within workloads that may be exploited upon multiprocessor platforms; we introduce this model in Sect. 2.1.3, but a more detailed discussion is postponed to Chap. 21. We start out in Chaps. 5–9 with an exploration of a restricted version of the three-parameter model that is commonly referred to as the *Liu and Layland (LL)* model. Much of the rest of this book is focused upon the 3-parameter model; we provide a summary of results concerning the DAG-based model in Chap. 21.

2.1.1 The Liu and Layland (LL) Task Model

This task model was formally defined in a paper [139] coauthored by C. L. Liu and J. Layland; hence the name. In this model, a task is characterized by two parameters—a *worst-case execution requirement* (WCET) C_i and a *period* (or *inter-arrival separation parameter*) T_i. A Liu and Layland task denoted τ_i is thus represented by an ordered pair of parameters: $\tau_i = (C_i, T_i)$. Such a task generates a potentially infinite sequence of jobs. The first job may arrive at any instant, and the arrival times of any two successive jobs are at least T_i time units apart. Each job has a WCET of C_i, and a deadline that is T_i time units after its arrival time. A Liu and Layland *task system* consists of a finite number of such Liu and Layland tasks executing upon a shared platform.

[2] This independence assumption represents a tradeoff between expressiveness and tractability of analysis; this trade-off is discussed in Sect. 2.3.

2.1.2 The Three-Parameter Sporadic Tasks Model

This model was proposed as a generalization to the Liu and Layland task model. As indicated by the name of the model, each task in this model [149] is characterized by three parameters: a *relative deadline* D_i in addition to the two parameters—WCET C_i and period T_i—that characterize Liu and Layland tasks. A 3-parameter sporadic task denoted by τ_i is thus represented by a 3-tuple of parameters: $\tau_i = (C_i, D_i, T_i)$. Such a task generates a potentially infinite sequence of jobs. The first job may arrive at any instant, and the arrival times of any two successive jobs are at least T_i time units apart. Each job has a WCET of C_i, and a deadline that occurs D_i time units after its arrival time. A 3-parameter sporadic *task system* consists of a finite number of such 3-parameter sporadic tasks executing upon a shared platform. A task system is often denoted as τ, and described by enumerating the tasks in it: $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$.

By allowing for the specification of a relative deadline parameter in addition to a period, the 3-parameter sporadic tasks model offers a means of specifying recurrent workloads that may occur infrequently (large period), but are urgent (have a small deadline). This is the model for recurrent tasks that has been most widely studied by the real-time scheduling theory community; for the most part in this book when we talk of "sporadic tasks" we mean tasks represented in this model.

Depending upon the relationship between the values of the relative deadline and period parameters of the tasks in it, a 3-parameter sporadic task system may further be classified as follows:

- In an *implicit-deadline* task system, the relative deadline of each task is equal to the task's period: $D_i = T_i$ for all $\tau_i \in \tau$. (Implicit-deadline task systems are exactly the same as Liu and Layland task systems).
- In a *constrained-deadline* task system, the relative deadline of each task is no larger than the task's period: $D_i \leq T_i$ for all $\tau_i \in \tau$.
- Tasks in an *arbitrary-deadline* task system do not need to have their relative deadlines satisfy any constraint with regards to their periods.

It is evident from these definitions that each implicit-deadline task system is also a constrained-deadline task system, and each constrained-deadline task system is also an arbitrary-deadline task system.

The ratio of the WCET of the task to its period is called the *utilization* of the task, and the ratio of the WCET to the smaller of its period and relative deadline is called its *density*. The *utilization of a task system* is defined to be the sum of the utilizations of all the tasks in the system. We will use the following notation in this book to represent the utilizations and densities of individual sporadic tasks, and of sporadic task systems.

Utilization: The utilization u_i of a task τ_i is the ratio C_i / T_i of its WCET to its period. The total utilization $U_{\text{sum}}(\tau)$ and the largest utilization $U_{\max}(\tau)$ of a task system τ are defined as follows:

$$U_{\text{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} u_i; \qquad U_{\max}(\tau) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau} (u_i)$$

Density: The density dens_i of a task τ_i is the ratio $(C_i/\min(D_i, T_i))$ of its WCET to the smaller of its relative deadline and its period. The total density $dens_{\text{sum}}(\tau)$ and the largest density $dens_{\text{max}}(\tau)$ of a task system τ are defined as follows:

$$\text{dens}_{\text{sum}}(\tau) \overset{\text{def}}{=} \sum_{\tau_i \in \tau} \text{dens}_i; \qquad \text{dens}_{\text{max}}(\tau) \overset{\text{def}}{=} \max_{\tau_i \in \tau} (\text{dens}_i)$$

Parallel Execution The different tasks in a task system are assumed, for the most part, to be independent of each other; hence, jobs of different tasks may execute simultaneously on different processors upon a multiprocessor platform. Parallelism within a single job is forbidden in our model: each job is assumed to represent a single thread of computation, which may execute upon at most one processor at any point in time.

As the scheduling windows of different jobs of the same task do not overlap in implicit-deadline and constrained-deadline task systems, each task in such a system executes upon at most one processor at each instant in time.

For tasks in arbitrary-deadline task systems that have relative deadline greater than period, the scheduling windows of multiple successive jobs of the task may overlap. The default assumption for such tasks is that multiple jobs of the same task may *not* execute simultaneously: a job must complete execution before the next job of the task begins to execute.

Thus, the 3-parameter sporadic task model does not in general allow for the modeling of parallelism within a single task. This has traditionally not been perceived as a shortcoming of the model, since the model was first developed in the context of uniprocessor systems, for which parallel execution is not possible in any case.

More recently, however, the trend toward implementing real-time systems upon multiprocessor and multicore platforms has given rise to a need for models that are capable of exposing any possible parallelism that may exist within the workload to the scheduling mechanism. Therefore, there has been a move in the real-time scheduling community toward considering new models that allow for partial parallelism within a task, as well as for precedence dependencies between different parts of each individual task. We describe such a model in Sect. 2.1.3 below.

2.1.3 The Sporadic DAG Tasks Model

Each recurrent task in this model is modeled as a DAG $G_i = (V_i, E_i)$, a (relative) deadline parameter D_i, and a period T_i. Each vertex $v \in V_i$ of the DAG corresponds to a sequential job of the kind discussed above, and is characterized by a WCET. Each (directed) edge of the DAG represents a precedence constraint: if (v, w) is a (directed) edge in the DAG then the job corresponding to vertex v must complete execution before the job corresponding to vertex w may begin execution. Groups of jobs that are not constrained (directly or indirectly) by precedence constraints in such a manner may execute in parallel if there are processors available for them to do so.

The task is said to release a *dag-job* at time-instant t when it becomes available for execution. When this happens, all $|V_i|$ of the jobs are assumed to become available for execution simultaneously, subject to the precedence constraints. The task may release an unbounded sequence of dag-jobs during runtime; all $|V_i|$ jobs that are released at some time-instant t must complete execution by time-instant $t + D_i$. A minimum interval of duration T_i must elapse between successive releases of dag-jobs. If $D_i > T_i$ it is not required that all jobs of a particular dag-job complete execution before execution of jobs of the next dag-job may begin—i.e., no precedence constraints are assumed between the jobs of different dag-jobs.

The sporadic DAGs task model will be discussed further in Chap. 21.

2.2 A Taxonomy of Multiprocessor Platforms

Various details must be provided in order to completely specify a multiprocessor platform. These include—how many processors comprise the platform? Are all these processors of the same kind, or do they possess different computing capabilities? How are the processors connected to each other? etc. In addition, we must specify whether the platform supports *preemption* and inter-processor *migration*.

As stated in Sect. 1.2, there is a classification of multiprocessor platforms as *identical*, *uniform*, or *unrelated*, depending upon the relative computing capabilities of the different processors; we will, for the most part, focus upon multiprocessor platforms comprised of multiple identical processors (we will briefly summarize some results concerning real-time scheduling upon unrelated multiprocessors in Chap. 22). We address the remaining questions below.

2.2.1 Preemption and Migration

Preemptive scheduling permits that a job executing upon a processor may be interrupted by the scheduler (perhaps because the scheduler needs to execute some other job), and have its execution resumed at a later point in time.

In nonpreemptive scheduling, such preemption is forbidden: once a job begins execution, it continues to execute until it has completed. (In *limited preemptive* scheduling, various kinds of restrictions are placed upon the occurrence of preemptions during scheduling).

Results concerning preemptive scheduling are typically obtained under the simplifying assumption that a preemption incurs no cost. As stated in Sect. 1.5, we will primarily study preemptive scheduling in this book.

Different algorithms for scheduling systems of recurrent real-time tasks upon multi-processor platforms place different restrictions as to where different tasks' jobs are permitted to execute. A *global* scheduling algorithm allows any job to execute upon any processor; by contrast, a *partitioned* scheduling algorithm first maps each recurrent task on to a particular processor, and only executes jobs of task upon the processor to which it has been mapped. Global and partitioned scheduling may both be considered as special cases of *clustered* scheduling, in which the processors comprising the multiprocessor platform are partitioned into clusters, and each recurrent task mapped on to a single cluster. Migration of a task's jobs is only allowed within the cluster to which the task is mapped.

In addition to the family of scheduling algorithms that may be considered to be specialized forms of clustered scheduling, there are scheduling algorithms that place various forms of restriction upon migration without forbidding it outright. Such algorithms are commonly called *semi-partitioned* or *limited migrative* scheduling algorithms; they may, for example, specify that no individual task is allowed to migrate between more than two processors, or they may restrict the total number of migratory tasks in a system, etc.

Inspired by some features that are available in modern multiprocessor operating systems such as Linux, some work has recently been done [39, 104] on a migrative model that is called the *processor affinities* model. In this model, a set of processors upon which a task may execute is specified for each task, and jobs of the task are allowed to migrate amongst these processors. The processor affinities model differs from clustered scheduling in that the sets of processors specified for the different tasks may overlap and hence not constitute a partitioning or a clustering of the set of available processors.

Although some interesting results concerning limited-migrative scheduling and the use of processor affinities have been obtained recently, we do not cover these paradigms; for the most part, the focus of this book remains on partitioned and global scheduling. We briefly state a few results concerning the semi-partitioned scheduling of Liu & Layland task systems in Sect. 6.5, and mention an interesting extension of partitioned scheduling called *federated scheduling* in the context of scheduling systems of sporadic DAG tasks, in Sect. 21.2.

2.3 Discussion

In this section, we attempt to justify some of the restrictions we have chosen to place upon the range of workload and machine models to be covered in this book. We particularly address the following questions:

1. Why do we primarily restrict ourselves to *sporadic* task models?
2. Why do we require *independence* amongst the tasks?
3. What are the consequences of ignoring preemption and migration costs?

2.3.1 The Restriction to Sporadic Task Models

We had stated, in Sect. 1.3, that we would be considering the scheduling of sporadic, rather than periodic, task systems in this book. As stated there, the main distinguishing feature of sporadic task systems is that *minimum*, rather than exact, inter-arrival separations are specified between the arrival of different pieces of work (jobs) from a task.

From a pragmatic perspective, our decision to restrict attention to sporadic models is driven by our desire, also stated in Chap. 1 (in Sect. 1.5), to only consider problems for which solutions are tractable—have polynomial or pseudo-polynomial run-time. It appears that defining a recurrent task model that violates the sporadic assumption in any meaningful way results in scheduling analysis problems that are highly intractable (nondeterministic polynomial time (NP)-hard in the strong sense)—an illustrative example is provided in Sect. 4.4, where it is shown that a very basic analysis problem for periodic task systems is highly intractable even upon single-processor platforms. Thus, our desire for efficiently solvable analysis problems restricts our choice of task models, and it appears that the various sporadic task models are about as general as one can get without running up against the intractability barrier (the separation between tractable and intractable models for uniprocessor scheduling is very precisely and methodically demarcated in Stigge's dissertation [168]; see also [169]).

From an application perspective, too, an argument can be made in favor of considering sporadic task models. Note that in sporadic task models, the different tasks do not really need a common notion of global time; the only requirement is that they all share a common notion of duration (i.e., they should all agree on the duration of "real time" represented by a unit of time). By contrast, stricter notions of recurrence such as periodic tasks assume that the different tasks generate jobs at specific "global" *instants* in time; one consequence of this is that periodic and similar models are more sensitive to differences between the notions of time maintained by different sources that may be responsible for generating the jobs of different tasks. Consider, for example, a distributed system in which each task (i.e., the associated process) maintains its own (very accurate) clock, and in which the clocks of different tasks are not synchronized with each other. The accuracy of the clocks permit us to assume that there is no clock drift, and that all tasks use exactly the same units for measuring time. However, the fact that these clocks are not synchronized rules out the use of a concept of an absolute time scale. This idea is explored further in Sect. 2.3.2 below, as the second task independence assumption.

We note that, since periodic behavior is one of the possible behaviors of a sporadic task system with corresponding parameter values, the worst-case behavior of a sporadic task system "covers" the behavior of the corresponding periodic task system; therefore, showing that a sporadic task system will always meet all deadlines implies that the corresponding periodic task system will also do so (although the converse of this statement is not true: a periodic task system may meet all deadlines but the corresponding sporadic task system may not).

A further note: upon uniprocessor platforms, the difference between periodic and sporadic behavior is sometimes brushed aside (see, e.g., [140, p. 40], which claims that scheduling analysis results for periodic task models remain valid even if the period parameters are reinterpreted to represent minimum inter-arrival separations). The rationale for this (backed by some empirical evidence) is that the worst-case behavior of the sporadic task system is evidenced when it behaves as a periodic system;

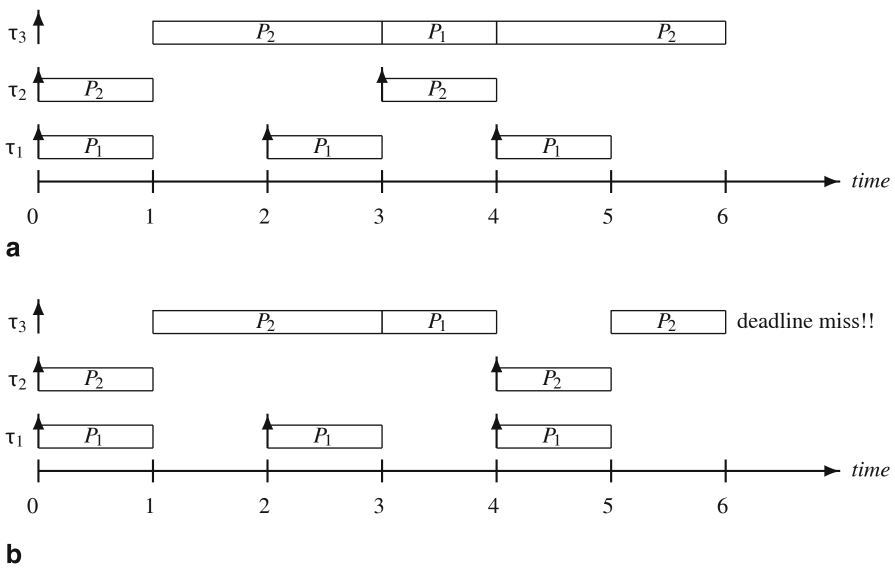


Fig. 2.2 **a** A 2-processor schedule for the job arrivals of task system τ of Example 2.1, interpreted as a *periodic* task system—repeat this schedule over all intervals $[6k, 6(k+1))$ for all $k = 0, 1, 2, \ldots$. Each up arrow denotes a job arrival: job-arrivals of τ_1 (at time-instants 0, 2, and 5) are depicted by the bottom-most row of up arrows, job-arrivals of τ_2 (at time-instants 0 and 3) by the middle row of up arrows, and the sole job-arrival of τ_3 (at time-instant 0) is depicted by the up arrow on the topmost row. **b** An infeasible sequence of job arrivals of the same task system interpreted as a sporadic task system

for example, it is known that a system of 3-parameter sporadic tasks is *infeasible*—cannot always be scheduled to meet all deadlines—on a single processor if and only if the arrival sequence for the equivalent periodic task system is also infeasible. But this is not necessarily true upon multiprocessors; consider the following example.

Example 2.1 As shown in Fig. 2.2, the task system τ comprised of the following three tasks

$$\tau_1 = (1, 1, 2), \tau_2 = (1, 1, 3), \tau_3 = (5, 6, 6)$$

is feasible on two processors under global scheduling if interpreted as periodic tasks, but not if interpreted as sporadic tasks (we cannot meet all deadlines for the sporadic task system if τ_1's second job arrives 3 time units after the first rather than arriving the minimum of 2 time units after).

2.3.2 The Task Independence Assumption

A pair of conditions collectively called the *task independence assumptions* were specified in [41]. These assumptions, listed below, dictate the process by which jobs are generated by the tasks in the system; once generated, the jobs (each characterized by an arrival time, an execution requirement, and a deadline) are independent of each other. That is, while the task independence assumptions restrict the job-generation process, they make no assertions about the interactions of the jobs once they have been generated.

1. *The runtime behavior of a task should not depend upon the behavior of other tasks.* That is, each task is an independent entity, perhaps driven by separate external events. It is not permissible for one task to generate a job directly in response to another task generating a job.
2. *The workload constraints are specified without referencing "absolute" time.* Hence, specifications such as "Task T generates a job at time-instant 3" are forbidden.
   (Note that periodic task systems in which an initial *offset* is specified for each task—see Sect. 4.4—violate the task independence assumption since these offsets are defined in terms of an absolute time scale).

In terms of sets of jobs that may legally be generated by a task system, the first task independence assumption implies that a set of jobs generated by an entire task system is legal in the context of the task system if and only if the jobs generated by each task are legal with respect to the constraint associated with that task. Examples of task systems *not* satisfying this assumption include systems where, for example, all tasks are required to generate jobs at the same time instant, or where it is guaranteed that certain tasks will generate jobs before certain other tasks. (To represent such systems in a manner satisfying this assumption, the interacting tasks must instead be modeled as a single task which is assumed to generate the jobs actually generated by the interacting tasks).

Letting an ordered 3-tuple (a, e, d) represent a job with arrival time a, execution requirement e, and deadline d, the second task independence assumption implies that if $\{(a_o, e_o, d_o), (a_1, e_1, d_1), (a_2, e_2, d_2) \ldots\}$ is a legal arrival set with respect to the workload constraint for some task in the system, then so is the set $\{(a_o - x, e_o, d_o - x), (a_1 - x, e_1, d_1 - x), (a_2 - x, e_2, d_2 - x) \ldots\}$, where x may be any real number.

The task independence assumptions are extremely general and are satisfied by a wide variety of the kinds of task systems one may encounter in practice. The various flavors of sporadic task systems discussed in Sect. 2.1 above satisfy these assumptions, as do "worst-case" periodic task systems—periodic task systems where each task may choose its offset.

So does a distributed system in which each task executes on a separate node (jobs correspond to requests for time on a shared resource), and which begins execution in response to an external event. All temporal specifications are made relative to the time at which the task begins execution, which is not *a priori* known.

It is noteworthy that answering interesting schedulability-analysis questions (such as determining feasibility) for many nontrivial task systems not satisfying the task independence assumptions (such as periodic task systems with deadlines not equal to period) turns out to be computationally difficult (often NP-hard in the strong sense), and hence of limited interest from the perspective of efficient analysis.

2.3.3 Preemption and Migration Costs

As stated in Sect. 2.2.1 above, in this book we are, for the most part, assuming a preemptive model of computation: a job executing on the processor may have its execution interrupted at any instant in time and resumed later, with no cost or penalty. In a similar vein, when we permit inter-processor migration in global scheduling, we assume that there is no cost or penalty associated with such migration. These are both clear approximations: in most actual systems, we would expect both a preemption and a migration to incur some delay and computational cost (we recommend the excellent discussions in Brandenburg's dissertation [65] for a detailed look at some of the issues arising in attempting to account for such overhead costs in actual computer systems).

There is a class of scheduling algorithms called *priority-driven* algorithms (Definition 3.3) that we will discuss in greater detail in Chap. 3; for now, it suffices to state that very many of the scheduling algorithms that we will look at in this book are priority-driven ones. Priority-driven algorithms possess the pleasing properties that

1. The number of preemptions in any preemptive schedule generated by such algorithms is strictly less than the number of jobs that are being scheduled, and
2. The number of job interprocessor migrations in any preemptive global schedule generated by such algorithms is also strictly less than the number of jobs that are being scheduled.

Hence, one can account for the overhead cost of preemptions and migrations by simply inflating the WCET parameters by the worst-case cost of one preemption and one migration—we will discuss this idea in further detail in Chap. 3.

Sources

A useful discussion of models (albeit primarily in the context of uniprocessor scheduling) is to be found in the survey paper [169] and in Stigge's dissertation [168]. The taxonomy of multiprocessor platforms may be found in textbooks on scheduling in the Operations Research context (where *processors* are commonly called *machines*). The task independence assumptions are from [41].

Chapter 3
Preliminaries: Scheduling Concepts and Goals

In this chapter, we define and discuss some concepts that form an important part of the background and vocabulary used in discussions of real-time scheduling. The concepts of feasibility and schedulability, discussed in Sect. 3.1 below, formalize required properties of hard-real-time scheduling algorithms. Section 3.2 introduces a classification of scheduling algorithms that we will be using a lot in the remainder of this book to categorize the different scheduling algorithms we will study. The notions of sustainability and predictability, discussed in Sect. 3.3, formalize some other desired properties we seek in our real-time scheduling algorithms.

3.1 Feasibility and Schedulability

It is evident from the definition of sporadic tasks that a given sporadic task system may generate infinitely many different collections of jobs during different executions. In order for a sporadic task system to be deemed feasible, it should be possible to construct schedules for each one of these collections of jobs that meet all job deadlines.

Definition 3.1 (feasiblility) A task system is said to be *feasible* upon a specified platform if schedules meeting all timing constraints exist upon the platform for all the collections of jobs that could legally be generated by the task system.

Feasibility is a very general property; it is merely required that a correct schedule *exists* for every legal collection of jobs. It may not always be possible to construct such a schedule without knowing the arrival times of all jobs beforehand[1].

A scheduling algorithm that knows all arrival times of all jobs beforehand is often referred to as a *clairvoyant* scheduler; a clairvoyant scheduler represents a nonrealizable ideal against which to compare the performance of some actual scheduling algorithm (via, e.g., the criterion of *optimality*—see the discussions

[1] The precise information that must be known beforehand to be able to make correct scheduling decisions in a multiprocessor setting is derived in [77].

S. Baruah et al., *Multiprocessor Scheduling for Real-Time Systems,*
Embedded Systems, DOI 10.1007/978-3-319-08696-5_3

about uniprocessor scheduling in Sects. 4.1 and 4.2—or the metric of *speedup factor*—Definition 5.2). In contrast to clairvoyant scheduling algorithms, the kinds of algorithms actually used for scheduling sporadic task systems typically have to deal with uncertainties regarding both

1. The arrival times of future jobs—all that is known is some lower bounds on these arrivals times, and in the durations between successive arrival times; and
2. The exact amount of execution that is needed by these jobs—although an upper bound on the execution requirement is provided by the WCET parameter, it is typically assumed that a scheduling algorithm only gets to know the exact amount of execution a job needs by actually executing the job until it completes.

That is, these two pieces of information—the actual arrival times and the actual execution times—about jobs are revealed *on line* to the scheduling algorithm during system execution. The concepts of sustainability and predictability, discussed later in this chapter in Sect. 3.3, elaborate upon some of the challenges arising from such uncertainty, and characterize some desirable attributes of scheduling algorithms that ameliorate the handling of these challenges.

In building hard-real-time applications, it is not sufficient to simply know that a task system is feasible; in addition, it must be guaranteed prior to system run time that all deadlines will be met by the scheduling algorithm that is used in the application. Such guarantees are made by schedulability tests.

Definition 3.2 (schedulability tests) Let A denote a scheduling algorithm. A sporadic task system is said to be *A-schedulable* upon a specified platform if A meets all deadlines when scheduling each of the potentially infinite different collections of jobs that could be generated by the sporadic task system, upon the specified platform.

An *A-schedulability test* accepts as input the specifications of a sporadic task system and a multiprocessor platform, and determines whether the task system is A-schedulable. An A-schedulability test is said to be *exact* if it identifies all A-schedulable systems, and *sufficient* if it identifies only some A-schedulable systems.

(Of course, an A-schedulability test may never incorrectly misidentify some system that is not A-schedulable, as being A-schedulable.) A sufficient schedulability test that is not exact is also called *pessimistic*, but for many situations an exact schedulability test is unknown or is computationally intractable. From an engineering point of view, a tractable schedulability test that is exact is ideal, while a tractable sufficient test with low pessimism is acceptable.

3.2 Priority Assignments

Run-time scheduling is essentially the process of determining, during the execution of a real-time application system, which job[s] should be executed at each instant in time. Run-time scheduling algorithms are typically implemented as follows: at each time instant, assign a *priority* to each active job, and allocate the available processors to the highest-priority jobs.

Depending upon the restrictions that are placed upon the manner in which priorities may be assigned, priority-based algorithms for scheduling sporadic task systems may be classified into the following categories.

1. In *fixed task priority (FTP)* scheduling, each sporadic task is assigned a unique priority, and each job inherits the priority of the task that generated it. The rate-monotonic scheduling algorithm [139], which assigns priorities to tasks according to their period parameters—tasks with smaller period are assigned greater priority (ties broken arbitrarily)—is an example of an FTP scheduling algorithm.
2. In *fixed job priority (FJP)* scheduling, different jobs of the same task may be assigned different priorities. However, the priority of a job, once assigned, may not change. The earliest deadline first scheduling algorithm [76, 139], in which the priority of a job depends upon its deadline parameter—jobs with earlier deadlines are assigned greater priority (ties broken arbitrarily)—is an example of a FJP scheduling algorithm.
3. In *dynamic priority (DP)* algorithms, there are no restrictions placed upon the manner in which priorities are assigned to jobs—the priority of a job may change arbitrarily often between its release time and its completion. The pfair scheduling algorithms that we will study in Chap. 7 are examples of DP scheduling. So are the Least Laxity algorithm discussed in Sect. 20.1 and EDZL [125] discussed in Sect. 20.2.

It is evident from the above classification that different scheduling algorithms may differ from one another in the manner in which priorities get assigned to individual jobs by the algorithms. Some scheduling algorithms are observed to have certain desirable features in terms of ease (and efficiency) of implementation, particularly upon multiprocessor platforms. Some of the important characteristics of such algorithms were studied by Ha and Liu [105–107], who proposed the following definition:

Definition 3.3 (priority-driven algorithms [107].) A scheduling algorithm is said to be a *priority driven* scheduling algorithm if and only if it satisfies the condition that *for every pair of jobs J_i and J_j, if J_i has higher priority than J_j at some instant in time, then J_i always has higher priority than J_j.*

Observe that FJP and FTP scheduling algorithms each assign a single priority to each job and are therefore classified as priority-driven (Definition 3.3); DP scheduling algorithms are not priority-driven according to this definition.

From an implementation perspective, there are significant advantages in using priority-driven algorithms in real-time systems; while it is beyond the scope of this book to describe in detail all these advantages, some important ones are listed below.

- Very efficient implementations of priority-driven scheduling algorithms have been designed (see, e.g., [150]).
- It can be shown that when a set of jobs is scheduled using a priority-driven algorithm then the total number of *preemptions* is bounded from above by the number of jobs in the set (and consequently, the total number of *context switches* is bounded at twice the number of jobs).

- For systems where *interprocessor migration* is permitted, it can similarly be shown that the total number of interprocessor migrations of individual jobs is bounded from above by the number of jobs.

3.3 Sustainability and Predictability

Systems are typically specified according to their worst-case parameters; for example in the three-parameter sporadic task model (Sect. 2.1.2), C_i denotes the worst-case execution requirement, T_i the minimum inter-arrival temporal separation between the arrival times of successive jobs, and D_i the maximum duration of time that may elapse between a job's arrival and the completion of its execution. Actual behavior during run-time if often "better" than the specifications—jobs may execute for less than C_i, arrive more than T_i time units apart, and can terminate before the deadline. The notion of sustainability [40] formalizes the expectation that a system determined to be schedulable under its worst-case specifications should remain schedulable when its real behavior is "better" than worst case: intuitively, sustainability requires that schedulability be preserved in situations in which it should be "easier" to ensure schedulability.

Within the context of the multiprocessor scheduling of sporadic task systems, sustainability may be discussed at several different levels. In particular, we distinguish between sustainable scheduling algorithms and sustainable schedulability tests. A *scheduling algorithm A* is said to be sustainable if any system that is A-schedulable under its worst-case specification remains so when its behavior is better than worst case. In order for an *A-schedulability test* to be deemed sustainable, it is required that the following two requirements be met:

1. Any task system deemed A-schedulable by this schedulability test should continue to meet its deadlines when its run-time behavior is "better" than mandated by the worst-case specifications, and
2. All task systems with "better" (less constraining) parameters than a task system found to be A-schedulable by the test should also be deemed A-schedulable by the test. (This second requirement is closely related to the notion of *self-sustainability* [23]; self-sustainability is discussed further in Sect. 14.5.1.)

Let us now delve a bit deeper into this definition of sustainable schedulability tests. Note that if A is a sustainable scheduling algorithm then all (exact and sufficient) A-schedulability tests will meet the first of these two requirements, but may or may not meet the second. If A is not a sustainable scheduling algorithm, on the other hand, no exact A-schedulability test can possibly be sustainable since it necessarily violates the first condition.[2]

[2] For such scheduling algorithms, it has been argued [40] that it is better engineering practice to use a sufficient schedulability test that is sustainable rather than an exact one that is not.

As stated above (and explored deeper in [40]), sustainability is a very basic and fundamental requirement in the engineering of hard-real-time systems, since it is only very rarely that a system can be characterized exactly (rather than by upper bounds) at system design time. This immediately motivates the first requirement in the definition of sustainable schedulability tests above, that for a schedulability test to be considered useful it must be the case that task systems deemed schedulable by the test should continue to meet all deadlines if the system behaves better than the specifications at run-time. Indeed, any schedulability test that fails to meet this requirement is not likely to be of much use in the engineering of hard-real-time systems.

The second requirement in the definition of sustainable schedulability tests—that all task systems with less constraining parameters than a system deemed schedulable should also be deemed schedulable—is a secondary requirement that has arisen from the needs of the incremental, interactive design process that is typically used in designing real-time systems. Ideally, such a design process allows for the interactive exploration of the state space of possible designs; this is greatly facilitated if making changes for the better to a design only results in improvements to system properties. If we were to only use sustainable schedulability tests during the system design process, then we would know that relaxing timing constraints (e.g., increasing relative deadlines or periods, or decreasing worst-case execution times) would not render schedulable subsystems unschedulable. For example, suppose that we were designing a composite system comprised of several subsystems, and we were at a point in the design state space where most subsystems are deemed schedulable by the schedulability test. We could safely consider relaxing the task parameters of the schedulable subsystems in order to search for neighboring points in the design state space in which the currently unschedulable subsystems may also be deemed schedulable, without needing to worry that the currently schedulable subsystems would unexpectedly be deemed unschedulable.

The notion of *predictability* is closely related to the concept of sustainability.

Let I denote a collection of jobs, in which each job has a WCET as well as a best case execution time (BCET) specified—during any execution, each job will have an actual execution requirement that is no smaller than its BCET and no larger than its WCET (in the models we have introduced thus far in this book, BCET is always assumed to be equal to zero). Within the context of any particular scheduling algorithm, let $S(I^+)$ and $S(I^-)$ denote the schedules for I when each job has an actual execution time equal to its WCET and BCET, respectively. The scheduling algorithm is said to be

Start time predictable if the actual start time of each job is no sooner than its start time in $S(I^-)$ and no later than its start time in $S(I^+)$;

Completion time predictable if the actual completion time of each job is no sooner than its completion time in $S(I^-)$ and no later than its completion time in $S(I^+)$; and

Predictable if it is both start time and completion time predictable.

Hence, with a predictable scheduling algorithm it is sufficient, for the purpose of bounding the worst-case response time of a task or proving schedulability of a task system, to look just at the jobs of each task whose actual execution times are equal to the task's worst-case execution time. Not every scheduling algorithm is predictable in this sense—for instance, many of the *anomalies* that have been identified in nonpreemptive scheduling (see, e.g., [97, 98]) are a direct consequence of nonpredictability.

However, Ha and Liu proved [107] that *all preemptable FTP and FJP scheduling algorithms are predictable*:

Theorem 3.1 ***(Ha and Liu)*** *Any preemptive FJP and FTP scheduling algorithm is predictable.* □

As a consequence, in considering such scheduling algorithms we need to only consider the case where each job's execution requirement is equal to the worst-case execution requirement of its generating task.

Sources

The classification of scheduling algorithms according to the priority assignment scheme is to be found in [67]. Priority-driven algorithms were defined in [107]; the term "predictability" as used here is also from [107]. The concept of sustainability was defined in [40].

Chapter 4
A Review of Selected Results on Uniprocessors

In this chapter we briefly review some important concepts and results from uniprocessor real-time scheduling theory that we will be using during our study of multiprocessor real-time systems in the later chapters. In Sect. 4.1, we prove the optimality of earliest-deadline-first (EDF) scheduling upon preemptive uniprocessor platforms; this powerful result favors the use of EDF as the run-time algorithm upon the individual processors in partitioned systems. Section 4.2 reviews some important results concerning deadline-monotonic and other fixed-task-priority (FTP) scheduling algorithms on unipocessors. Section 4.3 describes a particular collection of jobs called the synchronous arrival sequence (SAS) that may be generated by a sporadic task system, and discusses the reasons for the important role this collection of jobs plays in uniprocessor scheduling theory. Section 4.4 shows that answering interesting scheduling-theoretic questions concerning *periodic* task systems is often highly intractable even for uniprocessor systems, and thereby helps explain our decision to largely limit the scope of this book to the analysis of sporadic task systems.

4.1 The Optimality of EDF on Preemptive Uniprocessors

The EDF scheduling algorithm assigns scheduling priority to jobs according to their absolute deadlines: the earlier the deadline, the greater the priority (with ties broken arbitrarily). EDF is known to be *optimal* for scheduling a collection of jobs upon a preemptive uniprocessor platform, in the sense that if a given collection of jobs can be scheduled to meet all deadlines, then the EDF-generated schedule for this collection of jobs will also meet all deadlines. This fact is typically proved by induction, as follows.

Suppose that there is an optimal schedule that meets all deadlines of all the jobs, and suppose that the scheduling decisions made by EDF are identical to those made by this optimal schedule over the time interval $[0, t)$. At time-instant t, EDF observes that job j_1, with deadline d_1, is an earliest-deadline job needing execution, and schedules it over the interval $[t, t + \delta_1)$. However, the optimal schedule schedules some other job j_2, with deadline d_2, over the interval $[t, t+\delta_2)$. Let δ denote the smaller of δ_1 and δ_2: $\delta \overset{\text{def}}{=} \min\{\delta_1, \delta_2\}$. The critical observation is that since EDF, by definition, chooses a job with the earliest deadline, it must be the case that $d_1 \leq d_2$. Furthermore, since the optimal schedule also schedules j_1 to complete by its deadline, it must be the case that j_1 is executed in the optimal schedule to complete by time-instant to d_1. In the optimal schedule, we may therefore simply swap the execution of j_2 over $[t, t+\delta)$ with the first δ units of execution of j_1, thereby obtaining another optimal schedule for which the scheduling decisions made by EDF are identical to those made by this optimal schedule over the time interval $[0, t+\delta)$. The optimality of EDF follows, by induction on t.

This optimality of EDF on preemptive uniprocessors is a very powerful property. To show that a system is EDF-schedulable upon a preemptive uniprocessor, it suffices to show the *existence* of a schedule meeting all deadlines—the optimality of EDF ensures that it will find such a schedule. Consider, for example, an implicit-deadline sporadic task system τ. It is evident that if $U_{\text{sum}}(\tau) \leq 1$ then any collection of jobs generated by τ can be scheduled by an optimal clairvoyant scheduling algorithm to meet all deadlines—if we were to construct a "processor-sharing schedule"[1] in which each job of each task τ_i were assigned a fraction u_i of a processor at each instant between its arrival time and its deadline, each job would complete by its deadline (and since $U_{\text{sum}}(\tau) \leq 1$ we would never end up assigning a fraction of the processor greater than 1 at any instant in time). We may therefore immediately conclude, from EDF's optimality property that EDF schedules τ to always meet all deadlines. This establishes that any implicit-deadline sporadic task system τ with $U_{\text{sum}}(\tau) \leq 1$ is scheduled by EDF to always meet all deadlines.

4.2 FTP Scheduling Algorithms

In an FTP scheduling algorithm (see Sect. 3.2), each sporadic task in a task system is assigned a distinct priority and a job inherits the priority of the task that generates it.

Consider a three-parameter sporadic task system τ that is scheduled using some FTP algorithm (i.e., under some task priority assignment). Let $\text{hp}(\tau_k)$ denote the tasks that are assigned a priority greater than the priority of task τ_k.

The *worst-case response time* of task τ_k in a schedule denotes the maximum duration between the release of a job of τ_k and the time it completes execution in the schedule. For any constrained-deadline three-parameter sporadic task system τ, it has been shown [112, 124] that the smallest fix-point of the recurrence

$$R_k = C_k + \sum_{\tau_i \in \text{hp}(\tau_k)} \left\lceil \frac{R_k}{T_i} \right\rceil C_i \tag{4.1}$$

[1] The proof of Theorem 7.1 explains how such a processor-sharing schedule may be converted into one in which at most one job is executing at each instant in time.

is a tight upper bound on the response time of τ_k.

The rate-monotonic (RM) scheduling algorithm [137] is an FTP scheduling algorithm in which the priority assigned to a task depends upon its period: tasks with smaller period are assigned greater priority (with ties broken arbitrarily). It is known [137] that RM is an optimal FTP scheduling algorithm for scheduling implicit-deadline sporadic task systems upon preemptive uniprocessors: if there is any FTP scheduling algorithm that can schedule a given implicit-deadline sporadic task system to always meet all deadlines of all jobs, then RM will also always meet all deadlines of all jobs. A tight *utilization bound*[2] is also known for the RM scheduling of implicit-deadline sporadic task systems: any task system τ with $U_{\text{sum}}(\tau) \leq |\tau|(2^{1/|\tau|} - 1)$ is guaranteed RM-schedulable on a preemptive uniprocessor, and for each value of n there is an implicit-deadline sporadic task system of n tasks with utilization exceeding $n(2^{1/n} - 1)$ by an arbitrarily small amount that RM fails to schedule successfully.

The DM scheduling algorithm [133] is another FTP scheduling algorithm in which the priority assigned to a task depends upon its relative deadline parameter rather than its period: tasks with smaller relative deadline are assigned greater priority (with ties broken arbitrarily). Note that RM and DM are equivalent for implicit-deadline sporadic task systems, since all tasks in such systems have their relative deadline parameters equal to their periods. It has been shown [133] that DM is an optimal FTP scheduling algorithm for scheduling constrained-deadline sporadic task systems upon preemptive uniprocessors: if there is any FTP scheduling algorithm that can schedule a given constrained-deadline sporadic task system to always meet all deadlines of all jobs, then DM will also always meet all deadlines of all jobs. DM is however known to not be optimal for arbitrary-deadline sporadic task systems.

4.3 The Synchronous Arrival Sequence

In sporadic task models, tasks are characterized by a minimum inter-arrival separation parameter (amongst other parameters such as WCET and relative deadline, and perhaps additional parameters—as we will see in Chap. 21, in the sporadic DAG tasks model these may include an entire DAG). A sporadic task is said to generate a *synchronous arrival sequence (SAS)* of jobs if it generates jobs as rapidly as permitted to do so, i.e., a job arrives immediately the minimum inter arrival separation has elapsed since the arrival of the previous job. This notion is extended to task systems: a task system is said to generate a synchronous arrival sequence of jobs if all tasks in the task system generate a job at the same instant in time, and each task generates subsequent jobs as rapidly as permitted. The SAS is particularly significant in uniprocessor scheduling because there are many situations for which it is known that the SAS represents the worst-case behavior of a sporadic task system: if an algorithm is able to schedule the SAS for a system, then it is able to schedule all other collections of tasks that can legally be generated by the task system. Hence, many

[2] Utilization bounds are formally defined a bit later in this book—Definition 5.1.

uniprocessor schedulability results concerning sporadic task systems (such as the bound on response time described in Eq. 4.1 above and a pseudo-polynomial-time EDF schedulability test obtained [51]) are obtained by considering the SAS.

4.4 Intractability of Schedulability Analysis for Periodic Task Systems

In contrast to sporadic tasks, for which a minimum separation between successive arrivals by jobs of a task are specified, a *periodic* task has successive jobs arrive exactly a specified time-interval apart. In one popular model, a periodic task τ_i is characterized by the four parameters (A_i, C_i, D_i, T_i),with the interpretation that this task generates a job with worst-case execution requirement C_i and relative deadline D_i at each time-instant $(A_i + kT_i)$ for all $k \in \mathbf{N}$. As in the 3-parameter sporadic task model (Sect. 2.1.2), the parameters C_i, D_i, and T_i are referred to as the worst-case execution time (WCET), relative deadline, and period parameters of the task; the parameter A_i is called the *(initial) offset*. Analogously to sporadic task systems, a periodic task system τ is comprised of a number of periodic tasks. Periodic task system τ is said to be *synchronous* if all the tasks in τ have the same value of the offset $(A_i = A_j$ for all $\tau_i, \tau_j \in \tau)$. Also as with sporadic task systems, periodic task systems are characterized as implicit-deadline if $D_i = T_i$ for all tasks τ_i, constrained-deadline if $D_i \leq T_i$ for all tasks τ_i, and arbitrary-deadline otherwise.

It has been shown that many common schedulability problems for constrained-deadline and arbitrary-deadline periodic task systems are highly intractable—nondeterministic polynomial time (NP)-hard in the strong sense—even on preemptive uniprocessors. Further, we present the proof from [49, 50], showing that feasibility analysis of such systems is co-NP-hard in the strong sense (given the optimality of preemptive uniprocessor EDF mentioned in Sect. 4.1, this immediately implies that EDF-schedulability is also co-NP-hard in the strong sense).

Definition 4.1 (Simultaneous Congruences Problem (SCP)) Given a set A of n ordered pairs of positive integers $A = \bigcup_{i=1}^{n}\{(a_i, b_i)\}$ and a positive integer $k \leq n$, determine whether there is a subset $A' \subseteq A$ of k pairs and a natural number x such that for every $(a_i, b_i) \in A'$, $x \equiv a_i (\bmod\ b_i)$.

It has been shown [49; Theorem 3.2] that the SCP is NP-complete in the strong sense. Further, we will use this fact to establish the intractability of preemptive uniprocessor feasibility analysis for periodic task systems.

Theorem 4.1 *The feasibility problem for periodic task systems upon preemptive uniprocessor is co-NP-hard in the strong sense.*

Proof Let $\langle\{(a_1, b_1), \ldots, (a_n, b_n)\}, k\rangle$ denote an instance of SCP. Consider now the periodic task system of n tasks that is obtained in polynomial time from this instance of SCP, with the following parameters [49, 132]. For $1 \leq i \leq n$,

$$A_i \leftarrow (k-1) \times a_i$$

$$C_i \leftarrow 1$$

$$D_i \leftarrow k - 1$$
$$T_i \leftarrow (k - 1) \times b_i$$

It is evident (see [132] for details) that this task system is feasible if and only if there is no simultaneous collision of k pairs from the instance of SCP, i.e., if the instance $\langle\{(a_1, b_1), \ldots, (a_n, b_n)\}, k\rangle \notin$ SCP. Furthermore, all of the values in the task system are bounded by a polynomial in the values in the instance of SCP. □

The above theorem implies that there is no pseudo-polynomial-time algorithm for the feasibility problem for periodic task systems unless P = NP. In fact, the following corollary establishes that this intractability result holds even for task systems with utilization bounded from above by an arbitrarily small constant.

Corollary 4.1 *The feasibility problem for periodic task systems is co-NP-hard in the strong sense even if the systems are restricted to have processor utilization not greater than ϵ, where ϵ is any fixed positive constant.*

Proof Consider the construction in the proof of Theorem 3.2. If we multiply all of the start times and periods in this construction by the same positive integer, the proof still holds. In particular, if we multiply all of the start times and periods by some positive integer $c \geq n/(\epsilon(k-1)\min\{b_i\})$, then $\sum_{i=1}^{n} e_i/p_i \leq \epsilon$. □

Corollary 4.1 highlights the contrast between periodic and sporadic task systems: for sporadic task systems it has been shown [51] that feasibility analysis can be done in pseudo-polynomial time for task systems with utilization bounded from above by any constant strictly less than one.

We now list, without proof, some additional results concerning the preemptive uniprocessor scheduling of periodic task systems, that we will be using later in this book.

- It has been shown [49, 50] that the negative result of Corollary 4.1 does not hold for synchronous periodic task systems—those in which all tasks have the same offset. Specifically, for synchronous task systems with utilization bounded from above by a constant strictly less than one, EDF schedulability analysis can be performed in pseudo-polynomial time.
- Two important results were obtained in [133] concerning fixed-task-priority (FTP) scheduling of periodic task systems. It was shown that DM is not an optimal FTP algorithm for scheduling periodic task systems, even constrained-deadline ones. It was also proved that DM is, however, optimal for synchronous constrained-deadline periodic task systems.

Sources

The intractability results in Sect. 4.4 are from [49, 50].

Chapter 5
Implicit-Deadline (L&L) Tasks

This chapter, and the next few ones, are devoted to the multiprocessor scheduling of implicit-deadline sporadic task systems. We will see that a large amount of research has been conducted upon this topic, and quite a lot is known. Additionally, many of the techniques and approaches that underpin multiprocessor real-time scheduling theory, and that have subsequently gone on to be used in the analysis of systems represented using more sophisticated workload and platform models, were first discovered in the context of implicit-deadline task systems.

We briefly review the task model in this chapter, and define some of the metrics used for evaluating the effectiveness of scheduling algorithms and schedulability analysis tests. We consider partitioned scheduling in Chap. 6, and global scheduling in Chaps. 7–9, with Chap. 7 devoted to dynamic-priority (DP) scheduling, Chap. 8 to fixed-job-priority (FJP) scheduling, and Chap. 9 to fixed-task priority (FTP) scheduling.

We start out briefly reviewing the task and machine model we will be considering in this part of the book. Recall that an *implicit-deadline sporadic task* [137], also sometimes referred to as a *Liu & Layland task*, is characterized by an ordered pair of parameters: a *worst-case execution time (WCET)* and a *minimum inter-arrival separation* (that is, for historical reasons, also called the *period* of the task). Let τ_i denote an implicit-deadline sporadic task with WCET C_i and period T_i. Such a task generates a potentially infinite sequence of *jobs*, with the first job arriving at any time and subsequent job arrivals at least T_i time units apart. Each job has an execution requirement no greater than C_i; this must be met by a deadline that occurs T_i time units after the job's arrival.

We use the term *utilization* to denote the ratio of the WCET parameter of a task to its period: the utilization of τ_i, often denoted as u_i, is equal to C_i/T_i.

An implicit-deadline sporadic task *system* consists of a finite collection of sporadic tasks, each specified by the two parameters as described above. The utilization of a task system is defined to be the sum of the utilizations of all the tasks in the system; the utilization of task system τ is often denoted by $U_{\text{sum}}(\tau)$ (the largest utilization of any task is τ is analogously denoted by $U_{\max}(\tau)$).

S. Baruah et al., *Multiprocessor Scheduling for Real-Time Systems*,
Embedded Systems, DOI 10.1007/978-3-319-08696-5_5

An *identical multiprocessor platform* consists of a number m of processors ($m \geq 1$), each of which has the same computing capabilities as all the other processors in the platform.

Quantitative metrics allow us to compare the effectiveness of different scheduling algorithms. The utilization bound metric is a widely-used metric for quantifying the goodness of algorithms for scheduling implicit-deadline sporadic task systems; it is defined as follows:

Definition 5.1 Given a scheduling algorithm A for scheduling implicit-deadline sporadic task systems and a platform consisting of m unit-speed processors, the *utilization bound* of algorithm A on the m-processor platform is defined to be the largest number U such that all task systems with utilization $\leq U$ (and with each task having utilization ≤ 1) is successfully scheduled by A to meet all deadlines on the m-processor platform.

Utilization bounds have been widely used for evaluating the goodness of algorithms for the preemptive uniprocessor scheduling of implicit-deadline sporadic task systems; for example, the earliest deadline first (EDF) utilization bound of one and the rate-monotonic (RM) utilization bound of $\ln 2$ (approx. 0.69; [137]) are among the first results covered in many real-time scheduling courses. The first attempts at obtaining a better understanding of multiprocessor scheduling, therefore, quite naturally focused upon determining utilization bounds. Hence for example, Lopez et al. [140] have determined utilization bounds for various different approximate partitioning algorithms, and advocate the use of these utilization bounds as a quantitative measure of the goodness or effectiveness of these approximation algorithms. We will review these utilization bounds in Sect. 6.1.

One of the reasons why utilization bounds are significant in uniprocessor systems is that there is a direct relationship between feasibility and utilization: a necessary and sufficient condition for an implicit-deadline sporadic task system to be schedulable by an optimal algorithm on a unit-speed preemptive uniprocessor is that its utilization does not exceed one. However, this relationship breaks down for multiprocessor scheduling.[1] On the one hand, implicit-deadline sporadic task systems with utilization exceeding $(m + 1)/2$ by an arbitrarily small amount have been identified that cannot be scheduled by any partitioning or any global fixed-job-priority or fixed-task priority algorithm (see Sect. 3.2) on m unit-speed preemptive processors; on the other, there are task systems with utilization equal to m that can be scheduled by certain partitioning and certain global fixed-job-priority/fixed-task priority algorithms. This fact motivates the consideration of an additional metric, the speedup factor:

[1] This relationship does hold for global preemptive scheduling on multiprocessors: a necessary and sufficient condition for an implicit-deadline sporadic task system to be schedulable by an optimal algorithm on m unit-speed preemptive processors under global scheduling is that its utilization not exceed m [42]). However, we will see that all scheduling algorithms with utilization bound m are necessarily *dynamic-priority* algorithms (see Sect. 3.2); if we restrict our attention to fixed-job-priority (FJP) or fixed-task priority (FTP) algorithms, this relationship no longer holds.

Definition 5.2 (speedup factor) Scheduling algorithm A has speedup factor f, $f \geq 1$, if it successfully schedules any task system that can be scheduled upon a given platform by an optimal clairvoyant algorithm, provided A is scheduling the same task system upon a platform in which each processor is f times as fast as the processors available to the optimal algorithm.

It is evident from this definition that other factors being equal (or unimportant), we prefer scheduling algorithms with smaller speedup factors: an algorithm with speedup factor equal to one is an optimal algorithm.

The important point about this definition is that the optimal algorithm, against which the performance of algorithm A is compared, may be required to be subject to the same restrictions as the ones placed upon algorithm A (such as being a partitioning algorithm, and/or using an FJP algorithm upon each processor, etc.). Thus for example, the speedup factor of a fixed-job-priority scheduling algorithm would not penalize the algorithm for not being able to schedule some task system that cannot be scheduled even by an optimal clairvoyant fixed-job-priority scheduling algorithm.

In addition to deriving speedup factors for scheduling algorithms, we will on occasion derive a speedup factor for a schedulability test of a given schedulability algorithm. Such a metric is in essence bundling both the nonoptimality of a scheduling algorithm and the pessimism of its schedulability test into a single metric. It is reasonable to ask whether this makes sense; from a pragmatic perspective, we believe that the answer is "yes" for our domain of interest, which are hard-real-time systems. Since it must be a priori guaranteed in such hard-real-time systems that all deadlines will be met during run time, a scheduling algorithm is only as good as its associated schedulability test. In other words, a scheduling algorithm, no matter how close to optimal, cannot be used in the absence of an associated schedulability test able to guarantee that all deadlines will be met; what matters is that the *combination* of scheduling algorithm and schedulability test together have desirable properties.

Chapter 6
Partitioned Scheduling of L&L Tasks

In this chapter, we consider the partitioned preemptive scheduling of implicit-deadline sporadic task systems on identical multiprocessor platforms.

Given as input, an implicit-deadline sporadic task system τ and an m-processor platform, the tasks in the task system τ are to be partitioned into m disjoint subsystems, with each subsystem being assigned to execute upon a distinct processor. For most of the chapter, we consider that the tasks thus assigned to each processor are scheduled on that processor using the preemptive earliest-deadline-first (EDF) scheduling algorithm [76, 139]; in Sect. 6.4, we briefly describe the known results concerning fixed-task-priority scheduling. It follows from the results in [139] concerning the optimality of EDF upon preemptive uniprocessors (that we had described in Sect. 4.1) that a necessary and sufficient condition for the tasks assigned to each processor to be schedulable by EDF is that their utilizations sum to no more than the speed of the processor.

It is widely known that such partitioning is equivalent to the bin-packing [112, 113] problem, and is hence highly intractable: nondeterministic polynomial time (NP)-hard in the strong sense. We are therefore unlikely to be able to design partitioning algorithms that both achieve optimal resource utilization and have efficient (polynomial-time) implementations; instead, *approximation* algorithms that have polynomial run-time are widely used for performing such partitioning.

In Sect. 6.1, we briefly review utilization bounds for several such well-known approximate partitioning algorithms. In Sect. 6.2, we consider speedup factor in conjunction with utilization bounds to characterize more completely the performance of these different approximate partitioning algorithms. In Sect. 6.3, we describe a polynomial-time approximation scheme (PTAS) for determining, in-time polynomial in the representation of an implicit-deadline sporadic task system, a partitioning of the system upon an identical multiprocessor system that is within an additive constant ε removed from optimality with regards to the speedup factor, for any specified constant $\varepsilon > 0$.

S. Baruah et al., *Multiprocessor Scheduling for Real-Time Systems*,
Embedded Systems, DOI 10.1007/978-3-319-08696-5_6

6.1 Utilization Bounds for Some Common Partitioning Algorithms

Most polynomial-time heuristic algorithms for partitioning have the following common structure: first, they specify an order in which the tasks are to be considered. In considering a task, they specify the order in which to consider which processor upon which to attempt to allocate the task. A task is successfully allocated upon a processor if it is observed to "fit" upon the processor; within our context of the partitioned EDF-scheduling of implicit-deadline sporadic task systems, a task fits on a processor if the task's utilization does not exceed the processor capacity minus the sum of the utilizations of all tasks previously allocated to the processor. The algorithm declares success if all tasks are successfully allocated; otherwise, it declares failure.

Lopez et al. [142] compared several widely used heuristic algorithms extensively. In making these comparisons, Lopez et al. [142] classified the studied heuristics according to the following definitions:

Definition 6.1 (from [142]) A *reasonable allocation (RA)* algorithm is defined as one that fails to allocate a task to a multiprocessor platform only when the task does not fit into any processor upon the platform.

Within RA algorithms, *reasonable allocation decreasing (RAD)* algorithms consider the tasks for allocation in nonincreasing order of utilizations.

All the heuristic algorithms considered by Lopez et al. [142] were RA ones—indeed, there seems to be no reason why a system designer would ever consider using a non-RA partitioning algorithm. The algorithms considered by Lopez et al. [142] can be thought of as having been obtained by choosing different combinations of the following two factors:

1. *When a task is considered for assignment, to which processor does it get assigned?* The standard bin-packing heuristics of first-fit, worst-fit, and best-fit were considered:
   - In *first-fit (FF)* algorithms, the processors are considered ordered in some manner and the task is assigned to the first processor on which it fits.
   - In *worst-fit (WF)* algorithms, it is assigned to the processor with the maximum remaining capacity.
   - In *best-fit (BF)* algorithms, it is assigned to the processor with the minimum remaining capacity *exceeding its own utilization* (i.e., on which it fits).
2. *In what order are the tasks considered for assignment?*
   - In *decreasing (D)* algorithms, the tasks are considered in nonincreasing order of their utilizations.
   - In *increasing (I)* algorithms, the tasks are considered in nondecreasing order of their utilizations.
   - In *unordered (ε)* algorithms, the tasks are considered in arbitrary order. (In contrast to decreasing and increasing algorithms, unordered algorithms do not require that the tasks be sorted by their utilizations prior to allocation).

Different answers to each of these two questions results in a different heuristic; each of the combinations

$$\{\mathbf{FF}, \mathbf{BF}, \mathbf{WF}\} \times \{\mathbf{D}, \mathbf{I}, \varepsilon\}$$

yields a different heuristic, for a total of nine heuristics. They are referred to by two- or three-letter acronyms, in the obvious manner: e.g., FFD is "first fit" with tasks considered in nonincreasing order of utilization, while WF is "worst fit" with the tasks considered in arbitrary order.

Lopez et al. [142] compared these different partitioning algorithms from the perspective of their utilization bounds (Definition 5.1).

We now briefly list some of the results obtained by Lopez et al. [142]. Suppose we are seeking to partition n tasks on an m-processor platform. Let α denote an upper bound on the per-task utilization, and let $\beta \leftarrow \lfloor 1/\alpha \rfloor$. Upper and lower bounds were obtained in [142] on the utilization bounds of any reasonable allocation algorithm.

First, it was shown [142, Theorem 1] that any reasonable allocation algorithm has a utilization bound no smaller than

$$m - (m-1)\alpha. \tag{6.1}$$

This is immediately evident by observing that if a task τ_i with utilization u_i cannot be assigned to any processor, it must be the case where each processor already has been allocated tasks with total utilization strictly greater than $(1 - u_i)$. Summing over all the tasks—those already allocated to the m processors and the task τ_i—we conclude that the total utilization of all the tasks is no smaller than

$$\begin{aligned} & m(1-u_i) + u_i \\ & = m - (m-1)u_i \\ & \geq m - (m-1)\alpha \end{aligned}$$

thereby establishing the lower bound stated in Eq. 6.1.

The following lower bound was also proved in [142, Theorem 2]: No allocation algorithm, reasonable or not, can have a utilization bound larger than

$$\frac{\beta m + 1}{\beta + 1} \tag{6.2}$$

This lower bound was established by considering the task system consisting of the following n tasks, for any $n > \beta m$. There are $(\beta m + 1)$ "heavy" tasks each with utilization $\left(\frac{1}{\beta+1} + \frac{\varepsilon}{n}\right)$, while the remaining $(n - \beta m - 1)$ tasks are "light" with each having utilization $\frac{\varepsilon}{n}$; here, ε denotes some arbitrarily small positive real number. The total utilization of this task system is easily seen to be equal to $\left(\frac{\beta m+1}{\beta+1} + \varepsilon\right)$. To show that it cannot be partitioned upon m processors, observe that since there are $\beta m + 1$ heavy tasks, some processor would need to accommodate at least $\beta + 1$ heavy tasks in any partitioning. But this is not possible, since the total utilization of $\beta + 1$ heavy tasks exceeds one.

The following utilization bounds were shown for specific algorithms in [142]: WF and WFI have the lower bound (Eq. 6.1 above) while FF, BF, FFI, and BFI, as well as all the RAD algorithms considered, have the upper bound (Eq. 6.2 above). In the absence of any bound on the value of α, the largest utilization of any task (other than the obvious bound that it be no larger than one—the capacity of a processor), these bounds may be summarized as follows: when partitioning on a platform comprised of m unit-speed processors,

- WF and WFI have utilization bounds of one, regardless of the number of processors m;
- The remaining seven algorithms—FF, FFI, FFD, BF, BFI, BFD, and WFD—each has a utilization bound equal to $(m + 1)/2$.

Some Observations Observe that all nine partitioning algorithms analyzed by Lopez et al. [142] can be implemented extremely efficiently: sorting n tasks according to utilization takes $O(n \log n)$ time. On an m-processor platform, each of the task-allocation strategies FF, BF, and WF can be implemented in $O(m)$ time per task; WF can in fact be implemented in $O(\log m)$ time per task. Hence, in seeking to determine whether a given task system can be partitioned upon a specified platform by a particular partitioning algorithm, it seems reasonable to actually run the partitioning algorithm, rather than computing the utilization of the task system and comparing against the algorithm's (sufficient, not exact) utilization bound. Thus, from the perspective of actually implementing a real-time system using partitioned scheduling, there is no particular significance to using a utilization bound formula rather than actually trying out the algorithms. Rather, the major benefit to determining these bounds arises from the insight such bounds may provide regarding the efficacy of the algorithm.

6.2 Speedup Factors for Some Common Partitioning Algorithms

Let ε denote an arbitrarily small positive real number and m an arbitrary positive integer, and consider the following two task systems τ and τ'.

- Task system τ consists of $m + 1$ tasks, each of utilization $0.5 + \varepsilon$, to be scheduled on a platform comprised of m unit-capacity processors. This task system has a total utilization equal to $(m + 1) \times (0.5 + \varepsilon)$; this approaches $(m + 1)/2$ from above as $\varepsilon \to 0$.
- Task system τ' consists of $2(m + 1)$ tasks, each of utilization $(0.5 + \varepsilon)/2$, to be scheduled on a platform comprised of m unit-capacity processors. This task system, too, has a total utilization equal to $2(m + 1) \times \left(\frac{0.5+\varepsilon}{2}\right)$, which also approaches $(m + 1)/2$ from above as $\varepsilon \to 0$.

Both τ and τ' have the same utilization (and this utilization is larger than the upper bound of Inequality 6.2 on the utilization bound). Hence, neither τ nor τ' is guaranteed schedulable by any of the algorithms according to the utilization bounds computed in [142]. But it is quite evident that while *no* algorithm could possibly partition τ upon an m-processor platform, τ' is easily partitioned by most algorithms (including the RA and RAD ones studied by Lopez et al. [142]).

In other words, for task system τ one pays the penalty of a utilization loss of $(m - (m + 1)/2)$ *as a consequence of choosing to do partitioned scheduling*, regardless of which particular partitioning algorithm we use. By contrast, there is no similar utilization loss in partitioning τ': Any utilization loss arises from the choice of partitioning algorithm, not merely the decision to go with partitioned (as opposed to global) scheduling.

But the utilization bound metric does not distinguish between the two cases: It is unable to distinguish between task systems of the same utilization that are feasible (can be partitioned by an optimal algorithm) and those that are not.

Consider now the speedup factor metric (Definition 5.2). In the context of partitioned scheduling, the speedup factor of an approximation algorithm A for partitioned scheduling is the smallest number f such that any task system that can be partitioned by an optimal algorithm upon a particular platform can be partitioned by A upon a platform in which each processor is f times as fast.

The speedup factor of a partitioning algorithm quantifies the distance from optimality of the algorithm's resource-usage efficiency: the larger an algorithm's speedup factor, the less efficient its use of processor capacity. (At the extreme, a partitioning algorithm with speedup factor one is an *optimal* partitioning algorithm). Unlike the utilization metric, the speedup factor metric does not penalize an algorithm for not being able to partition a task system that is inherently not partitionable—cannot be partitioned by an optimal algorithm (such as the nonpartitionable task system that was constructed to show the upper bound of Eq. 6.2 on utilization bounds).

Common polynomial-time partitioning algorithms were evaluated and compared in [32] from the perspective of their respective speedup factors. The findings in [32] may be summarized as follows: when implemented upon m-processor platforms

- All RAD partitioning algorithms have a speedup factor of $\left(\frac{4}{3} - \frac{1}{3m}\right)$
- WF and WFI have a speedup factor of $\left(2 - \frac{2}{m}\right)$
- The remaining commonly-considered partitioning algorithms—FF, FFI, BF, and BFI—each have a speedup factor of $\left(2 - \frac{2}{m+1}\right)$

That is, the RAD algorithms have a speedup factor no larger than $4/3$ for all values of m, while the speedup factors of the remaining six algorithms considered in [142] approach 2 as $m \to \infty$. Contrasting to the utilization-bound metric, we note that while the utilization bounds of WF and WFI are less than that of the other algorithms, the remaining seven algorithms have the same utilization bound. Hence, utilization bounds alone cannot explain experimentally observed conclusions (see, in, e.g, [144; Fig. 6]) that FFD appears to be superior to FF or FFI, and BFD appears to be superior to BF or BFI; these findings are however consistent with the differences between the algorithms' speedup factors.

6.3 A PTAS for Partitioning

Hochbaum and Shmoys [108] designed a PTAS for the partitioning of implicit-deadline sporadic task systems[1] that behaves as follows. Given any positive constant ϕ, if an optimal algorithm can partition a given task system τ upon m processors each of speed s, then the algorithm in [108] will, in-time polynomial in the representation of τ, partition τ upon m processors each of speed $(1 + \phi)s$. This can be thought of as a *resource augmentation* result [116]: the algorithm of [108] can partition, in polynomial time, any task system that can be partitioned upon a given platform by an optimal algorithm, provided it (the algorithm of [106]) is given augmented resources (in terms of faster processors) as compared to the resources available to the optimal algorithm.

This is a theoretically significant result since it establishes that task partitioning can be performed to any (constant) desired degree of accuracy in polynomial time. However, the algorithm of [108] has poor implementation efficiency in practice: the constants in the run-time expression for this algorithm are prohibitively large. The ideas in [108] were applied in [68] to obtain an implementation that is efficient enough to often be usable in practice. We describe the implementation in [68] in the remainder of this section.

Overview The main idea behind the approach in [68] is to construct, for each identical multiprocessor platform upon which one will execute implicit-deadline sporadic task systems under partitioned EDF, a *lookup table* (LUT). Whenever a task system is to be partitioned upon this platform, this table is used to determine the assignment of the tasks to the processors.

The LUT is constructed assuming that the utilizations of all the tasks have values from within a fixed set of distinct values V. When this LUT is later used to actually partition of a given task system τ, each task in τ may need to have its worst-case execution time (WCET) parameter *inflated* so that the resulting task utilization is indeed one of these distinct values in V. (The *sustainability* [40] property—see Sect. 3.3—of preemptive uniprocessor EDF ensures that if the tasks with the inflated WCET's are successfully scheduled, then so are the original tasks). The challenge lies in choosing the values in V in such a manner that the amount of such inflation of WCET's that is required is not too large.

It should be evident (we will show this formally) that the larger the number of distinct values in the set V, the smaller the amount of inflation needed. Hence, an important design decision must be made prior to table-construction time: How large a table will we construct? This is expressed in terms of choosing a value for a parameter ε to the procedure that constructs the LUT. Informally speaking, the

[1] Actually, the result in [108] was expressed in terms of minimizing the *makespan*—the duration of the schedule—of a given finite collection of nonpreemptive jobs; however, the makespan minimization problem considered in [108] is easily shown to essentially be equivalent to the problem of partitioning implicit-deadline sporadic task systems.

smaller the value of ε, the smaller the WCET inflation that is needed, and the closer to optimal the resulting task-assignment. However, the size of the LUT and the time required to compute it, also depends on the value of ε: the smaller the value, the larger the table-size (and the amount of time needed to compute it).

Constructing the LUT We will now describe how the LUT is constructed for a multiprocessor platform consisting of m unit-speed processors. The steps involved in constructing the LUT for this platform are:

1. Choosing a value for the parameter ε
2. Based on the value chosen for ε, determining the utilization values that are to be included in the set V. (To make explicit the dependence of this set upon the value chosen for ε, we will henceforth denote this set as $V(\varepsilon)$)
3. Determining the combinations of tasks with utilizations in $V(\varepsilon)$ that can be scheduled together on a single processor
4. Using these single-processor combinations to determine the combinations of tasks with utilizations in $V(\varepsilon)$ that can be scheduled on m processors

Each of these steps is discussed in greater detail below.

Choosing ε We will see later (Theorem 6.2) that the performance guarantee that is made by the partitioning algorithm using the LUT is as follows: any task system that can be partitioned upon m unit-speed processors by an optimal partitioning algorithm will be partitioned by this algorithm on m processors each of speed $(1 + \varepsilon)$. Hence, in choosing a value for ε, we are in effect reducing the guaranteed utilization bound of each processor to equal $1/(1 + \varepsilon)$ times the actual utilization; so the decision in choosing a value for ε essentially becomes: what fraction of the processor capacity are we willing to sacrifice[2], in order to be able to do task partitioning more efficiently? For instance, if we were willing to tolerate a loss of up to 10 % of the processor utilization, ε would need to satisfy the condition

$$\left(\frac{1}{1+\varepsilon} \geq 0.9\right) \Leftrightarrow \left(\varepsilon \leq \frac{1}{0.9} - 1\right) \Leftrightarrow \left(\varepsilon \leq \frac{1}{9}\right).$$

As stated during the overview above, the size of the LUT, and the time required to compute it, also depend on the value of ε: the smaller the value, the larger the table-size (and the amount of time needed to compute it). Hence, ε is assigned the largest value consistent with the desired overall system utilization; in the example above, ε would in fact be assigned the value $1/9$.

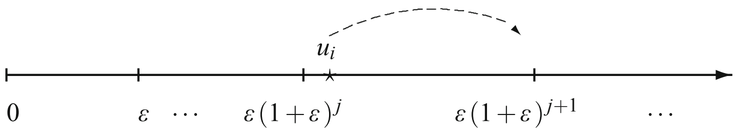
Determining the Utilization Values We next use the value of ε chosen above to determine which utilization values to include in the set $V(\varepsilon)$ of distinct utilization values that will be represented in the LUT we construct. In choosing the members of

[2] We note that this "sacrifice" is only in terms of *worst-case guarantees*, as formalized in Theorem 6.2. It is quite possible that some of this sacrificed capacity can in fact be used during the partitioning of particular task systems.

$V(\varepsilon)$, the objective is to minimize the amount by which the utilizations of the tasks to be partitioned must be *inflated*, in order to become equal to one of the values in $V(\varepsilon)$.

The choice we make is to have $V(\varepsilon)$ equal to the set of all real numbers of the form $\varepsilon \cdot (1 + \varepsilon)^k$, for all nonnegative integers k (up to the upper limit of one).

The rationale is as follows. When the table is used to perform task partitioning, the actual task utilizations will be rounded up to the nearest value present in the set $V(\varepsilon)$. Suppose that an actual utilization u_i is just a bit greater than one of the values present in V, say, $\varepsilon(1 + \varepsilon)^j$—this is depicted in the figure below by a "$\star$".


u_i
0
ε $\cdots$
$\varepsilon(1+\varepsilon)^j$
$\varepsilon(1+\varepsilon)^{j+1}$
$\cdots$

This utilization will be rounded up to $\varepsilon(1+\varepsilon)^{j+1}$; the fraction by which this utilization has been inflated is therefore

$$\frac{\varepsilon(1+\varepsilon)^{j+1}}{u_i} < \frac{\varepsilon(1+\varepsilon)^{j+1}}{\varepsilon(1+\varepsilon)^{j}} = (1+\varepsilon)\,.$$

Thus, if each task's utilization were to be inflated by this maximal factor, it follows that any collection of tasks with total utilization $\leq 1/(1 + \varepsilon)$ would have inflated utilization ≤ 1, and would hence be determined, based on our LUT, to fit on a single processor[3].

Let us now determine $|V(\varepsilon)|$, the number of elements in the set $V(\varepsilon)$. We wish to include each positive real number ≤ 1 that is of the form $\varepsilon(1 + \varepsilon)^j$ for nonnegative j. Since

$$\begin{aligned}
&\varepsilon(1+\varepsilon)^j \leq 1\\
&\Leftrightarrow (1+\varepsilon)^j \leq (1/\varepsilon)\\
&\Leftrightarrow j\log(1+\varepsilon) \leq \log(1/\varepsilon)\\
&\Leftrightarrow j \leq \frac{\log(1/\varepsilon)}{\log(1+\varepsilon)}\,,
\end{aligned}$$

we conclude that

$$|V(\varepsilon)| = \left\lfloor \frac{\log(1/\varepsilon)}{\log(1+\varepsilon)} \right\rfloor + 1. \tag{6.3}$$

[3] Note that this argument does not hold for actual utilizations—the u_i in the figure—less than $\varepsilon/(1+\varepsilon)$. If a task with utilization u_i arbitrarily close to zero ($u_i \to 0^+$) were to have its utilization rounded up to $\varepsilon/(1 + \varepsilon)$, the inflation factor would be $(\varepsilon/(1 + \varepsilon)) \div u_i$, which approaches ∞ as $u_i \to 0$. We will see—Step 3 of the pseudo-code listed in Fig. 6.1—that our task-assignment procedure handles tasks with utilization $< \varepsilon/(1 + \varepsilon)$ differently.

Task system τ, consisting of n implicit-deadline tasks with utilizations $u_1, u_2, \ldots, u_n$, is to be partitioned among m unit-speed processors.

1. For each task with utilization $\geq \varepsilon/(1+\varepsilon)$, *round up* its utilization (if necessary) so that it is equal to $\varepsilon \times (1+\varepsilon)^k$ for some nonnegative integer k. (Observe that such rounding up inflates the utilization of a task by at most a factor $(1+\varepsilon)$: the ratio of the rounded-up utilization to the original utilization of any task is $\leq (1+\varepsilon)$.) Now all the tasks with (original) utilization $\geq \varepsilon/(1+\varepsilon)$ have their utilizations equal to one of the distinct values that were considered during the table-generation step. Let k_i denote the number of tasks with modified utilization equal to $\varepsilon \times (1+\varepsilon)^{i-1}$, for each i, $1 \leq i \leq |V(\varepsilon)|$.
2. Determine whether this collection of modified-utilization tasks can be accommodated in one of the maximal m-processor configurations that had been identified during the preprocessing phase. That is, determine whether there is a maximal multiprocessor configuration

$$\big(\langle y_1, y_2, \ldots, y_{|V(\varepsilon)|}\rangle, \langle z_1, z_2, \ldots, z_m\rangle\big)$$

   in $L_m(\varepsilon)$, satisfying the condition that $y_i \geq k_i$ for each i, $1 \leq i \leq |V(\varepsilon)|$.
   - If the answer here is "no," then report *failure*: we are unable to partition τ among the m processors.
   - If the answer is "yes," however, then *a viable partitioning has been found* for the tasks with (original) utilization $\geq \varepsilon/(1+\varepsilon)$: assign these tasks according to the maximal m-processor configuration.
3. It remains to assign the tasks with utilization $< \varepsilon/(1+\varepsilon)$. Assign each to any processor upon which it will "fit;" i.e., any processor on which the sum of the (original—i.e., unmodified) utilizations of the tasks assigned to the processor would not exceed one if this task were assigned to that processor.
4. If all the tasks with utilization $< \varepsilon/(1+\varepsilon)$ are assigned to processors in this manner, then *a viable partitioning has been found* for all the tasks. However, if some task cannot be assigned in this manner, then report *failure*: we are unable to partition τ among the m processors.

Fig. 6.1 Algorithm for partitioning implicit-deadline sporadic tasks on an identical multiprocessor platform scheduled using preemptive partitioned EDF

Determining Legal Single-Processor Configurations We now seek to determine all the different ways in which a single processor can be packed with tasks that only have utilizations in $V(\varepsilon)$.

We refer to these as *single-processor configurations*.

For reasons of efficiency in storage (and subsequent lookup), we will seek only the maximal configurations of this kind: a single-processor configuration is said to be a *maximal* one if no additional task with utilization $\in V(\varepsilon)$ can be added without the sum of the utilizations exceeding the capacity of the processor:

Definition 6.2 (single-processor configuration) For a given value of ε, a single-processor configuration is a $|V(\varepsilon)|$-tuple

$$\langle x_1, x_2, \ldots, x_{|V(\varepsilon)|} \rangle$$

of nonnegative integers, satisfying the constraint that

$$\left(\sum_{i=1}^{|V(\varepsilon)|} (x_i \cdot \varepsilon \cdot (1+\varepsilon)^{i-1}) \right) \leq 1. \tag{6.4}$$

The single-processor configuration $\langle x_1, x_2, \ldots, x_{|V(\varepsilon)|} \rangle$ is *maximal* if

$$\left(\sum_{i=1}^{|V(\varepsilon)|} (x_i \cdot \varepsilon \cdot (1+\varepsilon)^{i-1}) \right) > (1-\varepsilon) \tag{6.5}$$

(which implies that no task with utilization $\in V(\varepsilon)$ can be added without exceeding the processor's capacity). □
Our objective is to determine a list $L_1(\varepsilon)$ of all possible maximal single-processor configurations for the selected value of ε (here, the subscript "1" denotes the number of processors; we will describe below how we use this to construct the list $L(\varepsilon)$ for m processors).

Since there are only finitely many distinct utilization values (as specified in Eq. 6.3) in $V(\varepsilon)$, all the elements of $L_1(\varepsilon)$ can in principle be determined by exhaustive enumeration: simply try all $|V(\varepsilon)|$-tuples with the i'th component no larger than $(1/(\varepsilon \cdot (1+\varepsilon)^{i-1}))$, adding the ones that satisfy Inequalities 6.4 and 6.5 to $L_1(\varepsilon)$. Such a procedure has run-time exponential in $(1/\varepsilon)$. Although this can be quite high for small ε, this run-time is incurred only once, when the multiprocessor platform is being put together. Once the list $L_1(\varepsilon)$ has been constructed, it can be stored and repeatedly reused for doing task partitioning.

Determining Legal Multiprocessor Configurations We can use the maximal single-processor configurations determined above to determine maximal configurations for a collection of m processors.

Intuitively, each such maximal multiprocessor configuration will represent a different manner in which m processors can be maximally packed with tasks having utilizations in $V(\varepsilon)$.

Definition 6.3 (multiprocessor configuration) For given m and ε, a multiprocessor configuration is an ordered pair of a $|V(\varepsilon)|$-tuple

$$\langle y_1, y_2, \ldots, y_{|V(\varepsilon)|} \rangle$$

of nonnegative integers, and an m-tuple

$$\langle z_1, z_2, \ldots, z_m \rangle$$

of positive integers $\leq |L_i(\varepsilon)|$. The z_j's denote configuration ID's of single-processor configurations (as previously computed, and stored in $L_1(\varepsilon)$); $\langle z_1, z_2, \ldots, z_m \rangle$ thus denotes the m-processor configuration obtained by configuring the j'th processor according to the single-processor configuration represented by ID z_j in $L_i(\varepsilon)$, for $1 \leq j \leq m$.

The tuples $\langle y_1, y_2, \ldots, y_{|V(\varepsilon)|}\rangle$ and $\langle z_1, z_2, \ldots, z_m\rangle$ must satisfy the constraint that for each i, $1 \leq i \leq |V(\varepsilon)|$, the i'th component of the tuples in $L_1(\varepsilon)$ with ID's $\in \langle z_1, z_2, \ldots, z_m\rangle$ sums to exactly y_i.

A multiprocessor configuration $(\langle y_1, y_2, \ldots, y_{|V(\varepsilon)|}\rangle, \langle z_1, z_2, \ldots, z_m\rangle)$ is *maximal* if there is no other multiprocessor configuration $(\langle y'_1, y'_2, \ldots, y'_{|V(\varepsilon)|}\rangle, \langle z'_1, z'_2, \ldots, z'_m\rangle)$ such that $y'_i \geq y_i$ for all i, $1 \leq i \leq |V(\varepsilon)|$. □

Let $L_m(\varepsilon)$ denote the list of all maximal multiprocessor configurations. $L_m(\varepsilon)$ can in principle be determined using exhaustive enumeration: simply consider all m-combinations of the single-processor configurations computed and stored in $L_1(\varepsilon)$. While the worst-case run-time could be as large as $|L_1(\varepsilon)|^m$ and thus once again exponential in ε and m, this step, like the computation of $L_1(\varepsilon)$, also needs to be performed only once during the process of synthesizing the multiprocessor platform. Furthermore, several pragmatic optimizations based upon various counting techniques and programming heuristics that may be used to reduce the run-time in practice are listed in [68]. After it has been computed, $L_m(\varepsilon)$ is stored in a lookup table that is provided along with the m-processor platform, and is used (in a manner discussed below) for partitioning specific task systems upon the platform.

Task Assignment Let τ denote a collection of n implicit-deadline sporadic tasks to be partitioned among the (unit-capacity) processors in the m-processor platform. Let u_i denote the utilization of the i'th task in τ. The task assignment algorithm, depicted in pseudo-code form in Fig. 6.1, operates in two phases

1. In the first phase (Steps 1 and 2 in the pseudo-code), it uses the LUT constructed above to attempt to assign all tasks with utilization $\geq \varepsilon/(1+\varepsilon)$.
2. Next, tasks with utilization $< \varepsilon/(1+\varepsilon)$ are considered during the second phase (Steps 3 and 4 in the pseudo-code). In this phase, the algorithm seeks to accommodate these small-utilization tasks in the remaining capacity remaining upon the processors after phase 1 completes.

Properties We will first show that the partitioning algorithm is *sound*:

Theorem 6.1 *If the partitioning algorithm of Fig. 6.1 succeeds in assigning all the tasks in τ, then the tasks that are assigned to each processor can be scheduled on that processor to meet all deadlines by preemptive uniprocessor* EDF.

Proof The sum of the inflated utilizations of all the tasks assigned on each processor during the first phase does not exceed the capacity of the processor. Hence, the sum of the original (i.e., noninflated) utilizations of tasks assigned to any particular processor does not exceed the capacity of the processor.

This property is preserved during the second phase of the algorithm, since a task is only added to a processor during this phase if the sum of the utilizations after doing so will not exceed the processor's capacity. Hence, if the task-assignment algorithm succeeds in assigning all the tasks to processors, then the sum of the utilizations of the tasks assigned to any particular processor is no larger than one. It follows from the optimality of EDF on preemptive uniprocessor platforms [76, 139] that each processor is consequently successfully scheduled by EDF. $\square$

Next, we show that if the algorithm fails to assign all the tasks in τ to the processors, then no algorithm could have partitioned τ upon an m-processor platform comprised of processors of slightly smaller computing capacity:

Theorem 6.2 *If the partitioning algorithm of Fig. 6.1 fails to partition the tasks in τ, then no algorithm can partition τ on a platform of m processors each of computing capacity* $1/(1+\varepsilon)$.

Proof The partitioning algorithm of Fig. 6.1 may declare failure at two points, one of which is in phase one and the other is in phase two. We consider each possible point of failure separately.

1. Suppose that the algorithm reports failure during phase one, while attempting to assign only the tasks with utilization $\geq \varepsilon/(1+\varepsilon)$ (Step 2 in the pseudo-code). Since each such task has its utilization inflated by a factor $< (1+\varepsilon)$, it must be the case that all such (original—i.e., unmodified-utilization) tasks cannot be scheduled by an optimal algorithm on a platform comprised of m processors each of computing capacity $1/(1+\varepsilon)$. In other words, even just the tasks in τ with unmodified utilizations $\geq \varepsilon$ cannot be partitioned among m processors of computing capacity $1/(1+\varepsilon)$ each, and consequently all of τ clearly cannot be partitioned on such a platform.
2. Suppose that the algorithm reports failure during phase two, while attempting to assign the tasks with utilization $< \varepsilon/(1+\varepsilon)$ (Step 4 in the pseudo-code). This would imply that while some task with utilization $< \varepsilon/(1+\varepsilon)$ remains unallocated to any processor, the sum of the utilizations of the tasks already assigned to each processor is $> (1-\varepsilon/(1+\varepsilon))$. Therefore, the total utilization of τ exceeds $m \times (1-\varepsilon/(1+\varepsilon)) = m(1/(1+\varepsilon))$, and τ cannot consequently be feasible on m processors of computing capacity $(1/(1+\varepsilon))$ each.

The theorem follows. $\square$

Theorems 6.1 and 6.2 together imply that the partitioning algorithm depicted in pseudocode form in Fig. 6.1 is able to partition upon m unit-speed processors any implicit-deadline sporadic task system that can be partitioned by an optimal algorithm upon m speed-$\left(\frac{1}{1+\varepsilon}\right)$ processors.

6.4 Fixed-Task-Priority (FTP) Partitioned Scheduling

We close this chapter with a brief overview of some results concerning partitioned scheduling of implicit-deadline sporadic task systems when we are constrained to using a FTP algorithm for scheduling each individual processor (see Sect. 3.2 for a review of the different restrictions upon priority assignment schemes).

As we had stated in Sect. 4.2, Liu and Layland [139] showed that the FTP priority assignment of ordering task priorities according to period parameters is optimal for implicit-deadline sporadic task systems in the sense that if a task system is schedulable under any FTP scheme, then the task system is schedulable if tasks are ordered according to period parameters, with smaller-period tasks being assigned greater priority. Recall from Sect. 4.2 that this priority order is called rate monotonic (RM), and that the following property of RM was proved in [139]:

Theorem 6.3 *(from [139]) Any implicit-deadline sporadic task system τ comprised of n tasks is schedulable upon a unit-capacity processor by RM if $U_{sum}(\tau) \leq n\,(2^{1/n} - 1)$.*

Oh and Baker [153] applied this uniprocessor utilization bound test to partitioned multiprocessor FTP scheduling using first-fit decreasing utilization (**FFD**, in the notation of Sect. 6.1) assignment of tasks to processors and RM local scheduling on each processor. They proved that any system of implicit-deadline sporadic tasks with total utilization $U < m(\sqrt{2}-1)$ is schedulable by this method on m processors. They also showed that for any $m \geq 2$ there is a task system with $U = (m+1)/(1+2^{1/(m+1)})$ that cannot be scheduled upon m processors using partitioned FTP scheduling. Lopez, Diaz and Garcia [141] refined and generalized this result, and along with other more complex schedulability tests, showed the following.

Theorem 6.4 *(from [141]) Any implicit-deadline sporadic task system τ comprised of n tasks is schedulable upon m unit-capacity processors by a FTP partitioning algorithm (specifically, FFD partitioning, and RM local scheduling) if*

$$U_{sum}(\tau) \leq (n-1)(\sqrt{2}-1) + (m-n+1)(2^{1/(m-n+1)} - 1)$$

6.5 Semi-partitioned FTP Scheduling

In the *semi-partitioned* approach to multiprocessor scheduling, a few tasks are split into "pieces" and the different pieces are assigned to different processors; each processor is then executed during run-time by a uniprocessor scheduler (although some communication between the uniprocessor schedulers upon the different processors is necessary in order to prevent different pieces of a task from executing simultaneously upon different processors). Guan et al. [101, 102] have devised a semi-partitioned algorithm for scheduling implicit-deadline sporadic task systems that has optimal utilization bounds. While we will not be exploring the semi-partitioned approach any further in this book, we consider these results very significant—there are not too many optimal results in multiprocessor scheduling—and direct the interested reader to [101, 102].

Sources

The EDF utilization bounds described in this chapter are to be found in Lopez et al. [141, 142]; the speedup bounds are from [32]. The PTAS described in Sect. 6.3 is from [68]; which applied the results in [108] concerning makespan minimization to come up with the LUT-based approach. Some of the FTP scheduling results are from [22].

Chapter 7
Global Dynamic-Priority Scheduling of L&L Tasks

We now turn our attention to global scheduling: interprocessor migration is permitted. The following result was established by Horn [107]:

Theorem 7.1 (from [107]) *Any implicit-deadline sporadic task system τ satisfying*

$$U_{sum}(\tau) \leq m \text{ and } U_{\max}(\tau) \leq 1$$

is feasible upon a platform comprised of m unit-capacity processors.

To see why this holds, observe that a "processor sharing" schedule, in which each job of each task τ_i is assigned a fraction u_i of a processor between its release time and its deadline, would meet all deadlines—such a processor-sharing schedule may subsequently be converted to one in which each job executes on zero or one processor at each time instant by means of the technique of Coffman and Denning [79; Sect. 3.6]. [1] Of course, such a schedule will see a very large number of preemptions and migrations and is highly unlikely to be implemented in practice; instead, powerful and efficient algorithms based upon the technique of *pfair scheduling* [42] have been developed for performing optimal global scheduling of implicit-deadline sporadic task systems upon multiprocessor platforms. We will study pfair scheduling in the remainder of this chapter.

Notice that task systems τ with $U_{\text{sum}}(\tau) > m$ or $U_{\max}(\tau) > 1$ cannot possibly be feasible upon a platform comprised of m unit-capacity processors; hence,

[1] Here is an informal description of the technique; for further detail, please see [79, p. 116]. Let δ denote an arbitrarily small positive real number, and $[t, t + \delta]$ a time interval during which the processor shares assigned to the tasks do not change. Each task τ_i that is active during this interval needs to execute for an amount $u_i \cdot \delta$. Considering the tasks sequentially in any order, we simply begin assigning them to the first processor starting at time-instant t until we have reached time-instant $t + \delta$, at which point we would "wrap around" and proceed to the next processor beginning again at time-instant t, and so on until all the tasks have been assigned. If a task has only been assigned a part of its execution requirement upon a processor, we assign it the remainder of its execution requirement upon the next processor: the fact that $u_i \leq 1$ for each task guarantees that the allocation to a task upon two consecutive processors will not overlap in time.

S. Baruah et al., *Multiprocessor Scheduling for Real-Time Systems*,
Embedded Systems, DOI 10.1007/978-3-319-08696-5_7

Theorem 7.1 represents an exact test for global feasibility of implicit-deadline task systems, and pfair and related algorithms are therefore optimal.

Recall the classification in Sect. 3.2 of scheduling algorithms into dynamic priority (DP), fixed-job priority (FJP), and fixed-task priority (FTP) ones, according to the restrictions that are placed upon the manner in which scheduling algorithms may assign priorities to jobs. It turns out that all known optimal scheduling algorithms (i.e., those that successfully schedule all task systems satisfying the condition of Theorem 7.1) are DP algorithms. We study such algorithms in the remainder of this chapter.

7.1 Overview

Pfair scheduling is characterized by the fact that tasks are explicitly required to make progress at steady *rates*. Consider a task $\tau_i = (C_i, T_i)$ in which both the parameters C_i and T_i are positive integers, and suppose that a job of this task is released at time-instant t_o. In a pfair schedule, scheduling decisions are made at integer time boundaries; hence, jobs are scheduled for execution an integer unit at a time (t_o, too, is assumed to be an integer). The "fair share" of execution that this job should receive by time-instant t, $t_o \le t \le t_o + T_i$, is $(t - t_o) \times u_i$, and a schedule is said to make "proportionate progress" [42] if it receives exactly $\lfloor (t - t_o) \times u_i \rfloor$ or $\lceil (t - t_o) \times u_i \rceil$ units of execution over $[t_o, t)$, for every integer t, $t_o \le t \le t_o + T_i$. Pfair scheduling algorithms ensure uniform execution rates by breaking jobs into smaller *subjobs*, each with execution requirement 1. Each subjob must execute within a computed *window* of time, the end of which acts as its *pseudo*-deadline. These windows divide each period of a task into subintervals of approximately equal length.

That was a brief informal description of pair scheduling; the remainder of the chapter elaborates on the details.

In Sect. 7.2 we formally define the notion of pfairness, and introduce the notation and terminology we will be using in the remainder of the chapter. We will see here that pfairness is a stronger notion than merely meeting all deadlines: All pfair schedules for implicit-deadline sporadic task systems meet all deadlines, but not all schedules that meet all deadlines are also pfair. In Sect. 7.3 we prove that pfair schedules always exist: it is always possible to schedule an implicit-deadline sporadic task system in a pfair manner. In Sect. 7.4, we derive a scheduling algorithm that schedules any implicit-deadline sporadic task system in a pfair manner.

7.2 Definitions and Notation

Let τ denote an implicit-deadline sporadic task system consisting of the n tasks τ_1, τ_2, τ_n, that is to be scheduled upon an identical multiprocessor platform comprised of m unit-speed processors. All task parameters are positive integers; i.e., $C_i, T_i \in \mathbf{N}$

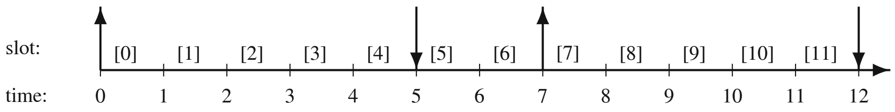


Fig. 7.1 The arrivals and deadlines of the two jobs ($n_1 = 2$) generated by task τ_1, for the example collection of jobs I discussed in Example 7.1

for all $i, 1 \leq i \leq n$. We further assume that total utilization $U_{\text{sum}}(\tau) \leq m$ and the maximum utilization $U_{\max}(\tau) \leq 1$.

Let I denote a legal collection of jobs generated by τ. Let n_i denote the number of jobs generated by τ_i, $1 \leq i \leq n$. Let $r_{i,k}$ denote the release time of the k'th job generated by τ_i for each τ_i and each $k \in [0, n_i)$ (Clearly, it is necessary that $r_{i,k+1} \geq r_{i,k} + T_i$ for all i and all k). Without loss of generality, assume that $\min_{\tau_i \in \tau}\{r_{i,0}\} = 0$, and let $t_{\max}$ denote $\max_{\tau_i \in \tau}\{r_{i,n_i-1} + T_i\}$.

Example 7.1 We will consider the following example throughout the remainder of this chapter. We have a task system τ comprised of the three tasks τ_1 with $C_1 = 2$ and $T_1 = 5$, τ_2 with $C_2 = 3$ and $T_2 = 4$, and τ_3 with $C_3 = 1$ and $T_3 = 3$. We will consider a collection I of eight jobs that are generated by this τ, with

- $n_1 = 2$; job release times $r_{1,0} = 0; r_{1,1} = 7$
- $n_2 = 3$; job release times $r_{2,0} = 0; r_{2,1} = 4; r_{2,2} = 8$
- $n_3 = 3$; job release times $r_{3,0} = 0; r_{3,1} = 3; r_{3,2} = 6$

The arrival-times and deadlines of the two jobs of τ_1 are depicted in Fig. 7.1. For this collection of jobs I, $t_{\max} = \max\{7 + 5, 8 + 4, 6 + 3\} = 12$.
We start with some conventions, definitions, and notations:

- The time interval between time-instants t and $t + 1$ (including t, excluding $t + 1$) will be referred to as *slot* t, $t \in \mathbf{N}$.
- Slot A processor is assigned to a particular task for the entire duration of a slot. Slots are designated in square brackets in Fig. 7.1: $[\ell]$ denotes the slot numbered ℓ, which is the time-interval $[\ell, \ell + 1)$.
- For integers a and b, let $[a, b) = \{a, \ldots, b - 1\}$. Furthermore, let $[a, b] = [a, b + 1)$, $(a, b] = [a + 1, b + 1)$, and $(a, b) = [a + 1, b)$.
- A *schedule* S for the legal collection of jobs I upon an m-processor platform is a function from $\tau \times \mathbf{N}$ to $\{0, 1\}$, where $\sum_{\tau_i \in \tau} S(\tau_i, t) \leq m$, $t \in \mathbf{N}$. Informally, $S(\tau_i, t) = 1$ if and only if task τ_i is scheduled in slot t.
- The *allocation* to a task τ_i at time t with respect to schedule S, denoted $\mathsf{alloc}(S, \tau_i, t)$, denotes the number of slots for which τ_i has been scheduled during the slots numbered $0, 1, 2, \ldots t - 1$. It is easily seen to be given by the following expression:

$$\mathsf{alloc}(S, \tau_i, t) = \textstyle\sum_{i \in [0,t)} S(x, i).$$

- The *lag* of a task τ_i at time t with respect to schedule S, denoted $\mathsf{lag}(S, \tau_i, t)$, is defined as:

$$\mathsf{lag}(S, \tau_i, t) = \sum_{r_{i,k} \le t} u_i \cdot \min(T_i, (t - r_{i,k})) - \mathsf{alloc}(S, \tau_i, t).$$

  Observe that the first term on the right-hand side specifies the amount of execution that task τ_i would receive by time-instant t, in a processor-sharing schedule in which each job of τ_i receives exactly a fraction u_i of a processor throughout the interval between its release time and its deadline. Informally, $\mathsf{lag}(S, \tau_i, t)$ thus measures the difference between the number of units of execution that task τ_i "should" have received in the set of slots $[0, t)$ and the number that it actually received.
- A schedule S is *pfair* if and only if

$$\forall \tau_i, t : \tau_i \in \tau, t \in \mathbf{N} : -1 < \mathsf{lag}(S, \tau_i, t) < 1. \tag{7.1}$$

- A schedule S is *pfair at time t* if and only if there exists a pfair schedule S' such that

$$\forall \tau_i : \tau_i \in \tau : \mathsf{lag}(S, \tau_i, t) = \mathsf{lag}(S', \tau_i, t).$$

Pfairness is a very strict requirement. It demands that the absolute value of the difference between the expected allocation and the actual allocation to every task always be strictly less than 1. In other words, a task never gets an entire slot ahead or behind. In general, it is not possible to guarantee a smaller variation in lag. Consider n identical tasks sharing a single processor, where the utilization of each task is $1/n$, that all have a job arrive simultaneously at time-instant zero. For n sufficiently large, we can make the lag of the first (resp., last) task scheduled become arbitrarily close to -1 (resp., 1).

7.3 The Existence of Pfair Schedules

In this section we will use a network flow argument to prove the existence of a pfair schedule upon m unit-speed processors, for the collection of jobs I defined above. In the next section, we will derive a run-time scheduling algorithm that generates such a schedule.

Let $\mathsf{earliest}(\tau_i, j)$ (resp., $\mathsf{latest}(\tau_i, j)$) denote the earliest (resp., latest) slot ($\mathsf{earliest}(\tau_i, t)$) ($\mathsf{latest}(\tau_i, t)$) during which task τ_i may be scheduled for the jth time, $j \in \mathbf{N}$, in any pfair schedule for the collection of jobs I of τ.

We can easily derive closed form expressions for $\mathsf{earliest}(\tau_i, j)$ and $\mathsf{latest}(\tau_i, j)$. Consider the ℓ'th time the k'th job of τ_i is being scheduled, $0 \le k < n_i$ and $0 \le \ell < C_i$. It follows from the definitions of $\mathsf{earliest}(\tau_i, j)$ and $\mathsf{latest}(\tau_i, j)$ that $\mathsf{earliest}(\tau_i, k \cdot C_i + \ell) = \min t : t \in \mathbf{N} : u_i \cdot (t + 1 - r_{i,k}) - (\ell + 1) > -1$ and

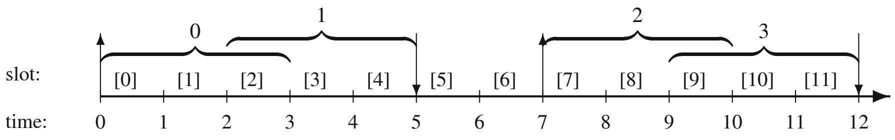


Fig. 7.2 The ranges of slots over which τ_1 may be scheduled for the zeroth, first, second, and third times, for the example collection of jobs I discussed in Example 7.1. Observe that successive ranges overlap by at most one slot

$\mathsf{latest}(\tau_i, k \cdot C_i + \ell) = \max t : t \in \mathbf{N} : u_i \cdot (t - r_{i,k}) - \ell < 1$. Hence, for all $k \in [0, n_i)$ and all $j \in [0, C_i)$,

$$\mathsf{earliest}(\tau_i, k \cdot C_i + \ell) = r_{i,k} + \left\lfloor \frac{\ell}{u_i} \right\rfloor, \text{and}$$

$$\mathsf{latest}(\tau_i, k \cdot C_i + \ell) = r_{i,k} + \left\lceil \frac{\ell + 1}{u_i} \right\rceil - 1$$

Note that $\mathsf{earliest}(\tau_i, j) \le \mathsf{latest}(\tau_i, j)$ for all $\tau_i \in \tau$, $j \in \mathbf{N}$. Note, too, that $\mathsf{earliest}(\tau_i, j + 1) - \mathsf{latest}(\tau_i, j) \ge 0$; i.e., *the range of slots during which a task may be scheduled for the j'th and the $(j + 1)$'th time in any pfair schedule overlap by at most one slot.*

Example 7.2 Consider once again the legal collection of jobs I introduced in Example 7.1. Instantiating $k \leftarrow 1$ $\ell \leftarrow 0$, we compute $\mathsf{earliest}(\tau_1, 1 \cdot 2 + 0) = \mathsf{earliest}(\tau_1, 2)$ to be

$$r_{1,1} + \left\lfloor \frac{0}{2/5} \right\rfloor = 7 + 0 = 7,$$

and $\mathsf{latest}(\tau_1, 1 \cdot 2 + 0) = \mathsf{latest}(\tau_1, 2)$ to be

$$r_{1,1} + \left\lceil \frac{1}{2/5} \right\rceil - 1 = 7 + 3 - 1 = 9.$$

In a similar vein, we can compute $\mathsf{earliest}(\tau_1, 0) = 0$, $\mathsf{earliest}(\tau_1, 1) = 2$, and $\mathsf{earliest}(\tau_1, 3) = 9$; and $\mathsf{latest}(\tau_1, 0) = 2$, $\mathsf{latest}(\tau_1, 1) = 4$, and $\mathsf{latest}(\tau_1, 3) = 11$. The ranges of slots for which task τ_i may be scheduled for the zeroth, first, second, and third times are depicted graphically in Fig. 7.2.

The remainder of this section is devoted to proving the existence of a pfair schedule for the collection of jobs I described above that are generated by the task system τ, upon m processors. Our strategy is as follows: First, we will describe a reduction from this collection of jobs to a weighted digraph G with a designated source and sink, such that certain flows in G correspond exactly (in a manner that will be made precise) to a pfair schedule for the collection of jobs I. Then we will prove the existence of such a flow in G.

We now describe below the construction of a weighted digraph G based upon the collection of jobs I. The graph is a "layered" one: The vertex set V of G is the union of six disjoint sets of vertices $V_0, \ldots, V_5$, and the edge set E of G is the union of five disjoint sets of edges $E_0, \ldots, E_4$, where E_i is a subset of $(V_i \times V_{i+1} \times \mathbf{N})$, $0 \le i \le 4$. G is thus a six-layered graph, in which all edges connect vertices in adjacent layers. The sets of vertices in G are as follows:

$$\begin{aligned}
V_0 &= \{\textsf{source}\},\\
V_1 &= \{\langle 1, \tau_i\rangle \mid \tau_i \in \tau)\},\\
V_2 &= \{\langle 2, \tau_i, j\rangle \mid \tau_i \in \tau, j \in [0, n_i \cdot C_i)\},\\
V_3 &= \{\langle 3, \tau_i, t\rangle \mid \tau_i \in \tau, t \in [0, t_{\max})\},\\
V_4 &= \{\langle 4, t\rangle \mid t \in [0, t_{\max})\}, \text{and}\\
V_5 &= \{\textsf{sink}\}.
\end{aligned}$$

An edge is represented by a 3-tuple. For $u, v \in V$ and $w \in \mathbf{N}$, the 3-tuple $(u, v, w) \in E$ represents an edge from u to v that has a capacity w. The sets of edges in G are as follows:

$$\begin{aligned}
E_0 &= \{(\textsf{source}, \langle 1, \tau_i\rangle, n_i \cdot C_i) \mid \tau_i \in \tau\},\\
E_1 &= \{(\langle 1, \tau_i\rangle, \langle 2, \tau_i, j\rangle, 1) \mid \tau_i \in \tau, j \in [0, n_i \cdot C_i)\},\\
E_2 &= \{(\langle 2, \tau_i, j\rangle, \langle 3, \tau_i, t\rangle, 1) \mid \tau_i \in \tau, j \in [0, C_i \cdot n_i), t \in [\textsf{earliest}(\tau_i, j),\\
&\qquad \textsf{latest}(\tau_i, j)]\},\\
E_3 &= \{(\langle 3, \tau_i, t\rangle, \langle 4, t\rangle, 1) \mid \tau_i \in \tau, t \in [0, t_{\max})\}, \text{and}\\
E_4 &= \{(\langle 4, t\rangle, \textsf{sink}, m) \mid t \in [0, t_{\max})\}.
\end{aligned}$$

Example 7.3 Figure 7.3 shows (a portion of) the digraph constructed for the example collection of jobs I described in Example 7.1. In this figure, we have shown all the vertices in V_0, V_1, V_4, and V_5, and only those vertices from V_2 and V_3 which are pertinent to the task τ_1; corresponding vertices pertaining to tasks τ_2 and τ_3 have been omitted from the diagram to keep things simple. Also, only edges between vertices that are depicted in the figure are shown (hence, not shown in the figure are nine outgoing edges from vertex $\langle 1, \tau_2\rangle$, and three outgoing edges from vertex $\langle 1, \tau_3\rangle$, and an additional two incoming edges into each vertex $\langle 4, t\rangle$).

Lemma 7.1 *If there is an integral flow of size $\sum_{\tau_i \in \tau} (n_i \cdot C_i)$ in G, then there exists a pfair schedule for the collection of jobs I.*

Proof Suppose there is an integral flow of size $\sum_{\tau_i \in \tau} (n_i \cdot C_i)$ in G. The total capacity of E_0, the set of edges leading out of the $\textsf{source}$ vertex, is equal to $\sum_{\tau_i \in \tau} (n_i \cdot C_i)$. Hence, each edge in E_0 is filled to capacity, and each vertex $\langle 1, \tau_i\rangle$ receives exactly $n_i \cdot C_i$ units of flow. As there are $n_i C_i$ vertices in V_2 each connected (by an edge of unit capacity) to vertex $\langle 1, \tau_i\rangle$, and no two vertices in V_1 are connected to the same

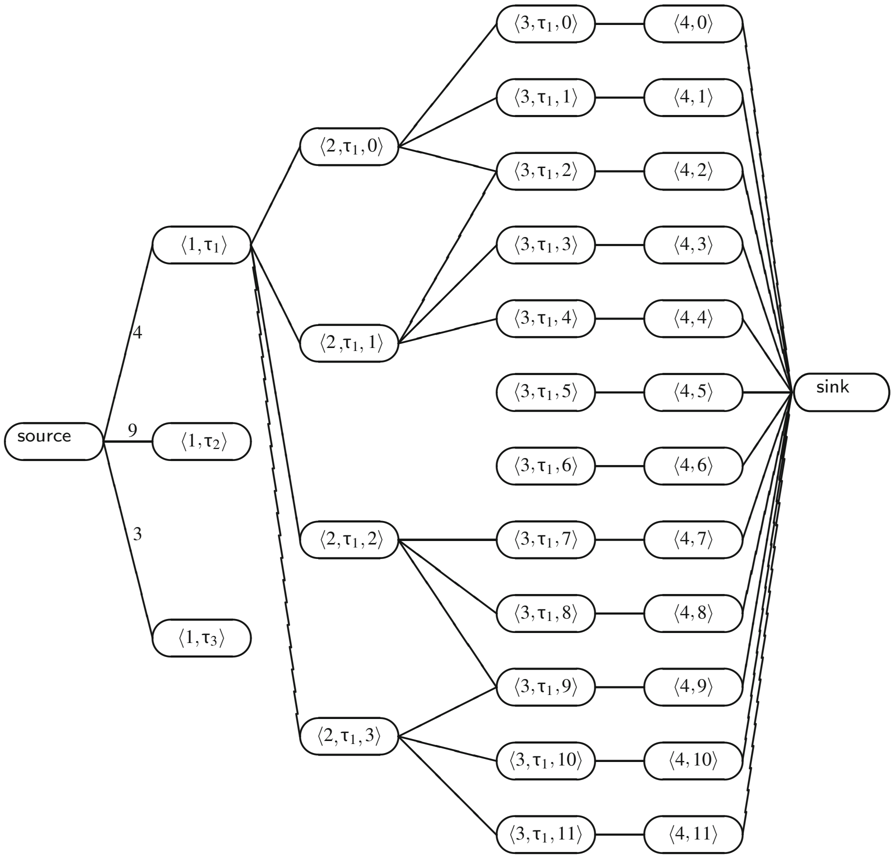


Fig. 7.3 Digraph construction illustrated. All edges flow let to right; edges in E_0 have capacities as labeled; each edge in E_5 has a capacity equal to the number of processors m; the remaining edges (those without labeled capacities in the figure) all have unit capacity. Only the vertices (and edges) pertinent to task τ_1 are depicted in layers V_3 and V_4

vertex in V_2, it follows that each vertex in V_2 receives a unit flow. Accordingly, each vertex in V_2 sends a unit flow to some vertex in V_3.

We will construct the desired schedule S from the given flow according to the following rule: Allocate a processor to task τ_i in slot t if and only if there is a unit flow from vertex $\langle 2, \tau_i, j\rangle$ to vertex $\langle 3, \tau_i, t\rangle$.

Each of the $t_{\max}$ edges of capacity m in E_4 carries at most m units of flow. Hence, for all $t \in [0, t_{\max})$, vertex $\langle 4, t\rangle$ receives at most m unit flows from vertices in V_3. Each vertex $\langle 3, \tau_i, t\rangle$ in V_3 is connected (by an edge of unit capacity) to vertex $\langle 4, t\rangle$, and is not connected to any other vertex in V_4. Thus, S schedules at most m tasks in each time slot t, for all $t \in [0, t_{\max})$. To see that no lag constraints are violated by S, observe that for each task τ_i and for all $j \in [0, n_i \cdot C_i)$, the jth scheduling of task

x occurs at a slot in the interval $[\mathsf{earliest}(\tau_i, j), \mathsf{latest}(\tau_i, j)]$ (the jth scheduling corresponds to the unique unit flow out of vertex $\langle 2, \tau_i, j\rangle$). □

Further, we will demonstrate the existence of an integral flow; in conjunction with Lemma 7.1 above, this will establish the existence of a pair schedule for this collection of jobs I.

Since all edges of the graph have integral capacity, it follows from the integer flow theorem of Ford and Fulkerson [90] that if there is a fractional flow of size $\sum_{\tau_i \in \tau}(n_i C_i)$ in the graph then there is an integral flow of that size. We will now show that a fractional flow of size $\sum_{\tau_i \in \tau}(n_i C_i)$ exists. We use the following flow assignments:

- Each edge $(\mathsf{source}, \langle 1, \tau_i\rangle, n_i C_i) \in E_0$ carries a flow of $n_i C_i$.
- Each edge $(\langle 1, \tau_i\rangle, \langle 2, \tau_i, j\rangle, 1) \in E_1$ carries a flow of size 1.
- Each edge $(\langle 3, \tau_i, t\rangle, \langle 4, t\rangle, 1) \in E_3$ carries a flow of size $\leq u_i$.
- Each edge $(\langle 4, t\rangle, \mathsf{sink}, m) \in E_4$ carries a flow of size $\leq U_{\text{sum}}(\tau)$ (and hence $\leq m$).
- The flows through edges in E_2 are determined as follows. For each $0 \leq k < n_i$ and $0 \leq \ell < C_i$, consider the ℓ'th time the k'th job of τ_i is being scheduled. Consider the edges emanating from the vertex $\langle 2, \tau_i, k \cdot C_i + \ell\rangle$.
  - The edge $(\langle 2, \tau_i, k \cdot C_i + \ell\rangle, \langle 3, \tau_i, \mathsf{earliest}(\tau_i, k \cdot C_i + \ell)\rangle, 1)$ carries a flow of size
  $$u_i - (\ell - u_i \cdot \lfloor \ell / u_i \rfloor),$$
  which is no larger than 1, the capacity of the edge.
  - For each $t \in [\mathsf{earliest}(\tau_i, k \cdot C_i + \ell), \mathsf{latest}(\tau_i, k \cdot C_i + \ell))$, the edge $(\langle 2, \tau_i, k \cdot C_i + \ell\rangle, \langle 3, \tau_i, t\rangle, 1)$ carries a flow of size u_i.
  - If $\mathsf{latest}(\tau_i, k \cdot C_i + \ell) = \mathsf{earliest}(\tau_i, k \cdot C_i + \ell + 1)$, then edge $(\langle 2, \tau_i, k \cdot C_i + \ell\rangle, \langle 3, \tau_i, \mathsf{latest}(\tau_i, k \cdot C_i + \ell)\rangle, 1)$ carries a flow of size
  $$(\ell + 1) - u_i \cdot \lfloor (\ell + 1)/u_i \rfloor,$$
  else, this edge $(\langle 2, \tau_i, k \cdot C_i + \ell\rangle, \langle 3, \tau_i, \mathsf{latest}(\tau_i, k \cdot C_i + \ell)\rangle, 1)$ carries a flow of size u_i. In either case, the flow is not larger than 1, the capacity of the edge.

Example 7.4 Let us consider the edges in E_2 that are depicted in Fig. 7.3.

- Consider first the edges coming out of $\langle 2, \tau_1, 0\rangle$. The top two edges—i.e., the edges $(\langle 2, \tau_1, 0\rangle, \langle 3, \tau_1, 0\rangle, 1)$ and $(\langle 2, \tau_1, 0\rangle, \langle 3, \tau_1, 1\rangle, 1)$—each carry $u_1 = 0.4$ units of flow. The edge $(\langle 2, \tau_1, 0\rangle, \langle 3, \tau_1, 2\rangle, 1)$ carries the remaining 0.2 units of flow.
- Consider next the edges coming out of $\langle 2, \tau_1, 1\rangle$. The top edge—the edge $(\langle 2, \tau_1, 1\rangle, \langle 3, \tau_1, 2\rangle, 1)$ carries 0.2 unit of flows, thereby ensuring that the total in-flow to vertex $\langle 3, \tau_1, 2\rangle$ is also $u_1 = 0.4$. The remaining two edges—$(\langle 2, \tau_1, 1\rangle, \langle 3, \tau_1, 3\rangle, 1)$ and $(\langle 2, \tau_1, 1\rangle, \langle 3, \tau_1, 4\rangle, 1)$—each carry $u_1 = 0.4$ units of flow.
- The edges coming out of $\langle 2, \tau_1, 2\rangle$ are analogous to those coming out of $\langle 2, \tau_1, 0\rangle$: the top two edges each carry 0.4 units of flow, while the third edge carries 0.2 units of flow. The edges coming out of $\langle 2, \tau_1, 3\rangle$ are similarly analogous to those

coming out of $\langle 2, \tau_1, 1\rangle$: the top edge carries 0.2 units of flow, while the other two edges each carry 0.4 units of flow.

Observe that in this example, the incoming flow to each vertex $\langle 3, \tau_1, t\rangle$ is either u_1 or zero. This is also the outgoing flow along each depicted edge of E_3.

We will now prove that the flow just defined is a valid flow of size $\sum_{\tau_i \in \tau} (n_i C_i)$. The capacity constraints have been met. The flow out of the source vertex is $\sum_{\tau_i \in \tau} (n_i C_i)$. We will now complete the proof by showing that flow is conserved at every interior vertex.

The flow into each vertex in V_1 is $n_i C_i$, and there are $n_i C_i$ edges leaving, each carrying a unit flow. The flow into each vertex in V_2 is 1. Further we will prove that the flow out of each vertex in V_2 is 1, and that the flow into each vertex in V_3 is either zero or u_i. Each vertex in V_3 has only one outgoing edge carrying a flow of u_i. Each vertex in V_4 has n incoming edges each carrying a flow size $\leq u_i$; since $\sum_{\tau_i \in \tau} u_i \leq m$, the flow in is $\leq m$, which can be accommodated on the one outgoing edge of capacity m.

It remains to prove that: (i) the flow out of each vertex in V_2 is 1, and (ii) the flow into each vertex in V_3 is $\leq u_i$. For each $0 \leq k < n_i$ and $0 \leq \ell < C_i$, consider the ℓ'th time the k'th job of τ_i is being scheduled. Consider the edges emanating from the vertex $\langle 2, \tau_i, k \cdot C_i + \ell\rangle$.

For (i), consider an arbitrary vertex $\langle 2, \tau_i, k \cdot C_i + \ell\rangle$ in V_2. There are latest$(\tau_i, k \cdot C_i + \ell)-$earliest$(\tau_i, k \cdot C_i + \ell)+1$, or $\lceil(\ell + 1)/u_i\rceil - \lfloor \ell/u_i \rfloor$, outgoing edges from $\langle 2, \tau_i, k \cdot C_i + \ell\rangle$. If earliest$(\tau_i, k \cdot C_i + \ell + 1) =$ latest$(\tau_i, k \cdot C_i + \ell)$ (equivalently, $\lceil(\ell + 1)/u_i\rceil - 1 = \lfloor(\ell + 1)/u_i\rfloor$), then the flow out of $\langle 2, \tau_i, k \cdot C_i + \ell\rangle$ is

$$\begin{aligned} &u_i - (j - u_i \cdot \lfloor \ell/u_i \rfloor) + u_i \cdot (\lceil(\ell + 1)/u_i\rceil - \lfloor \ell/u_i \rfloor - 2) \\ &+ (\ell + 1) - u_i \cdot \lfloor(\ell + 1)/u_i\rfloor, \end{aligned}$$

which simplifies to 1. Otherwise, earliest$(\tau_i, k \cdot C_i + \ell + 1) =$ latest$(\tau_i, k \cdot C_i + \ell)$ $+ 1$ (equivalently, $\lceil(\ell + 1)/u_i\rceil = \lfloor(\ell + 1)/u_i\rfloor = (\ell + 1)/u_i$), and the flow out of $\langle 2, \tau_i, k \cdot C_i + \ell\rangle$ is

$$u_i - (\ell - u_i \cdot \lfloor \ell/u_i \rfloor) + u_i \cdot (\lceil(\ell + 1)/u_i\rceil - \lfloor \ell/u_i \rfloor - 1),$$

which also simplifies to 1.

For (ii), consider an arbitrary vertex $\langle 3, \tau_i, t\rangle$ in V_3. If there are no incoming edges into this vertex, then the flow into this vertex is zero, which is $\leq u_i$. Otherwise, if $t =$ latest$(\tau_i, k \cdot C_i + \ell) =$ earliest$(\tau_i, k \cdot C_i + \ell + 1)$, then there are two incoming edges to $\langle 3, \tau_i, t\rangle$, namely $(\langle 2, \tau_i, k \cdot C_i + \ell\rangle, \langle 3, \tau_i, k \cdot C_i + \ell\rangle, 1)$ and $(\langle 2, \tau_i, k \cdot C_i + \ell + 1\rangle, \langle 3, \tau_i, t\rangle, 1)$. These edges carry flows of size $(\ell + 1) - u_i \cdot \lfloor(\ell + 1)/u_i\rfloor$ and $u_i - ((\ell + 1) - u_i \cdot \lfloor(\ell + 1)/u_i\rfloor)$, respectively, for a total incoming flow of u_i. Otherwise, there is only one incoming edge to $\langle 3, \tau_i, t\rangle$, and it carries a flow of u_i.

We demonstrated above, by construction of a fractional flow and an appeal to the integer flow theorem [90], the existence of an integral flow in the digraph; in conjunction with Lemma 7.1 above, this establishes the existence of a pair schedule for the collection of jobs I that was generated by the implicit-deadline sporadic task system τ.

7.4 A Pfair Scheduling Algorithm

The digraph construction above serves to establish that the collection of jobs I, generated by the implicit-deadline sporadic task system τ, has a pfair schedule. However, note that constructing the graph requires us to know beforehand the time-instants $r_{i,j}$ (the precise time-instants at which job arrivals occur), and thus constructing the scheduled prior to run-time requires clairvoyance. In this section, we will derive a non-clairvoyant online scheduling algorithm for scheduling the implicit-deadline sporadic task system τ upon an m-processor platform, and prove that it produces a pfair schedule for any collection of jobs that are legally generated by τ.

We start with some definitions. Let S denote a schedule that is pfair at time-instant t.

- Task τ_i is *contending* in schedule S at time-instant t if and only if

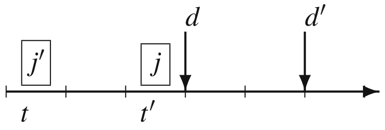
$$\mathsf{earliest}(\tau_i, \mathsf{alloc}(S, \tau_i, t) + 1) \leq t.$$

  That is, a task that is contending during the t'th time-slot in a schedule that is pfair at time-instant t may be allocated the processor during the t'th time-slot without violating pfairness.
- Task τ_i is *urgent* in schedule S at time-instant t if and only if

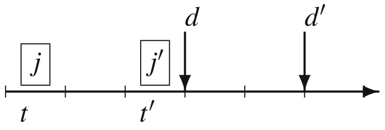
$$\mathsf{latest}(\tau_i, \mathsf{alloc}(S, \tau_i, t) + 1) = t.$$

  That is, an urgent task that is not allocated the processor during the t'th time-slot will violate pfairness.
- For any $p \in \mathbf{N}^+$, the p'th *pseudo-deadline* of contending task τ_i in schedule S at time-instant t is the latest slot by which τ_i must be allocated the processor for p more time-slots in order to not violate pfairness. (The first pseudo-deadline is thus the latest slot by which the the task next must be scheduled, in order to not violate pfairness). It is evident that the p'th pseudo-deadline of task τ_i is equal to $\mathsf{latest}(\tau_i, \mathsf{alloc}(\tau_i, S, t) + p)$.

Uniprocessor EDF: A Brief Digression In uniprocessor EDF, ties amongst jobs that have equal deadlines may be broken arbitrarily: if two jobs that are eligible to execute have the same deadline, we may choose to execute either one. Recall the following inductive argument in Sect. 4.1 establishing the optimality of preemptive uniprocessor EDF. Suppose that there is an optimal schedule that meets all deadlines of all the jobs, and suppose that the scheduling decisions made by EDF are identical to those made by this optimal schedule over the time interval $[0, t)$. At time-instant t, EDF observes that job j, with deadline d, is an earliest-deadline job needing execution, and schedules it during slot t. However, the optimal schedule schedules some other job j', with deadline d'. The critical observation is that since $d \le d'$ (EDF, by definition, chooses a job with the earliest deadline) and the optimal schedule also schedules j to complete by its deadline, it must be the case that j is executed in the optimal schedule during some slot $t', t < t' < d$. We may therefore simply swap the allocations in the optimal schedule during slots t and t', thereby obtaining an optimal schedule that agrees with the decisions made by EDF over the interval $[0, t+1)$:



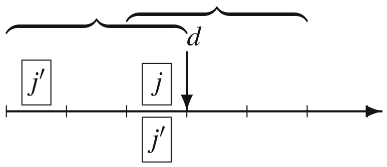
AN OPTIMAL SCHEDULE



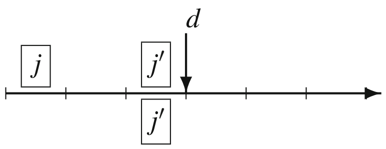
ANOTHER OPTIMAL SCHEDULE

The optimality of EDF follows, by induction on t.

Algorithm PF Let us now consider multiprocessors. We have established the existence of an optimal (in this case, pfair) schedule for the collection of jobs I via the digraph construction. However, the swapping argument used above for uniprocessor EDF does not work to show that scheduling according to the obvious generalization of EDF—comparing tasks according to their first pseudo-deadlines (i.e., when they would violate pfairness if not scheduled)—results in a pfair schedule. In particular, this argument fails to go through if the deadlines are *tied*, for the following reason. Suppose that the jobs j and j' have the same deadline—i.e., $d' = d$); EDF (or rather, the variant of EDF that seeks to generate a pfair schedule, and therefore compares the first pseudo-deadlines) breaks the tie in favor of j, but the optimal schedule breaks the tie in favor of j'. Recall that the range of slots during which j' may be scheduled for the ℓ'th and the $(\ell + 1)$'th time in a pfair schedule may overlap by up to one slot; hence, it is possible that the optimal schedule scheduled j in the last slot prior to its deadline (the slot $d - 1$), and scheduled j' for its *next* allocation in the same slot upon a different processor. In this case, swapping the allocations to j and j', as we had done in the uniprocessor case above, would result in an *illegal* schedule rather than another optimal one, since the slot $d - 1$ would now have j' concurrently scheduled upon two processors:



AN OPTIMAL SCHEDULE



ILLEGAL!!

It can be seen that this problem can only occur if two tasks have the same pseudo-deadline, *and* if successive scheduling "windows" for the tasks overlap by one slot. In Fig. 7.4, we therefore define an ordering $\succeq$ amongst tasks that does not break ties arbitrarily; rather, if corresponding pseudo-deadlines are equal and both tasks'

```
$\succeq(\tau_i, \tau_j)$
    $\triangleright$ $\tau_i$ and $\tau_j$ are both contending tasks. Without loss if generality, assume $i < j$
 1  if $\tau_i$ is urgent then **return** $\tau_i \succeq \tau_j$
 2  if $\tau_j$ is urgent then **return** $\tau_j \succeq \tau_i$
 3  $\ell_i \leftarrow \mathsf{alloc}(\tau_i, S, t) + 1$
 4  $\ell_j \leftarrow \mathsf{alloc}(\tau_j, S^{\text{`}} t) + 1$
    $\triangleright$ Compare the pseudo-deadlines of $\tau_i$ and $\tau_j$
 5  **if** $(\mathsf{latest}(\tau_j, \ell_j) < \mathsf{latest}(\tau_i, \ell_i))$ **then return** $\tau_j \succeq \tau_i$
 6  **if** $(\mathsf{latest}(\tau_i, \ell_i) < \mathsf{latest}(\tau_j, \ell_j))$ **then return** $\tau_i \succeq \tau_j$
    $\triangleright$ Otherwise, the pseudo-deadlines $\mathsf{latest}(\tau_i, \ell_i)$ and $\mathsf{latest}(\tau_j, \ell_j)$ are equal
 7  **if** $(\ell_j = n_j C_j - 1)$ **or** $\mathsf{earliest}(\tau_j, \ell_j + 1) \geq \mathsf{latest}(\tau_j, \ell_j) + 1)$ **then return** $\tau_i \succeq \tau_j$
 8  **if** $(\ell_i = n_i C_i - 1)$ **or** $\mathsf{earliest}(\tau_i, \ell_i + 1) \geq \mathsf{latest}(\tau_i, \ell_j) + 1)$ **then return** $\tau_j \succeq \tau_i$
    $\triangleright$ Otherwise, compare the *next* pair of pseudo-deadlines
 9  $\ell_i \leftarrow \ell_i + 1$
10  $\ell_j \leftarrow \ell_j + 1$
11  **go to** Line 5
```

Fig. 7.4 Comparing pseudo-deadlines

scheduling windows overlap, it compares the *next* pair of pseudo-deadlines, repeating the process until a tie-breaker is found.

Scheduling algorithm *PF* schedules, during each time-slot, the (up to) m contending tasks with greatest priority according to the $\succeq$ relationship defined in Fig. 7.4.

We now define (Figure 7.4) a total priority ordering $\succeq$ on the contending tasks as follows. An urgent task is $\succeq$ any contending task. (If τ_i and τ_j are both contending, we may break ties arbitrarily; let us do so in favor of the smaller-indexed task). Otherwise, we repeatedly compare corresponding pseudo-deadlines of the two tasks, according to the following rule:

- A task for which the pseudo-deadline is strictly smaller than the earliest slot at which the *next* allocation may be made to the task has lower priority according to this $\succeq$ relationship. (If this is true for both the tasks, we break ties in favor of the smaller-indexed one).

Until one task is deemed to be of greater priority than the other. Specific examples of such comparisons are depicted graphically in Figs. 7.5 and 7.6.

Computational Complexity It is evident that at most $\min(C_i, C_j)$ comparisons of pseudo-deadlines are necessary[2] in order to determine the $\succeq$ ordering amongst contending tasks τ_i and τ_j, yielding an over-all complexity that is pseudo-polynomial in the representation of the task system. Thus, a naive implementation of algorithm PF would have a run-time per slot that is pseudo-polynomial in the representation of the task system. A more efficient implementation was derived in [42], in which

[2] Actually, the potentially tighter bound of $\min(C_i/\gcd(C_i, T_i), C_j/\gcd(C_j, T_j))$ on the number of comparisons is easily seen to hold.

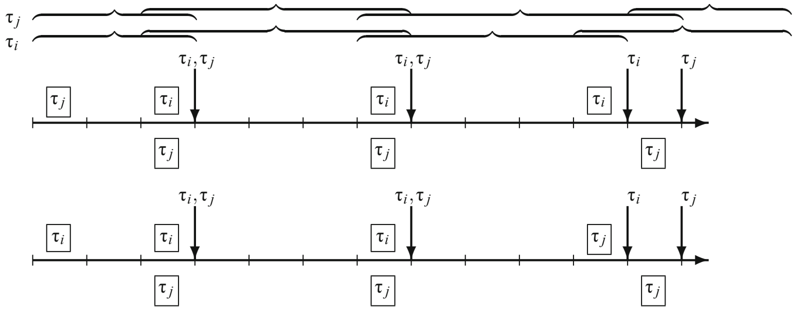


Fig. 7.5 The first two pseudo-deadlines of τ_i and τ_j coincide, but the third pseudo-deadline of τ_i is strictly smaller than the third pseudo-deadline of τ_j. (For these particular allocations to both τ_i and τ_j, successive scheduling windows overlap by one time-slot). Hence, $\tau_i \succeq \tau_j$. The *top* figure depicts a pfair schedule in which τ_j is scheduled at its earliest slots. Swapping the first allocation to τ_j with the last allocation to τ_i results in the pfair schedule in the *bottom* figure

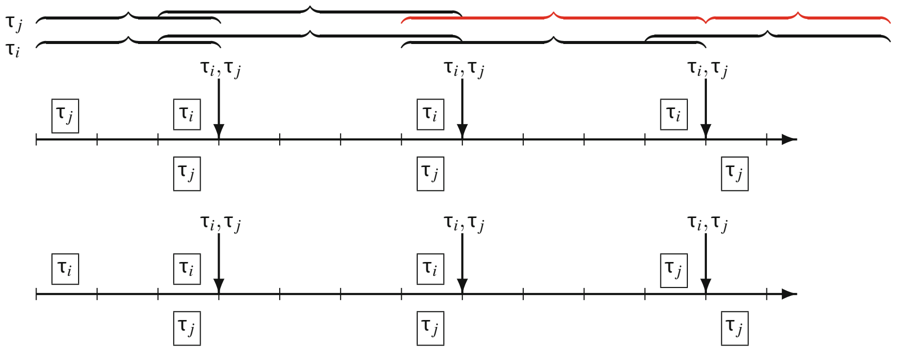


Fig. 7.6 The first two pseudo-deadlines of τ_i and τ_j coincide, and successive scheduling windows overlap by one time-slot. The third pseudo-deadline of τ_j coincides with the third pseudo-deadline of τ_i, but the third and fourth scheduling windows of τ_j do not overlap. Hence, $\tau_i \succeq \tau_j$. The *top* figure depicts a pfair schedule in which τ_j is scheduled at its earliest slots. Swapping the first allocation to τ_j with the last allocation to τ_i results in the pfair schedule in the *bottom* figure

the per-slot run-time complexity is linear in the representation of the task system—to be precise, the per-slot run-time complexity when scheduling τ was shown to be $O(\sum_{\tau_i \in \tau} \lceil \log(T_i + 1) \rceil)$. Further improvements have since been proposed; for instance, [48] proposed an algorithm called PD with a per-slot run-time that is $O(\min(m \log n, n))$. Extensions/modifications to pfairness have also since been proposed; e.g., [173] defined a variant called boundary fair (BF) that only enforces the lag constraint (Condition 7.1) at the boundaries of jobs rather than for each individual unit-sized subjob and thereby potentially reduces the number of preemptions and migrations. A generalized framework for considering pfairness and such related concepts called DP-FAIR was recently proposed in [134], and a DP-FAIR scheduling algorithm called DP-WRAP was derived there. The *R*eduction to *UN*iprocessor

(RUN) algorithm [159] is another efficient implementation that seeks to progressively transform a multiprocessor scheduling problem to a series of uniprocessor problems; this, too, has the effect of potentially reducing the number of preemptions and migrations.

Sources

The techniques and results described in this chapter were reported in [42, 53]. There is a large body of additional work on pfair scheduling that we have not reported here; see, e.g., the dissertation by Srinivasan [165].

Chapter 8
Global Fixed-Job-Priority Scheduling of L&L Tasks

Recall the classification in Sect. 3.2 of scheduling algorithms into dynamic priority (DP), fixed-job priority (FJP), and fixed-task priority (FTP) ones, according to the restrictions that are placed upon the manner in which scheduling algorithms may assign priorities to jobs. This chapter is devoted to FJP scheduling.

We saw in Chap. 7 that pfair scheduling algorithms are able to schedule implicit-deadline task systems optimally. They do so by breaking each job into a number of unit-sized subjobs, and scheduling each subjob separately: intuitively, it is easier to schedule uniform-sized jobs upon multiple processors than non-uniform ones. However, this enabling feature of pfair scheduling algorithms can also prove a disadvantage in certain implementations—one consequence of "breaking" each job of each task into subjobs, and making individual scheduling decisions for each subjob, is that jobs tend to get preempted after each of their constituent subjobs completes execution. As a result, pfair schedules are likely to contain a large number of job preemptions and context-switches. For some applications, this is not an issue; for others, however, the overhead resulting from too many preemptions may prove unacceptable. Pfair scheduling is not the appropriate scheduling approach for such application systems.

As discussed in Sect. 3.2, priority-driven scheduling algorithms (which include FJP and FTP ones) offer several implementation advantages. From among the priority-driven scheduling algorithms, FJP algorithms are more general than FTP ones; in the remainder of this chapter we focus our study on the global FJP scheduling of real-time systems that are represented as collections of independent implicit-deadline sporadic tasks. The global FTP scheduling of such systems will be discussed briefly in the next chapter (Chap. 9).

An important advance in our understanding of global scheduling on multiprocessors was obtained by Phillips et al. [153], who explored the use of *resource-augmentation* techniques for the online scheduling of real-time jobs[1]. The focus of Phillips et al. [153] was the scheduling of real-time systems that could be modeled as collections of independent jobs, not recurrent tasks. In [153], it was shown that

[1] Resource augmentation as a technique for improving the performance on online scheduling algorithms was formally proposed by Kalyanasundaram and Pruhs [113].

the obvious extension to the earliest deadline first (EDF) algorithm for identical multiprocessors could make the following performance guarantee:

Theorem 8.1 *(from [153]) If a collection of jobs is feasible on m identical processors, then the same collection of jobs is scheduled to meet all deadlines by* EDF *on m identical processors in which the individual processors are* $(2 - \frac{1}{m})$ *times as fast as in the original system.*

That is, the speedup factor (Definition 5.2) of global EDF in scheduling collections of independent jobs is (2 – 1/m), when compared to an optimal clairvoyant scheduler (that is *not* restricted to being FJP).

In this chapter, we will describe how the techniques introduced in [153] have been extended to obtain a wide variety of results concerning the global FJP scheduling of implicit-deadline sporadic task systems. The roadmap for the remainder of this chapter is as follows:

- Although our interest is in scheduling upon identical multiprocessor platforms, we start out considering the more general *uniform multiprocessor* platform model that was described in Sect. 1.2. In Sect. 8.1, we generalize Theorem 8.1 above is to be applicable to uniform multiprocessor platforms; this generalized version is presented here as Theorem 8.2.
- Since identical multiprocessors are a special case of uniform multiprocessors, Theorem 8.2 allows us to derive, in Sect. 8.2, a sufficient condition for an implicit-deadline sporadic task system to successfully meet all deadlines when scheduled using EDF—this is reported in Theorem 8.5.
- Further applications of Theorem 8.2 allow us to design, in Sect. 8.3, some FJP algorithms that are proved to be superior (in terms of both utilization bounds—Definition 5.1—and strict dominance) to EDF for the global scheduling of implicit-deadline sporadic task systems.

8.1 EDF Scheduling on Uniform Multiprocessors

We start out considering the scheduling of collections of independent jobs upon a uniform multiprocessor platform.

Definition 8.1 (uniform multiprocessor platform) A uniform multiprocessor platform π is comprised of $m(\pi)$ processors, where $m(\pi)$ is a positive integer. Each processor is characterized by a parameter denoting its *speed* or *computing capacity*. We use the notation $\pi = [s_1, s_2, \ldots, s_{m(\pi')}]$ to represent the uniform multiprocessor platform in which the processors have computing capacities $s_1, s_2, \ldots, s_{m(\pi')}$ respectively; without loss of generality, we assume that these speeds are indexed in a nonincreasing manner: $s_j \geq s_{j+1}$ for all $j,\ 1 \leq j < m(\pi)$.

We use the notation

$$S_{\text{sum}}(\pi) \overset{\text{def}}{=} \sum_{i=1}^{m(\pi')} s_i \text{ and } S_{\max}(\pi) \overset{\text{def}}{=} s_1$$

to denote the cumulative and maximum speeds of the processors in π. We further define a parameter $\lambda(\pi)$ as follows:

$$\lambda(\pi) \overset{\text{def}}{=} \max_{j=1}^{m(\pi')} \left\{ \frac{\sum_{k=j+1}^{m(\pi')} s_k}{s_j} \right\} . \tag{8.1}$$

The *lambda* parameter $\lambda(\pi)$ of uniform multiprocessor platform π as defined above is a crucial characteristic of a uniform multiprocessor system: informally speaking, $\lambda(\pi)$ measures the "degree" by which π differs from being an identical multiprocessor platform (in the sense that $\lambda(\pi) = (m - 1)$ if π is comprised of m identical processors, and becomes progressively smaller as the speeds of the processors differ from each other by greater amounts; in the extreme, if $s_1 > 0$ and $s_2 = s_3 = \cdots = s_{m(\pi')} = 0$ would have $\lambda(\pi) = 0$).

We will assume for this section that a hard real-time system is represented as an arbitrary collection of individual *jobs*. Each *job* $J_j = (r_j, c_j, d_j)$ is characterized by an arrival time r_j, an execution requirement c_j, and a deadline d_j, with the interpretation that this job needs to execute for c_j units over the interval $[r_j, d_j)$.

Our objective in this section may be stated as follows: given the specifications of a uniform multiprocessor platform π, we seek to derive a condition (Theorem 8.2) upon the specifications of any another uniform multiprocessor platform π' such that if π' satisfies this condition, then *any collection of independent jobs feasible on π will meet all deadlines when scheduled on π' using EDF.*

We introduce some additional notation.

Definition 8.2 ($\mathbf{W(A, \pi, I, t)}$) Let I denote any set of jobs, and π any uniform multiprocessor platform. For any algorithm A and time instant $t \geq 0$, let $W(A, \pi, I, t)$ denote the amount of work done by algorithm A on jobs of I over the interval $[0, t)$, while executing on π.

Definition 8.3 ($S_j(\pi)$) Let π denote a uniform multiprocessor platform with processor capacities $s_1, s_2, \ldots, s_{m(\pi')}$, $s_j \geq s_{j+1}$ for all j, $1 \leq j < m(\pi)$. Define $S_j(\pi)$ to be the sum of the speeds of the j fastest processors in π:

$$\left(S_j(\pi) \overset{\text{def}}{=} \sum_{\ell=1}^{j} s_\ell \right) \text{ for all } j,\ 1 \leq j \leq m(\pi) .$$

(Note that $S_{m(\pi')}(\pi)$ is equal to $S_{\text{sum}}(\pi)$—the sum of the computing capacities of all the processors in π.)

Work-Conserving Scheduling on Uniform Multiprocessors In the context of uniprocessor scheduling, a work-conserving scheduling algorithm is defined to be one that never idles the (only) processor while there is any active job awaiting execution. This definition extends in a rather straightforward manner to the identical multiprocessor case: an algorithm for scheduling on identical multiprocessors is defined to be work-conserving if it never leaves any processor idle while there remain active jobs awaiting execution.

We define a uniform multiprocessor scheduling algorithm to be work-conserving if and only if it satisfies the following conditions:

- No processor is idled while there are active jobs awaiting execution.
- If at some instant there are fewer active jobs than there are number of processors in the platform, then the active jobs are executed upon the fastest processors. That is, it is the case that at any instant t if the jth-slowest processor is idled by the work-conserving scheduling algorithm, then the kth-slowest processor is also idled at instant t, for all $k > j$.

EDF on Uniform Processors Recall that the earliest deadline first (EDF) scheduling algorithm chooses for execution at each instant in time the currently active job[s] that have the smallest deadlines. We assume that EDF is implemented upon uniform multiprocessor systems according to the following rules:

1. No processor is idled while there is an active job awaiting execution.
2. When there are fewer active jobs than there are number of processors in the platform, they are required to execute upon the fastest processors while the slowest processors are idled.
3. Higher priority jobs are executed on faster processors. Hence if the jth-slowest processor is executing job J_g at time t under our EDF implementation, it must be the case that the deadline of J_g is not greater than the deadlines of jobs (if any) executing on the $(j+1)$th-, $(j+2)$th-, ...-slowest processors.

The first two conditions above imply that EDF is a work-conserving scheduling algorithm.

Suppose that a given set of jobs is known to be feasible on a given uniform multiprocessor platform π. Lemma 8.1 specifies a condition (Condition 8.2 below) upon the parameters of any other uniform multiprocessor platform π', for ensuring that any work-conserving algorithm A' executing on π' is guaranteed to complete at least as much work by each instant in time t as any other algorithm A (including an optimal algorithm) executing on π, when both algorithms execute the same set of jobs I.

We will later use this lemma in Theorem 8.2 to determine conditions under which EDF executing on π' will meet all deadlines of a set of jobs known to be feasible on π.

Notice that Condition 8.2 is expressed as a constraint on the parameter $\lambda(\pi')$ of the uniform multiprocessor platform π'; it expresses the additional computing capacity needed by π' in terms of this $\lambda(\pi')$ parameter, and the speed of the fastest processor in π—the smaller the value of $\lambda(\pi')$ (i.e., the more π' deviates from being an identical multiprocessor), the smaller is the amount of this excess processing capacity it needs.

Lemma 8.1 *Let π and π' denote uniform multiprocessor platforms. Let A denote any uniform multiprocessor scheduling algorithm, and A' any work-conserving uniform multiprocessor scheduling algorithm. If the following condition is satisfied by the platforms π and π':*

$$S_{sum}(\pi') \geq \lambda(\pi') \cdot S_{max}(\pi) + S_{sum}(\pi) \tag{8.2}$$

then for any collection of jobs I and any time-instant $t \geq 0$,

$$W(A', \pi', I, t) \geq W(A, \pi, I, t) . \tag{8.3}$$

Proof The proof of this lemma, which is a direct generalization of proof techniques from [153], is somewhat tedious and detailed. It is presented here for the sake of completeness, but may be safely skipped at a first reading since only the lemma is used (and not the details of the proof) elsewhere in the book.

The proof is by contradiction. Suppose that π is comprised of processors with speeds $s_1, s_2, \ldots, s_{m(\pi')}$, $s_j \geq s_{j+1}$ for all j, $1 \leq j < m(\pi)$. Suppose that π' is comprised of processors with speeds $s'_1, s'_2, \ldots, s'_{m(\pi')}$, $s'_j \geq s'_{j+1}$ for all j, $1 \leq j < m(\pi')$.

Suppose that the lemma is not true, and let I denote a collection of jobs I on which work-conserving algorithm A' executing on π' has performed strictly less work than some other algorithm A executing on π by some time-instant. Let $J_a = (r_a, c_a, d_a)$ denote a job in I with the earliest arrival time such that there is some time-instant t_o satisfying

$$W(A', \pi', I, t_o) < W(A, \pi, I, t_o),$$

and the amount of work done on job J_a by time-instant t_o in A' is strictly less than the amount of work done on J_a by time-instant t_o in A.

By our choice of r_a, it must be the case that $W(A', \pi', I, r_a) \geq W(A, \pi, I, r_a)$. Therefore, the amount of work done by A over $[r_a, t_o)$ is strictly greater than the amount of work done by A' over the same interval.

Let x_ℓ denote the cumulative length of time over the interval $[r_a, t_o)$ during which A' is executing on ℓ processors, $1 \leq \ell \leq m(\pi')$ (Hence, $t_o - r_a = x_1 + x_2 + \cdots + x_{m(\pi')}$.) We make the following two observations.

- Since A' is a work-conserving scheduling algorithm, job J_a, which has not completed by instant t_o in the schedule generated by A', must be executing at all time-instants during which some processor is idled by A'. During the instants at which ℓ processors are non-idling, $\ell < m(\pi')$, all these nonidled processors have computing capacity $\geq s'_\ell$—this follows from the definition of "work-conserving" and the fact that A' is a work-conserving algorithm. Therefore, it follows that job J_a has executed for at least $\left(\sum_{j=1}^{m(\pi')-1} x_j s'_j\right)$ units by time t_o in the schedule generated by A' on π', while it could have executed for at most $S_{\max}(\pi) \cdot \left(\sum_{j=1}^{m(\pi')} x_j\right)$ units in the schedule generated by Algorithm A on π. We therefore have

$$\sum_{j=1}^{m(\pi')-1} x_j s'_j < S_{\max}(\pi) \left(\sum_{j=1}^{m(\pi')} x_j \right) . \tag{8.4}$$

Multiplying both sides of Inequality 8.4 above by $\lambda(\pi')$, and noting that

$$\left(x_j s'_j \lambda(\pi')\right) \geq \left(x_j s'_j \frac{S_{\text{sum}}(\pi') - S_j(\pi')}{s'_j} \right) = x_j (S_{\text{sum}}(\pi') - S_j(\pi')) ,$$

we obtain

$$\sum_{j=1}^{m(\pi')-1} \big(x_j(S_{\text{sum}}(\pi') - S_j(\pi'))\big) \;<\; S_{\max}(\pi)\lambda(\pi')\left(\sum_{j=1}^{m(\pi')} x_j\right). \tag{8.5}$$

- The total amount of work done by A' executing on π' during $[r_a, t_o)$ is given by $\sum_{j=1}^{m(\pi')}(x_j S_j(\pi'))$, while the total amount of work done by A executing on π during this same interval is bounded from above by the capacity of π, and is hence $\leq (\sum_{j=1}^{m(\pi')} x_j) \cdot S_{\text{sum}}(\pi)$. We thus obtain the inequality

$$\sum_{j=1}^{m(\pi')}(x_j S_j(\pi')) \;<\; \left(\sum_{j=1}^{m(\pi')} x_j\right) \cdot S_{\text{sum}}(\pi)\,. \tag{8.6}$$

Adding Inequalities 8.5 and 8.6, we obtain

$$\left(\sum_{j=1}^{m(\pi')}(x_j S_j(\pi')) + \sum_{j=1}^{m(\pi')-1}(x_j(S_{\text{sum}}(\pi') - S_j(\pi'))) < \left(\sum_{j=1}^{m(\pi')} x_j\right)\right.$$
$$\left.(S_{\max}(\pi)\lambda(\pi') + S_{\text{sum}}(\pi))\right)$$

$$\Leftrightarrow \left(x_{m'}S_{m'}(\pi') + \sum_{j=1}^{m(\pi')-1}\big[x_j(S_j(\pi') + S_{\text{sum}}(\pi') - S_j(\pi'))\big] < \left(\sum_{j=1}^{m(\pi')} x_j\right)\right.$$
$$\left.(S_{\max}(\pi)\lambda(\pi') + S_{\text{sum}}(\pi))\right)$$

$$\Leftrightarrow \left(x_{m'}S_{\text{sum}}(\pi') + \sum_{j=1}^{m(\pi')-1}(x_j S_{\text{sum}}(\pi')) < \left(\sum_{j=1}^{m(\pi')} x_j\right)(S_{\max}(\pi)\lambda(\pi') + S_{\text{sum}}(\pi))\right)$$

$$\Leftrightarrow \left(S_{\text{sum}}(\pi') \cdot \left(\sum_{j=1}^{m(\pi')} x_j\right) < \left(\sum_{j=1}^{m(\pi')} x_j\right)(S_{\max}(\pi)\lambda(\pi') + S_{\text{sum}}(\pi))\right)$$

$$\Leftrightarrow \Big(S_{\text{sum}}(\pi') < S_{\max}(\pi)\lambda(\pi') + S_{\text{sum}}(\pi)\Big)$$

which contradicts the assumption made in the statement of the lemma (Inequality 8.2). □

Theorem 8.2 below applies Lemma 8.1 to show that any collection of jobs I that is feasible on a uniform multiprocessor platform π will be scheduled to meet all deadlines by EDF on any platform π' satisfying Condition 8.2 of Lemma 8.1.

Theorem 8.2 *Let I denote an instance of jobs that is feasible on a uniform multiprocessor platform π. An* EDF *schedule of I on any uniform multiprocessor platform π' will also meet all deadlines, if Condition 8.2 of Lemma 8.1 is satisfied by platforms π and π'.*

Proof Suppose that π and π' satisfy Condition 8.2. From Lemma 8.1, we may conclude that the amount of work done at any time-instant t by EDF scheduling I on π' is at least as much as the work done by that time-instant t by an optimal scheduling algorithm executing I on π:

$$W(\mathsf{EDF}, \pi', I, t) \geq W(\textsc{opt}, \pi, I, t) \text{ for all } t \geq 0\,,$$

where OPT denotes an algorithm that generates a schedule for I which meets all deadlines on π—since I is assumed feasible on π, such a schedule exists.

We will now use induction to show that I is scheduled by EDF to meet all deadlines on π'. The induction is on the number of jobs in I. Specifically, let $I_k \overset{\text{def}}{=} \{J_1, \ldots, J_k\}$ denote the k jobs of I with the highest EDF priority.

Base Case I_o denotes the empty set; which is trivially scheduled to meet all deadlines on π' by EDF.

Induction Step Let us assume, as an induction hypothesis, that EDF can schedule I_k on π' for some k. Consider now the EDF-generated schedule of I_{k+1} on π'. Observe that *(i)* $I_k \subset I_{k+1}$, and *(ii)* the presence of J_{k+1} do not effect the scheduling decisions made by EDF on the jobs $\{J_1, J_2, \ldots, J_k\}$ while it is scheduling I_{k+1}. Hence the EDF schedule for $\{J_1, J_2, \ldots, J_k\}$ while scheduling I_{k+1}, is identical to the schedule generated by EDF while scheduling I_k; hence by the induction hypothesis, these k highest priority jobs $\{J_1, J_2, \ldots, J_k\}$ of I_{k+1} all meet their deadlines. It remains to prove that J_{k+1} also meets its deadline.

Consider now the schedule generated by OPT executing on π. Since I is feasible on π, I_{k+1}, being a subset of I, is also feasible on π and hence OPT will schedule I_{k+1} on π to meet all deadlines. That is,

$$W(\textsc{opt}, \pi, I_{k+1}, d_{k+1}) = \sum_{i=1}^{k+1} c_i,$$

where d_{k+1} denotes the latest deadline of a job in I_{k+1}. By applying Lemma 8.1, we have

$$W(\mathsf{EDF}, \pi', I_{k+1}, d_{k+1}) \geq W(\textsc{opt}, \pi, I_{k+1}, d_{k+1}) = \sum_{i=1}^{k+1} c_i.$$

Since the total execution requirement of all the jobs in I_{k+1} is $\sum_{i=1}^{k+1} c_i$ it follows that job J_{k+1} meets its deadline.

It is therefore the case that EDF successfully schedules all the jobs of I_{k+1} to meet their deadlines on π'. The theorem is thus proved. □

We point out that Theorem 8.1 concerning EDF-scheduling on identical multiprocessors, which was originally proved in [153], can be derived as an immediate corollary to Theorem 8.2 above:

Corollary 8.1 *If a set of jobs is feasible on an identical m-processor platform, then the same set of jobs will be scheduled to meet all deadlines by* EDF *on an identical* *m-processor platform in which the individual processors are* $(2 - \frac{1}{m})$ *times as fast as in the original system.*

Proof If we require the platforms π and π' of the statement of Theorem 8.2 to each be comprised of m identical processors of speeds s and s' respectively, the parameter $\lambda(\pi')$ takes on the value $(m-1)$:

$$\lambda(\pi') \overset{\text{def}}{=} \max_{j=1}^{m} \left\{ \frac{\sum_{k=j+1}^{m} s'_k}{s'_j} \right\} = \max_{j=1}^{m} \left\{ \frac{(j-1)s'}{s'} \right\} = m-1 \,.$$

The condition $S_{\text{sum}}(\pi') \geq \lambda(\pi') \cdot S_{\max}(\pi) + S_{\text{sum}}(\pi)$ from the statement of Theorem 8.2 is hence equivalent to

$$\begin{aligned} S_{\text{sum}}(\pi') &\geq \lambda(\pi') \cdot S_{\max}(\pi) + S_{\text{sum}}(\pi) \\ &\equiv m \cdot s' \geq (m-1) \cdot s + m \cdot s \\ &\equiv m \cdot s' \geq (m-1+m) \cdot s \\ &\equiv s' \geq \left(2 - \frac{1}{m}\right) \cdot s \end{aligned}$$

from which the corollary follows. □

8.2 Global EDF Scheduling

In this section, we will apply the results from Sect. 8.1 above concerning scheduling on uniform multiprocessors, to the EDF scheduling of systems of implicit-deadline sporadic tasks upon identical multiprocessor platforms. We begin with a more-or-less obvious result:

Theorem 8.3 *Let τ denote an implicit-deadline sporadic task system. There is a uniform multiprocessor platform π upon which τ is feasible, which satisfies the following two properties:*

1. $S_{max}(\pi) = U_{max}(\tau)$ *and*
2. $S_{sum}(\pi) = U_{sum}(\tau)$.

Proof The uniform multiprocessor platform in which each task $\tau_i \in \tau$ has a dedicated processor with computing capacity u_i bears witness to the correctness of this theorem. □

By a direct application of Theorems 8.3 and 8.2, we obtain below a sufficient condition for any implicit-deadline sporadic task system to be successfully scheduled by EDF.

Theorem 8.4 *Implicit-deadline sporadic task system τ comprised of n tasks can be EDF-scheduled upon an identical multiprocessor platform comprised of m unit-capacity processors, provided*

$$m \geq \min\left(n, \left\lceil \frac{U_{sum}(\tau) - U_{max}(\tau)}{1 - U_{max}(\tau)} \right\rceil\right) \tag{8.7}$$

Proof Clearly, no more than n processors are needed to schedule an n-task system; hence if the second term in the "min" above is no smaller than the first, the theorem is trivially seen to hold. For the remainder of this proof, consider therefore the case when $n > \lceil (U_{\text{sum}}(\tau) - U_{\max}(\tau))/(1 - U_{\max}(\tau)) \rceil$.

By Theorem 8.3, implicit-deadline sporadic task system τ is feasible on some uniform multiprocessor platform π_1 with cumulative computing capacity $S_{\text{sum}}(\pi_1) = U_{\text{sum}}(\tau)$, in which the fastest processor has speed $S_{\max}(\pi_1) = U_{\max}(\tau)$.

Note that the cumulative computing capacity of an m-processor identical multiprocessor platform consisting of unit-speed processors is m; furthermore as stated immediately following Definition 8.1, the parameter λ_π takes on the value $(m-1)$ for such a platform.

Consider now the statement of Theorem 8.2: any particular sequence of jobs generated by an implicit-deadline sporadic task system that is feasible upon a uniform multiprocessor platform π is successfully scheduled by EDF upon a uniform multiprocessor platform π' if

$$S_{\text{sum}}(\pi') \geq \lambda(\pi') \cdot S_{\max}(\pi) + S_{\text{sum}}(\pi)$$

Let us apply this theorem with π instantiated to the platform π_1 described above, and π' instantiated to the m-processor identical multiprocessor platform consisting of unit-speed processors. A sufficient condition for the task system τ to be EDF-schedulable on the m-processor identical multiprocessor platform consisting of unit-speed processors is that

$$\begin{aligned}
& S_{\text{sum}}(\pi') \geq \lambda(\pi') \cdot S_{\max}(\pi) + S_{\text{sum}}(\pi) \\
\Leftrightarrow\ & m \geq (m-1)U_{\max}(\tau) + U_{\text{sum}}(\tau) \\
\Leftrightarrow\ & m(1 - U_{\max}(\tau)) \geq U_{\text{sum}}(\tau) - U_{\max}(\tau) \\
\Leftrightarrow\ & m \geq \left\lceil \frac{U_{\text{sum}}(\tau) - U_{\max}(\tau)}{1 - U_{\max}(\tau)} \right\rceil
\end{aligned}$$

and the theorem is proved. □

Theorem 8.5 follows by algebraic simplification of the Expression 8.7:

Theorem 8.5 *Implicit-deadline sporadic task system τ can be EDF-scheduled upon m unit-speed identical processors, provided its cumulative utilization is bounded from above as follows:*

$$U_{sum}(\tau) \leq m - (m-1)U_{max}(\tau) \tag{8.8}$$

□

It turns out that the bounds of Theorem 8.4 and 8.5 are in fact tight:

Theorem 8.6 *Let m denote any positive integer > 1, u_1 any real number satisfying $0 < u_1 < 1$, and ϵ an arbitrarily small positive real number, $\epsilon \ll u_1$. EDF cannot schedule some implicit-deadline sporadic task systems with cumulative utilization $m - u_1(m-1) + \epsilon$ in which the largest-utilization task has utilization equal to u_1, upon m unit-speed processors.*

Proof Let p denote some positive number. We construct a task set τ as follows.

1. Task τ_1 has execution requirement $C_1 = u_1 \cdot p$ and period $T_1 = p$.
2. Tasks $\tau_2, \tau_2, \ldots, \tau_n$ all have period $T_i = p$ and execution requirement $C_i = C$ for all i, $1 < i \leq n$, satisfying

$$(n-1) \cdot C = m \cdot (1 - u_1) \cdot p + m\delta \ ,$$

where $\delta = (p \cdot \epsilon)/m$. Furthermore, n is chosen such that $n - 1$ is a multiple of m, and is large enough so that $C \ll u_1 \cdot p$.

The largest-utilization task in τ is τ_1, which has utilization equal to $(u_1 \cdot p)/p = u_1$. The cumulative utilization of tasks in τ is given by

$$\begin{aligned} U(\tau) &= \frac{u_1 \cdot p}{p} + \frac{C_2}{p} + \frac{C_3}{p} + \cdots + \frac{C_n}{p} \\ &= u_1 + \frac{(n-1) \cdot C}{p} \\ &= u_1 + m \cdot (1 - u_1) + m\frac{\delta}{p} \\ &= m - u_1 \cdot (m-1) + \epsilon \end{aligned}$$

Now consider the scheduling of the first jobs of each task, and suppose that EDF breaks ties such that τ_1's first job is selected last for execution[2]. Then EDF schedules the jobs of tasks $\tau_2, \tau_3, \ldots$ before scheduling τ_1's job; these jobs of $\tau_2, \tau_3, \ldots$ consume all m processors over the interval $[0, (1 - u_1) \cdot p + \delta)$, and τ_1's job can only begin execution at time-instant $(1 - u_1) \cdot p + \delta$. Therefore τ_1's job's completion time is $((1 - u_1) \cdot p + \delta + u_1 \cdot p) = (p + \delta)$, and it misses its deadline. Thus, the τ we have constructed above is an implicit-deadline sporadic task system with utilization $U_{\text{sum}}(\tau) = m - (m-1)U_{\max}(\tau) + \epsilon$, which EDF fails to successfully schedule upon m unit-speed processors. The theorem follows. □

Phillips et al. [153] had proved that any instance of jobs feasible upon m unit-capacity multiprocessors can be EDF-scheduled upon m processors each of capacity $(2 - \frac{1}{m})$. For implicit-deadline sporadic task systems, we see below (Theorem 8.7) that this follows as a direct consequence of the results above.

Lemma 8.2 *Any implicit-deadline sporadic task system τ satisfying*

$$U_{max}(\tau) \leq m/(2m-1) \text{ and } U_{sum}(\tau) \leq m^2/(2m-1)$$

is scheduled by Algorithm EDF *to meet all deadlines on m unit-capacity processors.*

[2] Alternatively, τ_1's period can be chosen to be infinitesimally larger than p—this would force EDF to schedule τ_1's job last, without changing the value of m.

Proof By Theorem 8.4, τ can be EDF-scheduled on $\left\lceil \frac{U_{\text{sum}}(\tau)-U_{\max}(\tau)}{1-U_{\max}(\tau)} \right\rceil$ unit-capacity processors; by substituting for $U_{\text{sum}}(\tau)$ and $U_{\max}(\tau)$, we obtain

$$\left\lceil \frac{U_{\text{sum}}(\tau)-U_{\max}(\tau)}{1-U_{\max}(\tau)} \right\rceil = \left\lceil \frac{\frac{m^2}{2m-1} - \frac{m}{2m-1}}{1-\frac{m}{2m-1}} \right\rceil = \left\lceil \frac{\frac{m}{2m-1}\cdot(m-1)}{\frac{2m-1-m}{2m-1}} \right\rceil$$
$$= \left\lceil \frac{m(m-1)}{(m-1)} \right\rceil = \lceil m \rceil = m\ .$$

□

Theorem 8.7 *The processor speedup factor for EDF-scheduling of implicit-deadline sporadic task systems is* $(2 - 1/m)$*; that is, any implicit-deadline sporadic task system that is feasible upon m unit-capacity processors will be scheduled by* EDF *to meet all deadlines on m* $(2 - \dfrac{1}{m})$*-capacity processors.*

Proof Sketch: Suppose that implicit-deadline sporadic task system $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is feasible upon m unit-capacity processors. It must be the case that

1. $u_i \le 1$ for all i, $1 \le i \le n$—that is, no individual task needs more than an entire processor, and
2. $U_{\text{sum}}(\tau)(\tau) \le m$—that is, the cumulative computing requirement of τ does not exceed the capacity of the platform.

The theorem now follows directly from Lemma 8.2, by scaling all utilizations and processor speeds by a factor of $(2 - \frac{1}{m})$. □

8.3 Global FJP Scheduling

Recall (Definition 5.1) that the *utilization bound* of a scheduling algorithm A for scheduling implicit-deadline task systems upon a platform consisting of m unit-speed processors is defined to be the largest number U such that all task systems with utilization $\le U$ (and with each task having utilization ≤ 1) is successfully scheduled by A to meet all deadlines on the m-processor platform. An examination of Theorem 8.6 reveals that there are implicit-deadline sporadic task systems with utilization larger than one by an arbitrarily small amount that are not EDF-schedulable upon m unit-speed processors for any m (such a task system with have $U_{\max}(\tau)$ – the utilization of the largest-utilization task—arbitrarily close to one). We illustrate this phenomenon, which is commonly called the *Dhall effect* [78], in the following example:

Example 8.1 Consider an implicit-deadline sporadic task system of $(m+1)$ tasks to be scheduled upon an m-processor platform, in which tasks $\tau_1, \ldots, \tau_m$ have parameters $(C_i = 1, T_i = x)$, and task τ_{m+1} has parameters $C_{m+1} = T_{m+1} = x + 1$. We observe that

- The utilization of this system is $(m \times \frac{1}{x}) + (\frac{x+1}{x+1})$, which for any given m, approaches 1 as x increases.
- If all the tasks were to release jobs simultaneously, then tasks $\tau_1, \cdots, \tau_m$ would be scheduled first, thereby causing τ_{m+1}'s job to miss its deadline.

This task system is thus not EDF-schedulable despite having a utilization close to 1. Hence, the utilization bound of global EDF is very poor: it is arbitrarily close to one regardless of the number of processors.

In Sect. 8.2 we considered the global EDF-scheduling of implicit-deadline sporadic task systems upon identical multiprocessor platforms. The reasons for considering EDF include the following:

- Very efficient implementations of EDF have been designed (see, e.g., [148]).
- It can be shown that when a set of jobs is scheduled by EDF then the total number of preemptions and migrations is bounded from above by the number of jobs in the set.

However, these desirable features are not unique to EDF, but instead hold for all priority-driven scheduling algorithms. In this section, therefore, we will study a variant of the EDF-scheduling algorithm that falls within the framework of priority-driven algorithms, and which is provably superior to EDF in the sense that it schedules all implicit-deadline sporadic task systems that EDF can schedule, and in addition schedules some implicit-deadline sporadic task systems for which EDF may miss some deadlines.

Before presenting the algorithm, we need to prove a preliminary result—Theorem 8.8 below.

Theorem 8.8 *Any implicit-deadline sporadic task system τ satisfying the following two properties*[3]

Property P1: $U_{sum}(\tau) \leq (m+1)/2$
Property P2: $U_{max}(\tau) \leq 1/2$

is correctly scheduled to meet all deadlines on m processors by EDF.

Proof By Theorem 8.5 (Condition 8.8), a sufficient condition for EDF-schedulability of τ is that $U_{\text{sum}}(\tau) \leq m - (m-1)\,U_{\max}(\tau)$. If τ satisfies Properties P1 and P2 above, the right hand side becomes $(m - \frac{m-1}{2}) = \frac{m+1}{2}$, and hence τ is schedulable by EDF. □

We are now ready to describe Algorithm fpEDF, our FJP scheduling algorithm for scheduling implicit-deadline sporadic task systems, and to derive a utilization-based sufficient schedulability condition for it.

Suppose that task system τ is to be scheduled by Algorithm fpEDF upon m unit-capacity processors, and let $\{\tau_1, \tau_2, \ldots, \tau_n\}$ denote the tasks in τ indexed according

[3] We point out that the constraints expressed by Theorem 8.8 is *incomparable* to those in Lemma 8.2, since $m/(2m-1) \geq 1/2$ but $m^2/(2m-1) \leq (m+1)/2$.

```
Algorithm fpEDF
Implicit-deadline sporadic task system $\tau = \{\tau_1, \tau_2, \ldots \tau_n\}$ to be scheduled on $m$ processors
(It is assumed that $u_i \geq u_{i+1}$ for all $i$, $1 \leq i < n$)

    for $i = 1$ to $(m-1)$ do
        if $(u_i > \frac{1}{2})$
                then $\tau_i$'s jobs are assigned highest priority (ties broken arbitrarily)
                else break
    the remaining tasks' jobs are assigned priorities according to EDF
```

Fig. 8.1 Algorithm fpEDF's priority-assignment rule

to nonincreasing utilization: $u_i \geq u_{i+1}$ for all i, $1 \leq i < n$. (Observe that $U_{\max}(\tau)$ is therefore equal to u_1.) Algorithm fpEDF first considers the $(m-1)$ "heaviest" (i.e., largest-utilization) tasks in τ. All the tasks from among these heaviest $(m-1)$ tasks that have utilization greater than one half are treated specially in the sense that all their jobs are always assigned highest priority (note that this is implemented trivially in an EDF scheduler by setting the deadline parameters of these jobs to $-\infty$). The remaining tasks' jobs—that is, the jobs of the tasks from among the heaviest $(m-1)$ with utilization $\leq 1/2$, as well as of the $(n-m+1)$ remaining tasks — are assigned priorities according to their deadlines (as in "regular" EDF). This priority-assignment rule is presented in pseudocode form, in Fig. 8.1.

Note that Algorithm fpEDF reduces to "regular" EDF when scheduling τ upon m processors if

1. $U_{\max}(\tau) \leq (1/2)$, in which case the "break" statement in the for-loop is executed for $i = 1$ and all tasks' jobs get EDF-priority; or
2. $m = 1$, in which case $(m-1) = 0$ and the for-loop is not executed at all.

Computational Complexity The runtime complexity of Algorithm fpEDF is identical to that of "regular" EDF, in the sense that, once it is determined which tasks' jobs always get highest priority, the runtime implementation of fpEDF is identical to that of EDF.

The process of determining which tasks' jobs always get highest priority would be performed according to the "for" loop in Fig. 8.1 in time linear in m, if the tasks in τ are presented sorted according to utilization. If the tasks are not presorted according to utilization, then the $(m-1)$ heaviest tasks can be determined in time linear in n using the standard linear-time selection algorithm. Since $m \leq n$, in either case the computational complexity of the pre-runtime phase is thus O(n), where n denotes the number of tasks in τ.

Properties The following theorem states that Algorithm fpEDF correctly schedules on m processors any implicit-deadline sporadic task system τ with utilization $U_{\text{sum}}(\tau) \leq (m+1)/2$.

Theorem 8.9 *Algorithm fpEDF has a utilization bound (Definition 5.1) no smaller than $\frac{m+1}{2}$ upon m processors.*

Proof The proof proceeds by induction on k, the number of processors. For a single processor, the correctness of the statement of the therorem follows from the optimality of EDF on uniprocessors [137].

Assume that the statement of the theorem is true for $k-1$ processors, and consider the case of k processors. Consider any implicit-deadline sporadic task system τ satisfying $U_{\text{sum}}(\tau) \leq (k+1)/2$. We consider two separate cases: **(i)** when $U_{\max}(\tau) \leq (1/2)$, and **(ii)** when $U_{\max}(\tau) > (1/2)$.

(i) If $U_{\max}(\tau) \leq 1/2$, then τ satisfies Properties P1 and P2 of Theorem 8.8. Therefore, it follows from Theorem 8.8 that τ is scheduled to meet all deadlines upon k processors by EDF. Since Algorithm fpEDF reduces to EDF when no task has utilization $>$ (1/2), we conclude that Algorithm fpEDF correctly schedules τ upon k processors.

(ii) Since $U_{\max}(\tau) > (1/2)$, Algorithm fpEDF assigns highest priority to all the jobs of τ_1.
Consider the system τ' obtained from τ by removing the task $\tau_1 = (\mathrm{C}_1, \mathrm{T}_\mathrm{i})$ of maximum utilization:

$$\tau' \overset{\text{def}}{=} (\tau \setminus \{\tau_1\})$$

Observe that

$$\begin{aligned} U_{\text{sum}}(\tau') &= U_{\text{sum}}(\tau) - U(\tau_1) \\ &= U_{\text{sum}}(\tau) - U_{\max}(\tau) \\ &\leq \frac{k+1}{2} - \frac{1}{2} \\ \Rightarrow \ U_{\text{sum}}(\tau') &\leq \frac{k}{2} \end{aligned}$$

By our inductive hypothesis above, Algorithm fpEDF therefore can successfully schedule τ' on $k-1$ processors.
Consider now the task system τ'', comprised of τ' plus a task $\hat{\tau_1} = (T_1, T_1)$ with utilization 1 and period equal to the period of τ_1:

$$\tau'' \overset{\text{def}}{=} (\tau \setminus \{\tau_1\}) \bigcup \{\hat{\tau_1} = (T_1, T_1)\}$$

A schedule for τ'' on k processors can be obtained from the fpEDF schedule for τ' on $k-1$ processors (which, according to our inductive hypothesis, is guaranteed to exist), by simply devoting one processor exclusively to the additional task $\hat{\tau_1}$, and scheduling the remaining $(k-1)$ exactly as in the fpEDF-schedule.
Furthermore, this schedule is exactly equivalent to the one that would be generated if Algorithm fpEDF were scheduling τ'' on k processors—this follows from the observations that

- Since task $\hat{\tau_1}$ has the highest utiliation of any task in τ'', its jobs would be assigned highest priority by Algorithm fpEDF.

- Jobs of the remaining tasks in τ'' would be assigned exactly the same priorities as they would in the $(k-1)$-processor fpEDF-schedule of τ'.
- The jobs of $\hat{\tau}_1$ completely occupy one processor (since $U(\tau_1) = 1$).

Thus, Algorithm fpEDF successfully schedules task-system τ'' upon k processors. Since Algorithm fpEDF is a fixed-priority algorithm, it follows by Theorem 3.1 that it is *predictable*; by the definition of predictability, it follows that Algorithm fpEDF successfully schedules τ, since τ may be obtained from τ'' by reducing the execution requirement of each of $\hat{\tau}_1$'s jobs by a quantity $(T_1 - C_1)$. □

We show below (Theorem 8.10) that no FJP-scheduling algorithm can have a greater schedulable utilization than Algorithm fpEDF.

Recall that an FJP-scheduling algorithm satisfies the condition that for every pair of jobs J_i and J_j, if J_i has higher priority than J_j at some instant in time, then J_i always has higher priority than J_j. In other words, individual jobs are assigned fixed priorities (although different jobs of the same task may have very different priorities).

Theorem 8.10 *No m-processor FJP scheduling algorithm has a schedulable utilization greater than $\frac{m+1}{2}$.*

Proof Consider the implicit-deadline sporadic task system comprised of $m+1$ identical tasks, each with execution requirement $1+\epsilon$ and period 2, where ϵ is an arbitrarily small positive number. Each task releases its first job at time-instant zero. Any FJP-scheduling algorithm must assign these jobs fixed priorities relative to each other and the task whose job is assigned the lowest priority at time-instant zero misses its deadline. Note that as $\epsilon \to 0$, $U_{\text{sum}}(\tau) \to \frac{m+1}{2}$; thus, the required result follows. □

As with partitioned scheduling [141], we can obtain better bounds upon schedulable utilization if the largest utilization $U_{\max}(\tau)$ of any task in τ is known.

Theorem 8.11 *Algorithm fpEDF correctly schedules any implicit-deadline sporadic task system τ satisfying*

$$U_{sum}(\tau) \leq \max\left(m - (m-1)U_{max}(\tau), \frac{m}{2} + U_{max}(\tau)\right) \tag{8.9}$$

upon m unit-capacity processors.

Proof We consider two cases separately: **(i)** when $U_{\max}(\tau) \leq (1/2)$, and **(ii)** when $U_{\max}(\tau) > (1/2)$. For $U_{\max}(\tau) \leq (1/2)$, observe that the first term in the "max" above is greater than or equal to the second; while for $U_{\max}(\tau) > (1/2)$, the second term in the "max" is greater than or equal to the first.

(i) For $U_{\max}(\tau) \leq (1/2)$, it is the first term in the "max" that defines the schedulable utilization for task systems τ satisfying $U_{\max}(\tau) \leq 1/2$. That is,

$$U_{\text{sum}}(\tau) \leq m - (m-1)U_{\max}(\tau) \tag{8.10}$$

for such systems.
However, we have previously observed that Algorithm fpEDF behaves exactly as EDF does when $U_{\max}(\tau) \leq 1/2$. The correctness of the theorem follows from the

observation that the bound of Eq. 8.10 above is the EDF-bound of Theorem 8.5 (Eq. 8.8).

(ii) As in the proof of Theorem 8.9, let τ' denote the task system obtained from τ by removing the task τ_1 of maximum utilization:

$$\tau' \overset{\text{def}}{=} (\tau \setminus \{\tau_1\})$$

As explained in the proof of Theorem 8.9, a sufficient condition for τ to be correctly scheduled on m processors by Algorithm fpEDF is that τ' be correctly scheduled on $(m-1)$ processors by Algorithm fpEDF. That is

$$\begin{aligned}
&\tau \text{is correctly scheduled on} m \text{processors}\\
&\Leftarrow \text{is correctly scheduled on} m-1 \text{processors}\\
&\Leftarrow U_{\text{sum}}(\tau') \le \frac{(m-1)+1}{2}\\
&\equiv (U_{\text{sum}}(\tau) - U_{\max}(\tau)) \le \frac{m}{2}\\
&\equiv U_{\text{sum}}(\tau) \le U_{\max}(\tau) + \frac{m}{2}
\end{aligned}$$

which is as stated in the theorem. □

8.3.1 A Pragmatic Improvement

The idea underpinning Algorithm fpEDF—circumvent the Dhall effect [78] (see also Example 8.1 above) by assigning greatest priority to jobs of the heaviest (largest-utilization) tasks, and use EDF upon the rest—can be exploited to often obtain performance (i.e., schedulability) superior to that offered by Algorithm fpEDF. We describe one such pragmatic improvement below.

Recall that we are assuming that tasks in the implicit-deadline sporadic task system τ are indexed according to nonincreasing utilization (i.e., $u_i \ge u_{i+1}$ for all i, $1 \le i < n$). Let $\tau^{(k)}$ denote τ with $(k-1)$ largest-utilization tasks removed:

$$\tau^{(k)} \overset{\text{def}}{=} \{\tau_k, \tau_{k+1}, \ldots, \tau_n\} \tag{8.11}$$

and consider the following priority-driven scheduling algorithm:

Algorithm EDF$^{(k)}$ assigns priorities to jobs of tasks in τ according to the following rule:

- For all $i < k$, τ_i's jobs are assigned highest priority (ties broken arbitrarily) — this is trivially achieved within an EDF implementation by setting the deadlines of all jobs of each such τ_i to be equal to $-\infty$.
- For all $i \ge k$, τ_i's jobs are assigned priorities according to EDF.

That is, Algorithm $\mathsf{EDF}^{(k)}$ assigns highest priority to jobs generated by the $k-1$ tasks in τ that have highest utilizations, and assigns priorities according to deadline to jobs generated by all other tasks in τ. (Thus, "pure" EDF is $\mathsf{EDF}^{(1)}$, while fpEDF is $\mathsf{EDF}^{(m'-1)}$, where m' denotes the smaller of the number of processors m and the number of tasks in τ with utilization $> 1/2$.)

Theorem 8.12 *Implicit-deadline sporadic task system τ will be scheduled to meet all deadlines on m unit-speed processors by Algorithm $\mathsf{EDF}^{(k)}$, where*

$$m = (k-1) + \left\lceil \frac{U(\tau^{(k+1)})}{1-u_k} \right\rceil \tag{8.12}$$

(Recall that $\tau^{(\ell)}$ denotes τ with $(\ell-1)$ largest-utilization tasks removed—Eqn 8.11.)

Proof By Theorem 8.3, $\tau^{(k)}$ is feasible on some uniform multiprocessor platform with cumulative computing capacity $U(\tau^{(k)})$, in which the fastest processor has speed u_k. Hence by Theorem 8.4, $\tau^{(k)}$ can be EDF-scheduled upon an identical multiprocessor platform comprised of $\hat{m}$ unit-capacity processors, where

$$\hat{m} \overset{\text{def}}{=} \left\lceil \frac{U(\tau^{(k+1)})}{1-u_k} \right\rceil .$$

It follows from the definition of m (Eq. 8.12) that

$$m = (k-1) + \hat{m} .$$

Now, consider the task system $\tilde{\tau}$ obtained from τ by replacing each task in $(\tau \setminus \tau^{(k)})$—that is, the k tasks in τ with the largest utilizations—by a task with the same period, but with utilization equal to one:

$$\tilde{\tau} \overset{\text{def}}{=} \bigcup_{j=1}^{k-1} \{(T_j, T_j)\} \;\bigcup\; \tau^{(k)} .$$

Let us consider the scheduling of $\tilde{\tau}$ by Algorithm $\mathsf{EDF}^{(k)}$, on m unit-capacity processors (where m is as defined in Eq. 8.12). Notice that Algorithm $\mathsf{EDF}^{(k)}$ will assign identical priorities to corresponding tasks in τ and $\tilde{\tau}$ (where the notion of "corresponding" is defined in the obvious manner). Also notice that when scheduling $\tilde{\tau}$, Algorithm $\mathsf{EDF}^{(k)}$ will

- Devote $(k-1)$ processors exclusively to the $(k-1)$ tasks that generate jobs of highest priority (since each has a utilization equal to unity), and thereby meet the deadlines of all jobs of these tasks.
- Execute EDF on the jobs generated by the remaining tasks (the tasks in $\tau^{(k)}$) upon the remaining $\hat{m}$ processors. As we have seen above, EDF schedules the tasks in $\tau^{(k)}$ upon $\hat{m}$ processors to meet all deadlines;

Hence, Algorithm $\mathsf{EDF}^{(k)}$ schedules $\tilde{\tau}$ upon m processors to meet all deadlines of all jobs

Finally, notice that an execution of Algorithm $\mathsf{EDF}^{(k)}$ upon m processors on task system τ can be considered to be an instantiation of a run of Algorithm $\mathsf{EDF}^{(k)}$ upon m processors on task system $\tilde{\tau}$, in which some jobs—the ones generated by tasks whose jobs are assigned highest priority—do not execute to their full execution requirement. It is straightforward to observe that Algorithm $\mathsf{EDF}^{(k)}$ is a *predictable* scheduling algorithm, and it hence follows from the result of Ha and Liu (Theorem 3.1) that each job of each task during the execution of Algorithm $\mathsf{EDF}^{(k)}$ on task system τ completes no later than the corresponding job during the execution of Algorithm $\mathsf{EDF}^{(k)}$ on task system $\tilde{\tau}$. And, we have already seen above that no deadlines are missed during the execution of Algorithm $\mathsf{EDF}^{(k)}$ on task system $\tilde{\tau}$. □

By Theorem 8.4, $\lceil (U(\tau) - u_1)/(1 - u_1) \rceil$ unit-capacity processors suffice to guarantee that all deadlines of implicit-deadline sporadic task system τ are met, if τ is scheduled using EDF. As the following corollary states, we can often make do with fewer than $\lceil (U(\tau) - u_1)/(1 - u_1) \rceil$ processors if we are not restricted to using the EDF-scheduling algorithm, but may instead choose one of the priority-driven algorithms Algorithm $\mathsf{EDF}^{(k)}$, for some k, $1 \le k < n$.

Corollary 8.2 *Implicit-deadline sporadic task system τ will be scheduled to meet all deadlines on*

$$m_{\min}(\tau) \overset{def}{=} \min_{k=1}^{n} \left\{ (k-1) + \left\lceil \frac{U(\tau^{(k+1)})}{1 - u_k} \right\rceil \right\} \tag{8.13}$$

unit-capacity processors by a priority-driven scheduling algorithm.

Proof Let $k_{\min}(\tau)$ denote the smallest value of k that minimizes the right-hand side of Eq. 8.13:

$$m_{\min}(\tau) \equiv (k_{\min}(\tau) - 1) + \left\lceil \frac{U(\tau^{(k_{\min}(\tau)+1)})}{1 - u_{k_{\min}(\tau)}} \right\rceil$$

It follows directly from Theorem 8.9 that τ can be scheduled to meet all deadlines upon $m_{\min}(\tau)$ unit-speed processors by the priority-driven algorithm Algorithm $\mathsf{EDF}^{(k_{\min}(\tau))}$. □

Algorithm PriD. Based upon Corollary 8.2 above, we propose the following priority-driven scheduling algorithm for scheduling implicit-deadline sporadic task systems upon identical multiprocessors: Given a implicit-deadline sporadic task system $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ with $u_i \le u_{i+1}$ for all i, $1 \le i < n$, Algorithm PriD computes $m_{\min}(\tau)$ according to Eq. 8.13, and schedules τ by Algorithm $\mathsf{EDF}^{(k_{\min}(\tau))}$.

It is evident that if a task system τ is schedulable by Algorithm $\mathsf{EDF}^{(k)}$ for any value of k, then it is schedulable by Algorithm PriD. This is formally stated in the following theorem.

Theorem 8.13 *Any implicit-deadline sporadic task system that is schedulable by Algorithm $\mathsf{EDF}^{(k)}$ for any value of k, $1 \le k \le |\tau|$, is schedulable by Algorithm PriD.*

Algorithm PriD is essentially choosing the “best” value for k — the one that results in the minimum number of processors being needed — for which Algorithm $\mathsf{EDF}^{(k)}$

would schedule a task system τ. It therefore includes consideration of the situation considered by Algorithm fpEDF, and it therefore follows that

Theorem 8.14 *Algorithm* PriD *dominates Algorithm* fpEDF*: each task system deemed scheulable by Algorithm* fpEDF *is also deemed schedulable by Algorithm* PriD*, and there are systems deemed schedulable by Algorithm* PriD *that Algorithm* fpEDF *fails to schedule.*

Sources

Many of the results presented in this chapter build upon the ideas in [153]. The results concerning the scheduling of sporadic task systems upon uniform multiprocessors are from [91, 92]. The utilization bound for EDF was derived in [95]; Algorithm PriD was also first proposed there. Algorithm fpEDF was presented in [29].

Chapter 9
Global Fixed-Task-Priority (FTP) Scheduling of L&L Tasks

In fixed-task-priority (FTP) scheduling algorithms, each task is assigned a distinct priority and all the jobs generated by a task inherit the priority of the task. FTP algorithms are therefore a special case of fixed-job-priority (FJP) scheduling algorithms: all FTP algorithms are also FJP algorithms, while not all FJP algorithms are FTP algorithms.

The global FTP scheduling of L&L task systems has also received a tremendous amount of attention from the real-time scheduling research community. However, we have chosen to not describe the research on global FTP scheduling in as much detail as we did global FJP scheduling in the previous chapter; instead, we will highlight a few particularly relevant results below.

It has long been known [137] that the rate monotonic (RM) scheduling algorithm, which assigns priorities to tasks in order of their period parameters (tasks with smaller period receiving greater priority, with ties broken arbitrarily), is an optimal FTP algorithm for implicit-deadline sporadic task systems upon preemptive uniprocessors, in the sense that if an implicit-deadline sporadic task system can be scheduled to always meet all deadlines upon a preemptive uniprocessor by any FTP algorithm, then it will also be scheduled to always meet all deadlines by RM. RM is no longer optimal for global scheduling upon multiprocessors; nevertheless, it is possible to prove a speedup bound for it—we do so in Sect. 9.1 below. As was the case with EDF (see Example 8.1), RM, too, suffers from the Dhall effect [78]; in Sect. 9.2, we present an FTP algorithm that somewhat circumvents the Dhall effect in much the manner that algorithm fpEDF had done in the case of FJP algorithms (Fig. 8.1 in Sect. 8.3). We briefly enumerate some other results concerning the FTP scheduling of implicit-deadline sporadic task systems, and present an upper bound on the utilization bound of any such scheduling algorithm, in Sect. 9.2.1.

9.1 Global RM Scheduling

In this section, we will study the properties of global RM. We first establish a processor speedup bound in Theorem 9.2; then in Theorem 9.3 we state without proof a utilization bound.

We start out defining a subclass of implicit-deadline sporadic task systems:

Definition 9.1 (RM-light task system) Implicit-deadline sporadic task system $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ is said to be *RM-light*(m) for any positive integer m if it satisfies the conditions

1. $U_{\max}(\tau) \le m/(3m-2)$, and
2. $U_{\text{sum}}(\tau) \le m^2/(3m-2)$.

Recall Theorem 8.3, which asserts that any implicit-deadline sporadic task system τ is feasible upon a uniform multiprocessor platform π with total computing capacity equal to $U_{\text{sum}}(\tau)$ and fastest processor having speed $U_{\max}(\tau)$.

Lemma 9.1 *Let τ denote an RM-light (m) implicit-deadline sporadic task system, π a uniform multiprocessor platform satisfying $S_{\text{sum}}(\pi) = U_{sum}(\tau)$ and $S_{\max}(\pi) = U_{max}(\tau)$, A any work-conserving scheduling algorithm, and π' an m-processor identical multiprocessor platform. The following relationship*

$$W(RM, \pi', I, t) \ge W(A, \pi, I, t) \tag{9.1}$$

holds for any collection of jobs I generated by the task system τ.

Proof By Lemma 8.1, this will hold if

$$\begin{aligned}
& S_{\text{sum}}(\pi') \ge \lambda(\pi') \cdot S_{\max}(\pi) + S_{\text{sum}}(\pi) \\
& \Leftarrow m \ge (m-1)\frac{m}{3m-2} + \frac{m^2}{3m-2} \\
& \Leftrightarrow m \ge \frac{2m^2 - m}{3m-2} \\
& \Leftrightarrow 3m^2 - 2m \ge 2m^2 - m \\
& \Leftrightarrow m^2 - m \ge 0 \\
& \Leftrightarrow m \ge 1
\end{aligned}$$

which is true. □

Theorem 9.1 *Any RM-light (m) implicit-deadline sporadic task system is scheduled by algorithm RM to meet all deadlines on an identical multiprocessor platform consisting of m unit-speed processors.*

Proof Suppose that this is not true. Let $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$ denote a minimal counterexample to the correctness of this theorem. Let I denote some minimal collection of jobs generated by τ for which RM misses a deadline for the first time at time-instant t_f—see Fig. 9.1. We observe that this must be a deadline of a job of τ_n (since if it were a deadline of a job of τ_k for some $k < n$ then the assumption that τ is a *minimal* counterexample would be violated), and lets $t_a = t_f - T_n$ denote the arrival time of this job.

As in the statement of Lemma 9.1 above, let π denote a uniform multiprocessor platform satisfying $S_{\text{sum}}(\pi) = U_{\text{sum}}(\tau)$ and $S_{\max}(\pi) = U_{\max}(\tau)$, and π' the

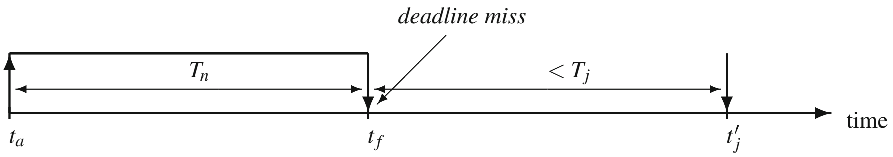


Fig. 9.1 Proof of Theorem 9.1: notation

m-processor identical multiprocessor platform. Let OPT denote an optimal algorithm executing on π, that executes each job of each task τ_i at a rate exactly equal to its utilization throughout the interval between its arrival time and its deadline.

Let us suppose that the latest deadline of any job of task τ_j (for $j < n$) in I is at time-instant t'_j. We observe that

- $t'_j < t_f + T_j$ (since any job of τ_j with deadline $\geq t_f + T_j$ arrives after t_f, and hence cannot be responsible for the deadline miss—this would contradict the assumed minimality of I); and
- The amount of execution for τ_j that remains to be done after time-instant t_a by OPT executing on π is bounded from above by $u_j \cdot (t_f + T_j - t_a)$, or $u_j \cdot (T_n + T_j)$.

Hence, the total amount of execution that remains to be done after time-instant t_a by OPT executing on π is bounded from above by

$$C_n + \sum_{j=1}^{n-1}(u_j \cdot (T_n + T_j)), \text{ which equals } C_n + T_n \sum_{j=1}^{n-1} u_j + \sum_{j=1}^{n-1} C_j.$$

It follows from Lemma 9.1 that RM executing on π' has completed at least as much total work as OPT has, when executing on π, by time-instant t_a. Hence, the amount of execution remaining for RM after time-instant t_a is also upper-bounded by the expression above.

Since the total processor capacity on the m-processor unit-speed identical multiprocessor platform π' over the duration $[t_a, t_f)$ is equal to mT_n, the amount of processor capacity left over for τ_n's job, after the higher-priority jobs have executed, is bounded from below by

$$mT_k - \left(T_n\left(\sum_{j=1}^{n-1} u_j\right) + \sum_{j=1}^{n-1} C_j\right);$$

in the worst case (i.e., to minimize availability for τ_n's job), this will all occur simultaneously upon all m processors. The duration for which τ_n's job gets to execute is therefore bounded from below by

$$T_k - \frac{1}{m}\left(T_n \sum_{j=1}^{n-1} u_j + \sum_{j=1}^{n-1} C_j\right).$$

For a deadline miss, we need this to be strictly less than C_n:

$$T_n - \frac{1}{m}\left(T_n \sum_{j=1}^{n-1} u_j + \sum_{j=1}^{n-1} C_j\right) < C_n$$

$$\Leftrightarrow \frac{C_n}{T_n} + \frac{1}{m}\left(\sum_{j=1}^{n-1} u_j + \sum_{j=1}^{n-1} \frac{C_j}{T_n}\right) > 1$$

$$\Leftarrow u_n + \frac{1}{m}\left(2\sum_{j=1}^{n-1} u_j\right) > 1 \text{ (Since } T_n \geq T_j)$$

Simplifying the LHS:

$$\begin{aligned}
& u_n + \frac{1}{m}\left(2\sum_{j=1}^{n-1} u_j\right) \\
&= u_n + \frac{1}{m}\left(2U_{\text{sum}}(\tau) - 2u_n\right) \\
&= \left(1 - \frac{2}{m}\right) u_n + \frac{2}{m} U_{\text{sum}}(\tau) \\
&\leq \left(1 - \frac{2}{m}\right)\left(\frac{m}{3m-2}\right) + \frac{2}{m}\left(\frac{m^2}{3m-2}\right) \\
&= 1
\end{aligned}$$

thereby contradicting the requirement that it be > 1. □

Theorem 9.2 *The processor speedup factor for RM scheduling of implicit-deadline sporadic task systems is* $(3 - 2/m)$.

Proof Sketch: We will show that any τ feasible upon a platform comprised of m speed-$\left(\frac{m}{3m-2}\right)$ processors is RM-schedulable upon m unit-speed processors. The theorem follows, by observing that $\left(1/\frac{m}{3m-2}\right)$ equals $(3 - 2/m)$.

Suppose that implicit-deadline sporadic task system τ is feasible upon m speed-$\left(\frac{m}{3m-2}\right)$ processors. It must be the case that

1. $U_{\text{max}}(\tau) \leq m/(3m-2)$
2. $U_{\text{sum}}(\tau)(\tau) \leq m^2/(3m-2)$.

τ is therefore an MS-light(m) task system. By Theorem 9.1, we conclude that τ is therefore RM-schedulable on m unit-speed processors □

A better utilization bound for global RM was proved in [59]:

Theorem 9.3 *(from [59]) Any implicit deadline periodic or sporadic task system τ satisfying*

$$U_{sum}(\tau) \leq \frac{m}{2}\left(1 - U_{max}(\tau)\right) + U_{max}(\tau) \qquad (9.2)$$

is successfully scheduled by RM on m unit-speed processors.

Algorithm RM-US(ξ)
Implicit-deadline sporadic task system $\tau = \{\tau_1, \tau_2, \ldots \tau_n\}$ to be scheduled on m processors
(It is assumed that $u_i \geq u_{i+1}$ for all i, $1 \leq i < n$)

```
for $i = 1$ to $(m-1)$ do
    if $(u_i > \xi)$
        then $\tau_i$ is assigned highest priority
        else break
the remaining tasks are assigned priorities according to RM
```

Fig. 9.2 Algorithm RM-US(ξ) priority-assignment rule

The above result is tighter than the one of Theorem 9.1, as can be seen applying Eq. 9.2 to task sets having $U_{\max}(\tau) \leq \frac{m}{3m-2}$. The RHS of Eq. 9.2 becomes

$$\begin{aligned}
&\frac{m}{2}(1 - U_{\max}(\tau)) + U_{\max}(\tau)\\
&= \frac{m}{2}\left(1 - U_{\max}(\tau)\left(1 - \frac{2}{m}\right)\right)\\
&\geq \frac{m}{2}\left(1 - \left(\frac{m}{3m-2}\right)\left(\frac{m-2}{m}\right)\right)\\
&= \frac{m}{2}\left(\frac{3m-2-m+2}{3m-2}\right)\\
&= \frac{m^2}{3m-2}
\end{aligned}$$

thereby establishing that the bound on $U_{\text{sum}}(\tau)$ is no smaller than $\frac{m^2}{3m-2}$.

Since Theorem 9.3 can also be applied to task systems τ with $U_{\max}(\tau) > \frac{m}{3m-2}$, it follows that it dominates the result of Theorem 9.1.

9.2 Global FTP Scheduling

Recall from Sect. 8.3, that the *Dhall effect* [78] (see Example 8.1) was responsible for high-utilization tasks compromising schedulability, and that the Dhall effect could sometimes be circumvented by assigning greater priority to jobs generated by high-utilization tasks. This same idea is explored in FTP scheduling as well—consider the algorithm RM-US(ξ) defined in Fig. 9.2.

The following result was proved in [6].

Theorem 9.4 *Algorithm RM-US($m^2/(3m-2)$) has a utilization bound (Definition 5.1) no smaller than $m^2/(3m-2)$ upon m unit-speed processors.*

Proof Let k_o denote the number of tasks in τ that have utilization $> m/(3m-2)$, and let $m_o \stackrel{\text{def}}{=} (m - k_o)$.

Let us consider the task system $\hat{\tau}$ obtained from τ by removing all the tasks with utilization $> m/(3m-2)$. The utilization of $\hat{\tau}$ is bounded as follows:

$$\begin{aligned} U_{\text{sum}}(\hat{\tau}) &< U_{\text{sum}}(\tau) - k_o \cdot \frac{m}{3m-2} \\ &\leq \frac{m^2}{3m-2} - \frac{k_o m}{3m-2} \\ &= \frac{m(m-k_o)}{3m-2} \\ &\leq \frac{(m-k_o)^2}{3(m-k_o)-2} \\ &\leq \frac{m_o^2}{3m_o-2} \end{aligned}$$

In addition, each $\tau_i \in \hat{\tau}$ satisfies

$$u_i \leq \frac{m}{3m-2} \leq \frac{m_o}{3m_o-2}$$

By Definition 9.1 we conclude that $\hat{\tau}$ is RM-light(m_o); by Theorem 9.1, it is therefore RM-schedulable on m_o processors. Since τ is obtained from $\hat{\tau}$ by adding k_o tasks, each with utilization $> m/(3m-2)$, and $m = m_o + k_o$, we can now use essentially the same argument as the one used in the proof of Theorem 8.12 to conclude that RM-US($m/(3m-2)$ will consequently schedule τ to always meet all deadlines upon m processors. □

An improvement to the above utilization bound was presented in [59]; since proving this bound required the development of a new schedulability test for uniprocessor RM, we state the result without proof here.

Theorem 9.5 *(Corollary 2 in [59]) Algorithm RM-US*$(1/3)$ *has a utilization bound of* $(m+1)/3$ *upon m unit-speed processors.*

Algorithm RM-US(ξ) is based on extending global RM in order to overcome the Dhall effect. An approach not based on RM scheduling at all was explored in [5]: rather than assigning priorities according to period, they are assigned according to slack, where the *slack* of a task $\tau_i = (C_i, T_i)$ is defined to be $(T_i - C_i)$. Algorithm SM-US(ξ) in [5] assigns greatest priority to tasks with utilization $> \xi$, just as algorithm RM-US(ξ) did. For the remaining tasks, however, algorithm SM-US(ξ) assigns priorities according to slack $\stackrel{\text{def}}{=} (T_i - C_i)$, with tasks having smaller slack being assigned greater priority. It is proved in [5] that the best utilization bound is obtained by setting $\xi \leftarrow 2/(3+\sqrt{5})$ or ≈ 0.382; the corresponding utilization bound for this value of ξ is $\approx 0.382m$:

Theorem 9.6 *(Theorem 1 in [5]) Algorithm SM-US*$(2/(3+\sqrt{5}))$ *has a schedulable utilization of* $\left(\frac{2m}{3+\sqrt{5}}\right)$ *upon m unit-speed processors.*

Since $\left(\frac{2m}{3+\sqrt{5}}\right) > (m+1)/3$ for $m \geq 7$, it follows that this approach offers a superior utilization bound to RM-US upon platforms with seven or more processors.

9.2.1 Upper Bounds

Baker [18, 21] showed that the utilization bound of global RM upon m unit-speed processors, expressed as a function of the maximum utilization of any individual task (denoted α), is no larger than

$$\alpha + m \ln\left(\frac{2}{1+\alpha}\right).$$

This bound was obtained by adapting ideas from [137] to construct a task system that is "barely schedulable" in the sense that an arbitrarily small increase in the total utilization of such a task set would result in a missed deadline.

An upper bound of $\left(\frac{m}{2} + \frac{1}{3}\right)$ on the utilization bound of *any* global FTP algorithm for scheduling implicit-deadline sporadic task systems was obtained in [9], via the following example:

Example 9.1 Consider a task set composed by m tasks τ_i with $C_i = 1$ and $T_i = 2$, and another task τ_{m+1} with $C_i = 1$ and $T_i = 3$. If we increase by an arbitrarily small ϵ the worst-case computation times of tasks $\tau_1, \ldots, \tau_m$, there is no priority assignment that can positively schedule the task set: if we give lowest priority to τ_{m+1}, then it will miss its deadline at time $t = 3$; otherwise, if τ_{m+1}'s priority is higher than the priority of one of the other tasks, the latter task will miss its deadline at time $t = 2$.

Sources

The derivation of the speedup bound for RM, and algorithm RM-US(ξ) as well as the derivation of its utilization bound, may be found in [4, 6]. As stated above, the improved utilization bound in Theorem 9.5 is from [59]. Some important ideas and insights into global multiprocessor scheduling are to be found in [7–10].

Chapter 10
The Three-Parameter Sporadic Tasks Model

We now move on from the implicit-deadline sporadic tasks model: this and the next several chapters discuss the multiprocessor scheduling and analysis of task systems that are modeled using the more general three-parameter sporadic tasks model. There is a lot of ground to cover here, particularly with regards to global scheduling. Some of the results and techniques that we will discuss may be considered to be generalizations of ideas previously introduced in the context of implicit-deadline sporadic task systems; many others are brand new and were developed specifically for three-parameter task systems.

In this chapter, we will briefly review the three-parameter sporadic tasks model, and introduce some functions for quantifying the computational demand of 3-parameter sporadic tasks, and of systems of such tasks. We will describe algorithms and analyses for partitioned scheduling in Chap. 11. Discussion on global scheduling algorithms and associated schedulability analysis will follow in the next nine chapters. Global scheduling of task systems more general than implicit-deadline ones is possibly the most intellectually challenging topic studied in depth in multiprocessor real-time scheduling theory, requiring the concurrent consideration of several factors and issues that do not arise in the scheduling and analysis of either implicit-deadline task systems or partitioned scheduling of three-parameter sporadic task systems. The general approach we adopt here is as follows.

- In Chap. 12 we make some observations regarding global scheduling of three-parameter sporadic task systems, that seek to highlight some of the challenges we face in attempting to devise suitable scheduling and analysis algorithms applicable to such systems.
- Many of the utilization-based results on the scheduling of implicit-deadline task systems can also be applied to three-parameter task systems by simply substituting *density* for utilization—this is explored in Chap. 13.
- In Chap. 14, we describe, at a rather high level, a general framework for developing global schedulability analysis tests. It turns out that most global schedulability analysis tests that have been devised can be viewed within this framework, as specific instantiations.

S. Baruah et al., *Multiprocessor Scheduling for Real-Time Systems,*
Embedded Systems, DOI 10.1007/978-3-319-08696-5_10

- Chapter 15 describes a couple of relatively widely-studied earliest-deadline-first (EDF) schedulability tests, the [BCL] and [BAR] tests, as instantiations of the general framework described in Chap. 14.
- Chapter 16 describes another widely-studied test, the [BAK] test. The [BAK] test is distinguished from the density tests of Chap. 13, as well as the [BCL] and [BAR] tests, in that quantitative worst-case bounds on its performance were derived, in terms of the speedup factor metric (Definition 5.2).
- Chapter 17 describes several enhancements to the [BCL] and [BAR] tests that appear (based on the evidence of extensive schedulability experiments on synthetically generated task-sets), to significantly improve the performance capabilities of these tests in terms of their ability to correctly identify schedulable task systems.
- Chapter 18 describes schedulability tests for task systems scheduled using global fixed-task-priority (FTP) scheduling algorithms.
- Most of the results in Chaps. 13–18 have been evaluated via schedulability experiments on synthetically generated task-sets (the main exception is the speedup bound for a variant of the [BAK] test, which we present in Chap. 16). The results we describe in Chap. 19 provide a primarily quantitative approach to global schedulability analysis. Specifically,
  - We characterize the capabilities and limitations of both the global fixed-job-priority (FJP) scheduling algorithm EDF and the global FTP scheduling deadline monotonic (DM) algorithm upon three-parameter sporadic task systems by deriving tight speedup bounds for these algorithms.
  - We describe corresponding schedulability tests that have pseudo-polynomial runtime complexity and near-optimal speedup factor.
  - We detail pragmatic improvements to these near-speedup-optimal schedulability tests that further improve their performance.

10.1 The Three-Parameter Sporadic Task Model

We start out briefly reviewing the task and machine model we will be considering in this chapter. Recall that a *3-parameter sporadic task* ([147]; henceforth often referred to simply as a sporadic task), is characterized by three parameters—a *worst-case execution requirement* (WCET) C_i, a *relative deadline* D_i, and a *period* (or *inter-arrival separation parameter*) T_i. A 3-parameter sporadic task denoted τ_i is thus represented by a 3-tuple of parameters: $\tau_i = (C_i, D_i, T_i)$. Such a task generates a potentially infinite sequence of jobs. The first job may arrive at any instant, and the arrival times of any two successive jobs are at least T_i time units apart. Each job has a WCET of C_i, and a deadline that occurs D_i time units after its arrival time. A 3-parameter sporadic *task system* consists of a finite number of such 3-parameter sporadic tasks executing upon a shared platform. A task system is often denoted as τ, and described by enumerating the tasks in it: $\tau = \{\tau_1, \tau_2, \ldots, \tau_n\}$.

Depending upon the relationship between the values of the relative deadline and period parameters of the tasks in it, a 3-parameter sporadic task system may further be classified as follows:

- In an *implicit-deadline* task system, the relative deadline of each task is equal to the task's period: $D_i = T_i$ for all $\tau_i \in \tau$. Implicit-deadline task systems are also called *Liu & Layland task systems*, since they were popularized in a paper [137] coauthored by C. L. Liu and J. Layland. We studied the scheduling of such task systems in Chaps. 5–9.
- In a *constrained-deadline* task system, the relative deadline of each task is no larger than that task's period: $D_i \le T_i$ for all $\tau_i \in \tau$.
- Tasks in an *arbitrary-deadline* task system do not need to have their relative deadlines satisfy any constraint with regards to their periods.

It is evident from these definitions that each implicit-deadline task system is also a constrained-deadline task system, and each constrained-deadline task system is also an arbitrary-deadline task system.

Recall, from Sect. 2.1.2, the following notation

- The *utilization* u_i of a task τ_i is the ratio C_i / T_i of its execution requirement to its period. The total utilization $U_{\text{sum}}(\tau)$ and the largest utilization $U_{\max}(\tau)$ of a task system τ are defined as follows:

$$U_{\text{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} u_i; \qquad U_{\max}(\tau) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau} (u_i)$$

- The *density* dens_i of a task τ_i is the ratio $(C_i/\min(D_i, T_i))$ of its execution requirement to the smaller of its relative deadline and its period. The total density $\text{dens}_{\text{sum}}(\tau)$ and the largest density $dens_{\max}(\tau)$ of a task system τ are defined as follows:

$$\text{dens}_{\text{sum}}(\tau) \stackrel{\text{def}}{=} \sum_{\tau_i \in \tau} \text{dens}_i; \qquad \text{dens}_{\max}(\tau) \stackrel{\text{def}}{=} \max_{\tau_i \in \tau} (\text{dens}_i)$$

10.2 Characterizing a Task's Demand

To determine whether a task system is schedulable or not upon a particular platform, it is necessary to determine bounds upon the amount of execution that could be generated by the tasks in the system. An obvious upper bound is obtained by observing that the maximum cumulative value of execution by jobs of a task τ_i with both arrival times and deadlines within an interval of duration t is $t \times \text{dens}_i$ (this result is formally stated in Lemma 10.1 below). A tighter bound called the demand bound function (DBF) has been identified—this is discussed in Sect. 10.3 below. The DBF essentially bounds the demand over an interval by the sum of the WCETs of all the jobs that may have both arrival times and deadline within the interval.

The DBF finds widespread use in uniprocessor schedulability analysis. For multiprocessor scheduling, it was observed [24, 62] that it was useful to consider some jobs arriving and/or having deadlines outside an interval, since such jobs could also sometimes contribute to the cumulative execution requirement placed on the computing platform within the interval. These observations motivated the definition of a refinement of the DBF called the forced-forward DBF, which we mention briefly in Sect. 10.4, and discuss in detail in Chap. 19.

It may sometimes be necessary to determine how much execution demand by a task can *arrive* over an interval of a particular duration. The quantification of such execution demand is formalized in Sect. 10.5 as the request bound function (RBF).

10.3 The Demand Bound Function

For any sporadic task τ_i and any real number $t \geq 0$, the *demand bound function* $\text{DBF}(\tau_i, t)$ is the largest cumulative execution requirement of all jobs that can be generated by τ_i to have both their arrival times and their deadlines within a contiguous interval of length t. It is evident that the cumulative execution requirement of jobs of τ_i over an interval $[t_o, t_o + t)$ is maximized if one job arrives at the start of the interval—i.e., at time-instant t_o—and subsequent jobs arrive as rapidly as permitted — i.e., at instants $t_o + T_i, t_o + 2T_i, t_o + 3T_i, \ldots$ (this fact is formally proved in [51]). Equation (10.1) below follows directly [51]:

$$\text{DBF}(\tau_i, t) \overset{\text{def}}{=} \max\left(0, \left(\left\lfloor \frac{t - D_i}{T_i} \right\rfloor + 1\right) \times C_i\right) \tag{10.1}$$

The following Lemma relates the density of a task to its demand bound function DBF.

Lemma 10.1 *For all tasks τ_i and for all $t \geq 0$,*

$$t \times dens_i \geq \text{DBF}(\tau_i, t) .$$

Proof : This lemma is easily validated informally by sketching $\text{DBF}(\tau_i, t)$ as a function of t, and comparing this with the graph for $t \times \text{dens}_i$, a straight line of slope $(C_i / \min(D_i, T_i))$ through the origin (see Fig. 10.1). $\text{DBF}(\tau_i, t)$ is a step function comprised of steps of height C_i, with the first step at $t = D_i$, and successive steps exactly T_i time units apart. It is seen that the graph of $t \times \text{dens}_i$ lies above the plot for $\text{DBF}(\tau_i, t)$, for all t. (For $D_i < T_i$, the graph for $t \times \text{dens}_i$ touches the plot for $\text{DBF}(\tau_i, t)$ at $t = D_i$; for $D_i = T_i$, the two touch at all-integer multiples of T_i; and for $D_i > T_i$ the two plots never touch). □

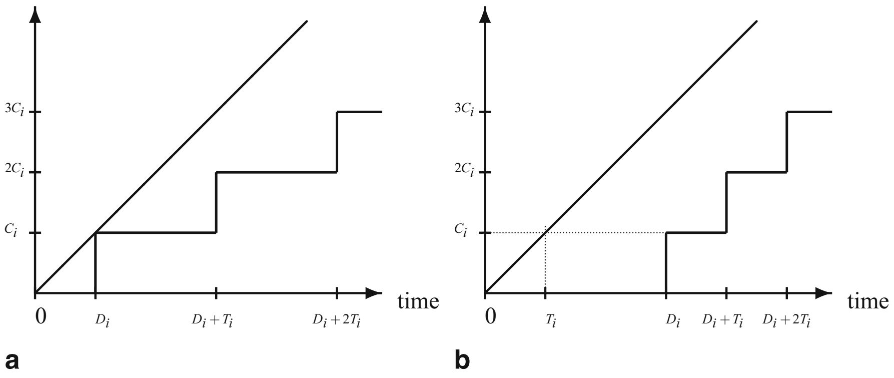


Fig. 10.1 Illustrating the proof of Lemma 10.1. The step plots represent DBF(τ_i, t), and the straight lines through the origin represent $t \times \text{dens}_i$. In **a** $D_i < T_i$, while in **b** $D_i > T_i$

The DBF of a task system τ is defined to be the sum of the DBF's of the individual tasks in τ; the LOAD of τ is defined to be largest value of DBF normalized by interval length:

$$\text{DBF}(\tau, t) \overset{\text{def}}{=} \sum_{\tau_\ell \in \tau} \text{DBF}(\tau_\ell, t)\ . \tag{10.2}$$

$$\text{LOAD}(\tau) \overset{\text{def}}{=} \max_{t>0} \left(\frac{\text{FF-DBF}(\tau, t)}{t} \right)\ . \tag{10.3}$$

10.3.1 Approximating the demand bound function

Computing DBF is a well-studied subject. It follows from results in [80] that we are unlikely to be able to compute DBF in polynomial time; however, very efficient pseudo-polynomial times algorithms are known for computing DBF [51, 160]. Polynomial-time algorithms for computing DBF approximately to any desired degree of accuracy have also been designed [26, 85, 88]. Equation 10.4 below gives such an approximation scheme for DBF; for any fixed value of k, DBF$^{(k)}(\tau_i, t)$ defines an approximation of DBF(τ_i, t) that is exact for the first k steps of DBF(τ_i, t), and an upper bound for larger values of t:

$$\text{DBF}^{(k)}(\tau_i, t) = \begin{cases} \text{DBF}(\tau_i, t) & \text{if t} \le (k-1)T_i + D_i \\ C_i + (t - D_i)u_i & \text{otherwise} \end{cases} \tag{10.4}$$

The following lemma provides a quantitative bound on the degree by which DBF$^{(k)}$ may deviate from DBF:

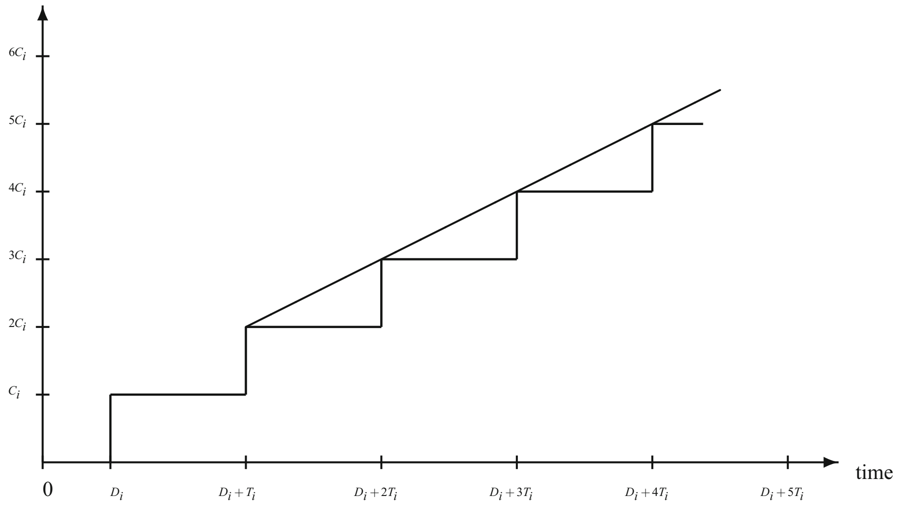


Fig. 10.2 Illustrating the proof of Lemma 10.2. The step plot represents $\text{DBF}(\tau_i, t)$. The plot for $\text{DBF}^{(k)}(\tau_i, t)$, for $k = 2$, is identical to the step plot for $t \le d_i + T_i$, and is denoted by the straight line for larger t

Lemma 10.2

$$\text{DBF}(\tau_i, t) \le \text{DBF}^{(k)}(\tau_i, t) < \left(1 + \frac{1}{k}\right) \text{DBF}(\tau_i, k)\ .$$

Proof: This lemma is easily validated informally by sketching $\text{DBF}(\tau_i, t)$ and $\text{DBF}^{(k)}(\tau, t)$ as functions of t for given k (see Fig. 10.2). As stated earlier in the proof of Lemma 10.1, $\text{DBF}(\tau_i, t)$ is a step function comprised of steps of height C_i, with the first step at $t = D_i$ and successive steps exactly T_i time units apart. The graph of $\text{DBF}^{(k)}(\tau, t)$ tracks the graph for $\text{DBF}(\tau_i, t)$ for the first k steps, and is a straight line with slope C_i / T_i after that. It is evident from the figure that $\text{DBF}^{(k)}(\tau_i, t)$ is always $\ge \text{DBF}(\tau_i, t)$,and that the ratio $\text{DBF}^{(k)}(\tau_i, t)/\text{DBF}(\tau_i, t)$ is maximized at t just a bit smaller than $kT_i + D_i$, where it is $< (k + 1)C_i)/(kC_i) = (1 + \frac{1}{k})$ as claimed. □

10.4 The Forced-Forward Demand Bound Function

It was observed [24, 62] that some jobs arriving and/or having deadlines outside an interval could also contribute to the cumulative execution requirement placed on the computing platform within the interval. This observation led to the introduction of closely related notions called *minimum demand* [24] and *forced-forward demand* [62]. We will discuss this characterization of a task's demand further in a later chapter (Sect. 19.1).

10.5 The Request-Bound Function

For any sporadic task τ_i and any real number $t \geq 0$, the *request-bound function* $\textsc{rbf}(\tau_i, t)$ is the largest cumulative execution requirement of all jobs that can be generated by τ_i to have their *arrival times* within a contiguous interval of length t, regardless of where their deadline may lie. The following function provides an upper bound on the total execution time requested by task τ_i at time t:

$$\textsc{rbf}(\tau_i, t) \overset{\text{def}}{=} \left\lceil \frac{t}{T_i} \right\rceil C_i. \tag{10.5}$$

The following function $\textsc{rbf}^*$ approximating $\textsc{rbf}$ has been proposed [86]:

$$\textsc{rbf}^*(\tau_i, t) \overset{\text{def}}{=} C_i + u_i \times t. \tag{10.6}$$

As stated earlier, it was shown [51] that the cumulative execution requirement of jobs of τ_i over an interval is maximized if one job arrives at the start of the interval, and subsequent jobs arrive as rapidly as permitted. Intuitively, approximation $\textsc{rbf}^*$ (Eq. 10.6 above) models this job-arrival sequence by requiring that the first job's deadline be met explicitly by being assigned C_i units of execution upon its arrival, and that τ_i be assigned an additional $u_i \times \Delta t$ of execution over time-interval $[t, t + \Delta t)$, for all instants t after the arrival of the first job, and for arbitrarily small positive Δt.

10.5.1 Relationship Between $\textsc{rbf}^*$ *and* $\textsc{dbf}$

These lemmas express upper bounds on the approximation $\textsc{rbf}^*(\tau_i, t)$ in terms of τ_i's utilization and demand-bound function.

Lemma 10.3 *The following inequality holds for any constrained-deadline*[1] *sporadic task* $\tau_i = (C_i, D_i, T_i)$ *and all* $t \geq D_i$,

$$\textsc{rbf}^*(\tau_i, t) \leq 3\textsc{dbf}(\tau_i, t) \tag{10.7}$$

Proof : This is visually evident by inspecting Fig. 10.3. The worst-case ratio occurs at the time-instant t just prior to $D_i + T_i$, when $\textsc{dbf}(\tau_i, t)$ has value $2C_i$ while $\textsc{rbf}^*(\tau_i, t)$ has value

$$< (C_i + u_i(D_i + T_i)) = \left(C_i + \frac{C_i}{T_i}(D_i + T_i)\right) \leq 3C_i$$

(since $D_i \leq T_i$ for constrained-deadline systems, and hence $\frac{C_i}{T_i}(D_i + T_i) \leq 2C_i$). $\square$

[1] Recall that a constrained-deadline task has $D_i \leq T_i$.

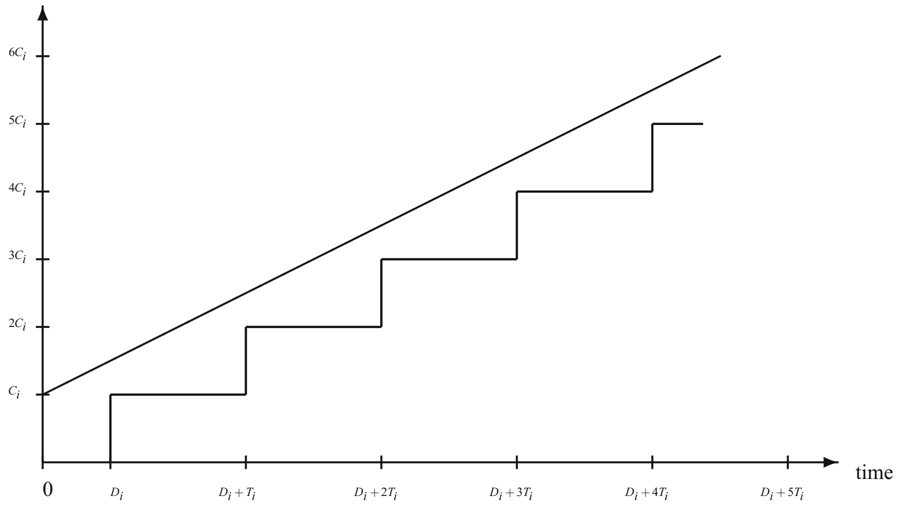


Fig. 10.3 Illustrating the proof of Lemma 10.3. The step plot represents DBF(τ_i, t). The plot for RBF$^*(\tau_i, t)$, is denoted by the straight line with a slope equal to u_i and y-intercept C_i

Lemma 10.4 *Given a sporadic task τ_i, the following inequality holds for all $t \geq D_i$,*

$$\text{RBF}^*(\tau_i, t) \leq \text{DBF}(\tau_i, t) + (u_i \times t) \tag{10.8}$$

Proof : Observe from the definition of DBF that for $t \geq d_i$,

$$\text{DBF}(\tau_i, t) \geq C_i.$$

Substituting this inequality into Eq. 10.6, we obtain Eq. 10.7. □

Sources

The demand bound function was originally defined in [51]; the idea of approximating it as the DBF* function was proposed by Albers and Slomka [3]. Further discussions on such approximations may be found in Fisher's dissertation [83].

Chapter 11
Partitioned Scheduling

In this chapter, we consider approximation algorithms for partitioning a sporadic task system upon an identical multiprocessor platform. Since earliest-deadline-first (EDF) is known to be an optimal algorithm for scheduling upon a preemptive uniprocessor, we will assume in this chapter that each individual processor will be EDF-scheduled during run-time. We will consider approximate approaches to partitioning when the scheduling algorithms upon each processor are restricted to being fixed-task-priority (FTP) ones, in Sect. 11.3.

As we saw in Chap. 6 with respect to the partitioned scheduling of systems of Liu and Layland tasks, most partitioning algorithms operate as follows:

1. *Order* the tasks according to some criterion.
2. Considering the tasks in this order, attempt to *assign* each to a processor upon which it fits. (Here, "fits" requires that the newly-assigned task plus all the tasks previously assigned to the processor, remain uniprocessor EDF-schedulable.)

11.1 Density-Based Partitioning

One approach to partitioning would be to consider the tasks in some order, and to assign tasks to processors such that the total density assigned to a particular processor does not exceed the capacity of the processor.

The correctness of this algorithm follows from the fact that a sufficient condition for a sporadic task system to be EDF-schedulable on a preemptive uniprocessor is that the total density of the system not exceed the capacity of the processor. It is fairly straightforward to show that such density-based partitioning guarantees to successfully schedule any sporadic task system τ upon m unit-capacity processors, for which

$$\text{dens}_{\text{sum}}(\tau) \leq m - (m-1) \times \text{dens}_{\text{max}}(\tau). \tag{11.1}$$

Although density-based partitioning is simple, it may perform very poorly–Sect. 13.1, in a later chapter, discusses the limitations of density as a criterion for scheduling three-parameter sporadic task systems. For instance, the following task

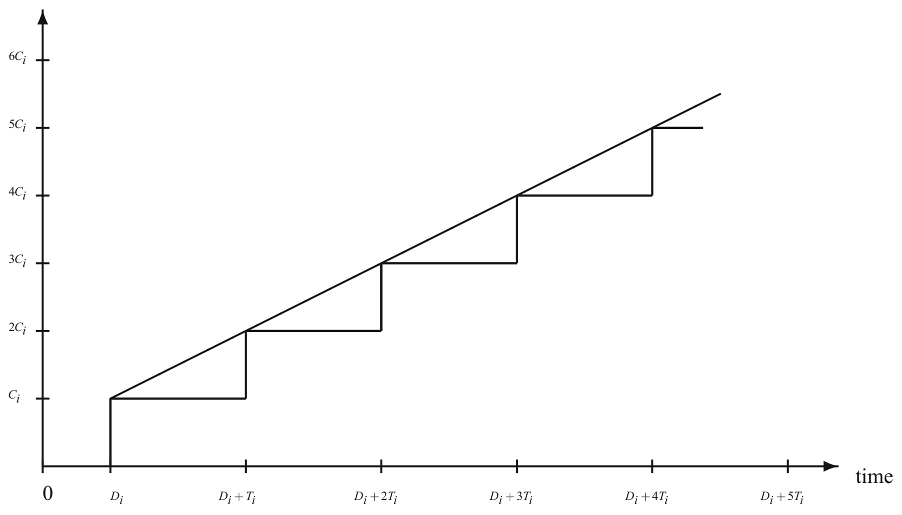


Fig. 11.1 DBF$^{(1)}$ as an approximation of the DBF

system (see also Example 13.1) is one on which density-based partitioning performs poorly.

Consider a task system with n tasks $\tau_1, \ldots, \tau_n$, with $\tau_i = (1, i, n)$. This system is feasible upon a unit-capacity uniprocessor for all values of n (this may be validated by observing that its synchronous arrival sequence is successfully scheduled by EDF—each τ_i's jobs execute over all the time intervals $[(i-1) \bmod n, i \bmod n))$. This task system's density is given by the harmonic sum $\sum_{i=1}^{n} \frac{1}{i}$, which increases without bound as n increases—for any fixed number of processors m there is a value of n for which this harmonic sum exceeds m.

11.2 Partitioning Based on Approximations to DBF

Recall the function DBF$^{(k)}$, an approximation to the DBF, that was defined in Eq. 10.4 (Sect. 10.3.1) for any fixed positive integer value of the integer k. For $k \leftarrow 1$, Eq. 10.4 instantiates to

$$\text{DBF}^{(1)}(\tau_i, t) \stackrel{\text{def}}{=} \begin{cases} 0, & t < D_i \\ C_i + (t - D_i)u_i, & t \geq D_i. \end{cases}$$

In essence, DBF$^{(1)}(\tau_i, t)$ tracks DBF(τ_i, t) exactly for $t \leq D_i$, and thenceforth represents a linear approximation of DBF(τ_i, t) by a straight line with slope U_i—see Fig. 11.1. This approximation upper bounds DBF(τ_i, t) at all other values of t, and is exactly equal to DBF(τ_i, t) at all values of t that are of the form $D_i + kT_i$ for all integer $k \geq 0$. Figure 11.2 depicts an alternative interpretation of the DBF$^{(1)}$ abstraction.

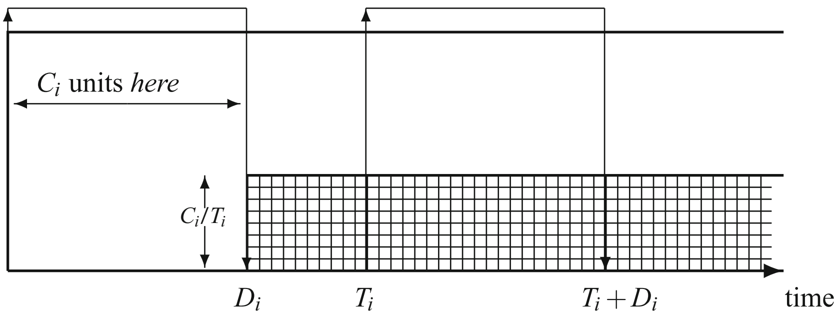


Fig. 11.2 Pictorial representation of task τ_i's reservation of computing capacity in a processor-sharing schedule

Intuitively, we can think of $\text{DBF}^{(1)}(\tau_i, t)$ as accommodating τ_i's worst-case computational demand by "reserving" C_i units of computing capacity for τ_i over the time interval $[0, D_i)$, followed by a processor-sharing schedule (see the discussion following Theorem 7.1) that reserves a fraction u_i of a processor for τ_i for all time-instants $\geq D_i$.

From Lemma 10.2, we conclude the following property of the $\text{DBF}^{(1)}$ approximation to the DBF: for all τ_i and t,

$$\text{DBF}(\tau_i, t) \leq \text{DBF}^{(1)}(\tau_i, t) < 2 \times \text{DBF}(\tau_i, t), \tag{11.2}$$

with the "worst case" (the factor of 2) occurring just prior to $t = (D_i + T_i)$.

A partitioning algorithm for sporadic task systems was derived in [44–46], based upon using the $\text{DBF}^{(1)}$ approximation in order to determine whether a task would "fit" upon a processor during the process of partitioning the tasks amongst the processors. Let τ denote a sporadic task system comprised of the n tasks $\tau_1, \tau_2, \ldots, \tau_n$, to be partitioned upon a platform of m unit-capacity processors. With no loss of generality, let us assume that the tasks in τ are indexed according to nondecreasing order of their relative deadline parameters (i.e., $D_i \leq D_{i+1}$ for all i, $1 \leq i < n$). The algorithm of [44], represented in pseudocode form in Fig. 11.3, considers the tasks in the order $\tau_1, \tau_2, \ldots$. Suppose that the $(i-1)$ tasks $\tau_1, \tau_2, \ldots, \tau_{i-1}$ with the smallest deadlines have all been successfully allocated, and task τ_i is being considered. Let $\tau(\ell)$ denote the tasks that have already been allocated to the ℓ'th processor, $1 \leq \ell \leq m$. Task τ_i is assigned to any processor k that satisfies the following two conditions:

$$\left(D_i - \sum_{\tau_j \in \tau(k)} \text{DBF}^{(1)}(\tau_j, D_i) \right) \geq C_i \tag{11.3}$$

and

$$\left(1 - \sum_{\tau_j \in \tau(k)} C_j/T_j \right) \geq C_i/T_i \;. \tag{11.4}$$

If no such processor exists, then the algorithm declares failure: it is unable to partition τ upon the m-processor platform.

```
PARTITION($\tau, m$)
    ▷ The collection of sporadic tasks $\tau = \{\tau_1,\ldots,\tau_n\}$ is to be partitioned on $m$ identical,
      unit-capacity processors. (Tasks are indexed according to non-decreasing value of
      relative deadline parameter: $D_i \le D_{i+1}$ for all $i$.) $\tau(k)$ denotes the tasks assigned to
      processor $k$; initially, $\tau(k) \leftarrow \{\ \}$ for all $k$.
1   for $i \leftarrow 1$ to $n$
2           for $k \leftarrow 1$ to $m$
3                   if $\tau_i$ satisfies Conditions 11.3 and 11.4 on processor $\pi_k$ then
                            ▷ assign $\tau_i$ to proc. $k$; proceed to next task
4                           $\tau(k) \leftarrow \tau(k) \bigcup \{\tau_i\}$
5                           goto line 7
6           end (of inner for loop)
7           if $(k > m)$ return PARTITIONING FAILED
8   end (of outer for loop)
9   return PARTITIONING SUCCEEDED
```

Fig. 11.3 Pseudocode for partitioning algorithm

Run-time Complexity In attempting to map task τ_i, observe that Algorithm PARTITION essentially evaluates, in Eqs. 11.3 and 11.4, the workload generated by the previously-mapped $(i - 1)$ tasks on each of the m processors. Since $\text{DBF}^{(1)}(\tau_j, t)$ can be evaluated in constant time, a straightforward computation of this workload would require $O(i + m)$ time. Hence the runtime of the algorithm in mapping all n tasks is no more than $\sum_{i=1}^{n} O(i + m)$, which is $O(n^2)$ under the reasonable assumption that $m \le n$.

The following lemma asserts that this algorithm never assigns more tasks to a processor than can be EDF-scheduled upon it to meet all declines.

Lemma 11.1 *If the tasks assigned to each processor prior to considering task τ_i are* EDF*-schedulable on that processor and the partitioning algorithm assigns task τ_i to the k'th processor, then the tasks assigned to each processor (including the k'th processor) remain* EDF*-schedulable on that processor.*

Proof : First, note that the EDF-schedulability of the processors other than the k'th processor is not affected by the assignment of task τ_i to the k'th processor. It remains to demonstrate that, if the tasks previously assigned to the k processor were EDF-schedulable prior to the assignment of τ_i and Conditions 11.3 and 11.4 are satisfied, then the tasks on the k'th processor remain EDF-schedulable after adding τ_i.

The scheduling of the k'th processor after the assignment of task τ_i to it is a uniprocessor scheduling problem. As stated in Chap. 4, it is known (see e.g., [51]) that a uniprocessor system of preemptive sporadic tasks is EDF-schedulable if and only if all deadlines can be met for the synchronous arrival sequence (i.e., when each task has a job arrive at the same time-instant, and subsequent jobs arrive as rapidly as legal). Also, recall that EDF is an optimal preemptive uniprocessor scheduling algorithm. Hence to demonstrate that the k'th processor remains EDF-schedulable after adding task τ_i to it, it suffices to demonstrate that all deadlines can be met for the synchronous arrival sequence.

This is easily seen to be true informally[1], by using the intuitive interpretation of DBF$^{(1)}$ presented in Fig. 11.2. In the context of this interpretation, Condition 11.3 can be thought of as validating whether C_i units of execution can be guaranteed to τ_i on the k'th processor, and Condition 11.4 whether a fraction U_i of the processor can henceforth be reserved for τ_i. □

The correctness of Algorithm PARTITION follows, by repeated applications of Lemma 11.1:

Theorem 11.1 *If Algorithm* PARTITION *returns* PARTITIONING SUCCEEDED *on task system τ, then the resulting partitioning is* EDF*-schedulable.*

Proof : Observe that Algorithm PARTITION returns PARTITIONING SUCCEEDED if and only if it has successfully assigned each task in τ to some processor.

Prior to the assignment of task τ_1, each processor is trivially EDF-schedulable. It follows from Lemma 11.1 that all processors remain EDF-schedulable after each task assignment as well. Hence, all processors are EDF-schedulable once all tasks in τ have been assigned. □

Algorithm PARTITION can be simplified further [45] if we restrict our attention to *constrained-deadline* sporadic task systems:

Lemma 11.2 *For constrained-deadline sporadic task systems, any τ_i satisfying Condition 11.3 during the execution of Line 3 in Algorithm* PARTITION *of Fig. 11.3 satisfies Condition 11.4 as well.*

Proof : To see this, observe that it follows from the definition of DBF$^{(1)}$ that for all $t \geq D_k$ it is the case that

$$\text{DBF}^{(1)}(\tau_k, t) = C_k + u_k \times (t - D_k) = u_k(T_k + t - D_k).$$

Since $T_k \geq D_k$ for any constrained-deadline task τ_k, it must therefore hold that DBF$^{(1)}(\tau_k, t) \geq u_k \times t$ for any such τ_k, and any $t \geq D_k$. Hence,

$$\begin{aligned}
&\text{Condition 11.3} \\
&\Leftrightarrow \left(D_i - \sum_{\tau_j \in \tau(k)} \text{DBF}^{(1)}(\tau_j, D_i) \geq C_i \right) \\
&\Rightarrow D_i - \sum_{\tau_j \in \tau(k)} (u_j \times D_i) \geq C_i \\
&\Rightarrow 1 - \sum_{\tau_j \in \tau(k)} u_j \geq \frac{C_i}{D_i} \\
&\Rightarrow 1 - \sum_{\tau_j \in \tau(k)} u_j \geq u_i \\
&\Leftrightarrow \text{Condition 11.4}
\end{aligned}$$

[1] A formal proof is also straightforward, but omitted.

That is, Condition 11.4 is guaranteed to hold if Condition 11.3 holds. □

Hence if the task system τ being partitioned is known to be a constrained task system, we need to only check Condition 11.3 (rather than both Condition 11.3 and Condition 11.4) on line 3 in Fig. 11.3.

11.2.1 Evaluation

Algorithm PARTITION is not an exact partitioning algorithm: it is possible that there are systems that are partitioned EDF-schedulable that will be incorrectly flagged as "unschedulable" by Algorithm PARTITION. This is only to be expected since a simpler problem—partitioning collections of implicit-deadline sporadic tasks—is known to be NP-hard in the strong sense while Algorithm PARTITION runs in $O(n^2)$ time. In this section, we offer a quantitative evaluation of the efficacy of Algorithm PARTITION, by deriving some properties (Theorem 11.2 and Corollary 11.1) of Algorithm PARTITION, which characterize its performance.

A Note Theorem 11.2 and Corollary 11.1 are not intended to be used as schedulability tests to determine whether Algorithm PARTITION would successfully schedule a given sporadic task system or not; since Algorithm PARTITION itself runs efficiently in polynomial time, the "best" (i.e., most accurate) polynomial-time test for determining whether a particular system is successfully scheduled by Algorithm PARTITION is to actually run the algorithm and check whether it performs a successful partition or not. These properties are instead intended to provide a quantitative measure of how effective Algorithm PARTITION is vis a vis the performance of an optimal scheduler.

Let us start out considering constrained-deadline task systems. Suppose that Algorithm PARTITION fails to assign some task τ_i to any processor while partitioning constrained-deadline sporadic task system τ upon an m-processor platform. By Lemma 11.2 we conclude that Condition 11.3 is violated upon each processor, i.e.,

$$D_i - \sum_{\tau_j \in \tau(k)} \text{DBF}^{(1)}(\tau_j, D_i) < C_i,$$

for each processor k. Summing over all the m processors, we see that it must be the case that

$$mD_i - \sum_{\tau_j \in (\tau \setminus \{\tau_i\})} \text{DBF}^{(1)}(\tau_j, D_i) < mC_i$$

$$\Leftrightarrow \left(mD_i - \sum_{\tau_j \in \tau} \text{DBF}^{(1)}(\tau_j, D_i) + C_i < mC_i \right)$$

$$\Leftrightarrow \left(\sum_{\tau_j \in \tau} \text{DBF}^{(1)}(\tau_j, D_i) > mD_i - (m-1)C_i \right)$$

We have seen (Eq. 11.2) that $\text{DBF}^{(1)}(\tau_i, t) < 2\,\text{DBF}(\tau_i, t)$ for all τ_i and t. Hence, we have

$$\sum_{\tau_j \in \tau} \text{DBF}^{(1)}(\tau_j, D_i) > mD_i - (m-1)C_i$$

$$\Rightarrow \left(\sum_{\tau_j \in \tau} 2\text{DBF}(\tau_j, D_i) > mD_i - (m-1)C_i \right)$$

$$\Leftrightarrow \left(2\frac{\sum_{\tau_j \in \tau} \text{DBF}(\tau_j, D_i)}{D_i} > m - (m-1)\frac{C_i}{D_i} \right)$$

$$\Rightarrow \left(2\text{LOAD}(\tau) > m - (m-1)\max_{\tau_i \in \tau}\left(\frac{C_i}{D_i}\right) \right)$$

thereby establishing the following sufficient schedulability condition for Algorithm PARTITION[2]:

Theorem 11.2 *Any constrained-deadline sporadic task system τ satisfying*

$$\text{LOAD}(\tau) \le \frac{1}{2} \times \left(m - (m-1)\mathit{dens}_{\max}(\tau) \right) \tag{11.5}$$

is guaranteed to be successfully scheduled by Algorithm PARTITION. □

Theorem 11.2 above may be used to establish the following speedup bound for Algorithm PARTITION:

Theorem 11.3 *Any constrained-deadline sporadic task system that can be scheduled on an m-processor platform by an optimal clairvoyant scheduling algorithm is successfully scheduled by Algorithm* PARTITION *on m processors that are each $(3 - \frac{1}{m})$ times as fast.*

Proof : Any sporadic task system τ that is feasible upon a platform comprised of m speed-$(m/(3m-1))$ processors necessarily has

$$\text{LOAD}(\tau) \le \frac{m^2}{3m-1} \text{ and } \text{dens}_{\max}(\tau) \le \frac{m}{3m-1}$$

Such systems therefore satisfy Inequality 11.5, since

$$\text{LOAD}(\tau) \le \frac{1}{2} \times \left(m - (m-1)\text{dens}_{\max}(\tau) \right)$$

$$\Leftarrow \frac{m^2}{3m-1} \le \frac{1}{2} \times \left(m - \frac{(m-1)m}{3m-1} \right)$$

$$\Leftrightarrow \frac{2m^2}{3m-1} \le \frac{3m^2 - m - m^2 + m}{3m-1}$$

[2] See Chap. 10 to review the definitions of $\text{LOAD}(\tau)$ and $\text{dens}_{\max}(\tau)$

$$\Leftrightarrow \frac{2m^2}{3m-1} \leq \frac{2m^2}{3m-1}$$

which is obviously true. This establishes the correctness of this theorem, since we have just shown that any constrained-deadline sporadic task system that is feasible upon m speed-$\frac{m}{3m-1}$ processors is successfully scheduled by Algorithm PARTITION on m speed-1 processors. □

The following result concerning arbitrary-deadline sporadic task systems was also proved in [44]; we omit the proof.

Corollary 11.1 *Algorithm* PARTITION *makes the following performance guarantee: if a sporadic task system is feasible on m identical processors each of a particular computing capacity, then Algorithm* PARTITION *will successfully partition this system upon a platform comprised of m processors that are each* $(4 - \frac{2}{m})$ *times as fast as the original.*

11.3 Partitioned FTP Scheduling

In this section, we study approximation algorithms for partitioning a sporadic task system upon an identical multiprocessor platform, under the restriction that only FTP (Fixed-Task-Priority—see Sect. 3.2) scheduling algorithms may be deployed on the individual processors during run-time. As in Sects. 11.1 and 11.2, the overall approach is to consider the tasks in some particular order; in considering a task, we seek to find a processor upon which it fits. The idea of a task "fitting" on a processor requires some elaboration: recall that a task τ_i fits on a processor if it can be assigned to the processor, and scheduled according to a given uniprocessor scheduling algorithm such that τ_i and all previously-assigned tasks will always meet all deadlines. The criteria for "fitting" on a processor are thus dependent on the choice of scheduling algorithm at the uniprocessor level. In this section, we are restricted to only consider the static-priority scheduling algorithms. Recall, from Chap. 4, that the Deadline Monotonic (DM) scheduling algorithm , which assigns priorities according to the relative deadline parameters—tasks with smaller relative deadlines are assigned greater priority—is known to be optimal for the static-priority scheduling of constrained-deadline sporadic task systems on uniprocessors [133]. We restrict our attention in this section to constrained-deadline sporadic task systems, and will therefore assume that each processor is scheduled using DM. Such a partitioning algorithm, called Algorithm FBB-FFD, was defined in [88]; we describe this algorithm below, and prove that it is correct (and has polynomial run-time)

Algorithm FBB-FFD is very similar to the EDF-partitioning algorithm of Sect. 11.2: tasks are again considered in nondecreasing order of their relative deadline parameters. The only difference is that the conditions for determining whether a task "fits" upon a processor is changed: the Conditions 11.3–11.4 of Sect. 11.2 are replaced by the single Condition 11.6 below. A detailed description follows.

Recall, from Sect. 10.5, the definition of the *request bound function* (RBF) of a sporadic task, and of the approximation RBF* to the RBF that was defined in Eq. 10.5 as follows:

$$\text{RBF}^*(\tau_i, t) = C_i + t_i \times t.$$

We use this approximation to perform partitioned DM scheduling below, much as we used the $\text{DBF}^{(1)}$ approximation to the demand bound function (DBF) to perform partitioned EDF scheduling in Sect. 11.2.

Given a sporadic task system τ comprised of n sporadic tasks $\tau_1, \tau_2, \ldots, \tau_n$, and a processing platform comprised of m unit-capacity processors, FBB-FFD will attempt to partition τ among the processors. Assume the tasks of τ_i are indexed in nondecreasing order of their relative deadline (i.e., $D_i \leq D_{i+1}$, for $1 \leq i < n$). The FBB-FFD algorithm considers the tasks in increasing index order. We now describe how to assign task τ_i assuming that tasks $\tau_1, \tau_2, \ldots, \tau_{i-1}$ have already successfully been allocated among the m processors. Let $\tau(\ell)$ denote the set of tasks already assigned to the ℓ'th processor, $1 \leq \ell \leq m$. We assign task τ_i to any processor k that satisfies the following condition

$$\left(D_i - \sum_{\tau_j \in \tau(k)} \text{RBF}^*(\tau_j, D_i) \right) \geq C_i \tag{11.6}$$

If no such π_k exists, then Algorithm FBB-FFD returns PARTITIONING FAILED: it is unable to conclude that sporadic task system τ is feasible upon the m-processor platform. Otherwise, FBB-FFD returns PARTITIONING SUCCEEDED.

Computational Complexity As with Algorithm PARTITION in Sect. 11.2, Algortihm FBB-FFD is computationally very efficient. Sorting the tasks in (nondecreasing) relative deadline order requires $\Theta(n \lg n)$ time. In attempting to map task τ_i, observe that Algorithm FBB-FFD essentially evaluates, in Eq. 11.6, the workload generated by the previously-mapped $(i-1)$ tasks on each of the m processors. Since $\text{RBF}^*(\tau_j, t)$ can be evaluated in constant time (see Eq. 10.6), a straightforward computation of this workload would require $O(i+m)$ time. Hence the runtime of the algorithm in mapping all n tasks is no more than $\sum_{i=1}^{n} O(i+m)$, which is $O(n^2)$ under the reasonable assumption that $m \leq n$.

The following lemma asserts that Algorithm FBB-FFD never overloads a processor.

Lemma 11.3 *If the tasks previously assigned to each processor were DM-schedulable on that processor and Algorithm* FBB-FFD *assigns task τ_i to the k'th processor, then the tasks assigned to each processor (including the k'th processor) remain DM-schedulable on that processor.*

Proof : Observe that the DM-schedulability of the processors other than the k'th processor is not affected by the assignment of task τ_i to the k'th processor. It remains to demonstrate that, if the tasks assigned to k were DM-schedulable on the k'th processor prior to the assignment of τ_i and Condition 11.6 is satisfied, then the tasks on the k'th processor remain DM-schedulable after adding τ_i.

The scheduling of the k'th processor after the assignment of task τ_i to it is a uniprocessor scheduling problem. First, observe that since tasks are assigned in order of nondecreasing deadlines the tasks previously assigned to the processor have greater priority that τ_i; hence, their schedulability is not compromised by assigning τ_i to the processor. It merely remains to show that τ_i will also meet all its deadlines on the k'th processor.

It is known (see e.g., [13]) that constrained-deadline sporadic task $\tau_i = (C_i, D_i, T_i)$ will always meet all its deadlines under preemptive uniprocessor DM scheduling if there is some t, $0 \leq t \leq T_i$, for which

$$C_i + \sum_{\tau_j \in \mathrm{hp}(\tau_i)} \mathrm{RBF}(\tau_j, t) \leq t, \tag{11.7}$$

where hp(τ_i) denotes the set of tasks sharing the processor with τ_i, that have greater DM priority than τ_i. But this is easily seen to be true for task τ_i on the k'th processor: since it is assigned to the k'th processor by Algorithm FBB-FFD, it must be the case that Condition 11.6 is satisfied:

$$\begin{aligned} & D_i - \sum_{\tau_j \in \tau(k)} \mathrm{RBF}^*(\tau_j, D_i) \geq C_i \\ \Leftrightarrow & C_i + \sum_{\tau_j \in \tau(k)} \mathrm{RBF}^*(\tau_j, D_i) \leq D_i \\ \Leftrightarrow & C_i + \sum_{\tau_j \in \mathrm{hp}(\tau_i)} \mathrm{RBF}^*(\tau_j, D_i) \leq D_i \\ \Rightarrow & C_i + \sum_{\tau_j \in \mathrm{hp}(\tau_i)} \mathrm{RBF}(\tau_j, D_i) \leq D_i \end{aligned}$$

and D_i therefore bears witness to the fact that Condition 11.7 is satisfied. □

The correctness of Algorithm FBB-FFD follows, by repeated applications of Lemma 11.3:

Theorem 11.4 *If the Algorithm* FBB-FFD *returns* PARTITIONING SUCCEEDED *when scheduling some constrained-deadline sporadic task system* τ*, then the resulting partitioning is DM-schedulable.*

Proof: Observe that Algorithm FBB-FFD returns PARTITIONING SUCCEEDED if and only if it has successfully assigned each task in τ to some processor.

Prior to the assignment of task τ_1, each processor is trivially DM-schedulable. It follows from Lemma 11.3 that all processors remain DM-schedulable after each task DM-schedulable once all tasks in τ have been assigned. □

Suppose that Algorithm FBB-FFD fails to assign some task τ_i to any processor while partitioning constrained-deadline sporadic task system τ upon an m-processor platform. Condition 11.6 is therefore violated upon each processor, i.e.,

$$D_i - \sum_{\tau_j \in \tau(k)} \mathrm{RBF}^*(\tau_j, D_i) < C_i$$

for each processor k. Summing over all the m processors, we see that it must be the case that

$$
\begin{aligned}
& mD_i - \sum_{\tau_j \in (\tau \setminus \{\tau_i\})} \text{RBF}^*(\tau_j, D_i) < mC_i \\
\Leftrightarrow & \left(mD_i - \sum_{\tau_j \in \tau} \text{RBF}^*(\tau_j, D_i) + C_i < mC_i \right) \\
\Leftrightarrow & \left(\sum_{\tau_j \in \tau} \text{RBF}^*(\tau_j, D_i) > mD_i - (m-1)C_i \right)
\end{aligned}
$$

By Lemma 10.3, $\text{RBF}^*(\tau_j, t) < 3\text{DBF}(\tau_j, t)$ for all τ_j and all $t \geq D_j$. Since tasks are considered in order of nondecreasing deadlines, this $D_i \geq D_j$ for all tasks D_j within the summation above, and hence we have

$$
\begin{aligned}
& \sum_{\tau_j \in \tau} \text{RBF}^*(\tau_j, D_i) > mD_i - (m-1)C_i \\
\Rightarrow & \left(\sum_{\tau_j \in \tau} 3\text{DBF}(\tau_j, D_i) > mD_i - (m-1)C_i \right) \\
\Leftrightarrow & \left(3\frac{\sum_{\tau_j \in \tau} \text{DBF}(\tau_j, D_i)}{D_i} > m - (m-1)\frac{C_i}{D_i} \right) \\
\Rightarrow & \left(3\text{LOAD}(\tau) > m - (m-1)\max_{\tau_i \in \tau}\left(\frac{C_i}{D_i}\right) \right)
\end{aligned}
$$

thereby establishing the following sufficient schedulability condition for Algorithm FBB-FFD:

Theorem 11.5 *Any constrained-deadline sporadic task system τ satisfying*

$$\text{LOAD}(\tau) \leq \frac{1}{3} \times \Big(m - (m-1)\mathit{dens}_{\max}(\tau) \Big) \tag{11.8}$$

is guaranteed to be successfully scheduled by Algorithm FBB-FFD. □

Theorem 11.5 above may be used to establish the following speedup bound for Algorithm FBB-FFD:

Theorem 11.6 *Any constrained-deadline sporadic task system that is feasible on an m-processor platform is successfully scheduled by Algorithm* FBB-FFD *on m processors that are each $(4 - \frac{1}{m})$ times as fast.*

Proof : Any sporadic task system τ that is feasible upon a platform comprised of m speed-$(m/(4m-1)$ processors must have

$$\text{LOAD}(\tau) \leq \frac{m^2}{4m-1} \text{ and } \text{dens}_{\max}(\tau) \leq \frac{m}{4m-1}$$

Such systems therefore necessarily satisfy Inequality 11.8, since

$$\begin{aligned}
&\textsc{load}(\tau) \le \frac{1}{3} \times \Big(m - (m-1)\text{dens}_{\max}(\tau)\Big) \\
\Leftarrow\ & \frac{m^2}{4m-1} \le \frac{1}{3} \times \left(m - \frac{(m-1)m}{4m-1}\right) \\
\Leftrightarrow\ & \frac{3m^2}{4m-1} \le \frac{4m^2 - m - m^2 + m}{4m-1} \\
\Leftrightarrow\ & \frac{3m^2}{4m-1} \le \frac{3m^2}{4m-1}
\end{aligned}$$

which is obviously true. This establishes the correctness of this theorem. □

Sources

EDF-based partitioned scheduling was discussed in [44–46]; DM-based partitioning was studied in [87]. A general task model was considered in [47]. Also see Fisher's dissertation [83].

Chapter 12
Global Scheduling: General Comments

The next several chapters delve into the global scheduling of three-parameter sporadic task systems. There is a lot of material to cover here: the real-time scheduling research community has devoted considerable effort to devise schedulability analysis tests for such systems. Since many proposed tests are *incomparable* to each other in the sense that there are task systems deemed schedulable by one that the others fail to identify as being schedulable, it is difficult to choose just one or a few representative "best" tests to include in this book.

Given the sheer volume of material here, we propose a few possibilities for pursuing parts of this material.

- To obtain a mainly *conceptual* understanding of the foundations of global schedulability analysis, we suggest that one read Chaps. 14,16, and 19 in-depth, skimming the others at a first reading. Chapter 19 in particular describes the best quantitative bounds known for global schedulability analysis. A high-level comparison of the different tests is provided in Sect. 14.5.
- Schedulability experiments on randomly generated systems of tasks have offered evidence that the tests in Chaps. 13,15, and 17 appear to be particularly effective for the schedulability analysis of earliest-deadline-first (EDF)-scheduled task systems. From a *pragmatic* perspective, therefore, these chapters are worth reading in-depth if one seeks to choose an EDF schedulability test to be implemented. Chapter 18 describes tests analogous to the ones described in Chaps. 13,15, and 17, but for the analysis of fixed task priority(FTP)-scheduled task systems; it should be read if one is seeking to implement an FTP-schedulability analysis algorithm.

From a scheduling–theoretic perspective, the global multiprocessor scheduling of systems of 3-parameter sporadic tasks is considerably more challenging than either the (global and partitioned) scheduling of implicit-deadline sporadic task systems, or the partitioned scheduling of 3-parameter sporadic task systems. We highlight a couple of novel challenges that arise in the remainder of this chapter.

- Since any sporadic task system may generate infinitely many different collections of jobs, each containing an unbounded number of jobs, it is not possible to validate schedulability by exhaustive enumeration of all these collections of jobs. Instead, the typical approach has been to identify one or a few *worst-case job* *arrival sequences*, each consisting of a bounded number of jobs, for a task system for which it can be proved that if these job collections are scheduled, then all legal job collections are schedulable. However, no small set of such worst-case collections of jobs are known for global scheduling, with respect to most scheduling algorithms—this point is discussed in Sect. 12.1 below.

- We saw in Chap. 7 that there are optimal algorithms for the global scheduling of implicit-deadline sporadic task systems—the pfair scheduling algorithm is one such example of optimal algorithm. In contrast, it is known that there can be no optimal algorithm for the global scheduling of 3-parameter sporadic task systems; this is discussed further in Sect. 12.2 below.

12.1 Global Scheduling and Worst-Case Behavior

Recall that in order for a sporadic task system to be considered A-schedulable for a given scheduling algorithm A, A must generate schedules meeting all deadlines for all legal job-arrival sequences of the task system. As discussed above, any sporadic task system has infinitely many different legal job arrival sequences; therefore, an approach of exhaustive enumeration of such sequences, followed by simulating algorithm A on each of these sequences, will not yield an effective schedulability test. Instead, the typical approach has been to identify one or a few worst-case job arrival sequences for a task system for which it can be proved that if algorithm A successfully schedules all these worst-case job arrival sequences, then A is guaranteed to successfully schedule all legal job arrival sequences of the task system. For example, with respect to EDF or deadline-monotonic (DM) scheduling upon preemptive uniprocessors it is known [51, 132, 137] that there is a unique worst-case job arrival sequence: Every task has one job arrive at the same instant in time, and each task has subsequent jobs arrive as soon as legally permitted to do so. (Such a job arrival sequence is often called the *synchronous arrival sequence* for the task system—see Sect. 4.3.) Since run-time scheduling is performed on a per-processor basis (i.e., as a collection of uniprocessor systems, one to each processor) under the partitioned paradigm, it follows that the synchronous arrival sequence is also the worst-case job arrival sequence for partitioned multiprocessor EDF and DM scheduling.

For global scheduling, however, the synchronous arrival sequence is not necessarily the worst-case arrival sequence, as the following example illustrates with respect to EDF scheduling.

Example 12.1 Consider the task system comprised of the three tasks $\tau_1 = (1, 1, 2)$, $\tau_2 = (1, 1, 3)$, and $\tau_3 = (5, 6, 6)$, executing upon two unit-capacity processors. It may be seen (Fig. 12.1a) that the synchronous arrival sequence is successfully scheduled by EDF to meet all deadlines. However, if task τ_1's second job were to arrive three, rather than two, time units after the first (Fig. 12.1b), then EDF would miss some deadline over the interval $[0, 6]$ (and indeed, no scheduling algorithm can possibly guarantee to meet all deadlines for this job arrival sequence—i.e., this task system is in fact infeasible).

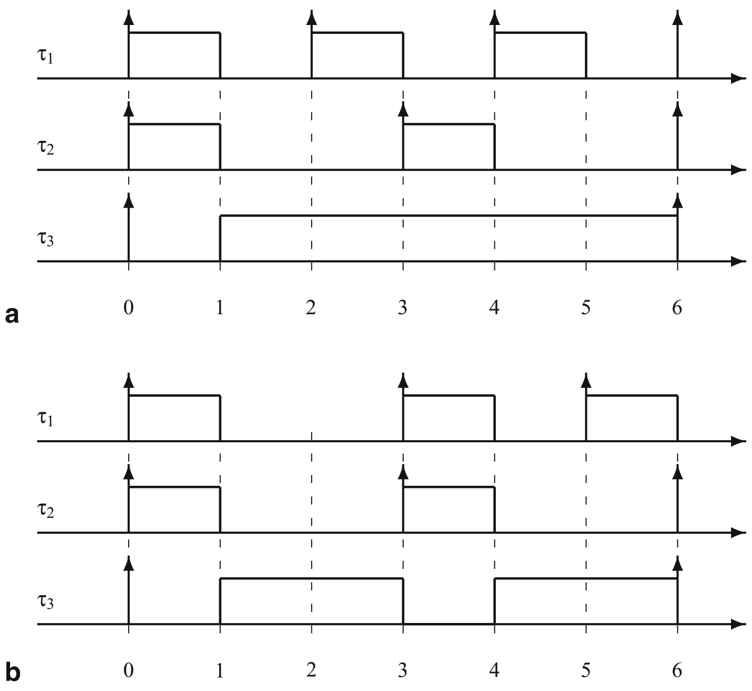


Fig. 12.1 Figure for Example 12.1, illustrating that the synchronous arrival sequence does not represent the worst case

Without knowing what the worst-case behavior of a sporadic task system may be, it is not possible to design simulation-based exact (necessary as well as sufficient) schedulability tests. To our knowledge, no finite collection of worst-case job arrival sequences has been identified for global scheduling of sporadic task systems (although a hopelessly intractable—doubly exponential in the size of the task system—exhaustive-search procedure is described in [61]). This, at a basic level, is the fundamental difference between our understanding of partitioned and global scheduling of sporadic task systems. As stated above, it is known that the synchronous arrival sequence represents the worst-case behavior of a task system under partitioned scheduling. Hence, partitioned schedulability testing can be solved in principle by determining whether it is possible to partition the tasks among the processors such that no deadlines are missed in the synchronous arrival sequence (although an algorithm for solving this problem exactly is provably highly intractable, since the partitioning step is itself intractable).

In the global case, however, we do not know how to determine schedulability even if computational tractability were not an issue. That is, we do not yet have an adequate understanding of what precisely the characteristics of a globally schedulable system are. (In addition, we point out that Example 12.1 above illustrates that the LOAD parameter of a sporadic task system is not an exact indicator of feasibility: While

LOAD(τ) $\leq m$ is clearly necessary for τ to be feasible on an m-processor platform, the task system in Example 12.1 illustrates that this is not sufficient.)

12.2 The Impossibility of Optimal Online Scheduling

It is known that EDF is an optimal algorithm for scheduling collections of independent jobs upon a preemptive uniprocessor (an informal proof of this fact is sketched in Sect. 7.4), and hence for scheduling systems of sporadic tasks as well. It has been shown [76] that in the multiprocessor case, no algorithm can be optimal for scheduling collections of independent jobs:

Theorem 12.1 *(Dertouzos and Mok 1989 [76]) For two or more processors, no online scheduling algorithm can be optimal for arbitrary collections of real-time jobs without a complete a priori knowledge of the deadline, execution time, and arrival time of each job.*

Of course, this negative result does not rule out the possibility of an optimal multiprocessor scheduling algorithm for sporadic task systems, since an optimal scheduling algorithm is only required to schedule all collections of jobs generated by *feasible* sporadic task systems. (Thus for example, the pfair algorithms described in Chap. 7 are optimal online algorithms for implicit-deadline sporadic task systems: They can schedule all collections of jobs generated by all feasible implicit-deadline sporadic task systems.) With regards to constrained-deadline (and therefore, arbitrary-deadline) 3-parameter sporadic task systems, it was shown [89] that the task system consisting of the following six tasks is (i) feasible under global

	C_i	D_i	T_i
τ_1	2	2	5
τ_2	1	1	5
τ_3	1	2	6
τ_4	2	4	100
τ_5	2	6	100
τ_6	4	8	100

scheduling upon a 2-processor platform, and (ii) cannot be scheduled to always meet all deadlines upon a 2-processor platform by any on-line scheduling algorithm, thereby establishing Theorem 12.2.

Theorem 12.2 *(Theorem 3 from [89]) No optimal online algorithm exists for the multiprocessor scheduling of real-time, constrained-deadline sporadic task systems on two or more processors.*

As stated above, this statement does *not* hold for implicit-deadline sporadic task systems—the pfair scheduling algorithms discussed in Chap. 7 are optimal online scheduling algorithms for such task systems.

Sources

The example illustrating that the synchronous arrival sequence does not correspond to worst-case behavior for global multiprocessor scheduling is from [54]. The example establishing the impossibility of optimal on-line scheduling is from [89].

Chapter 13
Density-Based Global Schedulability Tests

Most of the utilization-based schedulability tests presented in Chaps. 7–9 concerning the global scheduling of systems of implicit-deadline tasks can be generalized to the constrained- and arbitrary-deadline systems represented by the three-parameters task model, by replacing the utilizations with densities (recall that the density dens_i of the task $\tau_i = (C_i, D_i, T_i)$ is defined to be $(C_i/\min(D_i, T_i))$). Thus for example, a direct generalization of Theorem 7.1 yields

Theorem 13.1 *Any three-parameter sporadic task system τ satisfying*

$$dens_{sum}(\tau) \leq m \text{ and } dens_{max}(\tau) \leq 1$$

is feasible upon a platform comprised of m unit-capacity processors.

(Unlike Theorem 7.1 for implicit-deadline sporadic task systems, however, this is merely a sufficient, rather than an exact—necessary and sufficient—feasibility test for three-parameter sporadic task systems.)

Although this idea is very simple and not particularly deep, schedulability experiments on randomly generated task systems [20, 55, 56] provide evidence that the resulting EDF schedulability tests are remarkably effective.

13.1 Density as a Performance Metric

It is worth pointing out here some shortcomings of the density bound vis-à-vis the utilization bound (Definition 5.1) as a metric for evaluating the effectiveness of multiprocessor scheduling algorithms for sporadic task systems. Since the utilization of a task represents a tight upper bound on the fraction of the computing capacity of a processor that may be needed for executing jobs generated by the task, no scheduling algorithm, not even a clairvoyant optimal one, may have a utilization bound greater than the number of processors m. In other words, since no optimal algorithm has a utilization bound greater than m, the utilization bound of an algorithm provides an upper bound on its effectiveness compared to an optimal algorithm.

In contrast, there are three-parameter sporadic task systems with density arbitrarily large that are feasible (and hence schedulable by a clairvoyant optimal algorithm) upon an m-processor platform for any $m \geq 1$; hence, stating that a particular scheduling algorithm A is able to schedule all task systems with density $\leq c$ for some value of c does not tell us much about how A compares to an optimal algorithm. The example below illustrates how a task system with any desired density may be constructed, that is feasible, EDF schedulable, and DM schedulable on a unit-speed preemptive uniprocessor platform.

Example 13.1 Consider a sporadic task system τ consisting of the n tasks $\{\tau_i = (1, i, n)\}_{i=1}^{n}$. It is easily validated, using any of the well-known uniprocessor EDF schedulability tests, that this task system is EDF schedulable upon a single preemptive processor. This task system has a density $\text{dens}_{\text{sum}}(\tau)$ equal to

$$\sum_{i=1}^{n} \frac{1}{i},$$

which is the nth harmonic number. It is well known that the harmonic numbers increase without bound with increasing n; hence $\text{dens}_{\text{sum}}(\tau)$ can be made arbitrarily large by increasing the number of tasks n in τ.

Despite this fact, schedulability experiments on synthetically generated task sets have tended to indicate [20, 21, 55] that density-based tests for global schedulability perform quite well in practice for certain classes of task systems. We will therefore show below how the main results proved for global fixed job priority (FJP) and fixed task priority (FTP) scheduling of implicit-deadline sporadic task systems, described in Chaps. 8 and 9, can be generalized to yield density-based tests for arbitrary deadline systems.

13.2 Density-Based Tests for FJP Scheduling

The utilization-based test of Theorem 8.5 can be generalized to systems having deadlines different from periods as follows.

Theorem 13.2 *A task set τ with arbitrary deadlines is EDF schedulable upon a platform composed of m processors with unit capacity, if*

$$dens_{sum}(\tau) \leq m(1 - dens_{max}(\tau)) + dens_{max}(\tau). \tag{13.1}$$

Proof The proof essentially mimics the derivations in Chap. 8 that lead to the proof of Theorem 8.5. Analogously to Theorem 8.3, it is straightforward to conclude that the three-parameter sporadic task system τ is feasible upon some uniform multiprocessor platform π satisfying $S_{\text{sum}}(\pi) = \text{dens}_{\text{sum}}(\tau)$ and $S_{\text{max}}(\pi) = \text{dens}_{\text{max}}(\tau)$. Analogs of Theorems 8.4 and 8.5 immediately follow, with $U_{\text{sum}}(\tau)$ replaced by $\text{dens}_{\text{sum}}(\tau)$ and $U_{\text{max}}(\tau)$ replaced by $\text{dens}_{\text{max}}(\tau)$. □

Derivations analogous to those in Sect. 8.3 are also straightforward. Similarly to Theorem 8.8, we can show that any task system τ with $\text{dens}_{\text{sum}}(\tau) \leq (M+1)/2$

```
Algorithm fpEDF-density
Arbitrary-deadline sporadic task system $\tau = \{\tau_1, \tau_2, \ldots \tau_n\}$ to be scheduled on $m$ processors
(It is assumed that $\text{dens}_i \geq \text{dens}_{i+1}$ for all $i, 1 \leq i < n$)

  for $i = 1$ to $(m - 1)$ do
    if $(\text{dens}_i > \frac{1}{2})$
        then $\tau_i$'s jobs are assigned highest priority (ties broken arbitrarily)
        else break
  the remaining tasks' jobs are assigned priorities according to EDF
```

Fig. 13.1 The FTP scheduling algorithm fpEDF-density

and $\text{dens}_{\max}(\tau) \leq 1/2$ is successfully scheduled using EDF, allowing us to design Algorithm fpEDF-density (Fig. 13.1) analogously of Algorithm fpEDF, and prove that

Theorem 13.3 *Algorithm fpEDF-density successfully schedules any task system τ satisfying $dens_{sum}(\tau) \leq (m+1)/2$ upon m unit-speed processors.*

13.3 Density-Based Tests for FTP Scheduling

Utilization-based results for the FTP scheduling of implicit deadline sporadic task systems can similarly be generalized to arbitrary-deadline systems by using densities instead of utilizations.

The following analog of Theorem 9.3 was established in [60]:

Theorem 13.4 *(from [60]) Any periodic or sporadic task system τ satisfying*

$$dens_{sum}(\tau) \leq \frac{m}{2}(1 - dens_{max}(\tau)) + dens_{max}(\tau) \tag{13.2}$$

is successfully scheduled by DM on m unit-speed processors.

Analogously to Theorem 9.1, it can be shown that any constrained-deadline sporadic task system τ satisfying

$$\text{dens}_{\text{sum}}(\tau) \leq m^2/(3m-2) \text{ and } \text{dens}_{\max}(\tau) \leq m/(3m-2)$$

is successfully scheduled using DM. Similarly, Algorithm DM-DS(ξ) (Fig. 13.2) can be defined analogously to Algorithm RM-US(ξ), and Theorem 9.5 generalizes to the following:

Theorem 13.5 *Algorithm DM-DS$(1/3)$ schedules any constrained-deadline sporadic task system τ with $dens_{sum}(\tau) \leq (m+1)/3$ upon m unit-speed processors.*

```
Algorithm DM-DS($\xi$)
Constrained-deadline sporadic task system $\tau = \{\tau_1, \tau_2, \ldots \tau_n\}$ to be scheduled on $m$ processors
(It is assumed that $\text{dens}_i \geq \text{dens}_{i+1}$ for all $i$, $1 \leq i < n$)

    for $i = 1$ to $(m - 1)$ do
        if ($\text{dens}_i > \xi$)
                then $\tau_i$ is assigned highest priority
                else break
    the remaining tasks are assigned priorities according to DM
```

Fig. 13.2 The FTP scheduling algorithm DM-DS(ξ)

Sources

The observation that schedulability tests for implicit-deadline systems may be extended to three-parameter systems by substituting the density parameter in place of the utilization parameter was explicitly made in [59, 60].

Chapter 14
A Strategy for Global Schedulability Analysis

In this chapter, we describe, at a high level, a strategy that has proven effective in deriving global schedulability tests. Various specific instantiations of this strategy, yielding different sufficient schedulability tests, are described in the following chapters. (Although we do not cover nonpreemptive schedulability analysis in the book, we point out that this strategy has also been successfully applied to nonpreemptive schedulability analysis [103].)

This chapter is organized as follows. We describe the general strategy in Sect. 14.1, and illustrate its use in Sect. 14.2 by stating a well-known earliest-deadline-first-(EDF)uniprocessor schedulability test in the terms of this strategy. Two concepts—workload bounds and interference—that are central to the application of this strategy are explored in Sects. 14.3 and 14.4 respectively. In Sect. 14.5 we provide a high-level comparison of the features of the different tests that we will be studying in detail in the later chapters.

14.1 Description of the Strategy

In broad terms, this strategy starts out assuming that a given sporadic task system τ is not schedulable upon an m-processor platform by the scheduling algorithm under consideration, and derives conditions that must necessarily be satisfied in order for this to occur. A sufficient schedulability test is then obtained by negating these conditions.

This strategy may be considered as being comprised of the following steps.

1. ***An unschedulable collection of jobs:*** If τ is unschedulable, there are collections of jobs generated by τ upon which the scheduling algorithm will miss deadlines. We reason about a particular collection of such jobs. This collection of jobs is typically defined as possessing some additional properties—for instance, it may be defined to be a *minimal* collection of jobs upon which a deadline is missed; it may be a collection of jobs for which the deadline miss occurs at the earliest time-instant, etc.

2. ***An interval of interest:*** Next, we identify some time-interval of interest I such that the computational *demand* W of this collection of jobs—the amount of computation needed to meet all deadlines—over this interval must exceed some lower bound in order to cause this deadline-miss. We determine a value W_L for this lower bound.
3. ***An upper bound on demand:*** Next, we compute an upper bound W_U on W, denoting an upper bound on the maximum amount of work that could possibly be generated by the task system τ over the interval I.
4. ***A sufficient schedulability condition:*** Hence,

   $$\exists\, I \ :: W_U > W_L$$

   denotes a necessary condition for a deadline miss; consequently, the negation of this condition

   $$\forall\, I \ :: W_U \leq W_L$$

   denotes a sufficient schedulability condition.

Different schedulability tests differ from one another in the manner in which they identify the collection of unschedulable jobs and the interval of interest I, and in the manner they compute values for W_U's and W_L's.

14.2 Illustrating the Approach of Sect. 14.1: Uniprocessor EDF

Let us illustrate the strategy described in Sect. 14.1, by deriving the well-known uniprocessor EDF schedulability test[1].

Suppose that uniprocessor EDF misses a deadline at time-instant t_d while scheduling a collection of jobs generated by sporadic task system τ.

1. Remove all jobs with deadline $> t_d$ from the collection of jobs; since preemptive EDF schedules the earliest-deadline job at each instant, it follows that EDF will continue to miss a deadline at time-instant t_d, when scheduling this sequence of jobs (with larger-deadline jobs removed).
2. Let $t_o < t_d$ denote the latest instant at which the processor was idled by EDF while scheduling this (reduced) collection of jobs—*the interval* $[t_o, t_d)$ *is the interval of interest.*

   An obvious lower bound W_L on the demand W of τ over $[t_o, t_d)$ is given by the amount of actual execution received by this workload.

   Since EDF is a work-conserving algorithm, this is given by the interval-length $(t_d - t_o)$.

[1] Although the strategy of Sect. 14.1 was specified for global multiprocessor scheduling, it can be applied, as is being done here, to uniprocessor scheduling as well.

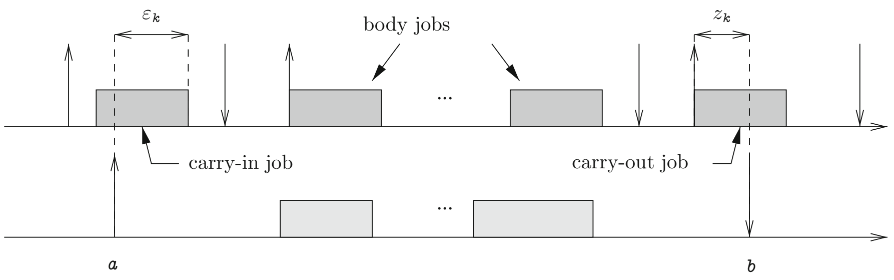


Fig. 14.1 Body, carry-in and carry-out jobs of task τ_k.

3. By definition of the demand bound function DBF, (Sect. 10.3) $\sum_{\tau_i \in \tau} \text{DBF}(\tau_i, t_d - t_o)$ is an upper bound W_U on W.
4. Letting $t \overset{\text{def}}{=} t_d - t_o$, we have

$$\exists\, t : t > 0 \ : \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, t) > t$$

as a necessary condition for uniprocessor EDF to miss a deadline. By negating this, we obtain the following sufficient condition for uniprocessor EDF schedulability:

$$\forall\, t : t > 0 \ : \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, t) \leq t.$$

Now, the "goodness" of such a test depends upon how tight the bounds W_U and W_L are. (In analyzing uniprocessor EDF scheduling above, e.g., it turns out that both the bounds are exact, and the test is therefore an exact schedulability test.)

14.3 Determining Workload Bounds

The various global schedulability tests that we will study in upcoming chapters all seem to more-or-less fit into the general strategy described in Sect. 14.1—they differ from one another primarily in the manner in which they determine the upper and lower bounds W_U and W_L on the workload within the interval of interest. When computing the upper bound W_U on the workload over some identified interval of interest $[a, b]$, the contributions of jobs to this workload are typically considered in three separate categories (see Fig. 14.1):

- *Body*: jobs with both release time and deadline in the considered interval; each such job contributes its entire execution requirement (its worst-case execution time (WCET)) C_k to the workload in that interval.
- *Carry-in*: jobs with release time before a and deadline within the interval $[a, b]$; only a part $\varepsilon_k \leq C_k$ of the execution requirements of such jobs contribute to the workload in the interval.

The challenges associated with accurately bounding this carry-in workload are discussed further in Sect. 16.1.1.

- *Carry-out*: jobs with release time before b and deadline after b; once again, only a part $z_k \leq C_k$ of the execution requirements of such jobs contribute to the workload in the interval.

According to these definitions, a job with release time before a and deadline after b is considered a carry-out job, while a job with release time before a and deadline coinciding with b is considered a carry-in job.

Note that with constrained-deadline sporadic task systems, there can be at most one carry-in job and one carry-out job for each task.

14.4 Interference

The workload contributing to the workload bounds discussed above can also be considered from the perspective of interference: If a job of task τ_k is active but not executing during some time-interval while a job of some τ_i, $(i \neq k)$ is executing during that interval, we say that τ_i is *interfering* with the execution of τ_k's job during that interval.

Definition 14.1 (interference (I_k, $I_{i,k}$)) The *maximum interference* I_k on a task τ_k is the cumulative length of all intervals in which the job of τ_k with the largest response time is ready to execute but it cannot execute due to higher priority jobs.

We also define the *interference* $I_{i,k}$ of task τ_i on task τ_k as the cumulative length of all intervals in which the job of τ_k with the largest response time is ready to execute but not executing, while τ_i is executing.

Notice that by definition:

$$I_{i,k} \leq I_k, \forall i, k. \tag{14.1}$$

The following theorem is valid for any kind of work-conserving global scheduler.

Lemma 14.1 *The interference that a constrained deadline task τ_k can suffer in interval $[a, b]$ is the sum of the interferences of all other tasks (in the same interval) divided by the number of processors:*

$$I_k = \frac{\sum_{i \neq k} I_{i,k}}{m}. \tag{14.2}$$

This follows from the observation that since the scheduling algorithm under consideration is work-conserving, in the time instants in which the job of τ_k is active but not executing, each of the m processors must be executing a job of some other task. Since $I_{k,k} = 0$, we can exclude the contribution of τ_k to the total interference.

14.5 Comparing Different Schedulability Tests

Given two different schedulability tests for a particular scheduling algorithm, one schedulability test is said to *dominate* the other if all task systems that can be shown schedulable by the second test can also be shown schedulable by the first, while the converse of this statement is not true: there are task systems that can be shown to be schedulable by the first test that the second test fails to establish as being schedulable.

Only a few dominance results can be claimed amongst the different global EDF schedulability tests described in this book. Namely, it is possible to analytically prove that:

1. The test described in Chap. 19 (we will call this the FF-DBF test) dominates the density-based test (Theorem 13.2), and
2. The [RTA] test described in Chap. 17 dominates the [BCL] test of Sect. 15.1.

All other pairs of tests are incomparable, meaning that for each pair of tests it is possible to find EDF-schedulable task sets that are correctly identified as being so by one of the tests but not by the other, and viceversa.

We now try to outline the main features that uniquely characterize each test. The differences among the tests are mainly related to the considered interval of interest, and the computed bound on the carry-in contributions to the workload—i.e., the interference due to jobs not entirely contained inside the considered interval of interest.

- The condition used in the [BAK] test (Chap. 16) is derived by defining the interval of interest in such a manner that a good bound can be derived on the maximum carry-in contribution of *each* task.
- The unique feature of the [BAR] test is that the interval of interest is defined in such a manner that only $(m - 1)$ jobs may contribute carry-in work, where m denotes the number of processors.
- Extensions to the [BAK] test, also described in Chap. 16, are able to limit both the number of jobs that contribute carry-in work and the amount of carry-in work by each such job.
- The advantageous feature of the [BCL] and [RTA] tests is an iterative procedure that allows for a reduction of the amount of carry-in work an individual job may contribute;
- The FF-DBF test (described in Chap. 19) derives a better bound than earlier tests on the total amount of carry-in work that is contributed by any task. A similar, although weaker, bound is found in the density-based test as well.

Bertogna [56] conducted extensive schedulability experiments upon randomly-generated task systems in an attempt to better characterize the relative effectiveness of the above EDF schedulability tests.

In addition to the tests mentioned above and described in later chapters of this book, a new test (denoted as [COMP]) was also considered, that was obtained by composing [RTA], [BAR] and the FF-DBF tests; Bertogna's experiments indicate that

[COMP] appears to be the most effective test. In order to test a given task system for EDF-schedulability, [COMP] first tests the task system using an "integration" of the [RTA] and [BAR] tests (the exact nature of this integration is detailed in [56, 57]); if this fails to determine that the task system is EDF-schedulable, then [COMP] tests the system using the FF-DBF test.

It is shown in [56] how the integration of [RTA] and [BAR] in the [COMP] test strictly dominates the simple serial combination of both tests.

(As explained in [56], the [COMP] test does not include the other tests mentioned above since the density-based test and [BCL] are already dominated by the FF-DBF test and [RTA], respectively, while [BAK] and its extension seemed to detect very few additional schedulable tasks—less than one out of every 10^6 randomly-generated task sets in the experiments reported in [56]).

14.5.1 Sustainability

Recall the concept of sustainability from Sect. 3.3, which seeks to formalize the intuitive notion that the performance of a scheduling algorithm or schedulability test should improve for systems that are "easier" to schedule: If a system is schedulable by a particular scheduling algorithm (or deemed schedulable by a particular schedulability test), then a system derived from the original by relaxing the parameters (e.g., decreasing the WCETs, increasing the periods and/or relative deadlines, etc.) should also be schedulable by that algorithm (deemed schedulable by the same schedulability test, respectively). It was argued in [40] that scheduling algorithms and schedulability tests that do not possess this property, i.e., are not sustainable—are not sufficiently robust for critical applications.

A scheduling policy or schedulability test for three-parameter sporadic task systems may be sustainable with respect to some, but not all three, job parameters. A scheduling policy or a schedulability test for sporadic task systems is said to be sustainable if it is sustainable with respect to all three parameters.

The notion of sustainability for schedulability tests for three-parameter sporadic task systems was strengthened somewhat in [23]:

Definition 14.2 (sustainable schedulability test [23]) Let A denote a scheduling policy, and F an A-schedulability test for three-parameter sporadic task systems. Let τ denote any such task system deemed to be A-schedulable by F. Let J denote a collection of jobs generated by τ. F is said to be a *sustainable* schedulability test if and only if scheduling policy A meets all deadlines when scheduling any collection of jobs obtained from J by changing the parameters of one or more individual jobs in any, some, or all of the following ways: (i) decreased execution requirements, (ii) larger relative deadlines, and (iii) later arrival times with the restriction that successive jobs of any task $\tau_i \in \tau$ arrive at least T_i time units apart.

As pointed out in [23], declaring a schedulability test for sporadic task systems to be sustainable represents a stronger claim than simply that a task system deemed schedulable would remain schedulable with ‘‘better’’ parameters (e.g., with a larger period or relative deadline, or a smaller execution requirement). In addition, sustainability demands that a system deemed schedulable continue to meet all deadlines even if the parameter changes are occurring ‘‘on line’’ during run-time. It is permitted that the parameters change back and forth, on a job-to-job basis, arbitrarily many times. The only restriction placed on such parameter-changing is that each generated job have exactly one arrival time, execution requirement, and deadline during its lifetime.

Sustainability of schedulability tests, as defined in Definition 14.2, provides a guarantee that if a task system is deemed schedulable by the test, it will not fail to meet all deadlines if the system behaves better at run time than the specifications.

A different kind of sustainability property for schedulability tests called *self-sustainability* was also defined in [23]:

Definition 14.3 (self-sustainability [23]) A schedulability test is *self-sustainable* if all task systems with “better” (less constraining) parameters than a task system deemed schedulable by the test are also be deemed schedulable by the test.

This form of sustainability was called *self-sustainability* to emphasize the difference between this and the preceding definition of sustainability: The system is not only required to remain schedulable under various parameter relaxations but also it is required to be verifiably schedulable by the same test. The notion of self-sustainability is motivated in [23] by the requirements of the incremental, interactive design process that is typically used in the design of real-time systems and in the evolutionary development of fielded systems. Ideally, such a design process allows for the interactive exploration of the space of possible designs by the system designer; such interactive design exploration is greatly facilitated if changes that are viewed as relaxations of constraints actually result in making feasible larger regions of the design space. If a self-sustainable schedulability test is used, then relaxing timing constraints (e.g, increasing relative deadlines or periods, or decreasing worst-case execution times) will not render schedulable subsystems unschedulable (or the practical equivalent: unverifiable). For example, suppose one were designing a composite system comprised of several subsystems, and had reached a point where most subsystems are deemed schedulable by the schedulability test. One could safely consider relaxing the task parameters of the schedulable subsystems in order to search for neighboring points in the design state space in which the currently unschedulable subsystems may also be deemed schedulable, without needing to worry that the currently schedulable subsystems would unexpectedly be deemed unschedulable.

The following results concerning sustainability of global EDF were established in [23]:

1. EDF is a sustainable scheduling policy with respect to WCET and periods.
2. It is not known whether EDF is sustainable with respect to the relative deadline parameter. That is, it is not known whether a task system τ' that is derived from

an EDF-schedulable task system τ by increasing the relative deadline parameters of some or all of the tasks in τ is also EDF-schedulable or not.[2]
3. The [BCL] test is not self-sustainable with respect to relative deadlines—i.e., it is possible that a task system will be deemed EDF-schedulable by the [BCL] test but another task system in which each task has a relative deadline no smaller than that of this original task system will not.
4. The EDF schedulability test specified in Theorem 16.2, which is derived from the [BAK] test, is also not self-sustainable with respect to relative deadlines.
5. However, the EDF schedulability test specified in Theorem 16.4, which is also derived from the [BAK] test, is indeed self-sustainable with respect to relative deadlines.

A few sustainability results in addition to the ones from [23] listed above are known:

1. All density-based tests (and, therefore, the corresponding utilization-based tests) are sustainable with respect to all the parameters—WCET, period, and relative deadline.
2. [BCL] and [RTA] are sustainable with respect to WCET and task periods.

And finally, an application of Lemma 14.2, also from [23], specifies that EDF is sustainable upon simultaneous relaxations of both periods and WCETs:

Lemma 14.2 *If a scheduling policy is sustainable under individual kinds of parameter relaxations, for sporadic task systems, then it is sustainable under the same relaxations in any combination.*

Sources

The general strategy for designing global multiprocessor schedulability tests that was described in Sect. 14.1 abstracted out from the many tests that we will study in the following chapters, as are the notions of workload bounds and interference discussed in Sects. 14.3 and 14.4 respectively. The uniprocessor test that is described in Sect. 14.2 in terms of the strategy of Sect. 14.1, is from [51]. The observations comparing the various schedulability tests that are presented in Sect. 14.5 are primarily taken from [56, 57]. The concept of sustainability was introduced in [40]; as stated in Sect. 3.3, it is closely related to the earlier notion of predictability that was introduced by Ha and Liu [105–107]. Sustainability properties in multiprocessor scheduling was studied in [23].

[2] As pointed out in [23], this fact is not a particularly severe impediment to the use of EDF for scheduling sporadic task systems in a robust manner: An EDF implementation that uses specified, rather than actual, job deadlines to determine job priority during run-time will maintain schedulability even in the event of the actual deadlines during run-time being greater than those specified for the system that was verified for schedulability.

Chapter 15
The [BCL] and [BAR] Tests

In this chapter, we will describe some of the tests that have been developed for global earliest-deadline-first (EDF) schedulability analysis of three-parameter sporadic task systems. Each test will be denoted with an acronym, generally derived from the names or initials of the authors of the publication in which the test was first presented[1]. Not surprisingly given the negative results in Chap. 12, these tests are all sufficient rather than exact. They are also by and large incomparable in the sense that none dominates all the others—there are task systems determined to be schedulable by each test that the others fail to show as being schedulable.

Throughout this chapter, we will consider a sporadic task system τ comprised of the n tasks $\tau_1, \tau_2, \ldots, \tau_n$, with τ_i having parameters (C_i, D_i, T_i), that is to be scheduled upon an m-processor platform. To keep the presentation simple we will assume that τ is a constrained-deadline task system: $D_i \leq T_i$ for all i, $1 \leq i \leq n$; we will occasionally, informally explain how the results described here may be extended to arbitrary-deadline sporadic task systems.

The first test ([BCL]) is described in Sect. 15.1; this test has the virtues of simplicity, elegance, and low run-time complexity. Second test, [BAR], is then described in Sect. 15.2; the [BAR] test essentially represents a direct extension of the technique outlined in Sect. 14.2 from uniprocessor to multiprocessor systems. It has a greater run-time complexity than the [BCL] test.

[1] Brian Kernighan, one of the authors of the AWK programming language, is said to have noted that "Naming a language after its authors [$\cdots$] shows a certain poverty of imagination." Nevertheless, this particular practice seems to have become established with respect to global schedulability tests.

15.1 The [BCL] Test

We first describe the test devised by Bertogna et al. [59], for constrained-deadline sporadic task systems. As per the template outlined in Chap. 14 we assume that the task system τ is not EDF schedulable upon m unit-speed processors, and identify conditions that are necessary for that to happen. The negate of these conditions will yield the sufficient schedulability test.

Identifying an Unschedulable Collection of Jobs Since τ is assumed to not be EDF schedulable, there must, by definition, exist collections of jobs generated by τ upon which deadlines are missed by EDF. Let us suppose that it is a job of task $\tau_k = (C_k, D_k, T_k)$ that misses its deadline for some such collection of jobs, and let t_d denote the instant at which this deadline miss occurs. We can obtain another unschedulable collection of jobs of τ from this collection, such that the amount of execution received by τ_k's job prior to its deadline is less than C_k by an arbitrarily small amount in the EDF schedule—such a collection of job is easily obtained from the original, by appropriately reducing the (actual) execution-times of jobs other than the job of τ_k, to allow τ_k's job more execution.

The Interval of Interest is the *scheduling window*—the interval between a job's arrival instant and its deadline—of the job of τ_k that misses its deadline.

Since EDF is a work-conserving algorithm (i.e., it never idles a processor while there are active jobs eligible to execute awaiting execution), τ_k's job must be executing at all time-instants during its scheduling window when some processor is available (it may execute at other instants as well). By the manner in which the unschedulable collection of jobs was selected above, we know that τ_k executes for less than C_k time units.

Hence, all m processors must be executing other jobs for $> (D_k - C_k)$ time units, and a lower bound on the demand W over the interval of interest is given by

$$W_L = C_k + m(D_k - C_k).$$

An Upper Bound on Demand Let $I_{i,k}$ denote an upper bound on the interference of τ_i on the job of τ_k that misses its deadline. An upper bound on the computational demand W by the considered collection of jobs over the interval of interest is then given by

$$W_U = C_k + \sum_{i \neq k} I_{i,k}$$

To provide an upper bound on the interference of τ_i on a job of τ_k, we compute an upper bound on the workload that τ_i can execute in the interval of interest.

We note that no carry-out job can interfere with task τ_k in the considered interval, since it has, by definition, a later deadline than τ_k's. We can therefore consider a

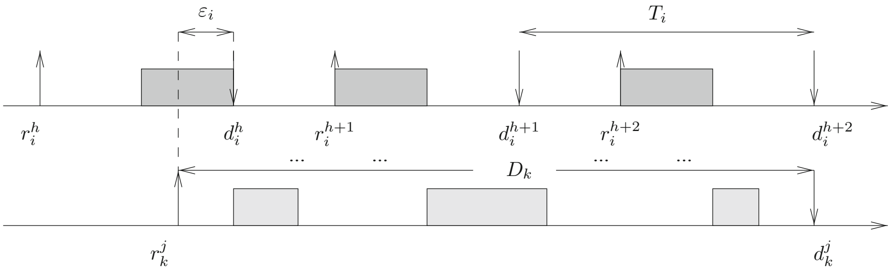


Fig. 15.1 Scenario that produces the maximum possible interference of task τ_i on a job of task τ_k when EDF is used

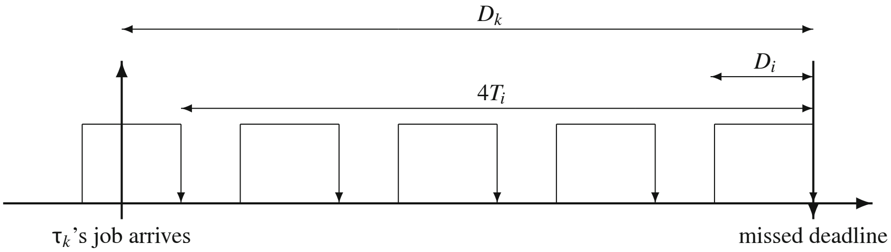


Fig. 15.2 Illustrating Eq. 15.1. There are four jobs of τ_i with both release time and deadline within the interval of interest (the scheduling window of τ_k's job). The second term on the right-hand side of Eq. 15.1 denotes the execution requirement of the carry-in job of τ_i, under the assumption that it executes with zero laxity. D_k mod T_i denotes the portion of carry-in job's scheduling window that intrudes into the scheduling window of τ_k's job is D_k mod T_i

situation in which the last job of τ_i has its deadline at the end of the interval, i.e., coinciding with τ_k's deadline, and every other instance of τ_i is executed as late as possible. The situation is depicted in Fig. 15.1.

To express τ_i's workload, we separate the contribution of the first job contained in the interval (not necessarily the carry-in job) from the rest of the jobs of τ_i. Each one of the jobs after the first one may contribute as much as its entire worst-case computation time C_i. There are $\lfloor(D_k/T_i)\rfloor$ such jobs. However, the first job may contribute no more than $\min(C_i, D_k \bmod T_i)$.

We therefore obtain the following expression (see Fig. 15.2):

$$I_{i,k} \le \left\lfloor \frac{D_k}{T_i} \right\rfloor C_i + \min\left(C_i, D_k \bmod T_i\right). \tag{15.1}$$

We can obtain a different bound on $I_{i,k}$, by the following argument. By the definition of interference, $I_{i,k}$ cannot exceed the duration of τ_k's job scheduling window *minus* the amount of execution received by τ_k's job prior to its deadline. Since we chose our unschedulable collection of jobs, such that τ_k's job executes for just a bit less than C_k time units over the scheduling window, it follows this interference can be no more than $(D_k - C_k + \epsilon)$, where ϵ is an arbitrarily small positive number. We therefore have

$$I_{i,k} < D_k - C_k + \epsilon.$$

Putting the pieces together, we obtain

$$I_{i,k} \leq \min\left(D_k - C_k + \epsilon, \left\lfloor \frac{D_k}{T_i} \right\rfloor C_i + \min\left(C_i, D_k \bmod T_i\right)\right) \tag{15.2}$$

Determining a Sufficient Schedulability Condition Since a necessary condition for τ_k to miss its deadline under EDF scheduling is that

$$(W_U > W_L) \equiv \left(C_k + \sum_{i \neq k} I_{i,k} > C_k + m(D_k - C_k)\right) \equiv \left(\sum_{i \neq k} I_{i,k} > m(D_k - C_k)\right).$$

Its negation yields a sufficient condition for EDF schedulability.

Theorem 15.1 *Task system τ is EDF-schedulable if, for each $\tau_k \in \tau$,*

$$\sum_{i \neq k} \min\left(D_k - C_k + \epsilon, \left\lfloor \frac{D_k}{T_i} \right\rfloor C_i + \min(C_i, D_k \bmod T_i)\right) \leq m(D_k - C_k)\,. \tag{15.3}$$

Note that in the special case when the time domain of the task parameters is integer, ϵ can be set to 1. The reason why this term is needed is to rule out the scenario in which there are a total of $m + 1$ tasks, and each of the other m tasks τ_i has $I_{i,k}$ strictly greater than $(D_k - C_k)$; in this case, these other m tasks could interfere for more than $D_k - C_k$ time units, leading to a deadline miss. Without the ϵ term, the test would instead incorrectly deem this system schedulable.

Alternatively, it is possible to avoid using the ϵ term by using a strict inequality, as follows

Theorem 15.2 *Task system τ is EDF schedulable if, for each $\tau_k \in \tau$,*

$$\sum_{i \neq k} \min\left(D_k - C_k, \left\lfloor \frac{D_k}{T_i} \right\rfloor C_i + \min(C_i, D_k \bmod T_i)\right) < m(D_k - C_k)\,. \tag{15.4}$$

Extension to Arbitrary-Deadline Systems As described above, the [BCL] test is only applicable to constrained-deadline systems. It has, however, been extended to systems of arbitrary-deadline tasks, by essentially "offsetting" all job times to the deadline of the prior job of the same task, and hence doing the equivalent of replacing each task with $D_i > T_i$ with one having $D_i = T_i$.

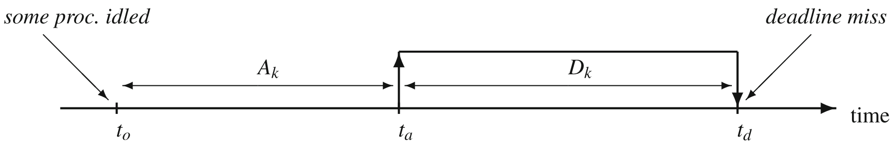


Fig. 15.3 Notation. A job of task τ_k arrives at t_a and misses its deadline at time-instant t_d. The latest time-instant prior to t_a when not all m processors are busy is denoted t_o

15.2 The [BAR] Test

We now present the test proposed in [31].

This test differs from the one in Sect. 15.1 in several ways: (i) it runs in time pseudo-polynomial in the representation of the task system, while it is evident from inspecting Theorem 15.2 that the [BCL] test has a polynomial run-time complexity; (ii) it considers intervals of intervals $[t_o, t_d)$ that do not necessarily coincide with the scheduling window of the task τ_k being assumed to miss its deadline; (iii) in computing interference, it allows us to bound the number of tasks carrying in work into the interval $[t_o, t_d)$ at $(m-1)$, where m is the number of processors in the system; however, the amount of interference allowed in by each of these $(m-1)$ tasks by this test may exceed the amount of carry-in interference allowed by the test in Sect. 15.1. It will be shown (Corollary 15.1) that this test, unlike the [BCL] test, generalizes the known exact EDF-schedulability test on uniprocessors. The two tests are known to be incomparable: Schedulable task systems have been identified that can be detected by each of the tests but not by the other. In particular, the test of the previous section appears to be superior upon task systems where there is a limited number of tasks with large C_i values, but suffers when there are $> m$ tasks with "large" execution requirements (C_i's).

As with the [BCL] test, this test, too, considers each task τ_k separately; when considering a specific τ_k, it seeks to identify sufficient conditions for ensuring that τ_k cannot miss any deadlines. To ensure that no deadlines are missed by any task in τ, these conditions must be checked for each of the n tasks $\tau_1, \tau_2, \ldots, \tau_n$.

Identifying an Unschedulable Collection of Jobs Consider any legal sequence of job requests of task system τ, on which EDF misses a deadline. Suppose that a job of task τ_k is the one to first miss a deadline, and that this deadline miss occurs at time-instant t_d (see Fig. 15.3). Let t_a denote this job's arrival time: $t_a = t_d - D_k$.

Discard from the legal sequence of job requests all jobs with deadline $> d_k$, and consider the EDF schedule of the remaining (legal) sequence of job requests. Since later-deadline jobs have no effect on the scheduling of earlier-deadline ones under preemptive EDF, it follows that a deadline miss of τ_k occurs at time-instant t_d (and this is the earliest deadline miss), in this new EDF schedule.

Identifying an Interval of Interest Let t_o denote the latest time-instant $\le t_a$ at which at least one processor is idled in this EDF schedule. (Since all processors are, by definition, idle prior to time-instant $t = 0$, such a t_o is guaranteed to exist.) The interval of interest is the interval $[t_o, t_d)$.

Let $A_k \overset{\text{def}}{=} t_a - t_o$.

Our goal now is to identify a lower bound on the computational demand over this interval of interest, that ensures a deadline miss for τ_k's job at time t_d, i.e., for τ_k's job to execute for strictly less than C_k time units over $[t_a, t_d)$. In order for τ_k's job to execute for strictly less than C_k time units over $[t_a, t_d)$, it is necessary that all m processors be executing jobs other than τ_k's job for strictly more than $(D_k - C_k)$ time units over $[t_a, t_d)$. Let us denote by Γ_k a collection of intervals, not necessarily contiguous, of cumulative length $(D_k - C_k)$ over $[t_a, t_d)$, during which all m processors are executing jobs other than τ_k's job in this EDF schedule.

For each i, $1 \le i \le n$, let $I(\tau_i)$ denote the contribution of τ_i to the work done in this EDF schedule during $[t_o, t_a) \bigcup \Gamma_k$. In order for the deadline miss to occur, it is necessary that the total amount of work that executes over $[t_o, t_a) \bigcup \Gamma_k$ satisfies the following condition

$$\sum_{\tau_i \in \tau} I(\tau_i) > m \times (A_k + D_k - C_k)\ ; \qquad (15.5)$$

this follows from the observation that all m processors are, by definition, completely busy executing this work over the A_k time units in the interval $[t_o, t_a)$, as well as the intervals in Γ_k of total length $(D_k - C_k)$.

Observe that the total length of the intervals in $[t_o, t_a) \bigcup \Gamma_k$ is equal to $(A_k + D_k - C_k)$.

An Upper Bound on Demand Recall that we say that τ_i has a *carry-in job* in this EDF schedule if there is a job of τ_i that arrives before t_o and has not completed execution by t_o. In the following, we compute upper bounds on $I(\tau_i)$ if τ_i has no carry-in job (this is denoted as $I_1(\tau_i)$), or if it does (denoted as $I_2(\tau_i)$).

- *Computing* $I_1(\tau_i)$. If a task τ_i contributes no carry-in work, then its contribution to this total amount of work that must execute over $[t_o, t_a) \bigcup \Gamma_k$ is generated by jobs arriving in, and having deadlines within, the interval $[t_o, t_d)$. Let us first consider $i \ne k$; in that case, it follows from the definition of the demand bound function (DBF —see Sect. 10.3) that the total work is at most $\text{DBF}(\tau_i, A_k + D_k)$; furthermore, this total contribution cannot exceed the total length of the intervals in $[t_o, t_a) \bigcup \Gamma_k$. Hence, the contribution of τ_i to the total work that must be done by EDF over $[t_o, t_a) \bigcup \Gamma_k$ is at most

  $$\min(\text{DBF}(\tau_i, A_k + D_k), A_k + D_k - C_k).$$

  Now, consider the case $i = k$. In that case, the job of τ_k arriving at time-instant t_a does not contribute to the work that must be done by EDF over $[t_o, t_a) \bigcup \Gamma_k$; hence, its execution requirement must be subtracted. Also, this contribution cannot exceed the length of the interval $[t_o, t_a)$, i.e., A_k.

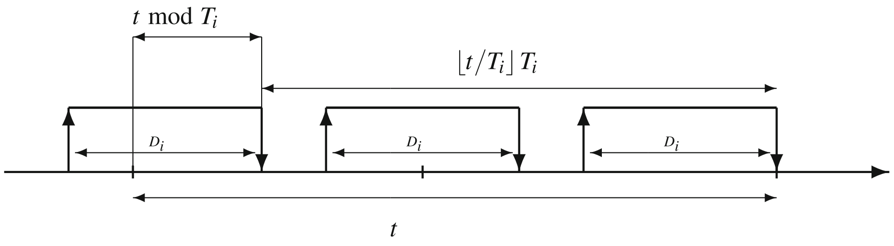


Fig. 15.4 Computing $\text{DBF}'(\tau_i, t)$

Putting these pieces together, we get the following bound on the contribution of τ_i to the total work that must be done by EDF over $[t_o, t_a) \bigcup \Gamma_k$:

$$I_1(\tau_i) \overset{\text{def}}{=} \begin{cases} \min(\text{DBF}(\tau_i, A_k + D_k), A_k + D_k - C_k) & \text{if } i \neq k \\ \min(\text{DBF}(\tau_i, A_k + D_k) - C_k, A_k) & \text{if } i = k \end{cases} \tag{15.6}$$

- *Computing* $I_2(\tau_i)$. Let us now consider the situation when τ_i is active at t_o, and hence potentially carries in some work.
  It was shown in [59] that the total work of τ_i in this case can be upper-bounded by considering the scenario in which some job of τ_i has a deadline at t_d, and all jobs of τ_i execute at the very end of their scheduling windows.
  Let us denote as $\text{DBF}'(\tau_i, t)$ the amount of work that can be contributed by τ_i over a contiguous interval of length t, if some job of τ_i has its deadline at the very end of the interval and each job of τ_i executes during the C_i units immediately preceding its deadline. It is easily seen (see Fig. 15.4), that there are exactly $\lfloor t/T_i \rfloor$ complete jobs of τ_i within this interval, and an amount $\min(D_i, t \bmod T_i)$ of the scheduling window of an additional, carry-in, job.
  This carry-in scheduling window may bring in at most C_i units of execution, yielding the following expression for $\text{DBF}'(\tau_i, t)$:

$$\text{DBF}'(\tau_i, t) \overset{\text{def}}{=} \left\lfloor \frac{t}{T_i} \right\rfloor \times C_i + \min(C_i, t \bmod T_i) \tag{15.7}$$

In computing τ_i's contribution to the total amount of work that must execute over $[t_o, t_a) \bigcup \Gamma_k$, let us first consider $i \neq k$. In that case, it follows from the definition of the demand bound function (DBF', as defined above) that the total work is at most $\text{DBF}'(\tau_i, A_k + D_k)$; furthermore, this total contribution cannot exceed the total length of the intervals in $[t_o, t_a) \bigcup \Gamma_k$. Hence, the contribution of τ_i to the total work that must be done by EDF over $[t_o, t_a) \bigcup \Gamma_k$ is at most

$$\min(\text{DBF}'(\tau_i, A_k + D_k), A_k + D_k - C_k).$$

Now, consider the case $i = k$. In that case, the job of τ_k arriving at time-instant t_a does not contribute to the work that must be done by EDF over $[t_o, t_a) \bigcup \Gamma_k$;

hence, its execution requirement must be subtracted. Also, this contribution cannot exceed the length of the interval $[t_o, t_a)$, i.e., A_k.
From the discussion above, we get the following bound on the contribution of τ_i to the total work that must be done by EDF over $[t_o, t_a) \bigcup \Gamma_k$:

$$I_2(\tau_i) \overset{\text{def}}{=} \begin{cases} \min(\text{DBF}'(\tau_i, A_k + D_k), A_k + D_k - C_k) & \text{if } i \neq k \\ \min(\text{DBF}'(\tau_i, A_k + D_k) - C_k, A_k) & \text{if } i = k \end{cases} \tag{15.8}$$

Deriving a Sufficient Schedulability Condition Let us denote by $I_{\text{DIFF}}(\tau_i)$ the difference between $I_2(\tau_i)$ and $I_1(\tau_i)$:

$$I_{\text{DIFF}}(\tau_i) \overset{\text{def}}{=} I_2(\tau_i) - I_1(\tau_i) \tag{15.9}$$

By definition of t_o, at most $(m-1)$ tasks are active at time-instant t_o. Consequently, there are at most $(m-1)$ tasks τ_i that contribute at amount $I_2(\tau_i)$, and the remaining $(n-m+1)$ tasks must contribute $I_1(\tau_i)$. Hence Eq. 15.5 may be rewritten as follows:

$$\sum_{\tau_i \in \tau} I_1(\tau_i) + \sum_{\text{the } (m-1) \text{ largest}} I_{\text{DIFF}}(\tau_i) > m(A_k + D_k - C_k) \tag{15.10}$$

Observe that all the terms in Eq. 15.10 are completely defined for a given task system, once a value is chosen for A_k. Hence for a deadline miss of τ_k to occur, there must exist some A_k such that Eq. 15.10 is satisfied. Conversely, in order for all deadlines of τ_k to be met it is sufficient that Eq. 15.10 be violated for all values of A_k. Theorem 15.3 follows immediately:

Theorem 15.3 *Task system τ is EDF-schedulable upon m unit-capacity processors if for all tasks $\tau_k \in \tau$ and all $A_k \geq 0$,*

$$\left(\sum_{\tau_i \in \tau} I_1(\tau_i) + \sum_{\textit{the } (m-1) \textit{ largest}} I_{\text{DIFF}}(\tau_i)\right) \leq m(A_k + D_k - C_k) \tag{15.11}$$

where $I_1(\tau_i)$ and $I_{\text{DIFF}}(\tau_i)$ are as defined in Eqs. 15.6 and 15.9 respectively. □

The earlier tests—the density and [BCL] tests—are not exact even for uniprocessor systems. (This is also indicated by the fact that all these prior tests have polynomial run-time, while EDF-schedulability analysis of sporadic task systems on uniprocessors has been shown to be NP-hard [81].) The following corollary asserts that the [BAR] test is superior to earlier tests in this regard:

Corollary 15.1 *The EDF schedulability test of Theorem 15.3 is a generalization of the exact uniprocessor EDF schedulability test of [51].*
Proof Sketch: For $m = 1$, there are $(m-1) = 0$ tasks that are active at time-instant t_o, i.e., t_o is the classical "idle instant" of uniprocessor real-time scheduling theory. By adding C_k to both the LHS and the RHS of Condition 15.11, it can be shown that the LHS reduces to the sum of the demand bound functions of all tasks over an

interval of size $A_k + D_k$, and the RHS reduces to the interval length. Hence, when $m = 1$ Condition 15.11 is asserting that the cumulative processor demand over all intervals must not exceed the interval length, which is exactly what the uniprocessor EDF schedulability test of [51] checks. □

15.2.1 Run-Time Complexity

For given τ_k and A_k, it is easy to see that Condition 15.11 can be evaluated in time linear in n:

- Compute $I_1(\tau_i)$, $I_2(\tau_i)$, and $I_{\text{DIFF}}(\tau_i)$ for each i—total time is O(n).
- Use linear-time selection [61] on $\{I_{\text{DIFF}}(\tau_1), I_{\text{DIFF}}(\tau_2), \ldots, I_{\text{DIFF}}(\tau_n)\}$ to determine the $(m-1)$ tasks that contribute to the second sum on the LHS.

How many values of A_k must be tested, in order for us to be able to ascertain that Condition 15.11 is satisfied for all $A_k \geq 0$? Theorem 15.4 provides the answer.

Theorem 15.4 *If Condition 15.11 is to be violated for any A_k, then it is violated for some A_k satisfying the condition below:*

$$A_k \leq \frac{C_\Sigma - D_k(m - U_{sum}(\tau)) + \sum_i (T_i - D_i)U_i + mC_k}{m - U_{sum}(\tau)} \tag{15.12}$$

where C_Σ denotes the sum of the $(m-1)$ largest C_i's.

Proof It is easily seen that $I_1(\tau_i) \leq \text{DBF}(\tau_i, A_k + D_k)$, and $I_2(\tau_i) \leq \text{DBF}(\tau_i, A_k + D_k) + C_i$. From this, it can be shown that the LHS of Condition 15.11 is $\leq C_\Sigma + \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, A_k + D_k)$.

For this to exceed the RHS of Condition 15.11, it is necessary that

$$C_\Sigma + \text{DBF}(\tau, A_k + D_k) > m(A_k + D_k - C_k)$$

$$\Rightarrow \text{(bounding DBF using the technique of [51])}$$

$$C_\Sigma + (A_k + D_k)U_{\text{sum}}(\tau) + \sum_i (T_i - D_i)U_i > m(A_k + D_k - C_k)$$

$$\Leftrightarrow \; C_\Sigma + D_k U_{\text{sum}}(\tau) + \sum_i (T_i - D_i)U_i - m(D_k - C_k) > A_k(m - U_{\text{sum}}(\tau))$$

$$\Leftrightarrow A_k \leq \frac{C_\Sigma - D_k(m - U_{\text{sum}}(\tau)) + \sum_i (T_i - D_i)U_i + mC_k}{m - U_{\text{sum}}(\tau)}$$

which is as claimed in the theorem. □

It can also be shown that Condition 15.11 need only be tested at those values of A_k at which $\text{DBF}(\tau_i, A_k + D_k)$ changes for some τ_i. Corollary 15.2 follows.

Corollary 15.2 *The condition in Theorem 15.3 can be tested in time pseudo-polynomial in the task parameters, for all task systems τ for which $U_{sum}(\tau)$ is bounded by a constant strictly less than the number of processors m.* □

15.2.2 A Simpler Formulation

Letting $\Delta \overset{\text{def}}{=} (t_d - t_o)$, Condition 15.10, our necessary condition for a deadline miss, can be expressed as

$$m\Delta - (m-1)C_k > \text{DBF}(\tau, \Delta) + C_\Sigma$$

for some $\Delta \geq D_k$. The contrapositive of the above statement, in conjunction with the result of Theorem 15.4, gives the following somewhat simpler formulation of the [BAR] schedulability test

Theorem 15.5 *A sufficient condition for EDF-schedulability is*

$$\forall\, \Delta \;: D_k \leq \Delta \leq \hat{\Delta} :\; m\Delta - (m-1)C_k \leq \text{DBF}(\tau, \Delta) + C_\Sigma,$$

where

$$\hat{\Delta} \leftarrow D_k + \frac{C_\Sigma - D_k(m - U_{sum}(\tau)) + \sum_i (T_i - D_i)U_i + mC_k}{m - U_{sum}(\tau)}$$

and C_Σ denotes the sum of the $(m-1)$ largest C_i's.

15.2.3 Non-preemptive Scheduling

Although we are not covering non-preemptive scheduling in this book, we note in passing that a clever extension of this idea for non-preemptive scheduling was explored in [103]. It was observed there that for non-preemptive scheduling, the necessary condition for a job of some task τ_k to miss its deadline at time at some time-instant t_d is that it not have *started* execution prior to time-instant $t_d - C_k$. Hence (with t_o defined as above—see Fig. 15.3), the interval of interest for non-preemptive scheduling becomes $[t_o, t_d - C_k)$ rather than $[t_o, t_d)$, and it must be the case that all m processors are executing other work throughout this interval. This observation was exploited to obtain a very effective schedulability test for non-preemptive EDF; details may be found in [103].

Sources

The [BCL] test was first described in [59]; it was elaborated upon and extended in Bertogna's dissertation [55]. The [BAR] test was first proposed in [31].

Chapter 16
The [BAK] Test

In this chapter, we present the sufficient schedulability test for global EDF that was obtained by Baker [19]. This test is based on some very deep insights, and contains some remarkably sophisticated ideas which were incorporated into tests developed later by other researchers. We consider the [BAK] test, and some related results that are also discussed in this chapter, to have played a crucial role in enabling the development of many advanced results concerning global multiprocessor real-time scheduling. However, the [BAK] test itself (as opposed to the ideas contained within it) is of limited significance today.

- From the practical perspective of actually testing sporadic task systems in order to determine whether they are schedulable or not, there is some evidence to suggest that the [BAK] test is, in general, inferior to the [BCL] test. This is partly because the more sophisticated analysis of the [BAK] test prevents the use of some of the simpler techniques for computing workload bounds that seem to make a big contribution to the effectiveness of the [BCL] test. (A specific example is being able to bound the contribution of τ_i, $i \neq k$, to $D_k - C_k$.) Here is one opinion[1] regarding the relativeness effectiveness of the [BAK] test vis a vis other tests: "It seems that many of the cases where [the [BAK]] test does better than the [BCL] test are also cases where the density test works, and many of the cases where [the [BAK]] does better than the density test the [BCL] test also works."
- In contrast to the density, [BCL], and [BAR] tests, the [BAK] test could be analyzed to quantify its effectiveness via the speedup factor metric (Definition 5.2). However, subsequent analysis [63] has yielded sufficient schedulability tests for global EDF with superior speedup bounds (these will be discussed in Chap. 19), which means that the [BAK] test is no longer the best-known one from the perspective of speedup factor, either.

The [BAK] test nevertheless remains worthy of examination, given the interesting ideas contained in it—we believe that pursuing these ideas provides a good deal of insight into the mechanism of global scheduling.

[1] Personal email communication from Ted Baker (22 January 2007).

16.1 Overview of the [BAK] Test

Recall the general strategy towards global schedulability analysis that was described in Sect. 14.1. We will assume that a task system τ is not global EDF-schedulable upon m unit-capacity processors, and consider a sequence of job requests of τ on which global EDF misses one or more deadlines. Suppose that a job of some task $\tau_k \in \tau$ is the first job that misses its deadline, at some time-instant t_d. We will analyze the situation over some time interval $[t_o, t_d)$. We will compute upper bounds on the amount of work that EDF is required to execute over the interval $[t_o, t_d)$, and obtain a necessary unschedulability condition by setting this bound to be large enough to deny τ_k's job C_k units of execution over its scheduling window. By negating this necessary unschedulability condition, we will obtain our desired schedulability condition.

16.1.1 The Problem of Carry-ins

The "goodness" of our general strategy depends on the accuracy of the upper bound on the amount of work that EDF is required to execute over the interval of interest. The major problem with obtaining an accurate upper bound is the problem of *carry-in jobs*, which were discussed in Sect. 14.3.

Recall that a carry-in job is a job that arrives prior to, and has a deadline within, the interval of interest. Uniprocessor schedulability tests are so accurate because they generally do not need to deal with carry-in jobs—by defining the start of the interval of interest to be the latest time-instant at which the processor is idled, we guarantee that there are no carry-in jobs to worry about.

For most global multiprocessor scheduling algorithms, it seems very difficult to accurately estimate the amount of work that the carry-in jobs contribute to the total work over the interval of interest.

Early global schedulability tests tended to be very pessimistic in this regard: For example, the [BAR] test adds the entire execution requirement of each carry-in job to the total work to be done over the interval of interest, while the [BCL] test assumes (see Fig. 15.2) that each carry-in job executes at the very end of its scheduling window.

16.1.2 The [BAK] *Breakthrough*

In [18], Baker came up with a clever technique for bounding the amount of work contributed by the carry-in jobs. The idea was to define the *start* of the interval of interest in a manner which allowed us to bound not just the number of carry-in jobs (as we saw being done for uniprocessors, and in the [BAR] test), but also their individual workload contributions. The basic idea, to be fleshed out in the remainder of this chapter, is this.

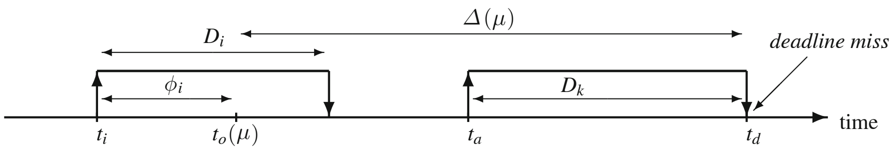


Fig. 16.1 Notation. A job of task τ_k arrives at t_a and misses its deadline at time-instant t_d

- Let us suppose that a job of τ_k, arriving at time-instant t_a, misses its deadline at $t_d = t_a + D_k$ – see Fig. 16.1.
- We can determine a lower bound on the amount of work that the scheduling algorithm needed (and failed) to complete over the interval $[t_a, t_d)$ (an example of such a bound is $(D_k - C_k)m + C_k$, since all m processors must have been busy whenever τ_k's job was not executing). Let μ denote this lower bound on work, normalized by the interval-length $(t_d - t_a)$ (for the example lower bound of $(D_k - C_k)m + C_k$, μ equals $m - (m-1)\frac{C_k}{D_k}$, since the interval is of length D_k).
- Define the start of the interval of interest to be the *earliest* $t \le t_a$ such that the work that the scheduling algorithm needs to complete over $[t, t_d)$ is greater than $(t_d - t)\mu$. Let $t_o(\mu)$ denote this start of the interval of interest.
- Consider a carry-in job of some task τ_i, that arrives at time-instant $(t_o(\mu) - \phi_i)$.
  - By definition of $t_o(\mu)$ as the earliest t for which work to be done by the scheduling algorithm over $[t, t_d)$ exceeds $(t_d - t)\mu$, it follows that the work to be done by the scheduling algorithm over $[t_o(\mu) - \phi_i, t_d)$ is no larger than $(t_d - t_o(\mu) + \phi_i)\mu$.
  - Therefore, the amount of work that was actually done by the scheduling algorithm over $[t_o(\mu) - \phi_i, t_o(\mu))$ can be bounded from above. (Intuitively, the amount of work to be done by time-instant t_d, normalized by the time remaining until t_d, has *increased* over $[t_o(\mu) - \phi_i, t_o(\mu))$; this allows us to compute an upper bound on the amount of work that was actually completed during this interval.)
  - Using this bound on the total amount of work that was done during $t_o(\mu) - \phi_i, t_o(\mu))$, we can obtain an upper bound on the duration over $[t_o(\mu) - \phi_i, t_o(\mu))$ during which all m processors could have simultaneously been busy.
  - The carry-in job of τ_i must have been executing during those instants $[t_o(\mu) - \phi_i, t_o(\mu))$ at which at least one processor is idle; this yields a lower bound on the amount of execution that this carry-in job *must* have received prior to the interval of interest.
- Only the portion of this carry-in job of τ_i that was not thus shown to *have* to execute over $[t_o(\mu) - \phi_i, t_o(\mu))$ contributes to the workload over the interval of interest.

This is a very clever and sophisticated technique, based on a deep understanding of the behavior of multiprocessing, for obtaining bounds on carry-in that are better than the naive bounds that were previously used. The [BAK] test, as well as several subsequent tests, have all exploited this technique for obtaining better upper bounds W_U upon the computational workload that must be executed within the interval of interest.

16.2 Details

We now present the detailed derivation of the [BAK] test. We will restrict our attention to constrained-deadline sporadic task systems; however, the test has been extended to arbitrary-deadline task systems using a technique similar to that used in extending the [BCL] test.

Identifying an Unschedulable Collection of Jobs Consider any legal sequence of job requests of task system τ, on which EDF misses a deadline. Suppose that a job of task τ_k is the one to first miss a deadline, and that this deadline miss occurs at time-instant t_d (see Fig. 16.1). Let t_a denote the arrival time of this job that misses its deadline: $t_a = t_d - D_k$.

Determining the Interval of Interest Discard from the legal sequence of job requests all jobs with deadline $> t_d$, and consider the EDF schedule of the remaining job requests. Since later-deadline jobs have no effect on the scheduling of earlier-deadline ones under preemptive EDF, it follows that a deadline miss of τ_k occurs at time-instant t_d (and this is the earliest deadline miss), in this new EDF schedule.

We introduce some notations now.

For any time-instant $t \le t_d$, let $W(t)$ denote the cumulative execution requirement of all the jobs in the collection of jobs, *minus* the total amount of execution completed by the EDF schedule prior to time-instant t. (Thus, $W(t)$ denotes the amount of work remaining to be done at time-instant t.)

Observe that the computational demand over the interval of interest, which had been referred to as W in the generic description of this technique, is equal to $W(t_s)$, where t_s denotes the (not yet defined) start of the interval of interest.

Let $\Omega(t)$ denote $W(t)$ normalized by the interval-length:

$$\Omega(t) \overset{\text{def}}{=} W(t)/(t_d - t). \tag{16.1}$$

Observe that

$$W(0) \le \sum_{\tau_i \in \tau} \text{DBF}(\tau_i, t_d)$$

This follows from the definition of the demand bound function, and the observation that all the work contributing to $W(0)$ has deadline at or before t_d. Thus,

$$\Omega(0) \le \frac{\sum_{\tau_i \in \tau} \text{DBF}(\tau_i, t_d)}{t_d} \le \text{LOAD}(\tau)\ . \tag{16.2}$$

Since τ_k's job receives strictly less than C_k units of execution over $[t_a, t_d)$, all m processors must be executing tasks other than τ_k's job for a total duration greater than $((t_d - t_a) - C_k)$ over this interval. Hence it must be the case that $W(t_a) > (t_d - t_a - C_k)m + C_k$; equivalently,

$$\Omega(t_a) > \frac{(t_d - t_a - C_k)m + C_k}{t_d - t_a}$$

$$= m - (m-1)\frac{C_k}{t_d - t_a}$$
$$\geq m - (m-1)\frac{C_k}{D_k}$$
$$= m - (m-1)\text{dens}_k$$

Let

$$\mu_k \overset{\text{def}}{=} m - (m-1)\text{dens}_k \tag{16.3}$$

We thus have $\Omega(t_a) > \mu_k$; previously (Inequality 16.2 above), we had seen that $\Omega(0) \leq \text{LOAD}(\tau)$. Assuming that $\text{LOAD}(\tau) < \mu_k$, we can therefore define, for each $\mu \in [\text{LOAD}(\tau), \mu_k]$, an earliest time-instant t at which $\Omega(t) \geq \mu$.

A further definition: Let us define $\mu(\tau)$ to be $\max_{\tau_k \in \tau}\{\mu_k\}$; equivalently,

$$\mu(\tau) \overset{\text{def}}{=} m - (m-1)\text{dens}_{\max}(\tau) \tag{16.4}$$

Rather than having just one interval of interest as in the [BCL] test, the [BAK] test considers *multiple intervals of interest*. Informally speaking, the different intervals are defined by different values of μ in the range $[\text{LOAD}(\tau), \mu_k]$—for each such value of μ, a different interval of interest $[t_o(\mu), d_k)$ is defined. We will later optimize over all these intervals of interest, choosing the "best" one to come up with our final schedulability test.

For any $\mu \in [\text{LOAD}(\tau), \mu_k]$, let $t_o(\mu)$ denote the smallest value of $t \leq t_a$ such that $\Omega(t) \geq \mu$. Let $\Delta(\mu) \overset{\text{def}}{=} t_d - t_o(\mu)$.

We now derive a lower bound W_L on the computational demand $W = W(t_o(\mu))$ over the interval of interest.

By definition of $t_o(\mu)$, $\Omega(t_o(\mu)) \geq \mu$. Hence

$$W = W(t_o(\mu)) = \Omega(t_o(\mu)) \times (t_d - t_o(\mu)) \geq \mu \times \Delta(\mu)$$
$$\Rightarrow W_L \overset{\text{def}}{=} \mu \times \Delta(\mu) \tag{16.5}$$

Computing an Upper Bound on Demand For each τ_i, the [BAK] test derives an upper bound $W_{i,k}$ on the amount of work that jobs of τ_i contribute to W. The upper bound W_U on the demand W over the interval of interest is then given by

$$W_U = \sum_{i=1}^{n} W_{i,k} \ .$$

Computing $W_{i,k}$. The [BAK] test is able to *bound* the amount of carry-in by a job of τ_i into the interval of interest, based on the property that the interval is defined to have a minimum load.

This is because, as we had briefly explained in Sect. 16.1 above, some of the carry-in job must have executed prior to the start of the interval.

We now elaborate upon the technique, outlined in Sect. 16.1 above, for bounding carry-in. Suppose that there is carry-in by task τ_i, for some $i \neq k$.

1. Assume that the job carrying in arrives at time-instant t_i, which is ϕ_i time units prior to time-instant $t_o(\mu)$ (i.e., $t_i = t_o(\mu) - \phi_i$—see Fig. 16.1).
2. Suppose that τ_i's job executed for y time-units over $[t_i, t_o(\mu))$ (leaving the rest of its execution as carry-in). All m processors must have been busy executing other jobs for $(\phi_i - y)$ time units during this interval; hence, the amount of execution that occurs over $[\phi_i, t_o(\mu))$ is at least $(m(\phi_i - y) + y)$.
3. By using this work bound along with the facts that $W(t_o(\mu), t_d) \geq \mu\Delta(\mu)$ while $W(t_i, t_d) < \mu(\Delta(\mu) + \phi_i)$, we obtain

$$\begin{aligned} & m(\phi_i - y_i) + y_i < \mu\phi_i \\ & \Rightarrow m\phi_i - (m-1)y_i < (m - (m-1)\text{dens}_k)\phi_i \\ & \Leftrightarrow y_i > \phi_i \text{dens}_k \end{aligned} \tag{16.6}$$

4. Since there are $\left\lfloor \frac{\Delta(\mu)+\phi_i-D_i}{T_i} + 1 \right\rfloor$ jobs of τ_i in the interval of length $\Delta(\mu) + \phi_i$, it follows that $W_{i,k}$, the total execution needed by τ_i over $[t_o, t_d)$ is bounded as follows

$$\begin{aligned} W_{i,k} &\leq \left\lfloor \frac{\Delta(\mu) + \phi_i - D_i}{T_i} + 1 \right\rfloor C_i - \phi_i \text{dens}_k \\ &\leq \frac{\Delta(\mu) + \phi_i + T_i - D_i}{T_i} C_i - \phi_i \text{dens}_k \\ &\leq (\Delta(\mu) + T_i - D_i)U_i + \phi_i(U_i - \text{dens}_k) \end{aligned} \tag{16.7}$$

5. If $(U_i \leq \text{dens}_k)$, the quantity in (16.7) above is $\leq (\Delta(\mu) + T_i - D_i)U_i$.
6. If $(U_i \geq \text{dens}_k)$, we can use the fact that $\phi_i \leq D_i$ to derive that the quantity in (16.7) above is $\leq (\Delta(\mu) + T_i)U_i - D_i \text{dens}_k$.
7. Hence when τ_i has a carry-in, the total execution needed by τ_i over $[t_o, t_d)$ is bounded as follows:

$$W_{i,k} \leq (\Delta(\mu) + T_i)U_i - D_i \min(\text{dens}_k, U_i)\,. \tag{16.8}$$

If there is no carry-in for τ_i, then the total execution needed by τ_i over $[t_o, t_d)$ is at most $(\Delta(\mu) + T_i - D_i)U_i$, which is clearly no greater than the quantity in (16.8) above.

We therefore have the following result: for all tasks τ_i,

$$W_{i,k} \leq (\Delta(\mu) + T_i)U_i - D_i \min(\text{dens}_k, U_i)\,.$$

The desired upper bound W_U on the demand W (which, as we had pointed out, is equal to $W(t_o(\mu))$) is thus

$$W_U \overset{\text{def}}{=} \sum_{\tau_i \in \tau} \Big((\Delta(\mu) + T_i)U_i - D_i \min(\text{dens}_k, U_i) \Big)\,. \tag{16.9}$$

Determining a Sufficient Schedulability Condition A necessary condition for τ_k to miss a deadline in an EDF schedule of τ is that

$$W_U > W_L$$

where W_U is as given in Eq. 16.9 and W_L is as given in Eq. 16.5. Now, this must be true for each $\mu \in [\textsc{load}(\tau), \mu_k]$:

$$\exists\tau_k : \tau_k \in \tau : \forall\mu : \textsc{load}(\tau) \le \mu \le \mu_k :$$
$$\sum_{\tau_i\in\tau}\big((\varDelta(\mu) + T_i)U_i - D_i \min(\lambda_k, U_i)\big) \ge \mu \times \varDelta(\mu)\,. \tag{16.10}$$

Taking the contrapositive (and dividing throughout by $\varDelta(\mu)$), we obtain a sufficient schedulability condition:

$$\forall\tau_k : \tau_k \in \tau : \exists\mu : \textsc{load}(\tau) \le \mu \le \mu_k :$$
$$\sum_{\tau_i\in\tau}\left(\left(1 + \frac{T_i}{\varDelta(\mu)}\right)U_i - \frac{D_i}{\varDelta(\mu)}\min(\lambda_k, U_i)\right) < \mu \tag{16.11}$$

Next, [BAK] uses the fact that $\varDelta(\mu)$ is, by definition, $\ge D_k$ to obtain the following sufficient schedulability condition[2]:

$$\forall\tau_k : \tau_k \in \tau : \exists\mu : \textsc{load}(\tau) \le \mu \le \mu_k :$$
$$\sum_{\tau_i\in\tau}\left(\left(1 + \frac{T_i}{D_k}\right)U_i - \frac{D_i}{D_k}\min(\lambda_k, U_i)\right) < \mu \tag{16.12}$$

The [BAK] test is stated in the following theorem.

Theorem 16.1 *Task system τ is global EDF schedulable on m unit-capacity processors if for each $\tau_k \in \tau$ there is a $\lambda_k \ge \frac{C_k}{D_k}$ such that*

$$\sum_{\tau_i\in\tau}\beta_{k,i} < m - (m-1)\lambda_k\,,$$

where $\beta_{k,i}$ is defined as follows:

$$\beta_{k,i} \overset{def}{=} (1 + T_i)U_i - \frac{D_i}{D_k}\min(\lambda_k, U_i)\,. \tag{16.13}$$

§. We need to only consider values of λ_k in the set

$$\left\{\frac{C_k}{D_k}\right\} \bigcup \left\{U_i \mid U_i \ge \frac{C_k}{D_k}\right\}\,.$$

[2] Observe that $\varDelta(\mu) \le D_k$ means that the LHS of Inequality 16.12 is no larger than the LHS of Inequality 16.11; consequently, Inequality 16.12 does indeed imply Inequality 16.11.

16.3 A Quantitative Bound

By computing the upper bound on demand somewhat differently than was done above, a modified version of the [BAK] test was obtained [35], for which a speedup bound could be determined. We describe this modified test below.

The first two steps of the general strategy for global schedulability analysis that was enumerated in Sect. 14.1—*identifying an unschedulable collection of jobs* and *determining the interval of interest*—proceed in exactly the same manner as in the [BAK] test as described above. We describe the remainder of the modified test below.

Computing an Upper Bound on Demand Recall that $W(t_o(\mu))$ denotes the amount of work that the EDF schedule needs (but fails) to execute over $[t_o(\mu), t_d)$. This work in $W(t_o(\mu))$ arises from two sources: (i) those jobs that arrived at or after $t_o(\mu)$, and (ii) the carry-in jobs, that arrived prior to $t_o(\mu)$ but have not completed execution in the EDF schedule by time-instant $t_o(\mu)$.

The total work contributed by those jobs that arrived at or after $t_o(\mu)$ can be bounded from above by using the DBF; in the next two lemmas, we will bound the amount of work that can be contributed by the carry-in jobs. We do this by bounding the number of tasks that may have carry-in jobs (Lemma 16.1), and the amount of work that each such task may contribute (Lemma 16.2). The product of these bounds yields an upper bound on the total amount of carry-in work.

Lemma 16.1 *The number of tasks that have carry-in jobs is* $\leq \lceil \mu \rceil - 1$.

Proof Let ϵ denote an arbitrarily small positive number. By definition of the instant $t_o(\mu)$, $\Omega(t_o(\mu) - \epsilon) < \mu$ while $\Omega(t_o(\mu)) \geq \mu$; consequently, strictly fewer than μ jobs must have been executing at time-instant $t_o(\mu)$. And since $\mu \leq m$ (as can be seen from Eq. 16.3 above), it follows that some processor was idled over $[t_o(\mu) - \epsilon, t_o(\mu))$, implying that all tasks with jobs awaiting execution at this instant would have been executing. This allows us to conclude that there are strictly fewer than μ—equivalently, at most $(\lceil \mu \rceil - 1)$—tasks with carry-in jobs. □

Lemma 16.2 *Each carry-in job has* $< \Delta(\mu) \times dens_{max}(\tau)$ *remaining execution requirement at time-instant* t_o.

Proof Let us consider a carry-in job of some task τ_i that arrives at time-instant $t_i < t_o(\mu)$ (see Fig. 16.1) and has, by definition of carry-in job, not completed execution by time-instant $t_o(\mu)$. As specified in Fig. 16.1, $\phi_i \overset{\text{def}}{=} t_o(\mu) - t_i$.

By definition of $t_o(\mu)$ as the earliest time-instant $t \leq t_a$ at which $\Omega(t) \geq \mu$, it must be the case that $\Omega(t_i) < \mu$. That is,

$$W(t_i) < \mu(\Delta(\mu) + \phi_i) \tag{16.14}$$

On the other hand, $\Omega(t_o(\mu)) \geq \mu$, meaning that

$$W(t_o(\mu)) \geq \mu\Delta(\mu) \tag{16.15}$$

From Inequalities 16.14 and 16.15 and the definition of $W(\cdot)$, it follows that the amount of work done in the EDF schedule over $[t_i, t_o(\mu))$ is *less than* $\mu\phi_i$.

Let y_i denote the amount of time over this interval $[t_i, t_o(\mu))$, during which some job of τ_i is executing. All m processors must be executing jobs from tasks other than τ_i for the remaining $(\phi_i - y_i)$ time units, implying that the total amount of work done in the EDF schedule over $[t_i, t_o(\mu))$ is *at least* $m(\phi_i - y_i) + y_i$. From these upper and lower bounds, we have

$$\begin{aligned}
& m(\phi_i - y_i) + y_i < \mu\phi_i \\
& \Rightarrow m(\phi_i - y_i) + y_i < \mu(\tau)\phi_i \quad \text{[Since by Eq. 16.3), } \mu \le \mu(\tau)] \\
& \Leftrightarrow m\phi_i - (m-1)y_i < (m - (m-1)\text{dens}_{\max}(\tau))\phi_i \\
& \Leftrightarrow y_i > \phi_i \text{dens}_{\max}(\tau)
\end{aligned} \tag{16.16}$$

As argued above, the total amount of work contributed to $W(t_o(\mu))$ by all the carry-in jobs of τ_i is bounded from above by $\text{DBF}(\tau_i, \phi_i + \Delta(\mu))$ minus the amount of execution received by jobs of τ_i over $[t_i, t_o(\mu))$. This is in turn bounded from above by

$$\begin{aligned}
& \text{DBF}(\tau_i, \phi_i + \Delta(\mu)) - y_i \\
& < \text{DBF}(\tau_i, \phi_i + \Delta(\mu)) - \phi_i \,\text{dens}_{\max}(\tau) \\
& \qquad \text{(by Inequality 16.16 above)} \\
& \le (\phi_i + \Delta(\mu))\text{dens}_i - \phi_i \,\text{dens}_{\max}(\tau) \text{ (from Lemma 10.1)} \\
& \le (\phi_i + \Delta(\mu))\text{dens}_{\max}(\tau) - \phi_i \,\text{dens}_{\max}(\tau) \\
& = \Delta(\mu) \,\text{dens}_{\max}(\tau)
\end{aligned}$$

as claimed in the lemma. □

We can now compute the upper bound W_U on the workload over the interval of interest $[t_o(\mu), t_d)$.

- First, there are the carry-in jobs: by Lemmas 16.1 and 16.2, there are at most $(\lceil \mu(\tau) \rceil - 1)$ of them, each contributing at most $\Delta(\mu)\text{dens}_{\max}(\tau)$ units of work. Therefore their total contribution to $W(t_o)$ is at most $(\lceil \mu(\tau) \rceil - 1)\Delta(\mu)\text{dens}_{\max}(\tau)$.
- All other jobs that contribute to $W(t_o)$ arrive in, and have their deadlines within the $\Delta(\mu)$-sized interval$[t_o, t_d)$. Their total execution requirement is therefore bounded from above by $\Delta(\mu)\text{LOAD}(\tau)$.

We therefore have the following upper bound on the amount of work over the interval of interest:

$$\Delta(\mu)\text{LOAD}(\tau) + (\lceil \mu(\tau) \rceil - 1)\Delta(\mu)\text{dens}_{\max}(\tau) \tag{16.17}$$

Determining a Sufficient Schedulability Condition Based upon Lemmas 16.1 and 16.2, we obtain the following:

Theorem 16.2 *Constrained-deadline sporadic task system τ is* EDF-*schedulable upon a platform comprised of m unit-speed processors provided*

$$\text{LOAD}(\tau) \le \mu(\tau) - (\lceil \mu(\tau) \rceil - 1)\textit{dens}_{\textit{max}}(\tau), \tag{16.18}$$

where $\mu(\tau)$ is as defined in Eq. 16.4.

Proof The proof is by contradiction: We obtain necessary conditions for the scenario above—when τ_k's job misses its deadline at t_d—to occur. Negating these conditions yields a sufficient condition for EDF schedulability.

By Eq 16.17 above, it follows that:

$$W(t_o(\mu(\tau))) < \Delta(\mu(\tau))\textsc{load}(\tau) + (\lceil\mu(\tau)\rceil - 1)\Delta(\mu(\tau))\text{dens}_{\max}(\tau) \qquad (16.19)$$

Since $\Omega(t_o(\mu(\tau)))\overset{\text{def}}{=}W(t_o(\mu(\tau)))/(t_d - t_o(\mu(\tau))) = W(t_o)/\Delta(\mu(\tau))$, we have

$$\Omega(t_o(\mu(\tau))) < \textsc{load}(\tau) + (\lceil\mu(\tau)\rceil - 1)\text{dens}_{\max}(\tau)$$

Since by the definition of $t_o(\mu(\tau))$ it is required that $\Omega(t_o(\mu(\tau))) \geq \mu(\tau)$, we must have

$$\textsc{load}(\tau) + (\lceil\mu(\tau)\rceil - 1)\text{dens}_{\max}(\tau) > \mu(\tau)$$

as a necessary condition to miss a deadline. By negating this, we get that

$$\textsc{load}(\tau) + (\lceil\mu(\tau)\rceil - 1)\text{dens}_{\max}(\tau) \leq \mu(\tau)$$

is sufficient for EDF-schedulability. The theorem follows. □

Theorem 16.3 *Any sporadic task system that is feasible upon a multiprocessor platform comprised of m speed-$\frac{3-\sqrt{5}}{2}$ processors is determined to be EDF-schedulable on m unit-capacity processors by the EDF-schedulability test of Theorem 16.2.*

Proof Suppose that τ is feasible upon m speed-x processors. For all values of $x \leq (3-\sqrt{5})/2$, we will show below that the test of Theorem 16.2 determines that τ is EDF-schedulable upon m unit-speed processors.

Any sporadic task system τ that is feasible in m speed-x processors satisfies

$$\text{dens}_{\max}(\tau) \leq x \ \text{ and } \ \textsc{load}(\tau) \leq mx$$

For τ to be deemed EDF-schedulable on m unit-capacity processors by the EDF-schedulability test of Theorem 16.2, it must be the case that

$$\begin{aligned}
&\textsc{load}(\tau) \leq \mu(\tau) - (\lceil\mu(\tau)\rceil - 1)\text{dens}_{\max}(\tau)\\
\Leftarrow\ &\text{(Since } \lceil x\rceil - 1 \leq x \text{ for any } x)\\
&\quad \textsc{load}(\tau) \leq \mu(\tau) - \mu(\tau)\text{dens}_{\max}(\tau)\\
\Leftrightarrow\ &\textsc{load}(\tau) \leq \mu(\tau)(1 - \text{dens}_{\max}(\tau))\\
\Leftrightarrow\ &\textsc{load}(\tau) \leq (m - (m-1)\text{dens}_{\max}(\tau))(1 - \text{dens}_{\max}(\tau))\\
\Leftarrow\ &mx \leq (m - (m-1)x)(1-x)\\
\Leftrightarrow\ &mx \leq (m(1-x) + x)(1-x)
\end{aligned}$$

$$
\begin{aligned}
&\Leftarrow mx \le m(1-x)(1-x)\\
&\Leftrightarrow x^2 - 3x + 1 \ge 0\\
&\Leftarrow x \le \frac{3-\sqrt{5}}{2}
\end{aligned}
$$

We have thus shown that τ being feasible upon m speed-$(3-\sqrt{5})/2$ processors implies its EDF-schedulability upon m unit-speed processors. □

Observing that

$$\frac{1}{(3-\sqrt{5})/2} = \frac{2}{3-\sqrt{5}} = \frac{3+\sqrt{5}}{2},$$

we have the following corollary:

Corollary 16.1 *The processor speedup factor of the* EDF*-schedulability test of Theorem 16.2 is no greater than* $\left(\frac{3+\sqrt{5}}{2}\right)$.

A somewhat superior EDF schedulability condition that is a bit more complex (to both state and prove) than the one presented in Theorem 16.2 above was derived in [27]; we present this theorem without proof below:

Theorem 16.4 *(from [27]) Constrained-deadline sporadic task system τ is* EDF*-schedulable upon a platform comprised of m unit-speed processors provided*

$$
\begin{aligned}
\textsc{load}(\tau) \le \max\big\{&\mu(\tau) - (\lceil\mu(\tau)\rceil - 1)dens_{max}(\tau), (\lceil\mu(\tau)\rceil - 1)\\
&- (\lceil\mu(\tau)\rceil - 2)dens_{max}(\tau)\big\}
\end{aligned}
$$

where $\mu(\tau)$ is as defined in Eq. 16.4.

Sources

In a series of papers [17–19], Ted Baker laid out the framework that formed the basis of many of the subsequently developed sufficient schedulability tests—the techniques and results that we have described in Sects. 16.1–16.2 were first proposed in those papers. The results in Sect. 16.3 build upon this framework to derive the schedulability tests of Theorem 16.2 and Theorem 16.4, for which quantitative speedup bounds may be derived. Theorem 16.2 was derived in [35]; Theorem 16.4 is from [27].

Chapter 17
Response Time Analysis: The [RTA] Test

We now describe a refinement to the basic [BCL] procedure that was proposed by Bertogna and Cirinei [58], that uses response times of tasks to further reduce potentially the workload of interfering tasks in the considered interval of interest. Since these response times are of course not known prior to performing schedulability analysis, the procedure takes on an iterative flavor—maximum (or minimum) values are estimated for response times of all tasks, and these estimated values are then used to iteratively decrease (or increase) the estimations. The details are described in Sect. 17.1 below. This test, which aggregates clever ideas from many prior tests into an integrated framework, appears to perform the best in schedulability experiments upon randomly generated workloads.

17.1 Response Time Analysis [RTA]

One of the main sources of pessimism in the global schedulability tests described above is related to the difficulties in obtaining tight upper bounds upon the carry-in contribution into the window of interest. The test presented in this section allows for a further reduction of the carry-in contributions of individual tasks, by computing a response time upper bound for each task. The analysis is similar to that of [BCL], using a smaller interval of interest.

Before deriving the analysis, let us first start by deriving some useful results regarding the interference that a task can suffer due to the simultaneous execution of other tasks. (It may be helpful to review the definitions and notation concerning interference that were introduced in Sect. 14.4.) These results are applicable to any system of sporadic tasks that are scheduled with a work-conserving policy.

Lemma 17.1 *For any $x \geq 0$, $I_k \geq x$ iff $\sum_{i \neq k} \min\left(I_{i,k}, x\right) \geq mx$.*

Proof *Only If.* Let ξ denote the number of tasks for which $I_{i,k} \geq x$ holds. If $\xi > m$, then $\sum_{i \neq k} \min(I_{i,k}, x) \geq \xi x > mx$. Otherwise $(m - \xi) \geq 0$ and, using Lemma 14.1 and Eq. 14.1, we get

S. Baruah et al., *Multiprocessor Scheduling for Real-Time Systems,*
Embedded Systems, DOI 10.1007/978-3-319-08696-5_17

$$\sum_{i\neq k}\min\left(I_{i,k},x\right)=\xi x+\sum_{i:I_{i,k}<x}I_{i,k}=\xi x+mI_k-\sum_{i:I_{i,k}\geq x}I_{i,k}$$
$$\geq \xi x+mI_k-\xi I_k=\xi x+(m-\xi)I_k\geq \xi x+(m-\xi)x=mx.$$

If. Note that if $\sum_i \min\left(I_{i,k},x\right)\geq mx$, it follows

$$I_k=\sum_{i\neq k}\frac{I_{i,k}}{m}\geq\sum_{i\neq k}\frac{\min\left(I_{i,k},x\right)}{m}\geq\frac{mx}{m}=x. \qquad \square$$

The following part of the analysis will consider the particular instance of task τ_k that is subject to the maximum possible interference. The finishing time of such an instance coincides with the response time of τ_k, which we denote as R_k. The interval of interest is then defined as the window between the release of the instance, denoted here as r_k, and its finishing time: $[r_k, r_k + R_k]$.

We now derive a result that will be useful to compute the worst-case response time of each task. For simplifying the schedulability condition, we assume all task parameters to be in an integer time domain[1].

Theorem 17.1 *A task τ_k has a response time bounded by R_k^{ub} if*

$$\sum_{i\neq k}\min\left(I_{i,k},R_k^{\mathrm{ub}}-C_k+1\right)<m(R_k^{\mathrm{ub}}-C_k+1)$$

Proof If the inequality holds for τ_k, from Lemma 17.1 we have

$$I_k<(R_k^{\mathrm{ub}}-C_k+1).$$

As we assumed an integer time-domain, each instance of τ_k will be therefore interfered for at most $R_k^{\mathrm{ub}}-C_k$ time units. From the definition of interference, it follows that every job of τ_k will be completed after at most R_k^{ub} time-units from its release. $\square$

Note that the above theorem formally proves that when computing an upper bound on the response time of a task τ_k, no other task can interfere more than (R_k-C_k+1) time-units.

To effectively use Theorem 17.1, it would be necessary to compute the interfering terms $I_{i,k}$. As it is hard to compute these terms, we adopt the following upper bound on the interfering workload, which follows directly from the definition of interference.

Observation 17.1 The interference $I_{i,k}$ that a task τ_i causes on a task τ_k cannot be greater than the effective workload of τ_i in the window of interest:

$$\forall i,k: \quad I_{i,k}\leq W_i(r_k,r_k+R_k)\leq R_k. \tag{17.1}$$

[1] A similar relation can be derived for systems without an integer time-domain, using a slightly more complex condition.

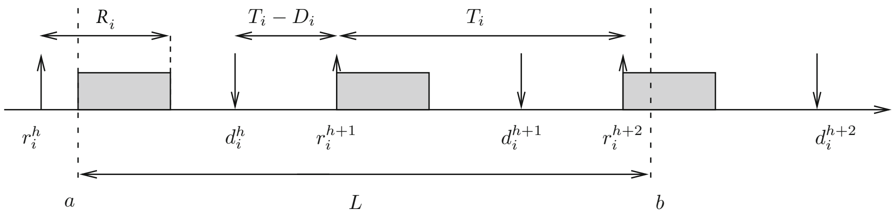


Fig. 17.1 Densest possible packing of jobs of τ_i, when R_i^{ub} is a safe upper bound on the response time of τ_i

17.1.1 [RTA] for Work-Conserving Schedulers

As exactly computing the effective workload executed by each interfering task within the interval of interest is hard, we seek for an upper bound of the workload that each task τ_i can execute in a generic window of length R_k. To do so, we seek to find the densest possible packing of jobs that can be generated by a legal schedule. Note that we are not relying on any particular scheduling policy, but are providing a bound that is valid for any work-conserving scheduler.

With these premises and as long as there are no deadline misses, a bound on the workload of a task τ_i in a generic interval $[a, b)$ can be determined by considering a situation in which the carry-in job begins executing right at the beginning of the interval of interest, with a response time equal to the worst-case response time of τ_i (therefore $r_i^{\varepsilon} = a + C_i - R_i$) and every other instance of τ_i is executed as soon as possible. The situation is represented in Fig. 17.1.

As by definition of R_i, each job J_i^j executes only in $[r_i^j, r_i^j + R_i)$ and for at most C_i time units, it immediately follows that the depicted situation provides the highest possible amount of execution in interval $[a, b)$: By "sliding" the interval backwards, the carry-in cannot increase while the carry-out decreases. Similarly, if the interval were slid forward, the carry-in decreases, while the carry-out can increase by at most the same amount.

Using Fig. 17.1, we now compute the effective workload of task τ_i in an interval $[a, b)$ of length L in the situation described above.

Note that the first job of τ_i after the carry-in is released at time $a + C_i - R_i + T_i$. The next jobs are then released periodically every T_i time units. Therefore the number of body jobs within the interval is

$$\left\lfloor \frac{(a+L) - (a + C_i - R_i + T_i)}{T_i} \right\rfloor = \left\lfloor \frac{L + R_i - C_i - T_i}{T_i} \right\rfloor .$$

Adding the (full) carry-in contribution, the number $N_i(L)$ of jobs of τ_i that contribute with an entire worst-case execution time (WCET) to the workload in an interval of length L is given by $\left(\left\lfloor \frac{L+R_i-C_i-T_i)}{T_i} \right\rfloor + 1 \right)$. Hence

$$N_i(L) = \left\lfloor \frac{L + R_i - C_i}{T_i} \right\rfloor . \tag{17.2}$$

The contribution of the carry-out job can then be bounded by

$$\min(C_i, (L + R_i - C_i) \bmod T_i).$$

A bound on the workload of a task τ_i in a generic interval of length L is then:

$$\widehat{W}_i(L) = \left\lfloor \frac{L + R_i - C_i}{T_i} \right\rfloor C_i + \min(C_i, (L + R_i - C_i) \bmod T_i). \tag{17.3}$$

We are now ready to state a first result valid for any work-conserving scheduler.

Theorem 17.2 *An upper bound on the response time of a task τ_k in a multiprocessor system scheduled with a work-conserving algorithm can be derived by the fixed point iteration on the value R_k^{ub} of the following expression, starting with $R_k^{\text{ub}} = C_k$:*

$$R_k^{\text{ub}} \leftarrow C_k + \left\lfloor \frac{1}{m} \sum_{i \neq k} \min(\widehat{W}_i(R_k^{\text{ub}}), R_k^{\text{ub}} - C_k + 1) \right\rfloor, \tag{17.4}$$

where $\widehat{W}_i(R_k)$ is given by Eq. 17.3.

Proof The proof is by contradiction. Suppose the iteration ends with a value $R_k^{\text{ub}} \leq D_k$, but the response time of τ_k is higher than R_k^{ub}. Since the iteration ends, it is

$$R_k^{\text{ub}} = C_k + \left\lfloor \frac{1}{m} \sum_{i \neq k} \min(\widehat{W}_i(R_k^{\text{ub}}), R_k^{\text{ub}} - C_k + 1) \right\rfloor$$

For Eq. (17.1) and $W_i(r_k, r_k + R_k) \leq \widehat{W}_i(R_k^{\text{ub}})$,

$$I_{i,k} \leq \widehat{W}_i(R_k^{\text{ub}}).$$

Therefore, as long as $R_k^{\text{ub}} \leq D_k$,

$$R_k^{\text{ub}} \geq C_k + \left\lfloor \frac{1}{m} \sum_{i \neq k} \min(I_{i,k}, R_k^{\text{ub}} - C_k + 1) \right\rfloor.$$

Since, by hypothesis, the response time of τ_k is higher than R_k^{ub}, the inverse of Theorem 17.1 gives

$$\sum_{i \neq k} \min\left(I_{i,k}, R_k^{\text{ub}} - C_k + 1\right) \geq m(R_k^{\text{ub}} - C_k + 1).$$

Therefore,

$$R_k^{\text{ub}} \geq C_k + \left\lfloor \frac{1}{m} m(R_k^{\text{ub}} - C_k + 1) \right\rfloor = R_k^{\text{ub}} + 1$$

reaching a contradiction.

It remains to show that the iteration converges in a finite amount of time. This is assured by the integer time assumption, and by noting that the assignment of Eq. (17.4) is a monotonically nondecreasing function of R_k^{ub}. □

A schedulability test could be performed by repeating the iteration described above for every task $\tau_k \in \tau$. If every iteration ends before the corresponding deadline value, then the task set is schedulable. However, the problem in applying this schedulability test is that, when computing $\widehat{W}_i(R_k^{\text{ub}})$ with Eq. 17.3, the response times R_i of all interfering tasks τ_i should be known.

To sidestep this problem, we will compute Eq. 17.3 using the response time upper bounds R_i^{ub}, instead of R_i. In this respect, two possible strategies can be adopted.

With the first strategy, all response time upper bounds are initially set to $R_i^{\text{ub}} = D_i, \forall \tau_i$. Then, for each considered task τ_k, Eq. (17.4) is used to compute a new response time bound. If the computed bound is $\leq D_k$, R_k^{ub} is replaced with the new value. Otherwise, the task is temporarily set aside, continuing to the next task. If for each task τ_k, a bound $\leq D_k$ is computed, the task system is deemed schedulable. Otherwise, another round of updates is performed for all tasks using the new (smaller) response time upper bounds. When no further update is possible for any of the n tasks, and at least one task has a bound exceeding the corresponding deadline, the test fails.

Basically, when the computed response time bound of a task is $> D_k$, the test does not immediately fail, but it tries decreasing the bounds R_i^{ub} of the interfering tasks τ_i. As $\widehat{W}_i(L)$ is a nonincreasing function of the response times of the interfering tasks, using smaller response time bounds R_i^{ub} in Eq. 17.3 may result in a smaller workload $\widehat{W}_i(L)$ imposed on τ_k's scheduling window. Equation (17.4) may then return a value $R_k^{\text{ub}} \leq D_k$. The convergence of the algorithm is guaranteed by the monotonicity of $\widehat{W}_i(L), \forall \tau_i$ with respect to L and to the response times of the interfering tasks.

A more formal version of the schedulability algorithm is shown in Fig. 17.2. Procedure ComputeResponse(τ_k) returns the response time upper bound R_k^{ub} computed using Theorem 17.2.

The second strategy for updating the response time upper bounds is initially setting all response time upper bounds $R_i^{\text{ub}} = C_i, \forall \tau_i$, i.e., to their minimum possible value. Then, for each task τ_k, Eq. (17.4) is used to compute a new tentative response time upper bound R_k^{ub}. If the computed bound is $> D_k$, the test fails, as there is no possibility for further improvements. Otherwise, if the new bound is larger than the previous one, R_k^{ub} is updated to the new value, passing to the next task. When an update took place, another round of updates needs to be performed for all n tasks. When no update takes place during a whole round, the test deems the task set schedulable.

This second strategy dominates the previous one, as it allows finding tighter worst-case response time upper bounds. It indeed considers a smaller (tentative) interval of interest, resulting in a possibly smaller interfering workload. However, it may require a larger run-time complexity for schedulable task systems. This is because every time there is an update in the response time bound of one of the tasks, another round of updates is triggered for the whole task system. This is not the case with the first strategy, which may deem a task set schedulable as soon as for each tasks τ_k a bound $R_k^{\text{ub}} \leq D_k$ is computed.

This second strategy is formalized in Fig. 17.3.

```
SCHEDCHECK($\tau$)
    ▷ Check the schedulability of a task set $\tau$.
    ▷ All response-time upper bounds are initially set to $R_k^{\text{ub}} = D_k, \forall k$;
    ▷ Updated = true.
1   while (Updated == true) do
2       Feasible ← true
3       Updated ← false
4       for $k = 0$ to $n$ do
            ▷ Try to update $R_k^{\text{ub}}$ for task $\tau_k$
5           NewBound ← COMPUTERESPONSE($\tau_k$)
6           if (NewBound > $D_k$) Feasible ← false
7           if (NewBound < $R_k^{\text{ub}}$)
8               { $R_k^{\text{ub}}$ ← NewBound
9               Updated ← true }
        end for
        ▷ When no task is infeasible, declare success
10      if (Feasible == true) return true
    done
    ▷ When no $R_k^{\text{ub}}$ can be updated anymore, stop the test
11  return false
```

Fig. 17.2 Response time analysis: first strategy

```
SCHEDCHECK($\tau$)
    ▷ Check the schedulability of a task set $\tau$.
    ▷ All response-time upper bounds are initially set to $R_k^{\text{ub}} = C_k, \forall k$;
    ▷ Updated = true.
1   while (Updated == true) do
2       Updated ← false
3       for $k = 0$ to $n$ do
            ▷ Try to update $R_k^{\text{ub}}$ for task $\tau_k$
4           NewBound ← COMPUTERESPONSE($\tau_k$)
5           if (NewBound > $D_k$) return false
6           if (NewBound > $R_k^{\text{ub}}$)
7               { $R_k^{\text{ub}}$ ← NewBound
8               Updated ← true }
        end for
    done
    ▷ When bounds converged and no task is infeasible, declare success
9   return true
```

Fig. 17.3 Response time analysis: second strategy

17.1.2 [RTA] for EDF

When adopting an earliest-deadline-first (EDF) scheduler, a tighter upper bound on the workload executed by the interfering tasks within the interval of interest can be derived, by noting that no job can be interfered with, by jobs with later deadlines. As in Sect. 15.1, we will consider the worst-case workload produced by an interfering

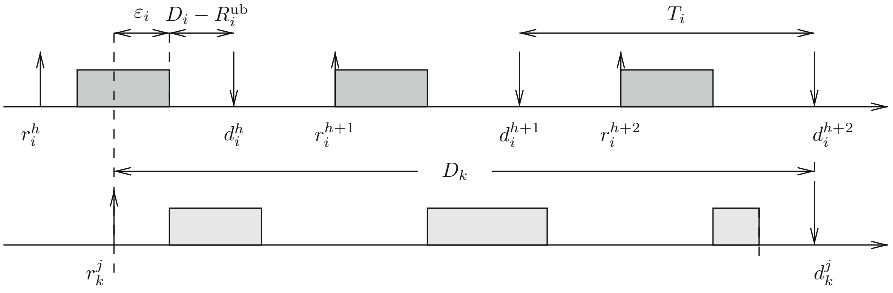


Fig. 17.4 Scenario that produces, with EDF, the maximum possible interference of task τ_i on a job of task τ_k, when R_i^{ub} is an upper bound on τ_i's response time

task τ_i when it has an absolute deadline coincident to a deadline of τ_k, and every other instance of τ_i is executed as late as possible.

Consider the situation in Fig. 17.4. Note that the interval of interest coincides with the full scheduling window of τ_k, i.e., from the release time to the deadline of the considered job. We express τ_i's workload separating the contributions of the first job that has a deadline inside the considered interval, from the contributions of later jobs of τ_i. There are $\left\lfloor \frac{D_k}{T_i} \right\rfloor$ later jobs, each one contributing for an entire worst-case computation time. Instead, the first job contributes for max $\left(0, D_k \bmod T_i - (D_i - R_i^{\mathrm{ub}})\right)$, when this term is lower than C_i. We therefore obtain the following expression:

$$I_{i,k} \le \widehat{I}_{i,k} \doteq \left\lfloor \frac{D_k}{T_i} \right\rfloor C_i + \min\left(C_i, \left(D_k \bmod T_i - D_i + R_i^{\mathrm{ub}}\right)_0\right). \tag{17.5}$$

Note that when $R_i^{\mathrm{ub}} = D_i$, Eq. (17.5) reduces to Eq. (15.1).

The above bound, along with the bounds presented for a general work-conserving scheduler, yields the following theorem.

Theorem 17.3 *An upper bound on the response time of a task τ_k in a multiprocessor system scheduled with global* EDF *can be derived by the fixed point iteration on the value R_k^{ub} of the following expression, starting with $R_k^{\mathrm{ub}} = C_k$:*

$$R_k^{\mathrm{ub}} \leftarrow C_k + \left\lfloor \frac{1}{m} \sum_{i \neq k} \min(\widehat{W}_i(R_k^{\mathrm{ub}}), \widehat{I}_{i,k}, R_k^{\mathrm{ub}} - C_k + 1) \right\rfloor, \tag{17.6}$$

where $\widehat{W}_i(L)$ and $\widehat{I}_{i,k}$ are given by Eq. 17.3 and 17.5, respectively.

Proof The proof is identical to that of Theorem 17.2. □

The iterative schedulability test we described for general work-conserving algorithms, applies as well to the EDF case. The only difference lies at line 2 of procedure SCHEDCHECK(τ) in Fig. 17.2, where procedure COMPUTERESPONSE(τ_k) returns the response time bound computed using Theorem 17.3.

This [RTA] test dominates the [BCL] test, improving the analysis by (i) reducing the individual carry-in contributions, and (ii) considering a smaller interval of interest (corresponding to the worst-case response time of the considered task instead of its full scheduling window) and thereby potentially decreasing the number of jobs of every other task that may overlap this interval of interest.

17.1.3 Computational Complexity

The complexity of the response time analysis presented in this section depends on the number of times the response time bound of a task can be updated. Consider a call to procedure SCHEDCHECK(τ), which will repeatedly invoke procedure COMPUTERESPONSE(τ_k). The complexity of procedure COMPUTERESPONSE(τ_k) is comparable to the classic response time analysis for uniprocessor systems (see Sect. 4.2). Assuming an integer model for time, the procedure will converge, or fail, in at most $(D_k - C_k)$ steps, each one requiring a sum of n contributions. The complexity is then $O(nD_{\max})$.

Since procedure COMPUTERESPONSE(τ_k) is invoked once for each task, (through the inner (*for*) cycle from 1 to n), the complexity of a single round of response time bound updates through all n tasks is $O(n^2 D_{\max})$. Now, the outer (*while*) cycle is iterated as long as there is an update in one of the response time bounds. Since, for the integer time assumption, each bound can be updated at most $D_k - C_k$ times, a rough upper bound on the total number of iterations of the outer cycle is $\sum_k (D_k - C_k) = O(nD_{\max})$. The overall complexity of the whole algorithm is then $O(n^3 D_{\max}^2)$.

While the worst-case complexity of the two different strategies in Fig. 17.2 and Fig. 17.3 is the same, the first strategy can be modified by stopping the test after a finite number of iterations N. If this is the case, the total number of steps taken by the schedulability algorithm is $O(n^3 N D_{\max})$.

The average run-time could be further improved by noting that a potential weakness is given by the contribution $(R_k^{\text{ub}} - C_k + 1)$ in the minimum of the RHS term of Eq. 17.6. This value can cause a slow progression of the iteration towards the final value, due to the low rate at which the response time is increased at each step. If the final response time is very late in time, the iteration will potentially converge after a lot of iterations. A simple modification can be made to provide a faster convergence, by splitting the procedure into two stages. In the first stage, the value $(R_k^{\text{ub}} - C_k + 1)$ is replaced by $(D_k^{\text{ub}} - C_k + 1)$. If the task converges to a value $R_k^{\text{ub}} \le D_k$, it is then possible to refine the derived bound on the response time in a second stage, using the original term $(R_k^{\text{ub}} - C_k + 1)$, starting with the value R_k^{ub} derived in the previous stage. This allows proceeding at bigger steps towards the final bound, eventually retreating if the step was too big. The simulations we ran with this alternative strategy did not show significant losses in the number of schedulable task sets detected in comparison with the original algorithm.

17.1.4 Reduction of Carry-in Instances

The interval of interest adopted by the [RTA] test does not allow for limiting the number of carry-in instances to at most $m-1$ of them, as done with the [BAR] test. Therefore, the tests are incomparable. However, it is possible to limit the number of carry-in instances to at most $m-1$ by changing the interval of interest considered in the [RTA] analysis. Consider an interval of interest that is similar to the one adopted for the [BAR] test. Let t_o be the start of the interval, coinciding with the last time-instant prior to the arrival of the problem job of τ_k at which some processor is idle or executing a job with lower priority. Different from the [BAR] test, the end of the interval is not the deadline of the problem job, but its finishing time.

For any work-conserving scheduler, no processor is idled when there are jobs awaiting execution. There are therefore at most $(m-1)$ carry-in jobs at the beginning of the interval. Note that from t_o to the release of the problem job, all processors are busy executing higher-priority jobs. Therefore, if the release of the problem job of τ_k is shifted left to the beginning of the interval of interest (while keeping its deadline and all previous instances of τ_k as they are), its response time cannot decrease.

An upper bound on the response time of τ_k can then be given considering a situation in which there are only $m-1$ carry-in instances. However, since previous instances of τ_k can execute at the beginning of the interval, the interfering workload should be computed extending the sum to all tasks contributions, including τ_k's (the [RTA] test, instead, limited the interfering workload to the tasks $\tau_{i\neq k}$, as evident from Eq. 17.6).

For each carry-in task $\tau_{i\neq k}$, the upper bound $\widehat{W}_i(L)$ on the workload in a window of length L given by Eq. 17.3 for a generic work-conserving scheduler can still be used. An upper bound on the workload executed in the interval of interest by previous jobs of τ_k can be derived by subtracting the problem job's contribution from $\widehat{W}_i(L)$. In both cases, it is sufficient to consider the amount of interfering contribution smaller than $L-C_k+1$. Therefore, an upper bound on the carry-in workload in the interval of interest is

$$\widehat{W}_i^{CI}(L) \doteq \begin{cases} \min\left(\widehat{W}_i(L), L-C_k+1\right), & \text{if } i \neq k \\ \min\left(\widehat{W}_i(L)-C_k, L-C_k+1\right), & \text{if } i = k. \end{cases} \tag{17.7}$$

For tasks without carry-in, the worst-case workload is computed considering the situation in Fig. 17.5, where each interfering task is released at the beginning of the window of interest.

The number of τ_i's jobs with both release and deadline within the interval of interest is $\lfloor L/T_i \rfloor$, and the contribution of the carry-out job is at most $\min(C_i, L \bmod T_i)$. The contribution of previous jobs of τ_k can again be derived by subtracting the problem job contribution. Considering the amount of interfering contribution smaller than $L-C_k+1$, an upper bound on the workload of a task with no carry-in in the interval

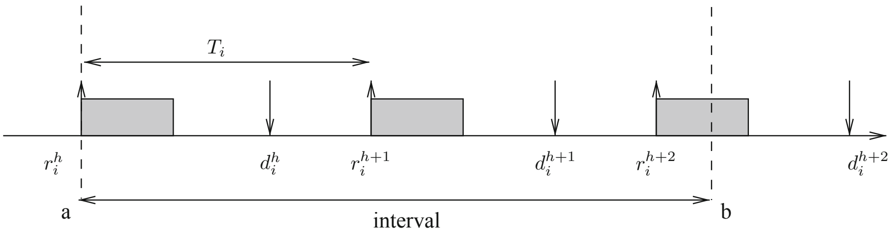


Fig. 17.5 Scenario that produces the densest possible packing of jobs of a task τ_i with no carry-in

of interest is then

$$\widehat{W}_i^{NC}(L) \doteq \begin{cases} \min\left(\left\lfloor \frac{L}{T_i} \right\rfloor C_i + \min(C_i, L \bmod T_i), L - C_k + 1\right), & \text{if } i \neq k \\ \min\left(\left\lfloor \frac{L}{T_i} \right\rfloor C_i + \min(C_i, L \bmod T_i) - C_k, L - C_k + 1\right), & \text{if } i = k. \end{cases} \tag{17.8}$$

Let us denote by $\widehat{W}_i^{\epsilon}$ the difference between $\widehat{W}_i^{CI}$ and $\widehat{W}_i^{NC}$:

$$\widehat{W}_i^{\epsilon}(L) \doteq \widehat{W}_i^{CI}(L) - \widehat{W}_i^{NC}(L). \tag{17.9}$$

The next theorem considers that at most $(m-1)$ tasks contribute at amount $\widehat{W}_i^{CI}$, and the remaining $(n-m+1)$ ones must contribute $\widehat{W}_i^{NC}$.

Theorem 17.4 *An upper bound on the response time of a task τ_k in a multiprocessor system scheduled with a global work-conserving scheduler can be derived by the fixed point iteration on the value R_k^{ub} of the following expression, starting with $R_k^{\text{ub}} = C_k$:*

$$R_k^{\text{ub}} \leftarrow C_k + \left\lfloor \frac{1}{m} \left(\sum_{\tau_i \in \tau} \widehat{W}_i^{NC}(L) + \sum_{i \mid (m-1)\, largest} \widehat{W}_i^{\epsilon}(L) \right) \right\rfloor, \tag{17.10}$$

where $\widehat{W}_i^{NC}(L)$ and $\widehat{W}_i^{\epsilon}(L)$ are given by Eq. 17.8 and 17.9, respectively .

As the sum of the interfering contribution is extended to all tasks, the above theorem is incomparable with the [RTA] test for work-conserving schedulers given by Theorem 17.2, as well as with the [RTA] test for systems scheduled with global EDF given by Theorem 17.3.

Sources

The response-time analysis for global work-conserving schedulers and EDF has been presented in [58]. The reduction of the carry-in instances to consider, presented in Sect. 17.1.4, is from [123].

Chapter 18
Global Fixed-Task-Priority Scheduling

We now describe some of the main schedulability tests for 3-parameter sporadic task systems that are globally scheduled using fixed-task-priority (FTP) scheduling algorithms. It will become evident that many of these tests are derived from the tests presented in the previous chapters for general work-conserving schedulers.

We start out presenting the FTP versions of the [BCL] (Sect. 15.1) and [RTA] (Sect. 17.1) tests. Then, we show that the idea described in Sect. 17.1.4 for reducing the number of carry-in instances that must be considered, is more effective when applied to the schedulability analysis of FTP scheduling algorithms.

Finally, we address the priority assignment problem and discuss a strategy that makes use of the presented schedulability tests.

Before presenting the tests, we make an important observation concerning the interference (see Sect. 14.4) imposed by lower-priority tasks in FTP schedules.

Observation 18.1 In FTP scheduling no task suffers interference from any lower-priority task. Hence

$$I_{i,k} = 0 \;\; \forall i, k \mid \tau_i \text{ has lower priority than } \tau_k$$

Assuming tasks are indexed by decreasing priorities ($i > k \Rightarrow \tau_k$ has scheduling priority over τ_i), Lemma 14.1 then specializes as follows.

Theorem 18.1 *The interference on a constrained deadline task τ_k is the sum of the interferences of all higher-priority tasks divided by the number of processors:*

$$I_k = \frac{\sum_{i<k} I_{i,k}}{m}. \tag{18.1}$$

18.1 The [BCL] Test for FTP Algorithms

The "workload versus demand" strategy adopted in the [BCL] test presented in Sect. 15.1 can be applied also in the FTP case. However, a different upper bound on the interference of each task τ_i on the problem job of τ_k needs to be provided. While no carry-out job interferes with the problem job in EDF, this is not the case with a

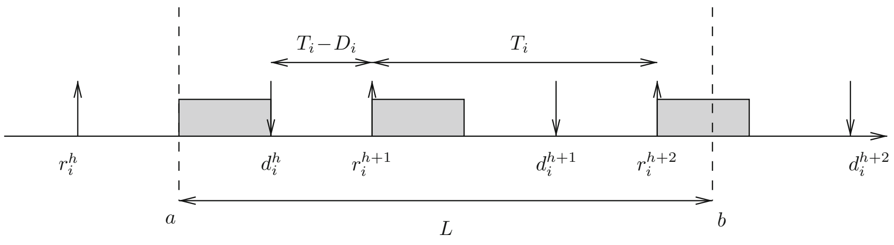


Fig. 18.1 Response-time analysis for FTP systems

general FTP scheduler. A different situation than the one depicted in Fig. 15.1 needs to be considered.

A bound on the workload of a task τ_i in the problem job's scheduling window $[r_k, r_k + D_k]$ can be computed by considering a situation in which the carry-in job starts executing as close as possible to its deadline and right at the beginning of the interval of interest, and every other instance of τ_i is executed as soon as possible. The situation is represented in Fig. 18.1. The analysis is similar to that presented for deriving Eq. 17.3 for the [RTA] test.

The first job of τ_i after the carry-in job is released at time $(r_k + C_i + T_i - D_i)$. The next jobs are then released periodically every T_i time units. Therefore the number of body jobs within the interval is

$$\left\lfloor \frac{(r_k + D_k) - (r_k + C_i - D_i + T_i)}{T_i} \right\rfloor = \left\lfloor \frac{D_k + D_i - C_i - T_i}{T_i} \right\rfloor .$$

Adding the (full) carry-in contribution, the number of τ_i's jobs that contribute with an entire Worst-case Execution Times (WCET) to the workload in an interval of length D_k is

$$\left\lfloor \frac{D_k + D_i - C_i}{T_i} \right\rfloor .$$

The contribution of the carry-out job can then be bounded by

$$\min(C_i, (D_k + D_i - C_i) \bmod T_i).$$

A bound on the workload of a task τ_i in the problem job's scheduling window is then:

$$\left\lfloor \frac{D_k + D_i - C_i}{T_i} \right\rfloor C_i + \min(C_i, (D_k + D_i - C_i) \bmod T_i). \tag{18.2}$$

While this bound is weaker than that for EDF given by Eq. (15.2), the performance of the [BCL] test for FTP systems is significantly improved by applying Theorem 18.1.

The FTP analog of the [BCL] Theorem 15.2 is given by the following theorem.

Theorem 18.2 *([BCL] for FTP) Task system τ is FTP schedulable if, for each $\tau_k \in \tau$,*

$$\sum_{i<k} \min\left(D_k - C_k, \left\lfloor \frac{D_k + D_i - C_i}{T_i} \right\rfloor C_i + \min(C_i, (D_k + D_i - C_i) \bmod T_i)\right)$$
$$< m(D_k - C_k)\ . \tag{18.3}$$

As in the EDF case, the schedulability test consists of n inequalities, each a sum of n terms. The computational complexity is therefore polynomial, at $O(n^2)$.

18.2 The [RTA] Test for FTP Scheduling

It is possible to similarly specialize the [RTA] test [58] to FTP-scheduled systems by applying Theorem 18.1, as the result of Theorem 17.1 can be specialized to FTP-scheduled systems by eliminating consideration of interference by all lower-priority tasks.

Theorem 18.3 *A task τ_k scheduled with FTP has a response time bounded by R_k^{ub} if*

$$\sum_{i<k} \min\left(I_{i,k}, R_k^{\text{ub}} - C_k + 1\right) < m(R_k^{\text{ub}} - C_k + 1)$$

The bound $\widehat{W}_i(L)$ on the generic workload in a window of length L given by Eq. 17.3 is still valid. It is then possible to compute an upper bound on the response time of a task τ_k as follows.

Theorem 18.4 *[[RTA] for FTP] An upper bound on the response time of a task τ_k in a multiprocessor system scheduled with FTP can be derived by the fixed point iteration on the value R_k^{ub} of the following expression, starting with $R_k^{\text{ub}} = C_k$:*

$$R_k^{\text{ub}} \leftarrow C_k + \left\lfloor \frac{1}{m} \sum_{i<k} \min(\widehat{W}_i(R_k^{\text{ub}}), R_k^{\text{ub}} - C_k + 1) \right\rfloor, \tag{18.4}$$

where $\widehat{W}_i(R_k)$ is given by Eq. 17.3.

The iterative schedulability test previously described for EDF and general work-conserving schedulers in Figs. 17.2 and 17.3 can be simplified for systems scheduled with FTP. By updating the response time upper bounds in priority order, the test can be stopped after the first round of updates. Indeed, tighter estimations of the interferences produced by lower priority tasks cannot possibly produce an improvement in the response time bound of a higher-priority task.

This observation enables the limiting of the number of updates to one for each task. Therefore, a simpler strategy can be adopted, as reported in Fig. 18.1 with procedure [SCHEDCHECKFP](τ). Function COMPUTERESPONSE(τ_k) returns the response time bound computed using Theorem 18.4.

As in the EDF case, the [RTA] test dominates the [BCL] test. However, the complexity of the [RTA] test for FTP systems is smaller than in the EDF and general cases. As mentioned in Sect. 17.1.3, computing a response time bound for a task takes $O(nD_{\max})$. Since there is only one round of slack updates for n tasks, the overall complexity is $O(n^2 D_{\max})$, instead of $O(n^3 D_{\max}^2)$. This reduced complexity, along with the improvement resulting from being able to neglect the interference of lower-priority tasks, makes the [RTA] particularly suitable for the schedulability analysis of FTP-scheduled systems. In fact, schedulability experiments on randomly generated workloads [55, 58] seem to indicate that procedure [SCHEDCHECKFP] outperforms the corresponding procedure [SCHEDCHECKFP] for EDF-scheduled systems, in terms of the number of schedulable task sets detected among randomly generated workloads.

18.3 Reducing the Number of Carry-in Instances

In Sect. 17.1.4, we showed how the number of carry-in tasks that need to be considered in the computation of interfering workload can be reduced to $m - 1$ for general work-conserving schedulers. A similar argument is applicable to FTP-scheduled systems, with an additional improvement: besides ignoring the interference from lower priority tasks, there is no need to consider previous jobs of τ_k in the computation of the workload interfering with the problem job of τ_k.

Theorem 18.5 *(Improved* [RTA] *for FTP) An upper bound on the worst-case response time R_k of a task τ_k in a multiprocessor system scheduled with global FTP can be derived by the fixed point iteration on the value R_k^{ub} of the following expression, starting with $R_k^{\mathrm{ub}} = C_k$:*

$$R_k^{\mathrm{ub}} \leftarrow C_k + \left\lfloor \frac{1}{m} \left(\sum_{i<k} \widehat{W}_i^{NC}(L) + \sum_{i<k|(m-1)\ \mathit{largest}} \widehat{W}_i^{\epsilon}(L) \right) \right\rfloor, \tag{18.5}$$

where $\widehat{W}_i^{NC}(L)$ and $\widehat{W}_i^{\epsilon}(L)$ are given by Eqs. (17.8) and (17.9), respectively.

Proof Consider an interval of interest that starts at the last time-instant, prior to the arrival of the problem job of τ_k, at which some processor is idle *or* is executing a task with priority lower than or equal to τ_k. Let t_o denote this time-instant.

Note that as long as $R_k \leq T_k$, the previous instances of the problem job do not contribute to the interfering workload in the considered interval. Also, from t_o to the release of the problem job, all processors are busy executing higher-priority jobs. Therefore the response time of the problem job cannot decrease if we were to shift all τ_k's instances to the left (i.e., earlier in time), so that the release of the problem job coincides with t_o.

By the definition of t_o, there are at most $(m - 1)$ carry-in tasks interfering with τ_k. An upper bound on the response time of τ_k can then be obtained by considering a situation in which there are only $m - 1$ carry-in instances.

The theorem follows by considering that at most $(m-1)$ tasks contribute at amount $\widehat{W}_i^{CI}$, given by Eq. (17.7), and the remaining $(n-m+1)$ ones must contribute $\widehat{W}_i^{NC}$. □

Note that there is no need to consider the case $i = k$ in the terms $\widehat{W}_i^{NC}(L)$ and $\widehat{W}_i^{CI}(L)$ given by Eqs. (17.8) and (17.7).

Differently from the EDF and general work-conserving cases, the above theorem dominates its original [RTA] counterpart given by Theorem 18.4, which considers the full carry-in contribution $\widehat{W}_i^{CI}$ for all tasks. However, when the number of tasks is $n \le 2m$, both theorems are equivalent [74].

Lemma 18.1 *The response time upper bounds of the m highest-priority tasks computed with Theorem 18.4 and 18.5 are equal to their worst-case execution times:*

$$R_k^{\text{ub}} = C_k, \forall 1 \le k \le m.$$

Proof This can be seen by considering Eqs. (18.4) and 18.5 starting with $R_k^{\text{ub}} = C_k$. The bounds $\widehat{W}_i^{NC}(L)$ and $\widehat{W}_i^{CI}(L)$ given by Eqs. (17.8) and (17.7) are limited by the $R_k^{\text{ub}} - C_k + 1$ term in the minimum, which is equal to 1. Since there are at most $m-1$ higher priority tasks, the floor function in Eqs. (18.4) and 18.5 gives $\lfloor (m-1)/m \rfloor = 0$. Hence, the fixed point iteration immediately terminates, returning a value of $R_k^{\text{ub}} = C_k, \forall 1 \le i \le m$. □

Theorem 18.6 *Theorem 18.4 and 18.5 are equivalent when $n \le 2m$.*

Proof From Lemma 18.1, we know that both theorems are equivalent when $n \le m$. Now, consider the tasks with priorities from $m+1$ to $2m$. The interference from the m highest priority tasks τ_i can be computed considering that $\forall 1 \le i \le m : R_i^{\text{ub}} = C_i$. In these conditions, the term $\widehat{W}_i^{CI}(L)$ given by Eq. (17.7) is equal to the corresponding term $\widehat{W}_i^{NC}(L)$ given by Eqs. (17.8), for all $1 \le i \le m$, so that $\widehat{W}_i^{\epsilon}(L) = 0$ for all $1 \le i \le m$. Out of the first $2m-1$ tasks, at most $2m-1-m = m-1$ of them can have $\widehat{W}_i^{\epsilon}(L) \ge 0$. Limiting the sum of the $\widehat{W}_i^{\epsilon}(L)$ terms in Eq. 18.5 to the largest $m-1$ contributions has therefore no effect for each of the $2m$ higher-priority tasks. Hence, Eq. 18.5 reduces to Equation (18.4), proving the theorem. □

18.3.1 Reducing the Number of Carry-in Instances for [BCL]

The idea explored above for limiting the contribution of carry-in in the [RTA] test is also applicable to the [BCL] test presented in Sect. 18.1; the proof is identical to that of Theorem 18.5.

Theorem 18.7 *(Improved* [BCL] *for FTP) Task system τ is FTP schedulable if, for each $\tau_k \in \tau$,*

$$\left(\sum_{i<k} \widehat{W}_{i,k}^{NC} + \sum_{i<k|(m-1)\ largest} \widehat{W}_{i,k}^{\epsilon} \right) < m(D_k - C_k), \tag{18.6}$$

where

$$\widehat{W}^{NC}_{i,k} = \min\left(\left\lfloor \frac{D_k}{T_i} \right\rfloor C_i + \min(C_i, D_k \bmod T_i), D_k - C_k\right), \tag{18.7}$$

$$\widehat{W}^{CI}_{i,k} = \min\left(\widehat{W}_i(D_k), D_k - C_k\right), \tag{18.8}$$

$$\widehat{W}^{\epsilon}_{i,k} \doteq \widehat{W}^{CI}_{i,k} - \widehat{W}^{NC}_{i,k}, \tag{18.9}$$

and $\widehat{W}_i(L)$ *is given by Eq. 17.3.*

18.4 The Priority Assignment Problem

All the FTP schedulability tests that we have described above check whether a task system is schedulable for a given priority assignment. We now consider the design problem of assigning the priorities: given a task system that must be scheduled using FTP scheduling, how do we assign priorities to the tasks in order to ensure schedulability?

We had seen in Sect. 4.2 that the priority assignment problem is solved for constrained-deadline 3-parameter sporadic task systems upon preemptive uniprocessors: deadline monotonic (DM) priority assignment is optimal in the sense that if a task system is FTP schedulable under any priority assignment then it is also FTP-schedulable under the DM priority assignment. However, it is easy to construct examples showing that DM is not optimal for global scheduling upon multiprocessors.

A very general scheme for assigning priorities on uniprocessor platforms was proposed by Audsley [12, 14, 15][1].

This scheme, depicted in pseudocode form in Fig. 18.2, seeks to recursively identify a task that may be assigned lowest priority; once such a task has been identified, it is removed from consideration and the process repeated with the remaining tasks.

This is guaranteed to find a priority assignment that is feasible according to schedulability test S, if one exists. The test is invoked at most $n(n+1)/2$ times where n denotes the number of tasks in the system, which is significantly better than inspecting all $n!$-possible priority orderings. However, S cannot be just any schedulability test; in [74], a set of three conditions are identified for a schedulability test to be *compatible* with the Audsley algorithm.

Theorem 18.8 *(from [74]) A schedulability test S is compatible with the Audsley algorithm if and only if the following three conditions are met:*

[1] This can be thought of as a generalization of the technique introduced earlier by Lawler [122] to recurrent tasks.

1. *The S-schedulability of a task does not depend on the relative priority ordering of higher-priority tasks.*
2. *The S-schedulability of a task does not depend on the relative priority ordering of lower-priority tasks.*
3. *A task that is S schedulable remains S schedulable if it is assigned a higher priority.*

It is easy to see that the above conditions are needed because the assignment of priorities with Audsley's algorithm is agnostic of the relative priority ordering mentioned in the conditions.

The schedulability tests presented in this chapter are not all compatible with the Audsley algorithm. While Condition 2 and 3 hold for all of them, the same is not true for Condition 1. In particular, both the [RTA] and the improved [RTA] tests do not meet Condition 1 and are, therefore, not compatible, as can be seen with the following example (from [74]).

Example 18.1 Consider a task set composed of the following four tasks: $\tau_1 = (10, 20, 20), \tau_2 = (10, 20, 20), \tau_3 = (10, 20, 100), \tau_4 = (20, 55, 55)$. When $m = 2$, the task set is deemed schedulable by [RTA] test with upper bounds on task response times of 10, 10, 20, and 55, respectively. However, by switching the priorities of τ_2 and τ_3, the response time upper bound of τ_2 increases to 20. This increases the interference bound of τ_2 on τ_4, so that τ_4 is no longer deemed schedulable by [RTA].

Using Theorem 18.6, the same example can be used to prove that the improved [RTA] is not compatible with the Audsley algorithm either.

A pessimistic version of the [RTA] that is compatible with the Audsley algorithm can be derived by considering $R_i = D_i$ for all higher-priority tasks τ_i when computing R_k^{ub}, as stated in the following theorem.

Theorem 18.9 *(Pessimistic* [RTA] *for FTP) An upper bound on the response time of a task τ_k in a multiprocessor system scheduled with FTP can be derived by the fixed point iteration on the value R_k^{ub} of the following expression, starting with $R_k^{\mathrm{ub}} = C_k$:*

$$R_k^{\mathrm{ub}} \leftarrow C_k + \left\lfloor \frac{1}{m} \sum_{i<k} \min\left(\widehat{W}_i^*(R_k^{\mathrm{ub}}), R_k^{\mathrm{ub}} - C_k + 1\right) \right\rfloor, \tag{18.10}$$

where

$$\widehat{W}_i^*(L) = \left\lfloor \frac{L + D_i - C_i}{T_i} \right\rfloor C_i + \min(C_i, (L + D_i - C_i) \bmod T_i). \tag{18.11}$$

Since $\widehat{W}_i^*(L) \geq \widehat{W}_i(L)$ as long as $R_i \leq D_i$, Theorem 18.9 is more pessimistic than the original [RTA]. However, the response time-bound R_k^{ub} does not depend on the response time bounds of higher priority tasks, so that it is insensitive of their relative priority ordering, meeting Condition 1.

A pessimistic version of the test of Theorem 18.5 can similarly be formulated using $\widehat{W}_i^*(L)$ instead of $\widehat{W}_i(L)$ when computing the $\widehat{W}_i^{\epsilon}(L)$ terms. Also in this case, this pessimistic version is compatible with the Audsley algorithm.

```
OPTIMALPRIORITY($\tau$)
    ▷ Optimal priority assignment for single processor systems.
1   for $k = n$ to 1 do
2       for each task $\tau_i \in \tau$ do
3           if $\tau_i$ is schedulable at priority $k$ using test $S$
4               Assign priority $k$ to $\tau_i$
5               Remove $\tau_i$ from $\tau$
                ▷ Exit the inner loop
6               break
        end for
7       if no task has been assigned priority $k$
8           return unschedulable
    end for
    ▷ When all priorities have been assigned, declare success
9   return schedulable
```

Fig. 18.2 The Audsley algorithm for priority assignment in single processor systems

Regarding the [BCL] test and its improved version (Theorems 18.2 and 18.7), it can be verified that both tests are compatible with the Audsley algorithm.

A priority assignment for multiprocessor systems globally scheduled with FTP can be derived by using the Audsley algorithm together with a compatible schedulability test. The tightest test among the compatible ones presented in this chapter appears to be the pessimistic version of Theorem 18.5 that replaces $\widehat{W}_i(L)$ with $\widehat{W}_i^*(L)$.

If the use of a tighter schedulability test that is not compatible with the Audsley algorithm is desired, an alternative is to choose a priority assignment, such as deadline monotonic or [DM-DS], and then check this priority assignment for schedulability. Davis and Burns [74] conducted schedulability experiments comparing a variety of different priority assignments, including DM, slack monotonic, etc., in terms of schedulable task sets among randomly generated workloads. Summarizing their conclusions (the detailed discussion may be found in [74]), it appears that the Audsley algorithm coupled with a compatible test generally outperforms any of the other studied priority assignments coupled with a tighter schedulability test.

Sources

The response-time analysis for global FTP has been presented in [58]. The reduction of the carry-in instances to consider, presented in Sect. 18.3, is from [100]. Theorem 18.6 and the considerations on the priority assignment problem are from [74].

Chapter 19
Speedup Bounds for Global Scheduling

In the preceding chapters, we have seen a variety of global schedulability analysis tests for the fixed job priority (FJP) scheduling algorithm EDF, the fixed task priority (FTP) scheduling algorithm deadline monotonic (DM), and the dynamic priority scheduling algorithm [EDZL]. When applied to systems of sporadic three-parameter task systems, the only one of these tests for which a quantitative metric of worst-case performance was obtained is the [BAK] test (Chap. 16): Corollary 16.1 showed that the processor speedup factor of the [BAK] test is no larger than $(3 + \sqrt{5})/2$, or ≈ 2.6181. In this chapter, we will describe the results of [63] deriving a tight speedup bound of $(2-\frac{1}{m})$ for global EDF scheduling of three-parameter sporadic task systems upon m processors, and a tight speedup bound of $(3 - \frac{1}{m})$ for global DM scheduling of three-parameter sporadic task systems upon m processors. We will also describe how these speedup bounds were used [36, 37, 63] to derive pseudopolynomial time schedulability tests that are within a factor ϵ away from optimal, for ϵ an arbitrarily small positive number.

Recall the definition of the *speedup factor* metric (Definition 5.2 in Chap. 5). A schedulability test is defined to have a processor speedup factor f, $f \geq 1$, if any task system not deemed to be schedulable by this test upon a particular platform is guaranteed to not be *feasible*—schedulable by an optimal clairvoyant scheduler—upon a platform in which each processor is at most $1/f$ times as fast. As stated above, we will now determine the processor speedup factor of global EDF and global DM when scheduling sporadic task systems; for ease of presentation, we restrict our attention here to constrained task systems.

The remainder of this chapter is organized as follows. We describe a characterization of a task's demand called *forced-forward demand* in Sect. 19.1 below. This allows for the derivation of a tighter lower bound than DBF (Sect. 10.3) upon the largest amount of execution that may be demanded by a task over an interval.

In Sect. 19.2, we determine sufficient schedulability conditions for both EDF and DM. From these, we derive speedup bounds in Sect. 19.3. The idea behind these speedup bounds can be expanded to develop sufficient schedulability tests with superior speedup bounds; we illustrate this by developing such a test for EDF in Sect. 19.4.

S. Baruah et al., *Multiprocessor Scheduling for Real-Time Systems*,
Embedded Systems, DOI 10.1007/978-3-319-08696-5_19

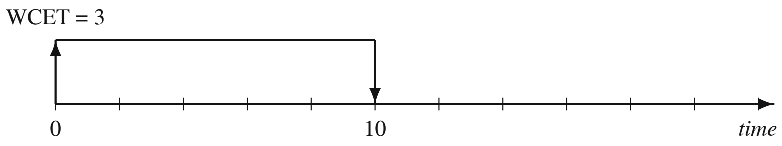


Fig. 19.1 Illustrating Example 19.1.

19.1 The Forced-Forward Demand Bound Function

It was observed [24, 63] that some jobs arriving and/or having deadlines outside an interval could also contribute to the cumulative execution requirement placed on the computing platform within the interval. This observation led to the introduction of a notion called *minimum demand* [24]. The minimum demand of a given collection of jobs in a particular interval of time is the minimum amount of execution that the sequence of jobs could require within that interval in order to meet all its deadlines.[1] We illustrate the difference between *demand* and *minimum demand* by a simple example.

Example 19.1 Consider a sequence of jobs comprised of a single job that arrives at time-instant zero, has an execution requirement equal to 5, and a deadline at time-instant 10 (Fig. 19.1). The *demand* of this sequence of jobs over the time-interval $[0, t)$ is 0 for all $t < 5$, and 3 for all $t \geq 5$. The *minimum demand* of this sequence of jobs over the time-interval $[0, t)$ is equal to

- zero, for values of $t \leq 2$;
- $t - 2$, for $t \in (2, 5]$, since the sole job must execute for at least $t - 2$ units over the interval if it is to meet its deadline; and
- 3, for $t > 5$.

The minimum demand concept is extended to sporadic tasks as follows. By definition, a sporadic task τ_i may generate infinitely many different collections of jobs. For a given interval-length t, τ_i's *maxmin demand* is defined to be the largest minimum demand over an interval of length t, by any collection of jobs that could legally be generated by τ_i.

The *maxmin demand* of a sporadic task system τ for interval-length t is defined as the sum of the maxmin demands of the tasks in τ, each for an interval-length t.

The *maxmin load* of τ is defined as the maximum value of the maxmin demand of τ for t, normalized by the interval length.

Forced-Forward Demand Bound Function In [36, 37, 63], these concepts of maxmin demand and maxmin load were generalized in the following manner to be applicable to execution on speed-σ processors, for arbitrary $\sigma > 0$.

[1] An essentially identical concept was independently introduced in [63], and called the *necessary demand*; a related concept called *forced-forward demand* was also introduced.

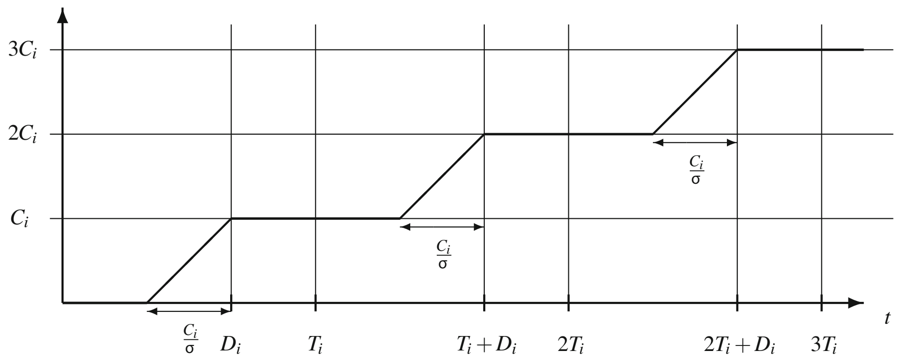


Fig. 19.2 Illustrating FF-DBF(τ_i, t, σ).

Let τ_i denote a sporadic task, t any positive real number, and σ any positive real number ≤ 1. The *forced-forward demand bound function* FF-DBF(τ_i, t, σ) is defined as follows:

$$\text{FF-DBF}(\tau_i, t, \sigma) \stackrel{\text{def}}{=} q_i C_i + \begin{cases} C_i & \text{if } r_i \geq D_i \\ C_i - (D_i - r_i)\sigma & \text{if } D_i > r_i \geq D_i - \frac{C_i}{\sigma} \\ 0 & \text{otherwise} \end{cases} \tag{19.1}$$

where

$$q_i \stackrel{\text{def}}{=} \left\lfloor \frac{t}{T_i} \right\rfloor \quad \text{and} \quad r_i \stackrel{\text{def}}{=} t \bmod T_i \ .$$

Informally speaking, FF-DBF(τ_i, t, σ) can be thought of as denoting the maxmin demand of τ_i for interval-length t, when execution *outside* the interval occurs on a speed-σ processor—see Fig. 19.2. The demand bound function DBF described in Sect. 10.3 is a special case of FF-DBF, in which it is assumed that execution outside the interval occurs on an infinite-speed processor:

$$\text{DBF}(\tau_i, t) = \text{FF-DBF}(\tau_i, t, \infty) \ .$$

Some additional notation: for a task system τ

$$\text{FF-DBF}(\tau, t, \sigma) \stackrel{\text{def}}{=} \sum_{\tau_\ell \in \tau} \text{FF-DBF}(\tau_\ell, t, \sigma) \ . \tag{19.2}$$

$$\text{FF-LOAD}(\tau, \sigma) \stackrel{\text{def}}{=} \max_{t>0} \left(\frac{\text{FF-DBF}(\tau, t, \sigma)}{t} \right) \ . \tag{19.3}$$

It follows from the results in [81] that the exact determination of FF-DBF and FF-LOAD is computationally intractable (co-NP hard).

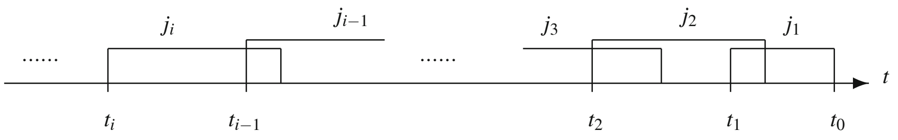


Fig. 19.3 Notation used in Sect. 19.2.

19.2 Sufficient Schedulability Conditions

Both EDF and DM are *work-conserving* algorithms: they never idle any processor while there are jobs awaiting execution. As we will see below, this work-conserving property implies that a deadline miss can only follow an interval during which a considerable amount of execution must have occurred. This property is formalized in Observation 19.1 below, based upon the following reasoning.

Let A denote some work-conserving algorithm that misses a deadline while scheduling some legal collection of jobs generated by τ on a unit-speed processor. Let us now examine A's behavior on some minimal[2] legal collection of jobs of τ on which it misses a deadline.

Let t_0 denote the instant at which the deadline miss occurs. Let j_1 denote a job that misses its deadline at t_0, and let t_1 denote j_1's arrival-time.

Let s denote any constant satisfying $\text{dens}_{\max}(\tau) \le s \le 1$.

(Observe that, since $s \ge \text{dens}_{\max}(\tau)$ and j_1 has not completed execution by t_0, it has executed for strictly less than $(t_0 - t_1) \times s$ units over the interval $[t_1, t_0)$.)

We define a sequence of jobs j_i, time-instants t_i, and an index k, according to the following pseudo-code (also see Fig. 19.3):

```
for $i \leftarrow 2, 3, \ldots$ do
  let $j_i$ denote a job that
    – arrives at some time-instant $t_i < t_{i-1}$;
    – has a deadline after $t_{i-1}$;
    – has not completed execution by $t_{i-1}$; and
    – has executed for strictly less than $(t_{i-1} - t_i) \times s$ units over the interval [t_i, t_{i-1}).
  if there is no such job then
    $k \leftarrow (i-1)$
    break (out of the for loop)
end if
end for
```

[2] By *minimal* we mean that A is able to successfully schedule every proper subset of this collection of jobs.

Let W denote the amount of execution that occurs in this schedule over the interval $[t_k, t_0)$. Observation 19.1 derives a lower bound on W for any work-conserving algorithm A.

Observation 19.1

$$W > (m - (m-1)s) \times (t_0 - t_k) .$$

Proof This is easily shown by an application of the technique in [155].

For each i, $1 \le i \le k$, let W_i denote the total amount of execution that occurs over the interval $[t_i, t_{i-1})$. Hence, $W = \sum_{i=1}^{k} W_i$.

Let x_i denote the total length of the time-intervals over $[t_i, t_{i-1})$ during which job j_i executes.

By choice of job j_i, it is the case that

$$x_i < (t_{i-1} - t_i) \cdot s .$$

By choice of job j_i, it has not completed execution by time-instant t_{i-1}. Hence over $[t_i, t_{i-1})$, all m processors must be executing whenever j_i is not; it follows that

$$\begin{aligned} W_i &\ge m(t_{i-1} - t_i - x_i) + x_i \\ &= m(t_{i-1} - t_i) - (m-1)x_i \\ &> m(t_{i-1} - t_i) - (m-1)(t_{i-1} - t_i)s \\ &= (m - (m-1)s) \times (t_{i-1} - t_i) . \end{aligned}$$

The observation, follows, by summing $\sum_{i=1}^{k} W_i$. □

Observation 19.1 above derived a lower bound on the amount of work W that is executed over $[t_k, t_0)$. In the following two observations, we will derive *upper* bounds on W when the scheduling algorithm is EDF and DM respectively; necessary conditions for nonschedulability under EDF and DM will follow, by requiring that the respective upper bounds be at least as large as the lower bound.

Observation 19.2 (EDF) If the work-conserving algorithm A is EDF, then

$$W \le \text{FF-DBF}(\tau, (t_0 - t_k), s) .$$

Proof Recall our assumption that we are analyzing a *minimal* unschedulable collection of jobs. If EDF is the scheduling algorithm, then such a minimal unschedulable collection will not contain any job that has its deadline $> t_0$ (since the presence of such jobs cannot effect the scheduling of jobs with deadline $\le t_0$, the collection of jobs obtained by removing all such jobs is also unschedulable by EDF.)

Thus all jobs that execute in $[t_k, t_0)$ (and thereby contribute to W) have their deadlines within the interval $[t_k, t_0)$. Some of them will also have arrived within this interval, while others may not.

Now it may be verified that the amount of execution that jobs of any task τ_ℓ contribute to W is bounded from above by the scenario in which a job of τ_ℓ has its deadline coincident with the end of the interval t_0, and prior jobs have arrived exactly T_ℓ time-units apart. Under this scenario, the jobs of τ_ℓ that may contribute to W include

- at least $q_\ell \stackrel{\text{def}}{=} \lfloor (t_0 - t_k)/T_\ell \rfloor$ jobs of τ_ℓ that lie entirely within the interval $[t_k, t_0)$; and
- (perhaps) an additional job that has its deadline at time-instant $t_k + r_\ell$, where $r_\ell \stackrel{\text{def}}{=} (t_0 - t_k) \bmod T_\ell$.

We now consider two separate cases:

1. $r_\ell \geq D_\ell$; i.e., the additional job with deadline at $t_k + r_\ell$ arrives at or after t_k. In this case, its contribution is C_ℓ.
2. $r_\ell < D_\ell$; i.e., the additional job with deadline at $t_k + r_\ell$ arrives prior to t_k. From the exit condition of the for-loop, it must be the case that this job has completed at least $(D_\ell - r_\ell) \times s$ units of execution prior to time-instant t_k; hence, its remaining execution is at most $\max(0, C_\ell - (D_\ell - r_\ell) \times s)$.

In either case, it may be seen that the upper bound on the total contribution of τ_ℓ to W is equal to $\text{FF-DBF}(\tau_\ell, t_0 - t_k, s)$ (see Eq. 19.1). Summing over all ℓ, we conclude that the total contribution of all the tasks to W is bounded from above by $\sum_{\ell=1}^{n} (\text{FF-DBF}(\tau_\ell, t_0 - t_k, s))$. □

Observation 19.3 **(DM)** If the work-conserving algorithm A is DM, then

$$W \leq \text{FF-DBF}(\tau, 2(t_0 - t_k), s) .$$

Proof Recall once again our assumption that we are analyzing a minimal unschedulable collection of jobs. If DM is the scheduling algorithm, then such a minimal unschedulable collection will not contain any job with *relative* deadline greater than the relative deadline of j_1, the job that misses its deadline at time-instant t_0 (since such jobs are assigned lower priority than j_1 under DM, and hence cannot effect the ability or otherwise of j_1 to meet its deadline.)

By the definition of t_1, the relative deadline of j_1 is $(t_0 - t_1)$. For any job with relative deadline $< (t_0 - t_1)$ to contribute to W (and hence execute prior to t_0), it must have a deadline $\leq (t_0 + (t_0 - t_1))$, i.e., $2t_0 - t_1$.

As in the proof of Observation 19.2 above, it may be verified that the amount of execution that jobs of any task τ_ℓ contribute to W is bounded from above by the scenario in which a job of τ_ℓ has its deadline coincident with the end of the interval (i.e., at $2t_0 - t_1$), and prior jobs have arrived exactly T_ℓ time-units apart. Under this scenario, it follows by an argument essentially identical to the one used in the proof of Observation 19.2 that the total contribution of all the tasks to W is bounded from above by $\text{FF-DBF}(\tau, 2t_0 - t_1 - t_k, s)$. But since $t_k \leq t_1$, $(2t_0 - t_1 - t_k) \geq 2(t_0 - t_k)$, and the observation is proved. □

Lemma 19.1 (EDF) *Suppose that constrained-deadline sporadic task system τ is not schedulable by global* EDF *upon m unit-speed processors. For each $s, s \geq dens_{\max}(\tau)$, there is an interval-length $L(s)$ such that*

$$\text{FF-DBF}(\tau, L(s), s) > (m - (m-1)s)L(s) .$$

Proof Follows by chaining the lower bound on W of Observation 19.1 with the upper bound of Observation 19.2, with $L(s) \leftarrow (t_0 - t_k)$. □

Lemma 19.2 *(DM) Suppose that constrained-deadline sporadic task system τ is not schedulable by global DM upon m unit-speed processors. For each $s, s \geq dens_{\max}(\tau)$, there is an interval-length $L(s)$ such that*

$$\text{FF-DBF}(\tau, L(s), s) > (m - (m-1)s)\frac{L(s)}{2} .$$

Proof Follows by chaining the lower bound on W of Observation 19.1 with the upper bound of Observation 19.3, with $L(s) \leftarrow 2(t_0 - t_k)$. □

Sufficient Schedulability Conditions The contrapositive of Lemma 19.1 above represents a global EDF schedulability condition: any task system τ satisfying

$$\Big(\exists\sigma : \sigma \geq \text{dens}_{\max}(\tau) : \big(\forall\Delta : \Delta \geq 0 : \big(\text{FF-DBF}(\tau, \Delta, \sigma) \leq (m - (m-1)\sigma) \times \Delta\big)\big)\Big) \tag{19.4}$$

is EDF-schedulable upon m unit-speed processors.

Similarly, the contrapositive of Lemma 19.2 represents a global DM schedulability condition: any task system τ satisfying

$$\Big(\exists\sigma : \sigma \geq \text{dens}_{\max}(\tau) : \big(\forall\Delta : \Delta \geq 0 : \big(\text{FF-DBF}(\tau, \Delta, \sigma) \leq (m - (m-1)\sigma) \times \frac{\Delta}{2}\big)\big)\Big) \tag{19.5}$$

is DM-schedulable upon m unit-speed processors.

19.3 Determining Processor Speedup Factor

We now determine, in Theorems 19.1 and 19.2 below, the processor speedup factors of schedulability tests for EDF and DM, respectively, based on checking the sufficient schedulability conditions derived in Sect. 19.2 above.

In [24] (Theorem 2), necessary conditions for global multiprocessor schedulability were identified; these necessary conditions generalize to our context and notation in the following manner:

Lemma 19.3 *If $\text{FF-LOAD}(\tau, \sigma) > m\sigma$ then τ is not feasible on m speed-σ processors.*

Proof Suppose that FF-LOAD$(\tau,\sigma) > m\sigma$.

Let t_0 denote a value for t that maximizes the RHS of Eq. 19.3, and hence determines the value of FF-LOAD(τ,σ):

$$\text{FF-LOAD}(\tau,\sigma) = \Big(\sum_{\tau_\ell \in \tau} \text{FF-DBF}(\tau_\ell, t_0, \sigma)\Big)/t_0.$$

By definition of FF-DBF, each task τ_ℓ can generate a sequence of jobs that together require $\geq$ FF-DBF(τ_ℓ, t_0, σ) units of execution over some interval of length t_0, when executing upon speed-σ processors. Since the different tasks of a sporadic task system are assumed to be independent of each other, such intervals for the different tasks can be aligned; the total execution requirement by all the tasks over the aligned interval is

$$\begin{aligned}
&\geq \sum_{\tau_\ell \in \tau} \text{FF-DBF}(\tau_\ell, t_0, \sigma)\\
&= \text{FF-LOAD}(\tau,\sigma) \times t_0 \ \ \text{(By definition of } t_0\text{)}\\
&> m\sigma t_0 \ \ \text{(Since FF-LOAD}(\tau,\sigma) \text{ is assumed to be } > m\sigma\text{)}.
\end{aligned}$$

But $m\sigma t_0$ denotes the total computing capacity over an interval of size t_0 upon m speed-σ processors. We therefore conclude that the total execution requirement by all the tasks in τ over the interval cannot be met, and some deadline must necessarily be missed. □

Lemma 19.4 *If task system τ does not satisfy Condition 19.4, then it is not feasible upon a platform comprised of m speed-$\frac{m}{2m-1}$ processors.*

Proof First, any τ not satisfying Condition 19.4 that has $\text{dens}_{\max}(\tau) > m/(2m-1)$ is trivially not feasible on speed-$\frac{m}{2m-1}$ processors.

Next, consider τ with $\text{dens}_{\max}(\tau) \leq m/(2m-1)$. Suppose that τ does not satisfy Condition 19.4: for all values of $\sigma > \text{dens}_{\max}(\tau)$, there is an interval-length $\Delta \geq 0$ such that FF-DBF$(\tau, \Delta, \sigma) > (m-(m-1)\sigma) \times \Delta$.

Let us instantiate this inequality for $\sigma \leftarrow m/(2m-1)$:

$$\begin{aligned}
\forall\, \Delta \geq 0 \ :\ &\text{FF-DBF}(\tau, \Delta, \frac{m}{2m-1})\\
&\qquad > \Big(m-(m-1)\frac{m}{2m-1}\Big) \times \Delta\\
\Rightarrow\ &\text{FF-LOAD}(\tau, \frac{m}{2m-1}) > m-(m-1)\frac{m}{2m-1}\\
\Leftrightarrow\ &\text{FF-LOAD}(\tau, \frac{m}{2m-1}) > \frac{2m^2-m-m^2+m}{2m-1}\\
\Leftrightarrow\ &\text{FF-LOAD}(\tau, \frac{m}{2m-1}) > m\frac{m}{2m-1}.
\end{aligned}$$

It therefore follows, from Lemma 19.3, that τ is not feasible upon a platform comprised of m speed-$\frac{m}{2m-1}$ processors. □

By taking the contrapositive of Lemma 19.4 above and observing that $1/(\frac{m}{2m-1})$ is equal to $(2 - \frac{1}{m})$, we have

Theorem 19.1 *The processor speedup factor of an* EDF *schedulability test based on checking Condition 19.4 has a processor speedup factor of* $(2-\frac{1}{m})$ *on an m-processor platform.*

Reasoning very similar to that used in Lemma 19.4 and Theorem 19.1 above are now applied to DM scheduling:

Lemma 19.5 *If task system τ fails the DM schedulability test of Condition 19.5, then it is not feasible upon a platform comprised of m speed-$\frac{m}{3m-1}$ processors.*

Proof Suppose that τ fails the schedulability test of Condition 19.5.

If $\text{dens}_{\max}(\tau) > (m/(3m - 1)$ then τ is trivially not feasible on a platform comprised of (any number of) speed-$(m/(3m - 1)$ processors, and we are done.

Assume now that $\text{dens}_{\max}(\tau) \le (m/(3m - 1)$. Suppose that τ does not satisfy Condition 19.5: for all values of $\sigma > \text{dens}_{\max}(\tau)$, there is an interval-length $\Delta \ge 0$ such that $\text{FF-DBF}(\tau, \Delta, \sigma) > (m - (m - 1)\sigma) \times \frac{\Delta}{2}$.

Let us instantiate this inequality for $\sigma \leftarrow m/(3m - 1)$:

$$\begin{aligned}
\forall\, \Delta \ge 0\ :\ & \text{FF-DBF}(\tau, \Delta, \frac{m}{3m-1}) \\
& \qquad > \Big(m - (m-1)\frac{m}{3m-1}\Big) \times \frac{\Delta}{2} \\
\Rightarrow\ & \text{FF-LOAD}(\tau, \frac{m}{3m-1}) > (m - (m-1)\frac{m}{3m-1}) \times \frac{1}{2} \\
\Leftrightarrow\ & \text{FF-LOAD}(\tau, \frac{m}{3m-1}) > \frac{3m^2 - m - m^2 + m}{2(3m-1)} \\
\Leftrightarrow\ & \text{FF-LOAD}(\tau, \frac{m}{3m-1}) > m\frac{m}{3m-1}.
\end{aligned}$$

It therefore follows from Lemma 19.3 above that τ is not feasible upon a platform comprised of m speed-$\frac{m}{3m-1}$ processors. □

By taking the contrapositive of Lemma 19.4 above and observing that $1/(\frac{m}{3m-1})$ is equal to $(3 - \frac{1}{m})$, we have Theorem 19.2

Theorem 19.2 *The processor speedup factor of a DM schedulability test based on checking Condition 19.5 has a processor speedup factor of* $(3-\frac{1}{m})$ *on an m-processor platform.*

19.4 Improved Schedulability Tests

Condition 19.4 is the contrapositive of the EDF unschedulability condition of Lemma 19.1; it states that in order to show that a task system τ is EDF-schedulable, it suffices to demonstrate that there exists a $\sigma \ge \text{dens}_{\max}(\tau)$ such that

$$\text{FF-DBF}(\tau, t, \sigma) \le (m - (m-1)\sigma) \times t \tag{19.6}$$

for all values of $t \ge 0$. Such a σ is called a *witness* to the EDF-schedulability of τ. In order to obtain a schedulability test with the optimal processor speedup factor of $(2 - (1/m))$, we have seen (Theorem 19.1 above) that we need only consider $\sigma \leftarrow m/(2m-1)$ as a potential witness, declaring the task set as not being EDF schedulable if this value of σ fails to satisfy Inequality 19.6 for all $t \ge 0$.

However, by testing only one out of all the values of σ that could bear witness to a task system's schedulability, this test clearly fails to make full use of the insight into EDF-schedulability that Lemma 19.1 affords us. We will derive an algorithm that fully exploits the insight of Lemma 19.1, by correctly identifying all task systems for which *any* σ would cause Inequality 19.6 to evaluate to true for all $t \ge 0$.

(We note that a similar exercise may be conducted for the DM schedulability test derived from Condition 19.5, the contrapositive of the DM unschedulability condition of Lemma 19.2. The steps are essentially identical to the ones for EDF as described below; hence, we will not describe the details for DM schedulability analysis here.)

Note that there are infinitely many different values of σ that could potentially be witnesses to the EDF-schedulability of a task system. For each such potential witness, there are infinitely many values of t for which it must be validated that Inequality 19.6 is satisfied. Two questions must therefore now be answered:

Q1: What values of σ need to be considered as potential witnesses to the EDF-schedulability of τ?

Q2: In order to determine whether a particular σ is indeed a witness or not, for which values of t does Condition 19.6 need to be evaluated?

We address the second of these questions first, in Sect. 19.4.1 below; the first question is addressed in Sect. 19.4.2.

19.4.1 Bounding the Range of Time-Values that Must Be Tested

We now address Q2, the second of the questions listed above: *for a given value of σ, for which values of t must we validate Condition 19.6 in order to be able to conclude that it holds for all t?*

Claim 19.1 For a given σ and τ, if Condition 19.6 is violated for any t then it is violated for some t in

$$\bigcup_{\tau_i \in \tau} \left\{ kT_i + D_i, kT_i + D_i - \min(C_i/\sigma, D_i) \mid k \in \mathbf{N} \right\} \tag{19.7}$$

Proof Sketch: This follows from the observation (also see Fig. 19.2) that $\text{FF-DBF}(\tau_i, t, \sigma)$ increases with a constant slope between $kT_i + D_i - \min(C_i/\sigma, D_i)$

and $kT_i + D_i$, and remains unchanged elsewhere, for all $k \in \mathbf{N}$. Hence the LHS of Condition 19.6 increases with constant slope with increasing t between two consecutive t's in this set; since the RHS also increases with constant slope with increasing t, it is guaranteed that if this condition is violated at some $\tilde{t}$ it will be violated at one of the two t's in this set that neighbor $\tilde{t}$. □

Claim 19.1 above tells us that we need evaluate Condition 19.6 for only countably many t's; Claims 19.2 and 19.3 below allow us to bound the actual number.

Claim 19.2 If Condition 19.6 is violated at some t for a given τ and m, and a particular $\sigma \le (m - U(\tau))/(m-1)$, then it is violated at some t no larger than $P(\tau)$.

Proof Recall that $P(\tau)$ denotes the hyperperiod—the least common multiple of the task period parameters—of τ.

Since for all τ_i the period T_i divides the hyperperiod $P(\tau)$, it follows from Eq. 19.1 (also see Fig. 19.2) that

$$\begin{aligned}\text{FF-DBF}(\tau, t + P(\tau), \sigma) &= P(\tau) \times U(\tau) + \text{FF-DBF}(\tau_i, t, \sigma)\\ &\le \; P(\tau) \times (m - (m-1)\sigma) + \text{FF-DBF}(\tau_i, t, \sigma)\end{aligned}$$

(since we are assuming that $\sigma \le (m - (U(\tau))/(m-1))$. Hence if Condition 19.6 is to be violated for some $t_v > P(\tau)$, it will also be violated for $t_v \bmod P(\tau)$. □

Claim 19.2 tells us that $P(\tau)$ is an upper bound on the values of t for which Condition 19.6 needs to be evaluated. Claim 19.3 below provides another upper bound.

Claim 19.3 If Condition 19.6 is violated at some t for a given τ and m, and a particular $\sigma \le (m - U(\tau))/(m-1)$, then t is no larger than

$$\frac{\sum_{\tau_i \in \tau} C_i}{m - (m-1)\sigma - U(\tau)}. \tag{19.8}$$

Proof We first observe that it directly follows from the definition of FF-DBF (Eq. 19.1, also see Fig. 19.2) that for all $t \ge D_i$ and for all σ,

$$\text{FF-DBF}(\tau_i, t, \sigma) \le \left(\frac{t}{T_i} + 1\right) C_i.$$

Suppose that $\text{FF-DBF}(\tau, t, \sigma) > (m - (m-1)\sigma)t$ for some $t > \max_{\tau_i \in \tau}\{D_i\}$. We then have

$$\begin{aligned}&\text{FF-DBF}(\tau, t, \sigma) > (m - (m-1)\sigma)t\\ &\Rightarrow \sum_{\tau_i \in \tau}\left(t\frac{C_i}{T_i} + C_i\right) > (m - (m-1)\sigma)t\\ &\Leftrightarrow tU(\tau) + \sum_{\tau_i \in \tau} C_i > (m - (m-1)\sigma)t\end{aligned}$$

$$\Leftrightarrow t < \frac{\sum_{\tau_i \in \tau} C_i}{m - (m-1)\sigma - U(\tau)}$$

and the lemma is proved. □

Testing Set For a given σ and τ, let $TS(\tau, \sigma)$ denote the *testing set* of values of t that lie in the set defined in Eq. 19.7 and are no larger than both $P(\tau)$ and the bound defined by Eq. 19.8.

How large can this testing set be? As shown in Claim 19.2, we need not consider any t exceeding the hyperperiod $P(\tau)$. It is easily seen that there are at most exponentially many points in the set defined in Eq. 19.7 not exceeding $P(\tau)$; hence, the testing set contains at most exponentially many points.

Suppose, however, that we were to enforce an additional restriction that we would not consider any σ greater than

$$\big(m - U(\tau) - \epsilon\big)/(m-1) \tag{19.9}$$

where ϵ is an arbitrarily small positive *constant*.

It would then follow from Inequality (19.8) that the upper bound on the values in $TS(\tau, \sigma)$ is guaranteed to be $\le (\sum_{\tau_i \in \tau} C_i)/\epsilon$, which is pseudopolynomial in the representation of the task system τ. Thus, this restriction immediately yields a pseudopolynomial upper bound on the number of elements in $TS(\tau, \sigma)$.

The consequence of enforcing the restriction of Eq. 19.9 above is that the test we develop is no longer able to identify all task systems satisfying Condition 19.4: task systems that only satisfy Condition 19.4 for values of σ in $(m - U(\tau) - \epsilon)/(m - 1)\ \epsilon, (m - U(\tau))/(m - 1)]$ would not be identified by our test. In exchange for this slight loss of optimality (the degree of which can be controlled by choosing ϵ to be appropriately small), we would restrict the size of the testing set to be pseudopolynomial.

19.4.2 Choosing Potential Witnesses to Test

We next address Q1, the first of the two questions listed earlier in this section. That is, we set about restricting the candidate field of σ's that need be tested as potential witnesses to the EDF-schedulability of τ.

Claim 19.4 No value of σ that is greater than $(m - U(\tau))/(m - 1)$ can possibly result in Condition 19.6 evaluating to true for all values of t (and hence, no such value of σ can attest to the EDF-schedulability of τ).

Proof Observe that FF-DBF(τ_i, t, σ) asymptotically approaches $t \times (C_i/T_i)$ as $t \to \infty$. Hence FF-DBF(τ, t, σ) asymptotically approaches $t \times U(\tau)$ with increasing t. In order to have FF-DBF$(\tau, t, \sigma) \le (m - (m-1)\sigma)t$ for all t, therefore, we need

$$U(\tau) \le m - (m-1)\sigma$$

$$\Leftrightarrow \sigma \leq \frac{m - U(\tau)}{m - 1} .$$

□

As a consequence of Claim 19.4 above, we can restrict the range of values for σ that are potential witnesses to the EDF-schedulability of τ. However, there are still infinitely many distinct values in this range, and we clearly cannot exhaustively check all these infinitely many values. Fortunately, it so happens that we can restrict the actual number of values of σ within this range that need be considered as potential witnesses to the EDF-schedulability of τ, as we will now show.

Let us suppose that we have identified a particular σ_{cur}, such that we know that no $\sigma < \sigma_{\text{cur}}$ can possibility bear witness to the EDF-schedulability of τ. Suppose that we then test σ_{cur}, and determine that it is not a witness to the EDF-schedulability of τ either. Let t_{cur} denote a value of t that causes Condition 19.6 to evaluate to false when $\sigma \leftarrow \sigma_{\text{cur}}$. Let σ_{new} denote the smallest value of $\sigma > \sigma_{\text{cur}}$ such that Condition 19.6 evaluates to true with $(\sigma \leftarrow \sigma_{\text{new}}; t \leftarrow t_{\text{cur}})$. (We describe below, in Sect. 19.4.4, how the value of σ_{new} is computed.) It is clear that t_{cur} rules out the possibility of any $\sigma \in [\sigma_{\text{cur}}, \sigma_{\text{new}})$ bearing witness to the EDF-schedulability of τ; hence, the *next* value of σ that we will need to consider is σ_{new}.

So we have seen how, if we know a constant σ_{cur} such that no $\sigma \leq \sigma_{\text{cur}}$ can be a witness to the EDF-schedulability of τ, we can determine the next potential witness σ_{new} that we must consider. Claim 19.5 below tells us that in considering σ_{new}, we need not revisit values of $\text{TS}(\tau, \sigma_{\text{new}})$ that are $\leq t_{\text{cur}}$:

Claim 19.5 Suppose that τ is not EDF-schedulable. By Lemma 19.1, it must be the case for each $s \geq \text{dens}_{\max}(\tau)$ there is an $L(s)$ such that
$\text{FF-DBF}(\tau, L(s), s) > (m - (m-1)s) \times L(s)$.
Consider some s_1 and $t_1 \leq L(s_1)$ such that

$$\text{FF-DBF}(\tau, t_1, s_1) > (m - (m-1)s_1) \times t_1 .$$

For any $s_2 > s_1$, there is a t_2, $t_1 \leq t_2 \leq L(s_2)$ such that

$$\text{FF-DBF}(\tau, t_2, s_2) > (m - (m-1)s_2) \times t_2 .$$

Proof The claim would follow if we were able to show that $L(s_1) \leq L(s_2)$; since the claim assumes that $t_1 \leq L(s_1)$, setting $t_2 \leftarrow L(s_2)$ would then bear witness to its correctness.

That $L(s_1)$ is no larger than $L(s_2)$ follows from the observation that the jobs $j_1, j_2, \ldots$ that are defined according to the pseudocode given in Sect. 19.2 for a given value of s (say, $s \leftarrow s_1$), are also valid for larger values of s (say, $s \leftarrow s_2$). This is easily shown by induction. Assume that $j_1, \ldots, j_{i-1}$ as defined for $s \leftarrow s_1$ are valid for $s \leftarrow s_2$: this implies that t_{i-1} is the same when $s \leftarrow s_1$ and $s \leftarrow s_2$. The job j_i as defined for $s \leftarrow s_1$ has executed for less than $(t_{i-1} - t_i)s_1$ prior to t_{i-1}. But since $s_2 > s_1$, it has also executed for less than $(t_{i-1} - t_i)s_2$ prior to t_{i-1}, and hence satisfies the condition to be considered as job j_i for $s \leftarrow s_2$ as well. □

19.4.3 *Putting the Pieces Together: The* EDF *Schedulability Test*

We are now ready to put the pieces together, and specify our schedulability test. This schedulability test is a methodical quest for a value of σ for which there is no t causing Condition 19.6 to evaluate to false (and which is thus a witness to the EDF-schedulability of τ). Based on Claim 19.5 above, we will start out testing a small value for σ; if this fails, we can try a larger value for σ and use the result of Claim 19.5 to trim the set of potential values of t that need to be tested for this larger value of σ. In greater detail, our algorithm is the following.

S1 Let σ_{cur} denote the value of σ currently being evaluated (i.e., the potential witness currently under consideration).
This is initialized as follows: $\sigma_{\text{cur}} \leftarrow \text{dense}_{\max}(\tau)$.
We will also use an additional variable t_{cur}, initialized to zero: $t_{\text{cur}} \leftarrow 0$.

S2 If σ_{cur} is larger than $(m - U(\tau) - \epsilon)/(m-1)$ where ϵ is an arbitrarily small positive constant that has been a priori determined, then we exit the test, having failed to show τ is EDF schedulable. (Here, we are using the result shown in Claim 19.4, modified as discussed in Eq. 19.9 to yield a testing set of pseudopolynomial size, to restrict the range of values of σ that we need test as potential witnesses to the EDF schedulability of τ.) Otherwise by the results of Sect. 19.4.1, we need only evaluate Condition 19.6 for values of $t \in \text{TS}(\tau, \sigma_{\text{cur}})$ to determine whether it is satisfiable or not.
We begin at the smallest value in $\text{TS}(\tau, \sigma_{\text{cur}})$ that is greater than t_{cur}, and consider the values in $\text{TS}(\tau, \sigma_{\text{cur}})$ in increasing order.
If no value of t in $\text{TS}(\tau, \sigma_{\text{cur}})$ causes Condition 19.6 to evaluate to false for this current value of σ_{cur}, then we exit the test, having succeeded in showing that τ is EDF schedulable.

S3 Suppose, however, that there *is* some value of t that causes Condition 19.6 to evaluate to false for this value of σ_{cur}. Assign t_{cur} this value of t.
By Claim 19.5, if τ is not EDF schedulable then for all values of $\sigma > \sigma_{\text{cur}}$ there is some $t \geq t_{\text{cur}}$ which causes Condition 19.6 to evaluate to false.
Let σ_{new} denote the smallest value of $\sigma' > \sigma_{\text{cur}}$, such that

$$\text{FF-DBF}(\tau, t_{\text{cur}}, \sigma') \leq (m - (m-1)\sigma') \times t_{\text{cur}} \; .$$

We compute σ_{new} using the technique described in Sect. 19.4.4 below, assign σ_{cur} this value σ_{new}, and go to Step S2.

Computational Complexity Observe that the values assigned to the variable t_{cur} during the above algorithm are monotonically increasing—once we assign t_{cur} a particular value, we never assign it a smaller value even after we have changed the value assigned to σ_{cur}. This observation can be used to show that the total number of values assigned to t_{cur} is no more than the cardinality of $TS(\tau, \sigma)$, for the largest value of σ that is tested. And we have seen in Sect. 19.4.1 that this number is pseudopolynomially bounded in the representation of the task system τ. We will see in Sect. 19.4.4 below that σ_{new} can be computed in polynomial time, while the rest of the processing above for a given value of t_{cur} is easily seen to also take polynomial time. This yields the following result:

Theorem 19.3 *This* EDF *schedulability test has pseudopolynomial time complexity.*

19.4.4 Computing σ_{new}

It remains to specify how the the value of σ_{new} is determined in the algorithm described above. That is, given *fixed* values for t_{cur} and σ_{cur} such that

$$\text{FF-DBF}(\tau, t_{\text{cur}}, \sigma_{\text{cur}}) > (m - (m-1)\sigma_{\text{cur}}) \times t_{\text{cur}} \ ,$$

we are to compute σ_{new}, the smallest $\sigma' > \sigma_{\text{cur}}$ such that

$$\text{FF-DBF}(\tau, t_{\text{cur}}, \sigma') \le (m - (m-1)\sigma') \times t_{\text{cur}} \ .$$

Let us examine how FF-DBF$(\tau_i, t_{\text{cur}}, \sigma)$ changes as σ is increased in the neighborhood of σ_{cur}, while the task τ_i and the interval-length t_{cur} are kept unchanged.

From Eq. 19.1, we know that FF-DBF$(\tau_i, t_{\text{cur}}, \sigma)$ depends on q_i and r_i,

where $q_i = \lfloor t_{\text{cur}}/T_i \rfloor$ and $r_i = t_{\text{cur}} \bmod T_i$. Notice that the values of q_i and r_i do not depend on σ. Hence,

(a) FF-DBF$(\tau_i, t_{\text{cur}}, \sigma)$ does not vary with σ if $r_i \ge D_i$ (the first case in Eq. 19.1).
(b) It varies linearly with σ while $D_i > r_i \ge D_i - \frac{C_i}{\sigma}$. That is, if $r_i < D_i$ then FF-DBF$(\tau_i, t_{\text{cur}}, \sigma)$ decreases linearly with increasing σ while $\sigma \le C_i/(D_i - r_i)$. This is the second case in Eq. 19.1.
(c) Once σ increases such that it is $> C_i/(D_i - r_i)$, FF-DBF$(\tau_i, t_{\text{cur}}, \sigma)$ remains unchanged with further increase in the value of σ. This is the third case in Eq. 19.1.

In order to compute σ_{new} given values for t_{cur} and σ_{cur}, we would therefore

L1 Classify each task τ_i as being either in class ***(a)***, ***(b)***, or ***(c)*** according to the above classification. That is, a task τ_i for which $r_i \ge D_i$ would be classified as being in class ***(a)***; one with $r_i < D_i$ and $r_i \ge D_i - \frac{C_i}{\sigma_{\text{cur}}}$ would be classified as being in class ***(b)***; while one with $r_i < D_i - \frac{C_i}{\sigma_{\text{cur}}}$ would be classified as being in class ***(c)***.
For each task τ_i in class ***(b)***, let $\hat{\sigma}_i \stackrel{\text{def}}{=} C_i/(D_i - r_i)$. Increasing σ to make it greater than $\hat{\sigma}_i$ would cause τ_i to no longer be a class ***(b)*** task.
L2 Observing that only tasks in class ***(b)*** have their FF-DBF's change—linearly—with changing σ, we can set up and solve a linear equation to determine σ_o, the smallest $\sigma' > \sigma_{\text{cur}}$ for which

$$\text{FF-DBF}(\tau, t_{\text{cur}}, \sigma_o) = (m - (m-1)\sigma_o) \times t_{\text{cur}} \ .$$

L3 If this computed value of σ_o is $\le \hat{\sigma}_i$ for all tasks τ_i in class ***(b)***, then we have computed the desired value for σ_{new}. Else,

a) assign σ_{cur} the value of the smallest $\hat{\sigma}_i$ from among all those computed for tasks in class ***(b)***, and
b) repeat from step L1 above.

Computational Complexity It is not difficult to see that σ_{new} can be computed in time polynomial in the representation of τ. This follows from the observations that

- Steps L1, L2, and L3 above each take polynomial time.
- During each iteration of the three-step process L1–L3 either (i) we determine the value of σ_{new} and exit; or (ii) at least one task that was in class ***(b)*** will henceforth be placed in class ***(c)*** in the subsequent iteration, whereas no additional tasks become class ***(b)*** tasks. Thus, the number of iterations of L1–L3 is bounded from above by the number of tasks initially in class ***(b)***, which is, of course, itself bounded by the number of tasks in τ.

Sources

The forced-forward demand bound function was proposed in [63] as a refinement to the demand bound function; ideas leading up to it were previously suggested in [24]. The speedup bounds for EDF and DM presented here were derived in [63]. The EDF schedulability test we describe here was first presented in [36, 37].

Chapter 20
Global Dynamic Priority Scheduling

In this chapter, we briefly describe the research that had been conducted in the dynamic-priority scheduling of systems of three-parameter sporadic tasks. In contrast to the situation with respect to Liu and Layland task systems (Chap. 7) where there is a very large body of work to discuss, the research on dynamic priority (DP) scheduling of three-parameter sporadic task systems is rather sparse. The one scheduling algorithm that has been explored in some detail—earliest deadline zero laxity (EDZL) (discussed in Sect. 20.2)—is "almost" a fixed job priority (FJP) algorithm in a sense that will become clearer upon reading Sect. 20.2, and hence most of the analyses conducted on EDZL are similar to the analyses seen in earlier chapters regarding earliest-deadline-first (EDF).

20.1 Least Laxity Scheduling

The *laxity* of a job at any instant in time in a schedule is defined to be its deadline minus the sum of its remaining processing time and the current time. The Least-Laxity-scheduling algorithm assigns greater priority to jobs with smaller laxity; since the laxities of jobs may change during run-time (the laxity a job that is executing remains the same, while that of a non-executing job increases), Least Laxity (LL) is a dynamic priority (DP) scheduling algorithm. It is known that LL is optimal upon preemptive uniprocessors in the same sense that EDF is (see Sect. 4.1). Although the results of Dertouzos and Mok [77] rule out the possibility of LL being optimal upon multiprocessors, it was generally believed (see, e.g., [133]) that LL is strictly superior to EDF for scheduling sporadic task systems upon preemptive multiprocessors. However, this was shown to not be true by Kalyanasundaram et al. [117]; this, in addition to the fact that it seems very difficult to obtain efficient implementations of LL schedulers, has resulted in this algorithm falling out of favor.

20.2 EDZL Scheduling

A somewhat more promising application of laxity-based scheduling is to be found in the EDZL scheduling algorithm [125]. The EDZL scheduling algorithm combines EDF with laxity-based scheduling: Jobs are initially assigned priorities according to their deadlines, but if the laxity of any job becomes equal to zero it gets promoted to the highest priority level. It is easy to show that EDZL strictly dominates EDF; since any job that has its laxity become equal to zero must immediately begin executing and continue to do so until it completes if it is to be guaranteed to complete by its deadline, the only cases where EDZL makes a scheduling decision different from that made by EDF is if EDF could have missed a deadline. This dominance result implies that every sufficient EDF-schedulability test is also an EDZL sufficient schedulability test.

In addition, Piao et al. proved in [156] the following utilization bound for EDZL-scheduled implicit deadline systems:

$$U_{\text{sum}}(\tau) \leq \frac{m+1}{2}, \tag{20.1}$$

and Baker et al. [25, 71] have derived sufficient schedulability conditions for EDZL-scheduled task systems with deadlines different from periods. We will not discuss these results in this book, but refer the interested reader to [25] for details.

The dominance of EDZL over EDF also means that EDZL exhibits good preemption and migration properties despite not being a priority-driven algorithm—Definition 3.3—for which, as discussed in Sect. 2.3.3, it is possible to bound the number of preemptions and migrations. Specifically, an EDZL-generated scheduled has no more preemptions and migrations than a correct EDF schedule; additionally, it can be shown that since any individual job has its priority changed only once—when/if it becomes a zero-laxity job—an EDZL schedule will have at most one additional preemption and migration per job.

Chapter 21
The Sporadic DAG Tasks Model

The Liu and Layland task model and the three-parameter sporadic task model both assume that there is a single thread of execution within each task; they do not allow for the modeling of parallelism within individual tasks. This was not seen as a shortcoming of the models when they were first being developed, since both were proposed in the context of uniprocessor real-time systems for which the presence of just a single processor ruled out parallel execution.

However, the trend toward implementing real-time systems upon multiprocessor and multicore platforms has given rise to a need for models that are capable of exposing any possible parallelism that may exist within the workload, to the scheduling mechanism. There has therefore recently been a move toward developing new models that allow for the representation of partial parallelism within a task itself, as well as for precedence dependencies between different parts of each individual task. Earlier models of this form include the moldable tasks model [146], the malleable tasks model [72], the fork-join or parallel synchronous task model [120], etc. The sporadic DAG tasks model [38], which we had briefly described informally in Sect. 2.1.3, generalizes these earlier models. In this chapter, we will formally define the sporadic directed acyclic graph (DAG) tasks model, and seek to obtain an intuitive understanding of the kinds of workload parallelism that can be modeled using this task model. We will study the schedulability of systems of sporadic DAG tasks, both for the special case of implicit-deadline systems and for more general task systems.

21.1 The Sporadic DAG Tasks Model

As we had stated in Sect. 2.1.3, each recurrent task in the sporadic DAG tasks model is modeled as a DAG $G_i = (V_i, E_i)$. Each vertex $v \in V_i$ of the DAG corresponds to a sequential job, and is characterized by a worst-case execution time (WCET) e_v. Each (directed) edge of the DAG represents a precedence constraint: If (v, w) is a (directed) edge in the DAG then the job corresponding to vertex v must complete execution before the job corresponding to vertex w may begin execution. Groups of jobs that are not constrained (directly or indirectly) by precedence constraints in such a manner may execute in parallel if there are processors available for them to do so.

The task is further characterized by a (relative) deadline parameter D_i and a period T_i. The interpretation of these parameters is as follows. We say the task G_i releases a dag-job at time-instant t when it becomes available for execution. When this happens, we assume that all $|V_i|$ of the jobs become available for execution simultaneously, subject to the precedence constraints. During any given run the task may release an unbounded sequence of dag-jobs; all $|V_i|$ jobs that are released at some time-instant t must complete execution by time-instant $t + D_i$. A minimum interval of duration T_i must elapse between successive releases of dag-jobs.

More formally, in the *sporadic DAG* model a task τ_i is specified as a three-tuple (G_i, D_i, T_i), where G_i is a DAG, and D_i and T_i are positive integers.

- The DAG G_i is specified as $G_i = (V_i, E_i)$, where V_i is a set of vertices and E_i a set of directed edges between these vertices (it is required that these edges do not form any cycle). Each $v \in V_i$ denotes a sequential operation (a "job"). Each job $v \in V_i$ is characterized by a WCET $e_v \in \mathbf{N}$. The edges represent dependencies between the jobs: if $(v_1, v_2) \in E_i$ then job v_1 must complete execution before job v_2 can begin execution. (We say a job becomes *available*—i.e., eligible to execute—once all its predecessor jobs have completed execution.)
- A *period* $T_i \in \mathbf{N}$. A *release* or arrival of a dag-job of the task at time-instant t means that all $|V_i|$ jobs $v \in V_i$ are released at time-instant t. The period denotes the minimum amount of time that must elapse between the release of successive dag-jobs: If a dag-job is released at t, then the next dag-job may not be released prior to time-instant $t + T_i$.
- A *deadline* $D_i \in \mathbf{N}$. If a dag-job is released at time-instant t then all $|V_i|$ jobs that were released at t must complete execution by time-instant $t + D_i$.

Analogously with three-parameter sporadic tasks, we may define a system τ of sporadic DAG tasks to be *implicit-deadline* if $D_i = T_i$ for all $\tau_i \in \tau$, *constrained-deadline* if $D_i \leq T_i$ for all $\tau_i \in \tau$, and *arbitrary-deadline* otherwise.

If $D_i > T_i$ for some task τ_i in an arbitrary-deadline system, then the task may release a dag-job prior to the completion of all jobs of the previously-released dag-jobs—the model does not require that all jobs of a dag-job complete execution before jobs of the next dag-job can start executing.

Some additional notation and terminology:

- A *chain* in the sporadic DAG task $\tau_i = (G_i, D_i, T_i)$ is a sequence of nodes $v_1, v_2, \ldots, v_k$ in G_i such that (v_i, v_{i+1}) is an edge in G_i, $1 \leq i < k$. The length of this chain is defined to be the sum of the WCETs of all its nodes: $\sum_{i=1}^{k} e_{v_k}$.
- len_i denotes the length of the longest chain in G_i. We point out that len_i can be computed in time linear in the number of vertices and the number of edges in G_i, by first obtaining a topological sorting of the vertices of the graph and then running a straightforward dynamic program.
- $\text{vol}_i = \sum_{v \in V_i} e_v$ denotes the total WCET of each dag-job of the task τ_i. Note that vol_i can be computed in time linear in the number of vertices in G_i.
- The *density* of task τ_i is defined as $\text{dens}_i \overset{\text{def}}{=} \text{len}_i / D_i$ and its *utilization* as $u_i \overset{\text{def}}{=} \text{vol}_i / T_i$.

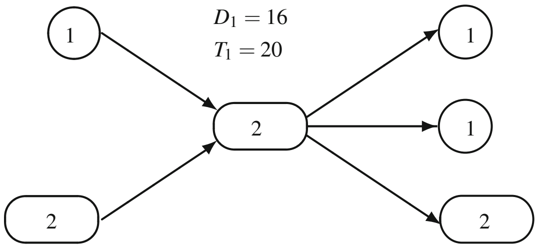


Fig. 21.1 Example sporadic DAG task τ_1. Vertices are labeled with the WCET's of the jobs they represent (the WCET is also indicated by the size of the vertex)

- For a DAG sporadic task system τ we define its *maximum density* $\text{dens}_{\max}(\tau)$ to be the largest density of any task in τ: $\text{dens}_{\max}(\tau) = \max_{\tau_i \in \tau}\{\text{dens}_i\}$; its *utilization* $U(\tau)$ to be the sum of the utilizations of all the tasks in τ: $U_{\text{sum}}(\tau) = \sum_{\tau_i \in \tau} u_i$; and its *hyperperiod* $P(\tau)$ to be the least common multiple of the period parameters of all the tasks in τ: $P(\tau) = \text{lcm}_{\tau_i \in \tau}\{T_i\}$.

Example 21.1 An example sporadic DAG task τ_1 is depicted graphically in Fig. 21.1. The DAG G_i for this task consists of six vertices and five directed edges denoting precedence constraints. The longest chain is of length $\text{len}(G_i) = 6$; by summing all the WCET's, we can see that $\text{vol}(G_i) = 9$. Also $\text{dens}_1 = 6/16 = 0.375$ and $u_1 = 9/20 = 0.45$.

21.1.1 Parallelism in the DAG

The sporadic DAG task model generalizes sequential recurrent task models such as the Liu and Layland model [139] or the three-parameter model [149] in the sense that each task may generate work that can be simultaneously executed upon more than one processor. We now seek to characterize the parallelism of a DAG task. Let us suppose that we had an unbounded number of unit-speed processors available to us, and let us consider the schedule of a dag-job of τ_i upon these processors. Let $\lambda_{i,0} = 0, \lambda_{i,1}, \ldots, \lambda_{i,k_i} = \text{len}_i$ denote the instants at which the number of processors used changes—this reflects a change in the degree of parallelism of the dag-job. It is evident that k_i is no larger than the number of vertices $|V_i|$ in the DAG. For $0 \leq j < k_i$, let $\nu_{i,j}$ denote the number of processors used (and hence, the degree of parallelism) over the duration $[\lambda_{i,j}, \lambda_{i,j+1}]$. For notational convenience, let $\nu_{i,k_i} = 0$.

Example 21.2 Fig. 21.2 illustrates the parallelism within each dag-job of the example sporadic DAG task τ_1 of Fig. 21.1.

The DAG has two source nodes (i.e., nodes with no predecessors); these may both execute immediately the dag-job is released. Therefore, $\nu_{1,0} = 2$. One of these completes upon having executed for one unit (thus, $\lambda_{1,1} = 1$); however, its successor

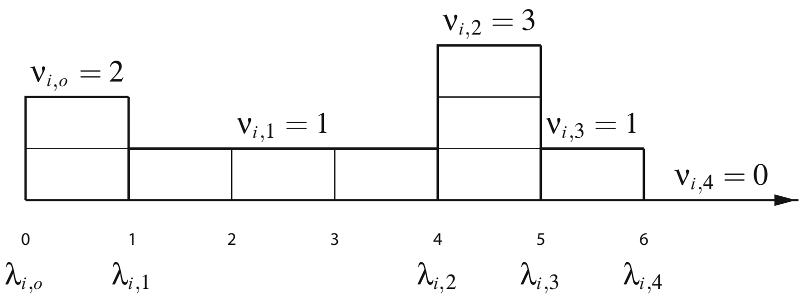


Fig. 21.2 The parallelism within each dag-job of the task τ_1 of Fig. 21.1

node can only begin executing when both source nodes have completed execution. When this happens, this single job executes for a further two units of execution. We therefore have $\lambda_{1,2} = 4$ and $\nu_{1,1} = 1$. Upon the completion of this job, its three successor jobs may each begin execution. Two of these successor jobs complete upon having executed for one unit; the remaining job executes for an additional unit. This is reflected by having $\lambda_{1,3} = 5$ with $\nu_{1,2} = 3$, and $\lambda_{1,4} = 6$ with $\nu_{1,3} = 1$.

21.1.2 Intractability of Feasibility Analysis

Determining whether a system of sporadic DAG tasks is feasible upon a multiprocessor platform under global scheduling is highly intractable. Indeed, even for a system consisting of just a single DAG task for which the relative deadline is $\leq$ period, determining feasibility is easily seen to be equivalent to the makespan minimization problem for preemptive scheduling of a set of precedence constrained jobs on identical processors, or $P|\text{prec}, \text{pmtn}|C_{\max}$ in the standard three-field scheduling notation [99]. Therefore, the problem of determining global feasibility for a system of sporadic DAG tasks is nondeterministic polynomial time (NP)-hard in the strong sense; in fact, it has been shown [127] to remain so even for schedulers given access to processors that are faster by a factor $(4/3 - \epsilon)$ for any $\epsilon > 0$. This result contrasts with the state of our knowledge concerning three-parameter sporadic task systems, where the problem is only known to be (co)NP-hard in the ordinary sense [81] (even on a single processor).

21.2 Implicit-Deadline DAGs

In an *implicit-deadline* sporadic task system τ, each sporadic DAG task $\tau_i = (G_i, D_i, T_i)$ in τ is required to have its relative deadline parameter D_i be equal to its period T_i: $D_i = T_i \ \forall \tau_i \in \tau$. The observation in Sect. 21.1.2 also means that feasibility analysis for such task systems is NP-hard in the strong sense; here we will seek efficient approximate scheduling algorithms and schedulability tests.

Capacity augmentation bounds [137] are similar to utilization bounds (Definition 5.1):

Definition 21.1 (implicit-deadline DAGs [137]) A scheduling algorithm for sporadic DAG task systems is said to have a capacity augmentation bound of b if it can schedule upon m unit-speed processors all task systems τ satisfying the following two conditions

1. $U_{\text{sum}}(\tau) \leq m/b$, and
2. $\text{dens}_{\text{max}}(\tau) \leq 1/b$

In contrast to speedup bounds, which do not necessarily directly yield schedulability tests, a capacity augmentation bound *can* be used as an efficient schedulability test, since both the total utilization $U_{\text{sum}}(\tau)$ and the maximum density $\text{dens}_{\text{max}}(\tau)$ of a task system τ can be computed very efficiently in time linear in the representation of the task system.

The following results are derived in [138] concerning the global scheduling of implicit-deadline sporadic DAG task systems:

- The capacity augmentation bound of global EDF is $\leq (3 + \sqrt{5})/2$, which is ≈ 2.618. For large m, this matches the lower bound established in [137].
- The capacity augmentation bound of global RM scheduling is $\leq (2 + \sqrt{3})$, which is ≈ 3.732.

Additionally, a *federated* scheduling algorithm is proposed and analyzed in [138]. Federated scheduling can be thought of as a generalization of partitioned scheduling to systems of DAG tasks; it dedicates a cluster of processors to each task τ_i with utilization u_i greater than one, and schedules the remaining tasks—those with utilization ≤ 1—upon a shared cluster of processors using a global scheduling algorithm. Specifically, each task τ_i with utilization ≥ 1 is assigned a cluster of

$$m_i \leftarrow \left\lceil \frac{\text{vol}_i - \text{len}_i}{D_i - \text{len}_i} \right\rceil$$

processors for its exclusive use. Let m' denote the number of processors remaining after all tasks with utilization > 1 have been assigned processors as above; the task system admits the task system if m' is $\geq$ twice the sum of the utilizations of the remaining tasks (those with utilization ≤ 1).

During run-time, the tasks on dedicated processors are scheduled using any greedy work-conserving scheduler; the tasks upon the m' shared processors are each considered as sequential tasks (i.e., their internal parallelism is not exploited) and scheduled using any of the strategies discussed in Chaps. 5–9.

It was proved in [138] that this federated scheduling algorithm has a capacity augmentation bound of 2, meaning that any sporadic DAG task system τ satisfying $U_{\text{sum}}(\tau) \leq m/2$, and $\text{dens}_{\text{max}}(\tau) \leq 1/2$ can be scheduled upon a multiprocessor platform with m unit-speed processors.

In addition to having such an efficient sufficient schedulability test, federated scheduling offers the run-time advantage that tasks with utilization > 1 get dedicated processors upon which to execute. By also adopting a partitioning approach the shared processors (recall, from Sect. 6.3, that such partitioning can be done in polynomial time by a PTAS to any desired degree of accuracy), one obtains an elegant generalization to partitioned scheduling for systems of implicit-deadline sporadic tasks, with the properties that (i) each task that executes on more than one processor does not share a processor with any other task, and (ii) tasks that share a processor with other tasks execute upon only one processor.

21.3 Normal Collections of Jobs

We now return our attention to arbitrary-deadline sporadic DAG task systems. The notion of *normal collection of jobs* was introduced in [64], as a generalization of the kinds of workloads that are generated by sporadic DAG tasks. Informally speaking, the distinguishing characteristic of normal collections of jobs is that all jobs with precedence constraints amongst them share a common release time and deadline; this idea is formalized in the following definition.

Definition 21.2 (normal collection of jobs [64]) A *collection of jobs* J is defined to be a sequence of jobs that are revealed online over time, i.e., a job $j \in J$ becomes known upon the release date of j. Each job $j \in J$ is characterized by a release date $r_j \in \mathbf{N}_0$, an absolute deadline $d_j \in \mathbf{N}$, an unknown execution time $e_j \in \mathbf{N}$, and a set of previous jobs J_j which are exactly the jobs which have to be finished before j can become available (the *predecessor jobs* of j). The actual execution time e_j of a job is discovered by the scheduler only when the job signals completion.

A collection of jobs J is said to be a *normal* collection of jobs if it satisfies the additional property that for every predecessor job j of each job k, $r_j = r_k$ and $d_j = d_k$.

Since in any collection of jobs generated by a sporadic DAG task system all jobs that constitute a particular dag-job have identical release date and deadline, and precedence constraints only exist within jobs on individual dag-jobs, it is evident that every collection of jobs generated by a sporadic DAG task system is normal,

Let us now consider the scheduling of collections of jobs. Suppose that an unbounded number of unit-speed processors were available, upon which to schedule a given collection of jobs J. Let A_∞ denote the scheduling algorithm that allocates a processor to each job as soon as it becomes available, and let S_∞ denote the resulting schedule. It is easy to see that the following claims hold:

- S_∞ starts and ends processing jobs always at integral time points (observe that all release dates r_j and all execution times e_j are assumed to be integers).
- At any point in time and for any job, S_∞ has executed at least as much of that job as any feasible schedule of J upon a platform of m unit speed processors.

21.4 A Speedup Bound for EDF

The following lemma from [64] characterizes global EDF schedules for normal collections of jobs.

Lemma 21.1 *Consider a normal collection J of jobs and let α denote any constant that is ≥ 1. At least one of the following statements is true:*

1. *All jobs in J meet their deadlines under EDF on m speed-α processors.*
2. *J is not schedulable by A_∞ (recall that A_∞ is defined upon unit-speed processors).*
3. *There is some interval I such that any feasible schedule for J must complete more than $(\alpha m - m + 1) \cdot |I|$ units of work within I.*

Proof Suppose that both 1 and 2 above do not hold; that is, (i) some job j fails to complete by its deadline d_j under EDF on m speed-α processors, and (ii) J is feasible upon infinitely many unit speed processors.

Without loss of generality, we can assume that there is no job j' in the instance with $d_{j'} > d_j$ (otherwise, since J is normal j' can be removed without affecting either EDF or A_∞). Let t^* denote the latest point in time such that at any time $t \in [0, t^*]$ EDF with α speedup has executed at least as much of *every* job as A_∞ at time t. Such a time exists, since $t^* = 0$ satisfies this property. As 1 and 2 are false, we also have $t^* < d_j$.

We claim that within the interval $I = [t^*, d_j]$, EDF completes more than $(\alpha m - m + 1) \cdot |I|$ units of work. This claim would yield the lemma due to the following reasoning. If EDF completes more than $(\alpha m - m + 1) \cdot |I|$ units of work, then the nonfailing algorithm A_∞ completes at least the same amount of work during I (by construction of I). Hence *every* feasible schedule has to complete more than $(\alpha m - m + 1) \cdot |I|$ units of work during I, since it could not do more work than A_∞ (and thereby more than EDF) before I.

We now prove the claim above regarding the amount of work done by EDF within interval I. Denote by X the total length of the intervals within I where in the EDF schedule all m processors are busy, and let $Y = |I| - X$. We distinguish two cases. First assume that $\alpha \cdot Y \geq |I|$. Denote by $Y_1, \ldots, Y_k \subseteq I$ all subintervals of I where not all processors are busy. We define t' such that $\alpha \cdot |[t^*, t'] \cap \bigcup_i Y_i| = \lceil t^* \rceil - t^*$. During all points in time within $[t^*, t'] \cap \bigcup_i Y_i$ all jobs are available for EDF which are scheduled by A_∞ during $[t^*, \lceil t^* \rceil\,]$. Since during all these points in time EDF does not use all processors and runs the processors with speed α, by time t' it has processed at least as much of every job as A_∞ by time $\lceil t^* \rceil$.

Next, define timepoints t_i, $i = 0, \ldots, d_j - \lceil t^* \rceil$ such that $\alpha \cdot |[t^*, t_i] \cap \bigcup_i Y_i| = \lceil t^* \rceil - t^* + i$ for each i. We prove by induction that up to time t_i EDF has processed as much of every job as A_∞ by time $\lceil t^* \rceil + i$. The case $i = 0$ was proven above. Now suppose that the claim is true for some value i. Then at each timepoint during $[t_i, t_{i+1}) \cap \bigcup_i Y_i$ all jobs are available for EDF that A_∞ works on during $[\,\lceil t^* \rceil + i, \lceil t^* \rceil + i + 1)$. Since during all these timepoints EDF does not use all processors and runs the processors with speed α, by time t_{i+1} it has processed at least as much of every job as A_∞ by time $\lceil t^* \rceil + i + 1$. By induction the claim is true for $i^* = d_j - \lceil t^* \rceil$

and hence at time $\lceil t^* \rceil + i^* = d_j$ EDF has finished as much of every job as A_∞. This yields a contradiction since we assumed that A_∞ constructs a feasible schedule and EDF does not.

Now assume that $\alpha \cdot Y < |I|$. Hence, in the interval I EDF finishes at least

$$\begin{aligned} \alpha m \cdot X + \alpha \cdot Y &= \alpha m \cdot (|I| - Y) + \alpha \cdot Y \\ &= \alpha m \cdot |I| - \alpha m Y + \alpha \cdot Y \\ &> \alpha m \cdot |I| - m \cdot |I| + |I| \\ &= (\alpha m - m + 1) \cdot |I| \end{aligned}$$

units of work, and by construction of I, any feasible schedule has to finish during the interval I all work that EDF finishes during I. □

Setting $\alpha \leftarrow (2 - 1/m)$ yields the following speedup bound for the EDF-scheduling of normal collections of jobs.

Theorem 21.1 *Any normal collection of jobs that is feasible on m unit-speed processors is EDF-schedulable on m processors each of speed* $(2 - 1/m)$.

Proof Since the instance is assumed feasible, it is clearly so upon a sufficiently high number of processors of unit speed. Also, the instance admits a valid schedule which finishes in any interval I at most $m \cdot |I|$ units of work. Note that if $\alpha = 2 - 1/m$ then $(\alpha m - m + 1) \cdot |I| = (2m - 1 - m + 1) \cdot |I| = m|I|$. Hence, Lemma 21.1 implies that EDF finishes all jobs by their respective deadline. □

Since every collection of jobs generated by a DAG task system is normal, we obtain the following corollary.

Corollary 21.1 *Any DAG task system that is feasible on m processors of unit speed is EDF-schedulable on m processors of speed* $(2 - 1/m)$.

The above bound is tight: examples are known, even without precedence constraints, of feasible collections of jobs that are not EDF-schedulable unless the speedup is at least $(2 - 1/m)$ [155].

21.5 A Speedup Bound for Deadline Monotonic (DM)

In scheduling collections of jobs, a DM scheduler would schedule, at each instant, the available jobs with with minimum relative deadline (breaking ties arbitrarily), where the relative deadline of a job j is the difference $(d_j - r_j)$ between its deadline and release date. By applying techniques similar to the ones used in the proof of Lemma 21.1, an analogous result is obtained:

Lemma 21.2 *Consider a normal collection J of jobs and let α denote any constant that is* ≥ 1. *At least one of the following holds:*

1. *All jobs in J are completed within their deadline under DM on m processors of speed α each, or*
2. *J is not schedulable by A_∞ (with unit speed), or*

3. *There is an interval I such that any feasible schedule for J must finish more than $(\alpha m - m + 1) \cdot |I|/2$ units of work within I.*

Proof Suppose that both (i) and (ii) do not hold, that is, under DM on m speed-α processors some job j fails its deadline d_j, and J is feasible if we are given a sufficiently large number of processors. We again will consider the idealized greedy algorithm A_∞. Without loss of generality, we can assume that there is no job j' in the instance with $d_{j'} > 2d_j - r_j$ where r_j is the release date of job j. In fact assume that in J there is a job j' that has deadline later than $2d_j - r_j$. If the relative deadline of job j' is at most $d_j - r_j$ then the job is released after d_j and we can ignore it; if the relative deadline of job j' is greater than $d_j - r_j$ then the execution of job j is not interrupted by job j' and hence by removing j' from J we obtain a smaller collection J' that violates the claim.

Let t^* denote the latest point in time such that at any time $t \in [0, t^*]$, DM has processed at least as much of *every* job as A_∞ at time t. Such a time exists, since $t^* = 0$ satisfies this property. Also, it must hold that $t^* < d_j$.

Let $\hat{t} = \min(t^*, r_j)$, $I = [\hat{t}, 2d_j - r_j]$ and $\hat{I} = [\hat{t}, d_j]$. Observe that the definition of DM implies that during $\hat{I}$ DM executes only jobs that have their deadline in I.

We claim that, within $\hat{I}$, DM finishes more than $(\alpha m - m + 1) \cdot |\hat{I}|$ units of work, hence A_∞ finishes at least the same amount of work during I (by construction of I and $\hat{I}$) and, hence, *every* feasible schedule has to finish more than $(\alpha m - m + 1) \cdot |\hat{I}|$ units of work during I.

Analogous to the case of EDF, we can show by contradiction that, within $\hat{I}$, DM finishes more than $(\alpha m - m + 1) \cdot |\hat{I}|$ units of work. Again, we denote by X the total length of the intervals within $\hat{I}$ where in the DM schedule all m processors are busy. Define $Y = |\hat{I}| - X$. As in the proof of EDF we distinguish two cases. First, if $\alpha \cdot Y \geq |\hat{I}|$, by the same argument as in the proof for EDF it is possible to show that DM has finished as much of every job as A_∞. This yields a contradiction since we assumed that A_∞ is feasible and DM is not.

If $\alpha \cdot Y < |\hat{I}|$, as in the proof of EDF it follows that during $\hat{I}$ DM finishes at least

$$\alpha m \cdot X + \alpha \cdot Y > (\alpha m - m + 1) \cdot |\hat{I}|$$

units of work, and by construction of I, any feasible schedule has to finish during the interval I all work that DM finishes during $\hat{I}$. Since $|\hat{I}| \geq |I|/2$, the lemma follows. □

Choosing $\alpha \leftarrow (3 - 1/m)$ yields the following speedup bound for the DM scheduling of normal collections of jobs.

Theorem 21.2 *Any normal collection of jobs that is feasible on m processors of unit speed is DM-schedulable on m processors of speed* $(3 - 1/m)$.

Proof Since we assumed the instance to be feasible, it is in particular feasible on a sufficiently high number of processors of unit speed. Also, the instance admits a valid schedule which finishes in any interval I at most $m \cdot |I|$ units of work. Note that if $\alpha = 3 - 1/m$ then $(\alpha m - m + 1) \cdot |I|/2 = (3m - 1 - m + 1) \cdot |I|/2 = m|I|$. Hence, Lemma 21.2 implies that DM finishes all jobs by their respective deadline. □

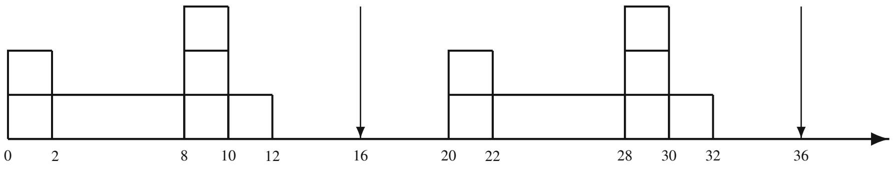


Fig. 21.3 The processors that are needed by $S_\infty(J, 0.5)$, if J consists of two releases of the sporadic DAG task τ_1 of Fig. 21.1, at time-instants 0 and 20, respectively

Corollary 21.2 *Any DAG task system that is feasible on m processors of unit speed is DM-schedulable on m processors of speed* $(3 - 1/m)$.

21.6 The Work Function

Bonifaci et al. [64] introduced the notion of a *work function* to characterize the amount of work that could be generated by a sporadic DAG task. We now describe this work function in terms somewhat more general than was used in [64].

Let s denote any positive real number. Let us generalize the ideas of A_∞ and S_∞ from Sect. 21.3 to speed-s processors: suppose that we had an infinite number of speed-s processors available upon which to execute a given sporadic DAG task system τ. Consider some collection of dag-jobs J generated by τ; let $S_\infty(J, s)$ denote the schedule obtained by allocating a speed-s processor to each job the instant it is ready to execute, and executing this job upon the allocated processor until it completes execution.

Example 21.3 Figure 21.3 depicts the number of processors used in $S_\infty(J, 0.5)$, when J consists of two releases of the sporadic DAG task of Fig. 21.1 at time-instants 0 and 20, respectively.

This is obtained by (i) scaling the figure depicting the parallelism of each dag-job (Fig. 21.2) by a factor of 0.5, thereby depicting the execution that is needed for the first dag-job, and (ii) replicating this scaled figure at $t = 20$ to depict the execution that is needed for the second dag-job.

Let τ_i denote a sporadic DAG task, and s any positive real number ≤ 1. Let J denote any collection of jobs legally generated by the task τ_i.

- For an interval I, let $\mathsf{work}(J, I, s)$ denote the amount of execution occurring within the interval I in the schedule $S_\infty(J, s)$, of jobs *with deadlines that fall within I*.
  Observe that since schedule $S_\infty(J, s)$ executes each job as soon as it becomes available, thereby leaving as little work to be done later as possible, every schedule for J on speed-s processors meeting all deadlines has to complete at least $\mathsf{work}(J, I, s)$ units of execution over the interval I.
- For any positive integer t, let $\mathsf{work}(J, t, s)$ denote the maximum value $\mathsf{work}(J, I, s)$ can take, over any interval I of duration equal to t.

- Finally, let $\mathsf{work}(\tau_i, t, s)$ denote the maximum value of $\mathsf{work}(J, t, s)$, over all job sequences J that may be generated by the sporadic DAG task τ_i.

Let us further extend the definition of the work function from individual tasks to task systems in an obvious manner: for any DAG task system τ, let $\mathsf{work}(\tau, t, s)$ denote the sum $\sum_{\tau_i \in \tau} \mathsf{work}(\tau_i, t, s)$. Claim 21.1 immediately follows from the definitions of maximum density and the work function:

Claim 21.1 Far a sporadic DAG task system τ to be feasible (always schedulable to meet all deadlines by an optimal, clairvoyant scheduler) upon m speed-s processors, it is necessary that

1. $\text{dens}_{\text{max}}(\tau) \leq s$, and
2. $\forall t : t \geq 0 : \mathsf{work}(\tau, t, s) \leq ms$.

21.6.1 Computing the Work Function

Above we have defined $\mathsf{work}(\tau_i, t, s)$ to be the *maximum* value of $\mathsf{work}(J, t, s)$, over all job sequences J that may be generated by the sporadic DAG task τ_i. It is evident that this maximum is achieved when the deadline of some dag-job of τ_i coincides with the rightmost endpoint of an interval of duration t, and the other dag-jobs of τ_i are released as closely as possible. This is illustrated in Example 21.4.

Example 21.4 We now show the manner in which $\mathsf{work}(\tau_i, t, s)$ is computed, by computing $\mathsf{work}(\tau_1, t, 0.5)$ for some example values of t (here τ_1 is the example sporadic DAG task depicted in Fig. 21.1).

$\mathbf{t = 10}$. Consider the schedule depicted in Fig. 21.3. As stated above, $\mathsf{work}(\tau_1, 10, 0.5)$ is maximized when the right end of the interval of duration 10 coincides with the deadline of some dag-job of τ_1. Consider, therefore, the interval $[6, 16]$; over this interval, one processor executes for the duration $[6, 12]$ of six time units, with a further two processors executing for a duration of two time units. The cumulative execution is therefore $(6+2+2) \times \frac{1}{2}$, or 5. Hence, $\mathsf{work}(\tau_1, 10, 0.5) = 5$.

$\mathbf{t = 25}$. Consider the interval of duration 25 ending at the deadline at time-instant 36: $[11, 36]$.

- One entire dag-job of τ_1, released at time-instant 20, executes within this interval—this dag-job contributes an amount equal to the sum of the WCETs of all the nodes in τ_1, i.e., 9.
- Since this dag-job has a deadline at time 36, the *previous* dag-job's deadline was at time-instant $36 - 20$, or 16.
- The dag-job with deadline 16 executes on one processor for the duration $[11, 12]$; this yields an additional $(1) \times \frac{1}{2}$, or 0.5.

Finally, $\mathsf{work}(\tau_1, 25, 0.5)$ is obtained as the sum of these two quantities: $\mathsf{work}(\tau_1, 25, 0.5) = 9 + 0.5 = 9.5$.

In Fig. 21.4, $\mathsf{work}(\tau_1, t, 0.5)$ is plotted as a function of t for values of $t \in [0, 34]$.

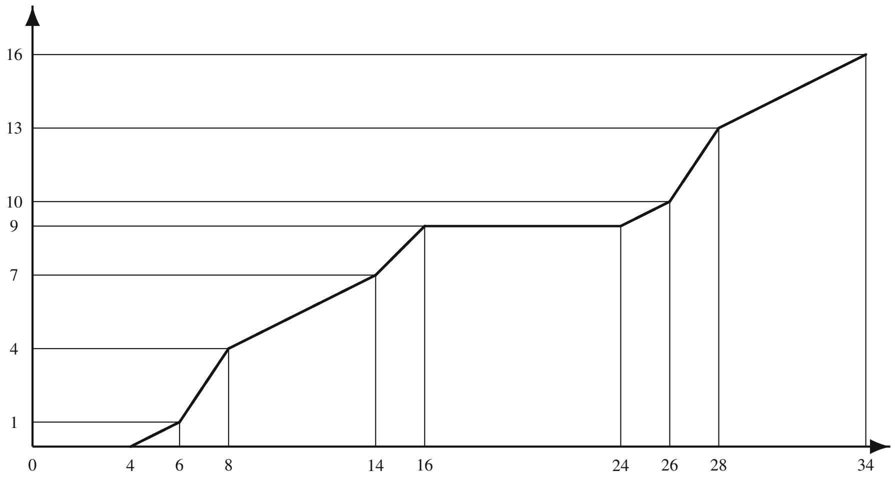


Fig. 21.4 Illustrating the work function: work$(\tau_i, t, 0.5)$ for the sporadic DAG task of Fig. 21.1

It follows from its definition that work(τ_i, t, s) is a piecewise linear function. If $D_i \le T_i$, then it is visually evident from the example in Fig. 21.4 that the number of linear "pieces" within any interval of duration T_i bounded from above by the number of vertices in the graph of τ_i, and work(τ_i, t, s) can be efficiently determined in polynomial time.

It is not clear how one would compute work(τ_i, t, s) exactly in polynomial time if $D_i > T_i$; for such tasks, [64] instead presents a polynomial-time technique for approximating work(τ_i, t, s), rather than computing it exactly.

21.7 Pseudo-Polynomial EDF-Schedulability Testing

We had obtained speedup bounds of $(2 - 1/m)$ and $(3 - 1/m)$ respectively for EDF and RM respectively in Sects. 21.4 and 21.5. However, speedup bounds do not in themselves necessarily let us determine whether a particular task system is schedulable or not.

In this section we will describe how the speedup bound results for EDF scheduling from Sect. 21.4 can be used to formulate an EDF-schedulability test (a similar exercise may be performed for DM scheduling; we do not provide the details here).

Let us reformulate Lemma 21.1 using the work function. Lemma 21.1 implies that, in order to assert that EDF feasibly schedules any job sequence J of jobs generated by τ upon m speed-α processors, it suffices to ensure that for any such job sequence J,

Condition 1: J is feasible under A_∞, and
Condition 2: there is no interval I during which any feasible schedule for J must finish more than $(\alpha m - m + 1) \cdot |I|$ units of work.

Furthermore if either of the two conditions fail (with $\alpha \geq 2 - 1/m$) then the system is infeasible on m unit-speed processors.

Note that these conditions are both monotonic with respect to the execution times of the individual jobs: if they are satisfied by a collection of jobs with certain execution times, they are also satisfied by a same collection of jobs with reduced execution times. It therefore suffices to validate these conditions using only the specified WCETs.

The first condition is satisfied if $\text{dens}_{\max}(\tau) \leq 1$. For the second to hold, it is sufficient that $\mathsf{work}(\tau, |I|, 1) \leq (\alpha m - (m - 1)) \times |I|$ for all intervals I.

Lemma 21.1 may therefore be restated as follows.

Theorem 21.3 *([64]) Let α denote any constant* ≥ 1. *Sporadic DAG task system τ is* EDF *schedulable on m speed-α processors if* $\textit{dens}_{\max}(\tau) \leq 1$, *and*

$$\mathsf{work}(\tau, t, 1) \leq (\alpha m - (m - 1)) \times t$$

for all values of $t \geq 0$.

In performing schedulability analysis, it is typical to assume that we have *unit-speed* processors available to us, and to determine whether a given system is guaranteed to be scheduled upon a platform comprised of such processors such that all jobs of all tasks will always complete by their deadlines. While Theorem 21.3 (in the form Lemma 21.1) enabled us to establish the speedup bound for global EDF in Sect. 21.4, it is not immediately obvious how it can be used to determine whether a given sporadic DAG task system is global EDF schedulable or not upon unit-speed processors. We now restate the result of Theorem 21.3 in a manner that allows us to answer this question.

Let σ denote any constant < 1. It is straightforward to mimic the derivation of Lemma 21.1 in [64], with "unit-speed" replaced by σ and α replaced by 1, to get

Theorem 21.4 *Let σ denote any constant* < 1. *Sporadic DAG task system τ is global-*EDF *schedulable on m unit-speed processors if* $\textit{dens}_{\max}(\tau) \leq \sigma$, *and*

$$\mathsf{work}(\tau, t, \sigma) \leq (m - (m - 1)\sigma) \times t \tag{21.1}$$

for all values of $t \geq 0$.

Hence, to show that a given τ is EDF-schedulable upon m unit-speed processors it suffices, according to Theorem 21.4, to produce a value for σ such that Condition 21.1 holds for all $t \geq 0$. We refer to such a σ as a *witness* to the EDF-schedulability of τ.

Definition 21.3 (witness) Any positive real number σ for which Condition 21.1 holds for all $t \geq 0$ is a *witness to the* EDF *schedulability of* sporadic DAG task system τ upon m unit-speed processors.

To show that a given sporadic DAG task system is EDF schedulable, we need to produce a witness to its schedulability—give a value of σ and show that Condition 21.1 holds for all $t \geq 0$ for this value of σ. A sufficient schedulability test with speedup $(2 - 1/m)$ is therefore immediately obtained by checking whether $\sigma \leftarrow (2 - 1/m)$ causes Condition 21.1 holds for all $t \geq 0$, and declaring the task system EDF-schedulable if and only if the answer is "yes." Pragmatic improvements to this test are possible, of the same kind as those detailed in Sect. 19.4 for the schedulability analysis of three-parameter sporadic task systems.

21.8 Polynomial-Time Sufficient Schedulability Tests

In addition to the speedup-optimal pseudo-polynomial-time schedulability tests described above, simpler EDF and DM schedulability tests with polynomial run-time were also derived in [64]; we present these tests without proof below. Assume, without loss of generality, that the DAG tasks τ_i are ordered according to nondecreasing D_i (breaking ties arbitrarily).

Theorem 21.5 *Any sporadic DAG task system τ satisfying the following conditions:*

i) $\mathrm{len}_k \leq D_k/3$*, $k = 1, 2, \ldots, n$,*
ii) for each k, $k = 1, 2, \ldots, n$,

$$\left(\sum_{i:T_i \leq D_k} u_i + \sum_{i:T_i > D_k} \frac{\mathrm{vol}_i}{D_k} \right) \leq \frac{m + 1/2}{3}.$$

is EDF-schedulable on m unit-speed processors.

Theorem 21.6 *Any sporadic DAG task system τ satisfying the following conditions:*

i) $\mathrm{len}_k) \leq D_k/5$*, $k = 1, 2, \ldots, n$,*
ii) for each k, $k = 1, 2, \ldots, n$,

$$\left(\sum_{i:T_i \leq 2D_k} u_i + \sum_{i:T_i > 2D_k} \frac{\mathrm{vol}_i}{4D_k} \right) \leq \frac{m + 1/4}{5}.$$

is DM-schedulable on m unit-speed processors.

Sources

The sporadic DAG tasks model described in this chapter was proposed in [38]. The speedup bounds and both the pseudo-polynomial and polynomial time schedulability tests are from [64]. The discussion of the work function is from [34]; pragmatic improvements to the EDF schedulability test presented in this chapter are also described in [34]. The notion of capacity bounds was introduced in [137]; the analysis of federated scheduling of implicit-deadline sporadic tasks is from [138].

Chapter 22
Real-time Scheduling upon Heterogeneous Multiprocessors

As the computational demands made by ever more complex embedded real-time applications continue to increase, there is a need for enhanced performance capabilities from these platforms. Initially, the approach adopted for obtaining such enhanced performance was to increase core counts in multiprocessor CPUs. Soon, however, chip makers began to distinguish themselves not only by offering more general-purpose cores but also by providing specialized hardware components that accelerate particular computations. Examples include multicore CPUs with specialized graphics processing cores, specialized signal-processing cores, specialized floating-point units, customizable FPGAs, etc. Computing platforms such as these with specialized components are called *unrelated* or *heterogeneous* [145] multiprocessor platforms.

One important consequence of this processor-specialization in heterogeneous platforms that needs to be taken into account during system implementation is that the same piece of code may require different amounts of time to execute upon different processing units. For example, a process responsible for rendering images may take far less time to execute upon a graphics coprocessor than on a CPU, while a number-crunching routine would execute more efficiently upon the CPU.

Although unrelated multiprocessors are becoming increasingly more important in real-time systems implementation, the scheduling-theoretic study of such systems is, relatively speaking, still in its infancy. In this chapter, we will survey the current state of the art in implementing real-time systems upon unrelated multiprocessors. We start out in Sect. 22.1 describing a model for representing sporadic task systems that are to be implemented upon an unrelated multiprocessor platform. In Sects. 22.2 and 22.3, we list what is currently known about the global and partitioned scheduling of implicit-deadline sporadic task systems upon unrelated multiprocessors; in Sect. 22.4, we discuss the partitioned scheduling of three-parameter sporadic task systems upon unrelated multiprocessors. (We do not consider the scheduling of systems of sporadic directed acyclic graph (DAG) tasks upon unrelated multiprocessors in this chapter— although this is a particularly interesting problem since the different vertices of a sporadic DAG task may execute differently upon different processors, not much is known about how this problem should be tackled.)

S. Baruah et al., *Multiprocessor Scheduling for Real-Time Systems,*
Embedded Systems, DOI 10.1007/978-3-319-08696-5_22

22.1 Task and Machine Model

A real-time system comprised of n sporadic tasks that is to be scheduled upon a given m-processor unrelated multiprocessor platform is specified by:

1. The collection of tasks $\{\tau_1, \tau_2, \ldots, \tau_n\}$. Each task τ_i is characterized, as before, by the three parameters C_i, D_i, and T_i, denoting respectively the worst-case execution requirement, relative deadline, and period parameters. (In Sects. 22.2 and 22.3 we will be dealing with implicit-deadline sporadic task systems, in which $D_i = T_i$ for all tasks τ_i.)
2. A *rate matrix* $\mathbf{R}_{[n,m]}$. This is an $(n \times m)$ matrix $[r_{i,j}]; i = 1, 2, \ldots, n; j = 1, 2, \ldots, m$ of nonnegative real numbers. The value of $r_{i,j}$ denotes the "rate" at which the j'th processor executes task τ_i—by executing task τ_i on the j'th processor for 1 time unit, $r_{i,j}$ units of work gets done. (If τ_i cannot be executed upon the j'th processor, then $r_{i,j} \leftarrow 0$.)

Most results concerning the scheduling of implicit-deadline sporadic task systems upon unrelated multiprocessors characterize the tasks according to their utilizations, rather than their C_i and T_i parameters. In such a characterization, the system may be represented as an $n \times m$ *utilization matrix* $\mathbf{U}_{[n \times m]}$ with $u_{i,j}$, the element in the i'th row, j'th column, having the value

$$u_{i,j} \leftarrow \frac{C_i}{r_{i,j} T_i}$$

and thus denoting the fraction of the computing capacity of the j'th processor that is needed to completely execute task τ_i.

22.2 Global Scheduling of Liu and Layland (LL) Task Systems

The preemptive scheduling of collections of independent jobs to minimize makespan upon unrelated multiprocessors was considered in [121]. The scheduling problem was formulated as a linear programming problem, and thereby solved in polynomial time. It was also shown that no more than $O(m^2)$ preemptions are necessary to schedule n jobs upon m processors.

These results can be adapted to obtain an optimal algorithm for the scheduling of implicit-deadline sporadic task systems upon unrelated multiprocessors, by observing that scheduling the implicit-deadline sporadic task system characterized by the utilization matrix $\mathbf{U}_{[n \times m]}$ is essentially equivalent to scheduling n jobs upon m unrelated processors, in which the i'th job has an execution requirement $u_{i,j}$ upon the j'th processor, with a makespan no larger than one. To see why this should be so, observe that a schedule for this collection of jobs with makespan ≤ 1 could have all its jobs "scaled down" by a factor Δ for Δ an arbitrarily small positive constant, to have makespan Δ and then replicated infinitely often to yield a schedule for the implicit-deadline sporadic task system. A schedule so constructed is likely have an unacceptably large number of preemptions and interprocessor migrations—as many as $\Theta(m^2)$ over each interval of duration Δ; however, techniques have been obtained [28] for ensuring that the total number of tasks executing upon more than one processor is strictly less than twice the number of processors. In addition, heuristic techniques may be applied to reduce the number of preemptions.

22.3 Partitioned Scheduling of LL Task Systems

Partitioning a system of implicit-deadline sporadic tasks upon an unrelated multiprocessor is a computationally intractable problem; we have seen earlier (Chap. 6) that such partitioning is NP-hard in the strong sense even on the simpler identical multiprocessor platforms. Optimal partitioning strategies are therefore unlikely to have polynomial-time implementations. Techniques have been developed for representing this partitioning problem as an integer linear program (ILP). Although solving an ILP exactly takes exponential time, approximation techniques [111, 128, 172] can be used to solve these ILPs approximately, in polynomial time. We will briefly describe the transformation of the scheduling problem to an ILP, and its approximate solution, in Sect. 22.3.1. These approximation algorithms make the following processor speedup factor (see Definition 5.2) guarantee: Any task system that can be partitioned by an optimal algorithm on a given unrelated multiprocessor platform can be partitioned by these algorithms upon a platform in which each processor is twice as fast. Furthermore it follows from results in [128] that under the assumption that P $\neq$ NP, there can be no polynomial-time algorithm for solving this problem that has a speedup factor smaller than 1.5.

22.3.1 A Linear-Programming Approach to Approximate Partitioning

Recall that in an ILP, one is given a set of n variables, *some or all of which are restricted to take on integer values only*, a collection of "constraints" that are expressed as linear inequalities over these n variables, and an "objective function," also expressed as a linear inequality of these variables. The set of all points in n-dimensional space over which all the constraints hold is called the *feasible region* for the ILP. The goal is to find the extremal (maximum or minimum, as specified) value of the objective function over the feasible region.

A linear program (LP) is like an ILP, without the constraint that some of the variables are restricted to take on integer values only. That is, in an LP over a given set of n variables, one is given a collection of constraints that are expressed as linear inequalities over these n variables, and an objective function, also expressed as a linear inequality of these variables. The region in n-dimensional space over which all the constraints hold is again called the feasible region for the LP, and the goal is to find the extremal value of the objective function over the feasible region. A region is said to be *convex* if, for any two points $\mathbf{p_1}$ and $\mathbf{p_2}$ in the region and any scalar $\lambda, 0 \le \lambda \le 1$, the point $(\lambda \cdot \mathbf{p_1} + (1-\lambda) \cdot \mathbf{p_2})$ is also in the region. A *vertex* of a convex region is a point $\mathbf{p}$ in the region such that there are no distinct points $\mathbf{p_1}$ and $\mathbf{p_2}$ in the region, and a scalar $\lambda, 0 < \lambda < 1$, such that $[\mathbf{p} \equiv \lambda \cdot \mathbf{p_1} + (1-\lambda) \cdot \mathbf{p_2}]$.

It is known that an LP can be solved in polynomial time by the ellipsoid algorithm [119] or the interior point algorithm [118]. (In addition, the exponential-time simplex algorithm [73] has been shown to perform extremely well "in practice," and is often the algorithm of choice despite its exponential worst-case behavior.)

We now state without proof some basic facts concerning such linear programming optimization problems.

Fact 22.1 The feasible region for a LP problem is convex, and the objective function reaches its optimal value at a vertex point of the feasible region.

An optimal solution to a LP problem that is a vertex point of the feasible region is called a *basic solution* to the LP problem.

Fact 22.2 Consider a LP on n variables $x_1, x_2, \ldots, x_n$, in which each variable is subject to the constraint that it be ≥ 0 (these constraints are called *nonnegativity constraints*). Suppose that there are a further m linear constraints. If $m < n$, then *at most m of the variables have nonzero values* at each vertex of the feasible region [1] (including the basic solution).

Returning to our scheduling problem of partitioning implicit-deadline sporadic task systems upon an unrelated multiprocessor platform, let us suppose that we are given an implicit-deadline sporadic task system characterized by the utilization matrix $\mathbf{U}_{[n \times m]}$. For any mapping of the n tasks on the m processors, let us define $(n \times m)$ *indicator variables* $x_{i,j}$, for $i = 1, 2, \ldots, n$; $j = 1, 2, \ldots, m$. Variable $x_{i,j}$ is set equal to one if the task τ_i is mapped onto the jth processor, and zero otherwise. A mapping of the n tasks upon the m processors would have these variables satisfy the following constraints:

$$\begin{aligned} x_{i,j} &= 0 \textbf{ or } 1, && (i = 1, 2, \ldots, n; \quad j = 1, 2, \ldots, m) \\ \sum_{j=1}^{m} x_{i,j} &= 1, && (i = 1, 2, \ldots, n) \end{aligned}$$

where these constraints restrict that each task be assigned to exactly one processor. We can therefore represent the scheduling problem of partitioning the tasks as the following integer programming problem, with the variables $x_{i,j}$ restricted to integer values.

Informally, U represents the maximum fraction of the capacity of any processor that is used, and is set to be the objective function (i.e., the quantity to be minimized) of the ILP problem. The first constraint asserts that each task be assigned some

[1] The feasible region in n-dimensional space for this linear program (LP) is the region over which all the $n + m$ constraints (the nonnegativity constraints, plus the m additional ones) hold.

ILP-Feas(U$_{[n\times m]}$).
Minimize U, subject to the following constraints:

1. $\sum_{j=1}^{m} x_{i,j} = 1$, $(i = 1, 2, \ldots, n)$
2. $\sum_{i=1}^{n} (x_{i,j} \cdot u_{i,j}) \le U$, $(j = 1, 2, \ldots, m)$
3. $x_{i,j}$ is a non-negative integer, $(i = 1, 2, \ldots, n;\ j = 1, 2, \ldots, m)$

LPR-Feas(U$_{[n\times m]}$).
Minimize U, subject to the following constraints:

1. $\sum_{j=1}^{m} x_{i,j} = 1$, $(i = 1, 2, \ldots, n)$
2. $\sum_{i=1}^{n} (x_{i,j} \cdot u_{i,j}) \le U$, $(j = 1, 2, \ldots, m)$
3. $x_{i,j}$ is a non-negative **real number**, $(i = 1, 2, \ldots, n;\ j = 1, 2, \ldots, m)$

processor; the second, that at most U of the j'th processor's capacity be used for each j, and the third, that it is semantically meaningless to assign negative values to the indicator variables. Let OPT(ILP-Feas($\mathbf{U}_{[n\times m]}$)) denote the minimum value of U, obtained by solving ILP-Feas($\mathbf{U}_{[n\times m]}$). If OPT(ILP-Feas($\mathbf{U}_{[n\times m]}$)) is at most one, it is not hard to see that an assignment of nonnegative integer values to the variables satisfying these constraints is equivalent to a partitioning of the n tasks upon the m processors. Thus, obtaining a solution to ILP-Feas($\mathbf{U}_{[n\times m]}$) is equivalent to determining whether the heterogeneous multiprocessor task system ($\mathbf{U}_{[n\times m]}$) is feasible. This is formally stated by the following theorem:

Theorem 22.1 *The integer linear programming problem ILP-Feas*($\mathbf{U}_{[n\times m]}$) *has a solution with* $U \le 1$ *if and only if the unrelated multiprocessor implicit-deadline sporadic task system* ($\mathbf{U}_{[n\times m]}$) *is feasible.*

The result in Theorem 22.1 allows us to transform the problem of determining whether an unrelated multiprocessor implicit-deadline sporadic task system is feasible to an ILP problem. At first sight, this may seem to be of limited significance, since ILP is also known to be intractable (NP-complete in the strong sense [154]). However, some recently-devised approximation techniques for solving ILP problems, based upon the idea of *LP relaxations* to ILP problems, may prove useful in obtaining approximate partitionings—we explore these approximation techniques below.

By relaxing the requirement that the $x_{i,j}$ variables in the integer linear programming formulation ILP-Feas($\mathbf{U}_{[n\times m]}$) described above be integers only, we have obtained the linear programming problem, which is referred to as the *LP-relaxation* [164] of ILP-Feas($\mathbf{U}_{[n\times m]}$):

Let $\mathbf{X}$ denote the $n\times m$ variables $x_{i,j}$. Let $\mathbf{X}_{\mathsf{OPT}}$ and U_{OPT} denote the values assigned to the variables in $\mathbf{X}$, and to U, in the basic solution to LPR-Feas($\mathbf{U}_{[n\times m]}$). Recall that OPT(ILP-Feas($\mathbf{U}_{[n\times m]}$)) denotes the optimal value of U obtained by solving of ILP-Feas($\mathbf{U}_{[n \times m]}$). Since LPR-Feas($\mathbf{U}_{[n \times m]}$) is a *less* constrained problem than ILP-Feas($\mathbf{U}_{[n \times m]}$), we have the following result.

Lemma 22.1

$$U_{OPT} \leq \text{OPT}(\textit{ILP-Feas}(\mathbf{U}_{[n \times m]})). \qquad \square$$

The constraints (3) of LPR-Feas($\mathbf{U}_{[n \times m]}$) above are nonnegativity constraints; hence, LPR-Feas($\mathbf{U}_{[n \times m]}$) is a LP on the $(n \cdot m + 1)$ variables (the $n \cdot m$ variables $\mathbf{X}$, and U), with only $(n + m)$ constraints other than nonnegativity constraints. By Fact 22.2 above, therefore, at most $(n + m)$ of the $(nm + 1)$ variables have nonzero values at the basic solution; in particular, *at most* $(n + m - 1)$ *of the values in* $\mathbf{X}_{OPT}$ *are nonzero.*

The crucial observation is that each of the n constraints (1) of LPR-Feas($\mathbf{U}_{[n \times m]}$) is on a *different* set of $x_{i,j}$ variables—the first such constraint has only the variables $x_{1,1}, x_{1,2}, \ldots, x_{1,m}$, the second has only the variables $x_{2,1}, x_{2,2}, \ldots, x_{2,m}$, and so on. Since there are at most $(n + m - 1)$ nonzero variables in $\mathbf{X}_{\text{OPT}}$, it follows from the pigeon-hole principle that at most $(m-1)$ of these constraints will have more than one nonzero value in $\mathbf{X}_{\text{OPT}}$. For each of the remaining (at least) $(n - m + 1)$ constraints, the sole nonzero $x_{i,j}$ variable must equal exactly 1, in order that the constraint be satisfied. Fact 22.3 follows.

Fact 22.3 For at least $(n - m + 1)$ of the integers i in $\{1, 2, \ldots, n\}$, exactly one of the variables $\{x_{i,1}, x_{i,2}, \ldots, x_{i,m}\}$ is equal to 1, and the remaining are equal to zero, in $\mathbf{X}_{\text{OPT}}$. $\square$

As a consequence of Fact 22.3, it follows that the solution to the LP problem LPR-Feas($\mathbf{U}_{[n \times m]}$) immediately yields a partial mapping of tasks to processors, in which all but at most $(m - 1)$ tasks get mapped.

It remains place the at most $(m-1)$ tasks that remain unmapped. For small values of m, exhaustive enumeration, with a time complexity of $O(m^m)$, can be used to find an optimal mapping of just these tasks on the capacity remaining upon the m processors. It is straightforward to show that the following property holds:

> If a given task system can be mapped on to a particular unrelated multiprocessor platform by an optimal algorithm, then the algorithm described above will find a mapping upon an unrelated multiprocessor platform in which each processor is twice as fast.

The overall complexity of this algorithm is a polynomial in n and m (for solving the LP) plus an expression that is $O(m^m)$. Although this is not a polynomial-time algorithm, it may prove adequate for small values of m. A true polynomial-time algorithm making the same performance guarantee may be obtained using the more advanced approximation techniques described in [129]; the interested reader is referred to [30, 129] for details.

22.3.2 Partitioning upon Limited Unrelated Multiprocessors

There has recently been an increasing recognition (see, e.g., [11]) that it is interesting to study unrelated multiprocessor platforms in which the number of distinct *types* of processors is a small constant. This is motivated by the plethora of currently-available and soon-to-be-available multicore CPUs with just a few distinct types of processors—typically one or a few general-purpose processing cores and one or more specialized graphics processors, or general-purpose processing cores and "synergistic" processors for executing single instruction multiple data (SIMD) instructions. Such example multicore CPUs provide motivation to consider the problem of partitioning LL task systems upon unrelated multiprocessors in which all the processors are of a relatively small number of distinct types. Unrelated multiprocessor platforms of this kind are referred to as *limited unrelated multiprocessors* [173].

Recall, from Sect. 6.3, that a *polynomial-time approximation scheme (PTAS)* is an algorithm for solving certain kinds of optimization problems approximately. A PTAS for a given optimization problem takes as input a parameter $\delta > 0$ and an instance of the optimization problem and, in time polynomial in the problem size (although not necessarily in the value of δ), produces a solution that is within a factor $(1 + \delta)$ of being optimal.

A PTAS was designed in [173] that approximately partitions systems of LL tasks upon limited unrelated multiprocessors to any desired degree of accuracy.

22.4 Partitioned Scheduling of Three-Parameter Sporadic Task Systems

In contrast to all the work discussed above in this chapter that deals with implicit-deadline sporadic task systems, Marchetti-Spaccamela et al. [147] considered the partitioned scheduling of three-parameter sporadic task systems upon unrelated multiprocessor platforms. The partitioning problem is expressed as an ILP, which is then solved approximately to yield a partitioning algorithm with a speedup factor of $(11 + 4\sqrt{3})$ or ≈ 17.9: If a task system can be partitioned upon a particular platform by an optimal algorithm, then it can be partitioned by the algorithm in [147] upon a platform in which each processor is faster by a factor 17.9. Although 17.9 is perhaps too large a factor for the algorithm in [147] to be considered practically significant, it does serve to establish the existence of a polynomial-time approximation algorithm and leaves open the possibility that the speedup factor could be reduced with further insights.

For the special case when the number of processors is a constant, the approximation algorithm in [147] reduces to a PTAS.

Chapter 23
Looking Ahead

As stated in the introduction, the sheer volume of excellent research on various aspects of multiprocessor scheduling theory meant that we could not possibly hope to detail, or even mention in passing, all the important and interesting results. Instead, we have selected a self-contained collection of topics from the vast body of research literature on multiprocessor real-time scheduling theory, and have attempted to provide a cohesive, relatively deep, and complete coverage of these topics. Our choice of topics for inclusion was primarily guided by their relevance to the discipline, the maturity of our knowledge about them, and the requirement that all taken together, they should comprise a complete narrative for a substantial and important subset of multiprocessor real-time scheduling theory. In addition, our personal preferences, biases, and expertise undoubtedly played a role in determining which topics got into this book, and which stayed out. In any event, there are many important aspects of multiprocessor real-time scheduling theory that did not see much discussion in this book; we briefly list some such topics in this chapter.

23.1 Soft and Firm Real-Time Scheduling

Here, we have focused exclusively on hard-real-time scheduling upon multiprocessors. There is a cohesive theory of *soft*-real-time scheduling centered around the concept of bounded tardiness—jobs are allowed to miss deadlines provided an a priori upper bound on the degree of such tardiness can be provided [78, 83, 130–132]. Several dissertations have been written on the subject of bounded-tardiness scheduling—see, e.g., [82], and the references therein. Time-utility functions (TUF) [160] represent another somewhat popular approach to soft-real-time scheduling; TUF-centered multiprocessor scheduling algorithms and schedulability analysis is reported in [69, 70, 95].

23.2 Multiple Resource Types

The research in this book has focused exclusively upon CPU scheduling. In an actual real-time system, there may be multiple resources, including memory, communication bandwidth, energy, etc., that are available in limited quantities and therefore need to be allocated in an efficient manner. There is a large volume of research on the concurrent consideration of multiple resources; see, e.g., [33, 43, 85, 94, 151, 152, 170, 174]. There has been some work on implementing resource-sharing protocols that were originally developed for uniprocessor systems such as the Priority Ceiling Protocol [166], to multiprocessor platforms (e.g., [158, 159]); and some new and exciting work on developing entirely new protocols exclusively for multiprocessor platforms (see, e.g., [65], and the many references therein).

23.3 Interprocessor Communication

The platform model we have adopted in this book assumes that communication between processors occurs instantaneously and incurs no cost. Although this is a reasonable abstraction to start with (in Sect. 2.3.3 we explained how for priority-driven scheduling algorithms some such costs—that of interprocessor migrations—could be accounted for by inflating the WCET parameters of jobs by the maximum cost of a migration), the abstraction results in excessive pessimism as platforms become more complex and communication costs become increasingly nonuniform. This is a significant shortcoming of much of the existing body of research into multiprocessor scheduling theory today; this shortcoming is being addressed in extended platform models that consider, for example, routing issues for networks-on-chip (see, e.g., [66, 110]).

References

1. Openmp* runtime library. https://www.openmprtl.org/
2. Ahmed, K., Schuegraf, K.: Transistor wars: Rival architectures face off in a bid to keep Moore's law alive. IEEE Spectrum **48**(11), 44–49 (2011)
3. Albers, K., Slomka, F.: An event stream driven approximation for the analysis of real-time systems. In: Proceedings of the EuroMicro Conference on Real-Time Systems, pp. 187–195. IEEE Computer Society Press, Catania, Sicily (2004)
4. Andersson, B.: Static-priority scheduling on multiprocessors. Ph.D. thesis, Department of Computer Engineering, Chalmers University (2003)
5. Andersson, B.: Global static-priority preemptive multiprocessor scheduling with utilization bound 38%. In: Proceedings of the 12th International Conference on Principles of Distributed Systems. IEEE Computer Society Press, Luxor, Egypt (2008)
6. Andersson, B., Baruah, S., Jansson, J.: Static-priority scheduling on multiprocessors. In: Proceedings of the IEEE Real-Time Systems Symposium, pp. 193–202. IEEE Computer Society Press (2001)
7. Andersson, B., Jonsson, J.: Fixed-priority preemptive multiprocessor scheduling: To partition or not to partition. In: Proceedings of the International Conference on Real-Time Computing Systems and Applications, pp. 337–346. IEEE Computer Society Press, Cheju Island, South Korea (2000)
8. Andersson, B., Jonsson, J.: Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling. In: Proceedings of the Real-Time Systems Symposium—Work-In-Progress Session. Orlando, FL (2000)
9. Andersson, B., Jonsson, J.: Some insights on fixed-priority preemptive non-partitioned multiprocessor scheduling. Tech. Rep. 01-2, Department of Computer Engineering, Chalmers University of Technology, Sweden (2001). Submitted for publication
10. Andersson, B., Jonsson, J.: The utilization bounds of partitioned and pfair static-priority scheduling on multiprocessors are 50%. In: Proceedings of the EuroMicro Conference on Real-Time Systems, pp. 33–40. IEEE Computer Society Press, Porto, Portugal (2003)
11. Andersson, B., Raravi, G., Bletsas, K.: Assigning real-time tasks on heterogeneous multiprocessors with two unrelated types of processors. In: Proceedings of the Real-Time Systems Symposium, pp. 239–248. IEEE Computer Society Press, San Diego, CA (2010)
12. Audsley, N.: On priority assignment in fixed priority scheduling. Information Processing Letters **79**(1), 39–44 (2001)
13. Audsley, N., Burns, A., Wellings, A.: Deadline monotonic scheduling theory and application. Control Engineering Practice **1**(1), 71–78 (1993)

S. Baruah et al., *Multiprocessor Scheduling for Real-Time Systems,*
Embedded Systems, DOI 10.1007/978-3-319-08696-5

14. Audsley, N.C.: Optimal priority assignment and feasibility of static priority tasks with arbitrary start times. Tech. rep., The University of York, England (1991)
15. Audsley, N.C.: Flexible scheduling in hard-real-time systems. Ph.D. thesis, Department of Computer Science, University of York (1993)
16. Baker, K.R., Trietsch, D.: Principles of Sequencing and Scheduling. Wiley Publishing (2009)
17. Baker, T.: An analysis of EDF schedulability on a multiprocessor. Tech. Rep. TR-030202, Department of Computer Science, Florida State University (2003)
18. Baker, T.: Multiprocessor EDF and deadline monotonic schedulability analysis. In: Proceedings of the IEEE Real-Time Systems Symposium, pp. 120–129. IEEE Computer Society Press (2003)
19. Baker, T.: An analysis of EDF schedulability on a multiprocessor. IEEE Transactions on Parallel and Distributed Systems **16**(8), 760–768 (2005)
20. Baker, T.: Comparison of empirical success rates of global vs. partitioned fixed-priority and EDF scheduling for hard real time. Tech. Rep. TR-050601, Department of Computer Science, Florida State University (2005)
21. Baker, T.: An analysis of fixed-priority schedulability on a multiprocessor. Real-Time Systems: The International Journal of Time-Critical Computing **32**(1–2), 49–71 (2006)
22. Baker, T., Baruah, S.: Schedulability analysis of multiprocessor sporadic task systems. In: S.H. Son, I. Lee, J.Y.T. Leung (eds.) Handbook of Real-Time and Embedded Systems. Chapman Hall/CRC Press (2007)
23. Baker, T., Baruah, S.: Sustainable multiprocessor scheduling of sporadic task systems. In: Proceedings of the EuroMicro Conference on Real-Time Systems. IEEE Computer Society Press, Dublin (2008)
24. Baker, T., Cirinei, M.: A necessary and sometimes sufficient condition for the feasibility of sets of sporadic hard-deadline tasks. In: Proceedings of the IEEE Real-time Systems Symposium, pp. 178–187. IEEE Computer Society Press, Rio de Janeiro (2006)
25. Baker, T., Cirinei, M., Bertogna, M.: EDZL scheduling analysis. Real-Time Syst. **40**, 264–289 (2008)
26. Baker, T., Fisher, N., Baruah, S.: Algorithms for determining the load of a sporadic task system. Tech. Rep. TR-051201, Department of Computer Science, Florida State University (2005)
27. Baker, T.P., Baruah, S.K.: An analysis of global edf schedulability for arbitrary-deadline sporadic task systems. Real-Time Syst. **43**(1), 3–24 (2009)
28. Baruah, S.: Feasibility analysis of preemptive real-time systems upon heterogeneous multiprocessor platforms. In: Proceedings of the 25th IEEE Real-Time Systems Symposium, RTSS 2004, pp. 37–46. IEEE Computer Society Press, Lisbon, Portugal (2004)
29. Baruah, S.: Optimal utilization bounds for the fixed-priority scheduling of periodic task systems on identical multiprocessors. IEEE Transactions on Computers **53**(6) (2004)
30. Baruah, S.: Partitioning real-time tasks among heterogeneous multiprocessors. In: Proceedings of the Thirty-third Annual International Conference on Parallel Processing, pp. 467–474. IEEE Computer Society Press, Montreal, Canada (2004)
31. Baruah, S.: Techniques for multiprocessor global schedulability analysis. In: Proceedings of the 28th Real-Time Systems Symposium, RTSS 2007, pp. 119–128. IEEE Computer Society Press, Tucson, AZ (2007)
32. Baruah, S.: Partitioned EDF scheduling: A closer look. Real-Time Systems: The International Journal of Time-Critical Computing **49**(6), 715–729 (2013). To appear
33. Baruah, S.: Partitioning sporadic task systems upon memory-constrained multiprocessors. ACM Transactions on Embedded Computing Systems **12**(3), 78:1–78:18 (2013)
34. Baruah, S.: Improved multiprocessor global schedulability analysis of sporadic dag task systems. In: Proceedings of the 2014 26th Euromicro Conference on Real-Time Systems, ECRTS '14, pp. 97–105. IEEE Computer Society Press, Madrid (Spain) (2014)
35. Baruah, S., Baker, T.: Schedulability analysis of global EDF. Real-Time Systems **38**(3), 223–235 (2008)

36. Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stiller, S.: Implementation of a speedup-optimal global EDF schedulability test. In: Proceedings of the EuroMicro Conference on Real-Time Systems. IEEE Computer Society Press, Dublin (2009)
37. Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stiller, S.: Improved multiprocessor global schedulability analysis. Real-Time Systems: The International Journal of Time-Critical Computing **46**(1), 3–24 (2010)
38. Baruah, S., Bonifaci, V., Marchetti-Spaccamela, A., Stougie, L., Wiese, A.: A generalized parallel task model for recurrent real-time processes. In: Proceedings of the IEEE Real-Time Systems Symposium, RTSS 2012, pp. 63–72. San Juan, Puerto Rico (2012)
39. Baruah, S., Brandenburg, B.: Multiprocessor feasibility analysis of recurrent task systems with specified processor affinities. In: Proceedings of the 34th IEEE Real-Time Systems Symposium, RTSS 2013. IEEE Computer Society Press, Vancouver, BC (2013)
40. Baruah, S., Burns, A.: Sustainable scheduling analysis. In: Proceedings of the IEEE Real-time Systems Symposium, pp. 159–168. IEEE Computer Society Press, Rio de Janeiro (2006)
41. Baruah, S., Chen, D., Gorinsky, S., Mok, A.: Generalized multiframe tasks. Real-Time Systems: The International Journal of Time-Critical Computing **17**(1), 5–22 (1999)
42. Baruah, S., Cohen, N., Plaxton, G., Varvel, D.: Proportionate progress: A notion of fairness in resource allocation. Algorithmica **15**(6), 600–625 (1996)
43. Baruah, S., Fisher, N.: A dynamic-programming approach to task partitioning among memory-constrained multiprocessors. In: Proceedings of the International Conference on Real-time Computing Systems and Applications. Springer-Verlag, Gothenburg, Sweden (2004)
44. Baruah, S., Fisher, N.: The partitioned multiprocessor scheduling of sporadic task systems. In: Proceedings of the IEEE Real-Time Systems Symposium, pp. 321–329. IEEE Computer Society Press, Miami, Florida (2005)
45. Baruah, S., Fisher, N.: The partitioned multiprocessor scheduling of deadline-constrained sporadic task systems. IEEE Transactions on Computers **55**(7), 918–923 (2006)
46. Baruah, S., Fisher, N.: The partitioned dynamic-priority scheduling of sporadic task systems. Real-Time Systems: The International Journal of Time-Critical Computing **36**(3), 199–226 (2007)
47. Baruah, S., Fisher, N.: Non-migratory feasibility and migratory schedulability analysis of multiprocessor real-time systems. Real-Time Systems: The International Journal of Time-Critical Computing (1–3) (2008)
48. Baruah, S., Gehrke, J., Plaxton, G.: Fast scheduling of periodic tasks on multiple resources. In: Proceedings of the Ninth International Parallel Processing Symposium, pp. 280–288. IEEE Computer Society Press (1995). Extended version available via anonymous ftp from `ftp.cs.utexas.edu`, as Tech Report TR–95–02
49. Baruah, S., Howell, R., Rosier, L.: Algorithms and complexity concerning the preemptive scheduling of periodic, real-time tasks on one processor. Real-Time Systems: The International Journal of Time-Critical Computing **2**, 301–324 (1990)
50. Baruah, S., Howell, R., Rosier, L.: Feasibility problems for recurring tasks on one processor. Theoretical Computer Science **118**(1), 3–20 (1993)
51. Baruah, S., Mok, A., Rosier, L.: Preemptively scheduling hard-real-time sporadic tasks on one processor. In: Proceedings of the 11th Real-Time Systems Symposium, pp. 182–190. IEEE Computer Society Press, Orlando, Florida (1990)
52. Baruah, S.K.: The non-preemptive scheduling of periodic tasks upon multiprocessors. Real-Time Syst. **32**(1–2), 9–20 (2006). DOI 10.1007/s11241-006-4961-9. URL http://dx.doi.org/10.1007/s11241-006-4961-9
53. Baruah, S.K., Cohen, N.K., Plaxton, C.G., Varvel, D.A.: Proportionate progress: A notion of fairness in resource allocation. In: Proceedings of the Twenty-fifth Annual ACM Symposium on Theory of Computing, STOC '93, pp. 345–354. ACM, New York, NY, USA (1993). DOI 10.1145/167088.167194. URL http://doi.acm.org/10.1145/167088.167194
54. Baruah, S.K., Pruhs, K.: Open problems in real-time scheduling. Journal of Scheduling **13**(6), 577–582 (2010)

55. Bertogna, M.: Real-time scheduling analysis for multiprocessor platforms. Ph.D. thesis, Scuola Superiore Santa Anna, Pisa, Italy (2008)
56. Bertogna, M.: Evaluation of existing schedulability tests for global EDF. In: ICPPW '09: Proceedings of the 2009 International Conference on Parallel Processing Workshops, pp. 11–18. IEEE Computer Society, Washington, DC, USA (2009). DOI http://dx.doi.org/10.1109/ICPPW.2009.12
57. Bertogna, M., Baruah, S.: Tests for global edf schedulability analysis. J. Syst. Archit. **57**(5), 487–497 (2011)
58. Marko Bertogna and Michele Cirinei. "Response-Time Analysis for Globally Scheduled Symmetric Multiprocessor Platforms", Proceedings of the IEEE International Real-Time Systems Symposium (RTSS 2007), Tucson, Arizona. December 2007.
59. Bertogna, M., Cirinei, M., Lipari, G.: Improved schedulability analysis of EDF on multiprocessor platforms. In: Proceedings of the EuroMicro Conference on Real-Time Systems, pp. 209–218. IEEE Computer Society Press, Palma de Mallorca, Balearic Islands, Spain (2005)
60. Bertogna, M., Cirinei, M., Lipari, G.: New schedulability tests for real-time tasks sets scheduled by deadline monotonic on multiprocessors. In: Proceedings of the 9th International Conference on Principles of Distributed Systems. IEEE Computer Society Press, Pisa, Italy (2005)
61. Blum, M., Floyd, R.W., Pratt, V., Rivest, R.L., Tarjan, R.E.: Time bounds for selection. Journal of Computer and System Sciences **7**(4), 448–461 (1973)
62. Bonifaci, V., Marchetti-Spaccamela, A.: Feasibility analysis of sporadic real-time multiprocessor task systems. In: M. de Berg, U. Meyer (eds.) ESA (2), *Lecture Notes in Computer Science*, vol. 6347, pp. 230–241. Springer (2010)
63. Bonifaci, V., Marchetti-Spaccamela, A., Stiller, S.: A constant-approximate feasibility test for multiprocessor real-time scheduling. In: Proceedings of the 16th Annual European Symposium on Algorithms, pp. 210–221. Karlsruhe, Germany (2008)
64. Bonifaci, V., Marchetti-Spaccamela, A., Stiller, S., Wiese, A.: Feasibility analysis in the sporadic DAG task model. In: Proceedings of the 2013 25th Euromicro Conference on Real-Time Systems, ECRTS '13, pp. 225–233. Paris (France) (2013)
65. Brandenburg, B.B.: Scheduling and locking in multiprocessor real-time operating systems. Ph.D. thesis, The University of North Carolina at Chapel Hill (2011)
66. Burns, A., Indrusiak, L.S., Shi, Z.: Schedulability analysis for real time on-chip communication with wormhole switching. Int. J. Embed. Real-Time Commun. Syst. **1**(2), 1–22 (2010)
67. Carpenter, J., Funk, S., Holman, P., Srinivasan, A., Anderson, J., Baruah, S.: A categorization of real-time multiprocessor scheduling problems and algorithms. In: J.Y.T. Leung (ed.) Handbook of Scheduling: Algorithms, Models, and Performance Analysis. CRC Press LLC (2003)
68. Chattopadhyay, B., Baruah, S.: A lookup-table driven approach to partitioned scheduling. In: Proceedings of the IEEE Real-Time Technology and Applications Symposium (RTAS). IEEE Computer Society Press, Chicago (2011)
69. Cho, H., Ravindran, B., Jensen, E.D.: Utility accrual real-time scheduling for multiprocessor embedded systems. Journal of Parallel and Distributed Computing **70**(2), 101–110 (2010). DOI http://dx.doi.org/10.1016/j.jpdc.2009.10.003.
URL http://www.sciencedirect.com/science/article/pii/S0743731509001865
70. Cho, H., Wu, H., Ravindran, B., Jensen, E.D.: On multiprocessor utility accrual real-time scheduling with statistical timing assurances. In: In IFIP Embedded and Ubiquitous Computing (EUC (2006))
71. Cirinei, M., Baker, T.P.: EDZL scheduling analysis. In: Proceedings of the EuroMicro Conference on Real-Time Systems. IEEE Computer Society Press, Pisa, Italy (2007)
72. Collette, S., Cucu, L., Goossens, J.: Integrating job parallelism in real-time scheduling theory. Information Processing Letters **106**(5), 180–187 (2008)
73. Dantzig, G.B.: Linear Programming and Extensions. Princeton University Press (1963)

74. Davis, R., Burns, A.: Improved priority assignment for global fixed priority pre-emptive scheduling in multiprocessor real-time systems. Real-Time Systems **47**(1), 1–40 (2011). DOI 10.1007/s11241-010-9106-5. URL http://dx.doi.org/10.1007/s11241-010-9106-5
75. Davis, R., Burns, A., Marinho, J., Nelis, V., Petters, S., Bertogna, M.: Global fixed priority scheduling with deferred pre-emption. In: Embedded and Real-Time Computing Systems and Applications (RTCSA), 2013 IEEE 19th International Conference on, pp. 1–11 (2013)
76. Dertouzos, M.: Control robotics : the procedural control of physical processors. In: Proceedings of the IFIP Congress, pp. 807–813 (1974)
77. Dertouzos, M., Mok, A.: Multiprocessor scheduling in a hard real-time environment. IEEE Transactions on Software Engineering **15**(12), 1497–1506 (1989)
78. Devi, U., Anderson, J.: Tardiness bounds for global EDF scheduling on a multiprocessor. In: Proceedings of the IEEE Real-Time Systems Symposium, pp. 330–341. IEEE Computer Society Press, Miami, FL (2005)
79. Dhall, S.K., Liu, C.L.: On a real-time scheduling problem. Operations Research **26**, 127–140 (1978)
80. E. Coffman, J., Denning, P.J.: Operating Systems Theory. Prentice-Hall, Englewood Cliffs, NJ (1973)
81. Eisenbrand, F., Rothvoß, T.: EDF-schedulability of synchronous periodic task systems is coNP-hard. In: Proceedings of the Annual ACM-SIAM Symposium on Discrete Algorithms (2010)
82. Erickson, J.: Managing tardiness bounds and overload in soft real-time systems. Ph.D. thesis, The University of North Carolina at Chapel Hill (2014)
83. Erickson, J., Devi, U., Baruah, S.: Improved tardiness bounds for global edf. In: Proceedings of the EuroMicro Conference on Real-Time Systems. IEEE Computer Society Press, Brussels (2010)
84. Fisher, N.: The multiprocessor real-time scheduling of general task systems. Ph.D. thesis, Department of Computer Science, The University of North Carolina at Chapel Hill (2007)
85. Fisher, N., Anderson, J., Baruah, S.: Task partitioning upon memory-constrained multiprocessors. In: Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 416–421. IEEE Computer Society Press, Hong Kong (2005)
86. Fisher, N., Baker, T., Baruah, S.: Algorithms for determining the demand-based load of a sporadic task system. In: Proceedings of the International Conference on Real-time Computing Systems and Applications. IEEE Computer Society Press, Sydney, Australia (2006)
87. Fisher, N., Baruah, S.: A polynomial-time approximation scheme for feasibility analysis in static-priority systems with bounded relative deadlines. In: Proceedings of the 13th International Conference on Real-Time Systems. Paris, France (2005)
88. Fisher, N., Baruah, S.: Global static-priority scheduling of sporadic task systems on multiprocessor platforms. In: Proceeding of the IASTED International Conference on Parallel and Distributed Computing and Systems. IASTED, Dallas, TX (2006)
89. Fisher, N., Baruah, S., Baker, T.: The partitioned scheduling of sporadic tasks according to static priorities. In: Proceedings of the EuroMicro Conference on Real-Time Systems, pp. 118–127. IEEE Computer Society Press, Dresden, Germany (2006)
90. Fisher, N., Goossens, J., Baruah, S.: Optimal on-line multiprocessor scheduling of sporadic real-time tasks is impossible. Real-Time Systems: The International Journal of Time-Critical Computing **45**, 26–71 (2010)
91. Ford, L., Fulkerson, D.: Flows in Networks. Princeton University Press, Princeton, NJ (1962)
92. Funk, S.: EDF scheduling on heterogeneous multiprocessors. Ph.D. thesis, Department of Computer Science, The University of North Carolina at Chapel Hill (2004)
93. Funk, S., Goossens, J., Baruah, S.: On-line scheduling on uniform multiprocessors. In: Proceedings of the IEEE Real-Time Systems Symposium, pp. 183–192. IEEE Computer Society Press (2001)
94. Gai, P., Lipari, G., di Natale, M.: Minimizing memory utilization of real-time task sets in single and multi-processor systems-on-a-chip. In: Proceedings of the IEEE Real-Time Systems Symposium. IEEE Computer Society Press (2001)

95. Garyali, P., Dellinger, M., Ravindran, B.: On best-effort utility accrual real-time scheduling on multiprocessors. In: C. Lu, T. Masuzawa, M. Mosbah (eds.) Principles of Distributed Systems, *Lecture Notes in Computer Science*, vol. 6490, pp. 270–285. Springer Berlin Heidelberg (2010)
96. Goossens, J., Funk, S., Baruah, S.: Priority-driven scheduling of periodic task systems on multiprocessors. Real Time Systems **25**(2–3), 187–205 (2003)
97. Graham, R.: Bounds for certain multiprocessing anomalies. Bell System Technical Journal **45**, 1563–1581 (1966)
98. Graham, R.: Bounds on multiprocessor timing anomalies. SIAM Journal on Applied Mathematics **17**, 416–429 (1969)
99. Graham, R.L., Lawler, E.L., Lenstra, J.K., Kan, A.H.G.R.: Optimization and approximation in deterministic sequencing and scheduling: A survey. Ann. Discrete Mathematics **5**, 287–326 (1979)
100. Guan, N., Stigge, M., Yi, W., Yu, G.: Fixed-priority multiprocessor scheduling with Liu and Layland's utilization bound. In: Real-Time and Embedded Technology and Applications Symposium (RTAS), 2010 16th IEEE, pp. 165–174 (2010)
101. Guan, N., Stigge, M., Yi, W., Yu, G.: Parametric utilization bounds for fixed-priority multiprocessor scheduling. In: Parallel Distributed Processing Symposium (IPDPS), 2012 IEEE 26th International, pp. 261–272 (2012)
102. Guan, N., Yi, W., Gu, Z., Deng, Q., Yu, G.: New schedulability test conditions for non-preemptive scheduling on multiprocessor platforms. In: Proceedings of the Real-Time Systems Symposium. IEEE Computer Society Press, Barcelona (2008)
103. Gujarati, A., Cerqueira, F., Brandenburg, B.: Schedulability analysis of the Linux Push and Pull Scheduler with arbitrary processor affinities. In: Proceedings of the 2013 25th Euromicro Conference on Real-Time Systems, ECRTS '13. IEEE Computer Society Press, Paris (France) (2013)
104. Ha, R.: Validating timing constraints in multiprocessor and distributed systems. Ph.D. thesis, Department of Computer Science, University of Illinois at Urbana-Champaign (1995). Available as Technical Report No. UIUCDCS-R-95-1907
105. Ha, R., Liu, J.W.S.: Validating timing constraints in multiprocessor and distributed real-time systems. Tech. Rep. UIUCDCS-R-93-1833, Department of Computer Science, University of Illinois at Urbana-Champaign (1993)
106. Ha, R., Liu, J.W.S.: Validating timing constraints in multiprocessor and distributed real-time systems. In: Proceedings of the 14th IEEE International Conference on Distributed Computing Systems. IEEE Computer Society Press, Los Alamitos (1994)
107. Hochbaum, D., Shmoys, D.: Using dual approximation algorithms for scheduling problems: Theoretical and practical results. Journal of the ACM **34**(1), 144–162 (1987)
108. Horn, W.: Some simple scheduling algorithms. Naval Research Logistics Quarterly **21**, 177–185 (1974)
109. Indrusiak, L.S.: End-to-end schedulability tests for multiprocessor embedded systems based on networks-on-chip with priority-preemptive arbitration. Journal of Systems Architecture **60**(7), 553–561 (2014). DOI http://dx.doi.org/10.1016/j.sysarc.2014.05.002.
URL http://www.sciencedirect.com/science/article/pii/S1383762114000800
110. Jansen, K., Porkolab, L.: Improved approximation schemes for scheduling unrelated parallel machines. In: Proceedings of the thirty-first annual ACM symposium on Theory of computing, pp. 408–417. ACM Press (1999). DOI http://doi.acm.org/10.1145/301250.301361
111. Johnson, D.: Fast algorithms for bin packing. Journal of Computer and Systems Science **8**(3), 272–314 (1974)
112. Johnson, D.S.: Near-optimal bin packing algorithms. Ph.D. thesis, Department of Mathematics, Massachusetts Institute of Technology (1973)
113. Joseph, M., Pandya, P.: Finding response times in a real-time system. The Computer Journal **29**(5), 390–395 (1986)
114. Kalyanasundaram, B., Pruhs, K.: Speed is as powerful as clairvoyance. In: 36th Annual Symposium on Foundations of Computer Science (FOCS'95), pp. 214–223. IEEE Computer Society Press, Los Alamitos (1995)

115. Kalyanasundaram, B., Pruhs, K.: Speed is as powerful as clairvoyance. Journal of the ACM **37**(4), 617–643 (2000)
116. Kalyanasundaram, B., Pruhs, K., Torng, E.: Errata: A new algorithm for scheduling periodic, real-time tasks. Algorithmica **28**(3), 269–270 (2000)
117. Karmakar, N.: A new polynomial-time algorithm for linear programming. Combinatorica **4**, 373–395 (1984)
118. Khachiyan, L.: A polynomial algorithm in linear programming. Dokklady Akademiia Nauk SSSR **244**, 1093–1096 (1979)
119. Lakshmanan, K., Kato, S., Rajkumar, R.: Scheduling parallel real-time tasks on multi-core processors. In: RTSS, pp. 259–268. IEEE Computer Society (2010)
120. Lawler, E., Labetoulle, J.: On preemptive scheduling of unrelated parallel processors by linear programming. Journal of the ACM **25**(4), 612–619 (1978)
121. Lawler, E.L.: Optimal sequencing of a single machine subject to precedence constraints. Management Science **19**(5), 544–546 (1973)
122. Lee, J., Shin, I.: Limited carry-in technique for real-time multi-core scheduling. Journal of Systems Architecture—Embedded Systems Design **59**(7), 372–375 (2013)
123. Lee, J., Shin, K.: Controlling preemption for better schedulability in multi-core systems. In: Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd, pp. 29–38 (2012)
124. Lee, S.K., Epley, D.: On-line scheduling algorithms of real-time sporadic tasks in multiprocessor systems. Tech. Rep. 92-3, University of Iowa. Dept. of Computer Science (1992)
125. Lehoczky, J., Sha, L., Ding, Y.: The rate monotonic scheduling algorithm: Exact characterization and average case behavior. In: Proceedings of the Real-Time Systems Symposium—1989, pp. 166–171. IEEE Computer Society Press, Santa Monica, California, USA (1989)
126. Lenstra, J.K., Rinnooy Kan, A.H.G.: Complexity of scheduling under precedence constraints. Operations Research **26**(1), 22–35 (1978)
127. Lenstra, J.K., Shmoys, D., Tardos, E.: Approximation algorithms for scheduling unrelated parallel machines. In: A.K. Chandra (ed.) Proceedings of the 28th Annual Symposium on Foundations of Computer Science, pp. 217–224. IEEE Computer Society Press, Los Angeles, CA (1987)
128. Lenstra, J.K., Shmoys, D., Tardos, E.: Approximation algorithms for scheduling unrelated parallel machines. Mathematical Programming **46**, 259–271 (1990)
129. Leontyev, H., Anderson, J.: Tardiness bounds for EDF scheduling on multi-speed multicore platforms. In: Proceedings of the IEEE International Conference on Embedded and Real-Time Computing Systems and Applications, pp. 103–111. IEEE Computer Society Press (2007)
130. Leontyev, H., Anderson, J.: Tardiness bounds for FIFO scheduling on multiprocessors. In: Proceedings of the EuroMicro Conference on Real-Time Systems, pp. 71–80. IEEE Computer Society Press (2007)
131. Leontyev, H., Anderson, J.: Generalized tardiness bounds for global multiprocessor scheduling. Real Time Systems (2010)
132. Leung, J.Y.T.: A new algorithm for scheduling periodic real-time tasks. Algorithmica **4**, 209–219 (1989)
133. Leung, J.Y.T., Merrill, M.: A note on the preemptive scheduling of periodic, real-time tasks. Information Processing Letters **11**, 115–118 (1980)
134. Leung, J.Y.T., Whitehead, J.: On the complexity of fixed-priority scheduling of periodic, real-time tasks. Performance Evaluation **2**, 237–250 (1982)
135. Levin, G., Funk, S., Sadowski, C., Pye, I., Brandt, S.: Dp-fair: A simple model for understanding optimal multiprocessor scheduling. In: Real-Time Systems (ECRTS), 2010 22nd Euromicro Conference on, pp. 3–13 (2010)
136. Li, J., Agrawal, K., Lu, C., Gill, C.D.: Analysis of global EDF for parallel tasks. In: Proceedings of the 2013 25th Euromicro Conference on Real-Time Systems, ECRTS '13, pp. 3–13. Paris (France) (2013)

137. Li, J., Saifullah, A., Agrawal, K., Gill, C., Lu, C.: Capacity augmentation bound of federated scheduling for parallel dag tasks. In: Proceedings of the 2012 26th Euromicro Conference on Real-Time Systems, ECRTS '14. IEEE Computer Society Press, Madrid (Spain) (2014)
138. Liu, C., Layland, J.: Scheduling algorithms for multiprogramming in a hard real-time environment. Journal of the ACM **20**(1), 46–61 (1973)
139. Liu, J.W.S.: Real-Time Systems. Prentice-Hall, Inc., Upper Saddle River, New Jersey 07458 (2000)
140. Lopez, J.M., Diaz, J.L., Garcia, D.F.: Minimum and maximum utilization bounds for multiprocessor rate-monotonic scheduling. IEEE Transactions on Parallel and Distributed Systems **15**(7), 642–653 (2004)
141. Lopez, J.M., Diaz, J.L., Garcia, D.F.: Utilization bounds for EDF scheduling on real-time multiprocessor systems. Real-Time Systems: The International Journal of Time-Critical Computing **28**(1), 39–68 (2004)
142. Lopez, J.M., Garcia, M., Diaz, J.L., Garcia, D.F.: Worst-case utilization bound for EDF scheduling in real-time multiprocessor systems. In: Proceedings of the EuroMicro Conference on Real-Time Systems, pp. 25–34. IEEE Computer Society Press, Stockholm, Sweden (2000)
143. Lupu, I., Courbin, P., George, L., Goossens, J.: Multi-criteria evaluation of partitioning schemes for real-time systems. In: Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on, pp. 1–8 (2010)
144. Maheswaran, M., Braun, T.D., Siegel, H.J.: Heterogeneous distributed computing. In: J.G. Webster (ed.) Encyclopedia of Electrical and Electronic Engineering, vol. 8. John Wiley, New York, NY (1999)
145. Manimaran, G., Murthy, C.S.R., Ramamritham, K.: A new approach for scheduling of parallelizable tasks in real-time multiprocessor systems. REAL-TIME SYSTEMS **15**, 39–60 (1998)
146. Marchetti-Spaccamela, A., Rutten, C., van der Ster, S., Wiese, A.: Assigning sporadic tasks to unrelated parallel machines. In: A. Czumaj, K. Mehlhorn, A.M. Pitts, R. Wattenhofer (eds.) ICALP (1), *Lecture Notes in Computer Science*, vol. 7391, pp. 665–676. Springer (2012)
147. Marinho, J., Nelis, V., Petters, S., Bertogna, M., Davis, R.: Limited pre-emptive global fixed task priority. In: Real-Time Systems Symposium (RTSS), 2013 IEEE 34th, pp. 182–191 (2013)
148. Mok, A.: Fundamental design problems of distributed systems for the hard-real-time environment. Ph.D. thesis, Laboratory for Computer Science, Massachusetts Institute of Technology (1983). Available as Technical Report No. MIT/LCS/TR-297
149. Mok, A.: Task management techniques for enforcing ED scheduling on a periodic task set. In: Proceedings of the 5th IEEE Workshop on Real-Time Software and Operating Systems, pp. 42–46. Washington D.C. (1988)
150. Niemeier, M.: Approximation Algorithms for Modern Multi-Processor Scheduling Problems. Ph.D. thesis, Lausanne (2012). 10.5075/epfl-thesis-5561
151. Niemeier, M., Wiese, A., Baruah, S.: Partitioned real-time scheduling on heterogeneous shared-memory multiprocessors. In: Proceedings of the EuroMicro Conference on Real-Time Systems. IEEE Computer Society Press, Porto, PT. (2011)
152. Oh, D.I., Baker, T.: Utilization bounds for N-processor rate monotone scheduling with static processor assignment. Real-Time Systems: The International Journal of Time-Critical Computing **15**, 183–192 (1998)
153. Papadimitriou, C.H.: On the complexity of integer programming. Journal of the ACM **28**(4), 765–768 (1981)
154. Phillips, C.A., Stein, C., Torng, E., Wein, J.: Optimal time-critical scheduling via resource augmentation. In: Proceedings of the Twenty-Ninth Annual ACM Symposium on Theory of Computing, pp. 140–149. El Paso, Texas (1997)
155. Piao, X., Han, S., Kim, H., Park, M., Cho, Y., Cho, S.: Predictability of earliest deadline zero laxity algorithm for multiprocessor real-time systems. In: Object and Component-Oriented

Real-Time Distributed Computing, 2006. ISORC 2006. Ninth IEEE International Symposium on (2006)
156. Pinedo, M.L.: Scheduling: Theory, Algorithms, and Systems, 4th edn. Springer Publishing Company, Incorporated (2012)
157. Rajkumar, R.: Real-time synchronization protocols for shared memory multiprocessors. In: Proceedings of the International Conference on Distributed Computing Systems, pp. 116–123 (1990)
158. Rajkumar, R., Sha, L., Lehoczky, J.: Real-time synchronization protocols for multiprocessors. In: Proceedings of the Ninth IEEE Real-Time Systems Symposium, pp. 259–269. IEEE (1988)
159. Ravindran, B., Jensen, E.D., Li, P.: On recent advances in time/utility function real-time scheduling and resource management. In: Proceedings of the IEEE International Symposium on Object-oriented Real-time Distributed Computing, pp. 55–60. IEEE Computer Society Press (2005)
160. Regnier, P., Lima, G., Massa, E., Levin, G., Brandt, S.: Run: Optimal multiprocessor real-time scheduling via reduction to uniprocessor. In: Real-Time Systems Symposium (RTSS), 2011 IEEE 32nd, pp. 104–115 (2011)
161. Ripoll, I., Crespo, A., Mok, A.K.: Improvement in feasibility testing for real-time tasks. Real-Time Systems: The International Journal of Time-Critical Computing **11**, 19–39 (1996)
162. Schranzhofer, A.: Efficiency and predictability in resource sharing multicore systems. Ph.D. thesis, ETH Zurich (2011). Diss. ETH No. 19556
163. Schrijver, A.: Theory of Linear and Integer Programming. John Wiley and Sons (1986)
164. Sha, L.: Migrating real-time software from single-core to multicore chips. Keynote speech given at the 9th IEEE International Symposium on Industrial Embedded Systems (SIES 2014), Pisa, Italy (June 18–20, 2014)
165. Sha, L., Rajkumar, R., Lehoczky, J.P.: Priority inheritance protocols: An approach to real-time synchronization. IEEE Transactions on Computers **39**(9), 1175–1185 (1990)
166. Srinivasan, A.: Efficient and flexible fair scheduling of real-time tasks on multiprocessors. Ph.D. thesis, Department of Computer Science, The University of North Carolina at Chapel Hill (2003)
167. Stigge, M.: Real-time workload models: Expressiveness vs. analysis efficiency. Ph.D. thesis, Ph.D. thesis, Uppsala University (2014)
168. Stigge, M., Yi, W.: Models for real-time workload: A survey. Proceedings of a conference organized in celebration of Professor Alan Burns sixtieth birthday p. 133 (2013)
169. Szymanek, R.W., Kuchcinski, K.: A constructive algorithm for memory-aware task assignment and scheduling. In: International Workshop on Hardware/Software Co-Design (CODES). ACM Press, Copenhagen, Denmark (2001)
170. Thekkilakattil, A., Baruah, S., Dobrin, R., Punnekkat, S.: The global limited preemptive earliest deadline first feasibility of sporadic real-time tasks. In: Proceedings of the 2012 26th Euromicro Conference on Real-Time Systems, ECRTS '14. IEEE Computer Society Press, Madrid (Spain) (2014)
171. Verschae, J., Wiese, A.: On the configuration-lp for scheduling on unrelated machines. In: C. Demetrescu, M.M. Halldórsson (eds.) ESA, *Lecture Notes in Computer Science*, vol. 6942, pp. 530–542. Springer (2011)
172. Wiese, A., Bonifaci, V., Baruah, S.: Partitioned EDF scheduling on a few types of unrelated multiprocessors. Real-Time Systems: The International Journal of Time-Critical Computing **49**(2), 219–238 (2013)
173. Yun, H., Yao, G., Pellizzoni, R., Caccamo, M., Sha, L.: Memory access control in multiprocessor for real-time systems with mixed criticality. In: Proceedings of the 2012 24th Euromicro Conference on Real-Time Systems, ECRTS '12. IEEE Computer Society Press, Pisa (Italy) (2012)
174. Zhu, D., Mosse, D., Melhem, R.: Multiple-resource periodic scheduling problem: how much fairness is necessary? In: Real-Time Systems Symposium, 2003. RTSS 2003. 24th IEEE, pp. 142–151 (2003)

Index